The jOOQ™ User Manual

SQL was never meant to be abstracted. To be confined in the narrow boundaries of heavy mappers, hiding the beauty and simplicity of relational data. SQL was never meant to be object-oriented. SQL was never meant to be anything other than... SQL!

Overview

This manual is divided into six main sections:

- Getting started with jOOQ
  This section will get you started with jOOQ quickly. It contains simple explanations about what jOOQ is, what jOOQ isn't and how to set it up for the first time
- SQL building
  This section explains all about the jOOQ syntax used for building queries through the query DSL and the query model API. It explains the central factories, the supported SQL statements and various other syntax elements
- Code generation
  This section explains how to configure and use the built-in source code generator
- SQL execution
  This section will get you through the specifics of what can be done with jOOQ at runtime, in order to execute queries, perform CRUD operations, import and export data, and hook into the jOOQ execution lifecycle for debugging
- Reference
  This section is a reference for elements in this manual

© 2009 - 2025 by Data Geekery™ GmbH.

Table of contents

1. Copyright, License, and Trademarks.... 17
2. Getting started with jOOQ.... 22
2.1. How to read this manual.... 22
2.2. The sample database used in this manual.... 23
2.3. Different use cases for jOOQ.... 24
2.3.1. jOOQ as a SQL builder without code generation.... 25
2.3.2. jOOQ as a SQL builder with code generation.... 26
2.3.3. jOOQ as a SQL executor.... 27
2.3.4. jOOQ for CRUD.... 27
2.3.5. jOOQ for PROs.... 28
2.4. Downloading jOOQ.... 28
2.5. Tutorials.... 31
2.5.1. jOOQ in 7 easy steps.... 31
2.5.1.1. Step 1: Preparation.... 31
2.5.1.2. Step 2: Your database.... 33
2.5.1.3. Step 3: Code generation.... 33
2.5.1.4. Step 4: Connect to your database.... 35
2.5.1.5. Step 5: Querying.... 36
2.5.1.6. Step 6: Iterating.... 36
2.5.1.7. Step 7: Explore!.... 37
2.5.2. Using jOOQ with Flyway.... 37
2.5.3. Using jOOQ with jbang.... 43
2.6. jOOQ and Java 8.... 44
2.7. jOOQ and Scala.... 45
2.8. jOOQ and Groovy.... 46
2.9. jOOQ and Kotlin.... 47
2.10. jOOQ and NoSQL.... 47
2.11. jOOQ and JPA.... 47
2.12. Build your own.... 48
2.13. jOOQ and backwards-compatibility.... 49
3. SQL building.... 51
3.1. The query DSL type.... 51
3.2. The DSLContext API.... 52
3.2.1. SQL Dialect.... 53
3.2.2. SQL Dialect Family.... 54
3.2.3. Connection vs. DataSource.... 54
3.2.4. Custom data.... 55
3.2.5. Custom ExecuteListeners.... 56
3.2.6. Custom Unwrappers.... 56 (new)
3.2.7. Custom Settings.... 57
3.2.7.1. Auto-attach Records.... 58
3.2.7.2. Backslash Escaping.... 58
3.2.7.3. Execute Logging.... 59
3.2.7.4. Fetch Warnings.... 59
3.2.7.5. Identifier style.... 59
3.2.7.6. Inline Threshold.... 60 (new)
3.2.7.7. IN-list Padding.... 61
3.2.7.8. JDBC Flags.... 62
3.2.7.9. Keyword style.... 62
3.2.7.10. Listener Invocation Order.... 63 (new)

© 2009 - 2025 by Data Geekery™ GmbH.

3.2.7.11. Locales ... 63 (new)
3.2.7.12. Map JPA Annotations ... 64
3.2.7.13. Object qualification ... 64
3.2.7.14. Optimistic Locking ... 64
3.2.7.15. Parameter name prefix ... 65 (new)
3.2.7.16. Parameter types ... 65
3.2.7.17. Parser Configuration ... 66 (new)
3.2.7.18. Reflection caching ... 67
3.2.7.19. Return all columns on store ... 67
3.2.7.20. Return Identity Value On Store ... 67 (new)
3.2.7.21. Runtime schema and table mapping ... 68
3.2.7.22. Scalar subqueries for stored functions ... 70
3.2.7.23. Statement Type ... 71
3.2.7.24. Updatable Primary Keys ... 71
3.2.8. Thread safety ... 72
3.3. SQL Statements (DML) ... 72
3.3.1. jOOQ's DSL and model API ... 73
3.3.2. The WITH clause ... 74
3.3.3. The WITH RECURSIVE clause ... 75
3.3.4. The SELECT statement ... 77
3.3.4.1. SELECT clause ... 78
3.3.4.1.1. Projection type safety ... 78
3.3.4.1.2. SelectField ... 78
3.3.4.1.3. SELECT * ... 79
3.3.4.1.4. SELECT * EXCEPT (...) ... 79 (new)
3.3.4.1.5. SELECT DISTINCT ... 81
3.3.4.1.6. SELECT DISTINCT ON ... 81
3.3.4.1.7. Convenience methods ... 83
3.3.4.2. FROM clause ... 83
3.3.4.2.1. JOIN operator ... 84
3.3.4.2.2. Implicit path JOIN ... 84
3.3.4.3. WHERE clause ... 87
3.3.4.4. CONNECT BY clause ... 87
3.3.4.5. GROUP BY clause ... 88
3.3.4.5.1. GROUP BY columns ... 88
3.3.4.5.2. GROUP BY ROLLUP ... 89
3.3.4.5.3. GROUP BY CUBE ... 91
3.3.4.5.4. GROUP BY GROUPING SETS ... 92
3.3.4.5.5. GROUP BY empty grouping set ... 94
3.3.4.6. HAVING clause ... 95
3.3.4.7. WINDOW clause ... 96
3.3.4.8. QUALIFY clause ... 97 (new)
3.3.4.9. ORDER BY clause ... 100
3.3.4.9.1. Ordering by field index ... 100
3.3.4.9.2. Ordering and NULLS ... 101
3.3.4.9.3. Ordering using CASE expressions ... 102
3.3.4.9.4. Oracle's ORDER SIBLINGS BY clause ... 104
3.3.4.10. LIMIT .. OFFSET clause ... 104
3.3.4.11. WITH TIES clause ... 105
3.3.4.12. SEEK clause ... 107
3.3.4.13. FOR UPDATE clause ... 109
3.3.4.14. Set operations ... 110
3.3.4.14.1. Type safety ... 111
3.3.4.14.2. Projection rowtype ... 111
© 2009 - 2025 by Data Geekery™ GmbH.

3.3.4.14.3. Differences to standard SQL ... 111
3.3.4.14.4. UNION ... 115
3.3.4.14.5. UNION ALL ... 117
3.3.4.14.6. INTERSECT ... 118
3.3.4.14.7. INTERSECT ALL ... 120
3.3.4.14.8. EXCEPT ... 121
3.3.4.14.9. EXCEPT ALL ... 122
3.3.4.15. Lexical and logical SELECT clause order ... 123
3.3.5. The INSERT statement ... 125
3.3.5.1. INSERT .. VALUES ... 125
3.3.5.2. INSERT .. DEFAULT VALUES ... 126
3.3.5.3. INSERT .. SET ... 128
3.3.5.4. INSERT .. SELECT ... 129
3.3.5.5. INSERT .. ON DUPLICATE KEY UPDATE ... 131
3.3.5.6. INSERT .. ON DUPLICATE KEY IGNORE ... 135
3.3.5.7. INSERT .. ON CONFLICT ... 140
3.3.5.8. INSERT .. RETURNING ... 144
3.3.6. The UPDATE statement ... 146
3.3.6.1. UPDATE .. SET ... 146
3.3.6.2. UPDATE .. SET ROW ... 147
3.3.6.3. UPDATE .. FROM ... 149
3.3.6.4. UPDATE .. WHERE ... 152
3.3.6.5. UPDATE .. ORDER BY .. LIMIT ... 154 (new)
3.3.6.6. UPDATE .. RETURNING ... 157
3.3.7. The DELETE statement ... 159
3.3.7.1. DELETE .. WHERE ... 159
3.3.7.2. DELETE .. ORDER BY .. LIMIT ... 160 (new)
3.3.7.3. DELETE .. RETURNING ... 162
3.3.8. The MERGE statement ... 164
3.3.8.1. USING .. ON ... 164
3.3.8.2. WHEN MATCHED THEN UPDATE ... 166
3.3.8.3. WHEN MATCHED THEN DELETE ... 168
3.3.8.4. WHEN NOT MATCHED THEN INSERT ... 169
3.4. SQL Statements (DDL) ... 171
3.4.1. The ALTER statement ... 171
3.4.1.1. ALTER INDEX ... 172
3.4.1.1.1. ALTER INDEX .. RENAME ... 172
3.4.1.1.2. ALTER INDEX IF EXISTS ... 173
3.4.1.2. ALTER SCHEMA ... 175
3.4.1.2.1. ALTER SCHEMA .. RENAME ... 175
3.4.1.2.2. ALTER SCHEMA IF EXISTS ... 176
3.4.1.3. ALTER SEQUENCE ... 177
3.4.1.3.1. ALTER SEQUENCE .. RENAME ... 177
3.4.1.3.2. ALTER SEQUENCE .. RESTART ... 178
3.4.1.3.3. ALTER SEQUENCE IF EXISTS ... 179
3.4.1.4. ALTER TABLE ... 180
3.4.1.4.1. ALTER TABLE .. ADD COLUMN ... 181
3.4.1.4.2. ALTER TABLE .. ADD COLUMNS ... 182
3.4.1.4.3. ALTER TABLE .. ADD COLUMN IF NOT EXISTS ... 185
3.4.1.4.4. ALTER TABLE .. ADD PRIMARY KEY ... 187
3.4.1.4.5. ALTER TABLE .. ADD UNIQUE ... 189
3.4.1.4.6. ALTER TABLE .. ADD FOREIGN KEY ... 190
3.4.1.4.7. ALTER TABLE .. ADD CHECK ... 191
3.4.1.4.8. ALTER TABLE .. RENAME ... 192
© 2009 - 2025 by Data Geekery™ GmbH.

3.4.1.4.9. ALTER TABLE .. COMMENT ... 193
3.4.1.4.10. ALTER TABLE .. ALTER COLUMN .. SET DEFAULT ... 195
3.4.1.4.11. ALTER TABLE .. ALTER COLUMN .. SET NOT NULL ... 196
3.4.1.4.12. ALTER TABLE .. ALTER COLUMN .. DROP NOT NULL ... 198
3.4.1.4.13. ALTER TABLE .. ALTER COLUMN .. SET TYPE ... 199
3.4.1.4.14. ALTER TABLE .. RENAME COLUMN ... 201
3.4.1.4.15. ALTER TABLE .. RENAME CONSTRAINT ... 203
3.4.1.4.16. ALTER TABLE .. RENAME INDEX ... 204
3.4.1.4.17. ALTER TABLE .. DROP COLUMN ... 205
3.4.1.4.18. ALTER TABLE .. DROP COLUMN RESTRICT ... 206
3.4.1.4.19. ALTER TABLE .. DROP COLUMN CASCADE ... 207
3.4.1.4.20. ALTER TABLE .. DROP COLUMNS ... 208
3.4.1.4.21. ALTER TABLE .. DROP COLUMN IF EXISTS ... 210
3.4.1.4.22. ALTER TABLE .. DROP CONSTRAINT ... 212
3.4.1.4.23. ALTER TABLE .. DROP PRIMARY KEY ... 213 (new)
3.4.1.4.24. ALTER TABLE .. DROP FOREIGN KEY ... 214 (new)
3.4.1.4.25. ALTER TABLE IF EXISTS ... 215
3.4.1.5. ALTER VIEW ... 217
3.4.1.5.1. ALTER VIEW .. COMMENT ... 217
3.4.1.5.2. ALTER VIEW .. RENAME ... 218
3.4.1.5.3. ALTER VIEW IF EXISTS ... 219
3.4.2. The COMMENT statement ... 221
3.4.2.1. COMMENT ON COLUMN ... 221
3.4.2.2. COMMENT ON TABLE ... 222
3.4.2.3. COMMENT ON VIEW ... 223
3.4.3. The CREATE statement ... 225
3.4.3.1. CREATE INDEX ... 225
3.4.3.2. CREATE SCHEMA ... 226
3.4.3.3. CREATE SEQUENCE ... 227
3.4.3.3.1. CREATE SEQUENCE IF NOT EXISTS ... 228
3.4.3.4. CREATE TABLE ... 230
3.4.3.4.1. Columns ... 230
3.4.3.4.2. Nullability ... 232
3.4.3.4.3. Defaults ... 234
3.4.3.4.4. Identities ... 236
3.4.3.4.5. Primary key ... 240
3.4.3.4.6. Unique constraints ... 243
3.4.3.4.7. Foreign keys ... 245
3.4.3.4.8. Check constraints ... 248
3.4.3.4.9. From a SELECT ... 250
3.4.3.4.10. Global temporary tables ... 252
3.4.3.5. CREATE TYPE ... 254 (new)
3.4.3.6. CREATE VIEW ... 254
3.4.3.6.1. CREATE OR REPLACE VIEW ... 255
3.4.3.6.2. WITH CHECK OPTION ... 257
3.4.3.6.3. WITH READ ONLY ... 258
3.4.4. The DROP statement ... 262
3.4.4.1. DROP INDEX ... 262
3.4.4.1.1. IF EXISTS ... 263
3.4.4.2. DROP SCHEMA ... 265
3.4.4.2.1. IF EXISTS ... 267
3.4.4.3. DROP SEQUENCE ... 268
3.4.4.3.1. IF EXISTS ... 269
3.4.4.4. DROP TABLE ... 271
© 2009 - 2025 by Data Geekery™ GmbH.

3.4.4.4.1. CASCADE 271
3.4.4.4.2. IF EXISTS 272
3.4.4.5. DROP TYPE 274 (new)
3.4.4.5.1. IF EXISTS 275 (new)
3.4.4.6. DROP VIEW 276
3.4.4.6.1. IF EXISTS 276
3.4.5. The GRANT statement 278
3.4.6. The REVOKE statement 279
3.4.7. The SET statement 279
3.4.7.1. SET CATALOG 279
3.4.7.2. SET SCHEMA 280
3.4.8. The TRUNCATE statement 282
3.4.9. Generating DDL from objects 283
3.5. Procedural statements 283 (new)
3.5.1. Block statement 284 (new)
3.5.2. CONTINUE statement 287 (new)
3.5.3. EXECUTE statement 289 (new)
3.5.4. EXIT statement 291 (new)
3.5.5. FOR statement 293 (new)
3.5.6. GOTO statement 297 (new)
3.5.7. IF statement 298 (new)
3.5.8. Labels 300 (new)
3.5.9. LOOP statement 301 (new)
3.5.10. REPEAT statement 303 (new)
3.5.11. Variables 306 (new)
3.5.12. WHILE statement 310 (new)
3.6. Catalog and schema expressions 312
3.7. Table expressions 313
3.7.1. Generated Tables 313
3.7.2. Aliased Tables 313
3.7.2.1. Aliased generated tables 314
3.7.2.2. Aliased table expressions 315
3.7.2.3. Derived column lists 315
3.7.2.4. Unnamed derived tables 319
3.7.3. Joined tables 319
3.7.3.1. CROSS JOIN 320
3.7.3.2. INNER JOIN 322
3.7.3.3. OUTER JOIN 323
3.7.3.4. SEMI JOIN 325
3.7.3.5. ANTI JOIN 327
3.7.3.6. ON clause 328
3.7.3.7. ON KEY clause 329
3.7.3.8. USING clause 330
3.7.3.9. NATURAL clause 330
3.7.3.10. LATERAL 331
3.7.3.11. APPLY 332
3.7.3.12. PARTITION BY 334
3.7.4. The VALUES() table constructor 334
3.7.5. Derived tables 338
3.7.6. Inline derived tables 338 (new)
3.7.7. The Oracle PIVOT clause 339
3.7.8. Relational division 339
3.7.9. Array and cursor unnesting 340
3.7.10. Table-valued functions 341
© 2009 - 2025 by Data Geekery™ GmbH.

3.7.11. GENERATE_SERIES ... 341
3.7.12. The DUAL table ... 346
3.8. Column expressions ... 348
3.8.1. Table columns ... 348
3.8.1.1. Generated table columns ... 349
3.8.1.2. Dereferenced table columns ... 349
3.8.1.3. Named table columns ... 349
3.8.2. Aliased columns ... 350
3.8.3. Cast expressions ... 351
3.8.4. Datatype coercions ... 353
3.8.5. Collations ... 353
3.8.6. Arithmetic expressions ... 354
3.8.7. String concatenation ... 355
3.8.8. Case sensitivity with strings ... 355
3.8.9. General functions ... 355
3.8.9.1. CHOOSE ... 355
3.8.9.2. COALESCE ... 357
3.8.9.3. DECODE ... 358
3.8.9.4. IIF ... 361
3.8.9.5. NULLIF ... 362
3.8.9.6. NVL ... 363
3.8.9.7. NVL2 ... 364
3.8.10. Numeric functions ... 365
3.8.10.1. ABS ... 366
3.8.10.2. ACOS ... 366
3.8.10.3. ASIN ... 367
3.8.10.4. ATAN ... 368
3.8.10.5. ATAN2 ... 369
3.8.10.6. CEIL ... 370
3.8.10.7. COS ... 372
3.8.10.8. COSH ... 373
3.8.10.9. COT ... 374
3.8.10.10. COTH ... 375
3.8.10.11. DEG ... 376
3.8.10.12. E ... 378
3.8.10.13. EXP ... 379
3.8.10.14. FLOOR ... 380
3.8.10.15. GREATEST ... 381
3.8.10.16. LEAST ... 383
3.8.10.17. LN ... 384
3.8.10.18. LOG ... 385
3.8.10.19. NEG ... 386
3.8.10.20. PI ... 387
3.8.10.21. POWER ... 388
3.8.10.22. RAD ... 389
3.8.10.23. RAND ... 391
3.8.10.24. ROUND ... 392
3.8.10.25. SIGN ... 394
3.8.10.26. SIN ... 395
3.8.10.27. SINH ... 396
3.8.10.28. SQRT ... 397
3.8.10.29. TAN ... 398
3.8.10.30. TANH ... 399
3.8.10.31. TRUNC ... 400
© 2009 - 2025 by Data Geekery™ GmbH.

3.8.10.32. WIDTH_BUCKET ... 403
3.8.11. Bitwise functions ... 404
3.8.11.1. BIT_AND ... 404
3.8.11.2. BIT_COUNT ... 405
3.8.11.3. BIT_NAND ... 409
3.8.11.4. BIT_NOR ... 410
3.8.11.5. BIT_NOT ... 412
3.8.11.6. BIT_OR ... 414
3.8.11.7. BIT_XNOR ... 415
3.8.11.8. BIT_XOR ... 418
3.8.11.9. SHL ... 420
3.8.11.10. SHR ... 422
3.8.12. String functions ... 424
3.8.12.1. ASCII ... 424
3.8.12.2. BIT_LENGTH ... 425
3.8.12.3. CONCAT (|| operator) ... 426
3.8.12.4. LEFT ... 427
3.8.12.5. LENGTH ... 428
3.8.12.6. LOWER ... 429
3.8.12.7. LPAD ... 430
3.8.12.8. LTRIM ... 432
3.8.12.9. MD5 ... 433
3.8.12.10. MID ... 434
3.8.12.11. OCTET_LENGTH ... 434
3.8.12.12. POSITION ... 435
3.8.12.13. REPEAT ... 437
3.8.12.14. REPLACE ... 438
3.8.12.15. REVERSE ... 440
3.8.12.16. RIGHT ... 440
3.8.12.17. RPAD ... 442
3.8.12.18. RTRIM ... 443
3.8.12.19. SPACE ... 444
3.8.12.20. SUBSTRING ... 445
3.8.12.21. TRANSLATE ... 447
3.8.12.22. TRIM ... 448
3.8.12.23. UPPER ... 449
3.8.13. Datetime functions ... 449
3.8.13.1. CENTURY ... 450 (new)
3.8.13.2. CURRENT_DATE ... 452
3.8.13.3. CURRENT_LOCALDATE ... 454
3.8.13.4. CURRENT_LOCALDATETIME ... 455
3.8.13.5. CURRENT_LOCALTIME ... 456
3.8.13.6. CURRENT_OFFSETDATETIME ... 458
3.8.13.7. CURRENT_OFFSETTIME ... 460
3.8.13.8. CURRENT_TIME ... 462
3.8.13.9. CURRENT_TIMESTAMP ... 463
3.8.13.10. DATE ... 465
3.8.13.11. DATEADD ... 466
3.8.13.12. DATEDIFF ... 469
3.8.13.13. DATESUB ... 473
3.8.13.14. DAY ... 476
3.8.13.15. DAY_OF_YEAR ... 478 (new)
3.8.13.16. DECADE ... 479 (new)
3.8.13.17. EPOCH ... 482 (new)
© 2009 - 2025 by Data Geekery™ GmbH.

3.8.13.18. EXTRACT ... 483
3.8.13.19. HOUR ... 485
3.8.13.20. ISO_DAY_OF_WEEK ... 487 (new)
3.8.13.21. LOCALDATE ... 490
3.8.13.22. LOCALDATEADD ... 491
3.8.13.23. LOCALDATESUB ... 494
3.8.13.24. LOCALDATETIME ... 497
3.8.13.25. LOCALDATETIMEADD ... 498
3.8.13.26. LOCALDATETIMESUB ... 501
3.8.13.27. LOCALTIME ... 504
3.8.13.28. MILLENNIUM ... 506 (new)
3.8.13.29. MINUTE ... 508
3.8.13.30. MONTH ... 510
3.8.13.31. QUARTER ... 512 (new)
3.8.13.32. SECOND ... 514
3.8.13.33. TIME ... 516
3.8.13.34. TIMESTAMP ... 518
3.8.13.35. TIMESTAMPADD ... 519
3.8.13.36. TIMESTAMPSUB ... 522
3.8.13.37. TO_DATE ... 525
3.8.13.38. TO_LOCALDATE ... 526
3.8.13.39. TO_LOCALDATETIME ... 527
3.8.13.40. TO_TIMESTAMP ... 528
3.8.13.41. TRUNC ... 529
3.8.13.42. YEAR ... 530
3.8.14. ARRAY functions ... 532
3.8.14.1. ARRAY constructor ... 533
3.8.15. JSON functions ... 534 (new)
3.8.15.1. JSON_ARRAY ... 534 (new)
3.8.15.2. JSON_OBJECT ... 536 (new)
3.8.16. CONNECT BY functions ... 537
3.8.16.1. CONNECT_BY_ISCYCLE ... 537
3.8.16.2. CONNECT_BY_ISLEAF ... 539
3.8.16.3. CONNECT_BY_ROOT ... 540
3.8.16.4. LEVEL ... 541
3.8.16.5. PRIOR ... 542
3.8.16.6. SYS_CONNECT_BY_PATH ... 543
3.8.17. System functions ... 544
3.8.17.1. CURRENT_SCHEMA ... 544
3.8.17.2. CURRENT_USER ... 546
3.8.18. Aggregate functions ... 547
3.8.18.1. Grouping ... 548
3.8.18.2. Distinctness ... 549
3.8.18.3. Filtering ... 550
3.8.18.4. Ordering ... 552
3.8.18.5. Ordering WITHIN GROUP ... 552
3.8.18.6. Keeping ... 553
3.8.18.7. ARRAY_AGG ... 553
3.8.18.8. AVG ... 554
3.8.18.9. BOOL_AND ... 555
3.8.18.10. BOOL_OR ... 557
3.8.18.11. COLLECT ... 560
3.8.18.12. COUNT ... 560
3.8.18.13. CUME_DIST ... 561
© 2009 - 2025 by Data Geekery™ GmbH.

3.8.18.14. DENSE_RANK ... 562
3.8.18.15. EVERY ... 563
3.8.18.16. GROUP_CONCAT ... 565
3.8.18.17. LISTAGG ... 567
3.8.18.18. MAX ... 569
3.8.18.19. MEDIAN ... 570
3.8.18.20. MIN ... 571
3.8.18.21. MODE ... 572
3.8.18.22. PERCENT_RANK ... 573
3.8.18.23. PERCENTILE_CONT ... 574
3.8.18.24. PERCENTILE_DISC ... 575
3.8.18.25. PRODUCT ... 576
3.8.18.26. RANK ... 580
3.8.18.27. SUM ... 581
3.8.19. Window functions ... 582
3.8.19.1. PARTITION BY ... 582
3.8.19.2. ORDER BY ... 584
3.8.19.3. ROWS, RANGE, GROUPS (frame clause) ... 585
3.8.19.4. EXCLUDE ... 587
3.8.19.5. NULL treatment ... 588
3.8.19.6. FROM FIRST, FROM LAST ... 590
3.8.19.7. Nested aggregate functions ... 590
3.8.19.8. Window aggregation ... 591
3.8.19.9. Window ordered aggregate ... 591
3.8.19.10. ROW_NUMBER ... 592
3.8.19.11. RANK ... 593
3.8.19.12. DENSE_RANK ... 594
3.8.19.13. PERCENT_RANK ... 595
3.8.19.14. CUME_DIST ... 596
3.8.19.15. NTILE ... 597
3.8.19.16. LEAD ... 598
3.8.19.17. LAG ... 599
3.8.19.18. FIRST_VALUE ... 601
3.8.19.19. LAST_VALUE ... 602
3.8.19.20. NTH_VALUE ... 603
3.8.20. User-defined functions ... 604
3.8.21. User-defined aggregate functions ... 604
3.8.22. The CASE expression ... 606
3.8.23. Sequences and serials ... 606
3.8.24. Scalar subqueries ... 607
3.8.25. Tuples or row value expressions ... 608
3.9. Conditional expressions ... 609
3.9.1. Condition building ... 609
3.9.2. TRUE and FALSE condition ... 610
3.9.3. BOOLEAN columns ... 611
3.9.4. AND, OR, NOT boolean operators ... 611
3.9.5. Boolean operator precedence ... 612
3.9.6. Comparison predicate ... 612
3.9.7. Comparison predicate (degree > 1) ... 613
3.9.8. Quantified comparison predicate ... 615
3.9.9. BETWEEN predicate ... 616
3.9.10. BETWEEN predicate (degree > 1) ... 618
3.9.11. DISTINCT predicate ... 620
3.9.12. DISTINCT predicate (degree > 1) ... 622
© 2009 - 2025 by Data Geekery™ GmbH.

3.9.13. EXISTS predicate ... 625
3.9.14. IN predicate ... 626
3.9.15. IN predicate (degree > 1) ... 627
3.9.16. JSON predicate ... 629 (new)
3.9.17. LIKE predicate ... 630
3.9.18. LIKE REGEX predicate ... 632
3.9.19. Quantified LIKE predicate ... 633 (new)
3.9.20. NULL predicate ... 635
3.9.21. NULL predicate (degree > 1) ... 636
3.9.22. OVERLAPS predicate ... 637
3.9.23. SIMILAR TO predicate ... 638 (new)
3.9.24. UNIQUE predicate ... 639 (new)
3.9.25. Query By Example (QBE) ... 641
3.10. Operator precedence ... 642
3.11. Synthetic SQL clauses ... 642
3.12. Dynamic SQL ... 643
3.12.1. Optional conditional expressions ... 644
3.13. Plain SQL ... 645
3.13.1. Plain SQL API ... 645
3.13.2. Plain SQL templating language ... 647
3.14. Hints ... 648
3.14.1. MySQL hints ... 649
3.14.1.1. Index hints ... 649
3.14.1.2. STRAIGHT_JOIN ... 649
3.14.1.3. Oracle style hints in MySQL ... 650
3.14.2. Oracle hints ... 650
3.14.3. SQL Server hints ... 651
3.14.3.1. WITH ... 651
3.14.3.2. OPTION ... 653
3.15. SQL Parser ... 653
3.15.1. SQL Parser API ... 653
3.15.2. SQL Parser CLI ... 655
3.15.3. SQL translator ... 655
3.15.4. SQL Parser Grammar ... 656
3.16. Names and identifiers ... 656
3.17. Bind values and parameters ... 657
3.17.1. Indexed parameters ... 657
3.17.2. Named parameters ... 658
3.17.3. Inlined parameters ... 659
3.17.4. SQL injection ... 659
3.18. QueryParts ... 660
3.18.1. SQL rendering ... 660
3.18.2. Declaration vs reference ... 662
3.18.3. Pretty printing SQL ... 662
3.18.4. Variable binding ... 663
3.18.5. Custom data type bindings ... 664
3.18.6. Custom syntax elements ... 667
3.18.7. Plain SQL QueryParts ... 668
3.18.8. Serializability ... 670
3.18.9. Custom SQL transformation with VisitListener ... 670
3.18.9.1. Example: Logging abbreviated bind values ... 671
3.19. Zero-based vs one-based APIs ... 672
3.20. SQL building in Scala ... 673
3.21. Compile time validation ... 675
© 2009 - 2025 by Data Geekery™ GmbH.

4. SQL execution.... 678
4.1. Comparison between jOOQ and JDBC.... 679
4.2. Query vs. ResultQuery.... 679
4.3. Fetching.... 680
4.3.1. Record vs. TableRecord.... 682
4.3.2. Record1 to Record22.... 683
4.3.3. Arrays, Maps and Lists.... 684
4.3.4. ResultQuery as Iterable.... 684
4.3.5. RecordMapper.... 684
4.3.6. POJOs.... 685
4.3.7. RecordMapperProvider.... 689
4.3.8. Lazy fetching.... 689
4.3.9. Lazy fetching with Streams.... 690
4.3.10. Many fetching.... 691
4.3.11. Later fetching.... 692
4.3.12. Reactive Fetching.... 694 (new)
4.3.13. ResultSet fetching.... 694
4.3.14. Auto data type conversion.... 695
4.3.15. Custom data type conversion.... 696
4.3.16. Data type lookups.... 697
4.4. Static statements vs. Prepared Statements.... 698
4.5. Reusing a Query's PreparedStatement.... 699
4.6. JDBC flags.... 700
4.7. Using JDBC batch operations.... 701
4.8. Sequence execution.... 702
4.9. Stored procedures and functions.... 703
4.9.1. Oracle Packages.... 705
4.9.2. Oracle member procedures.... 705
4.10. Exporting to XML, CSV, JSON, HTML, Text, Charts.... 706
4.10.1. Exporting XML.... 706
4.10.2. Exporting CSV.... 707
4.10.3. Exporting JSON.... 707
4.10.4. Exporting HTML.... 707
4.10.5. Exporting Text.... 708
4.10.6. Exporting Charts.... 708
4.11. Importing data.... 709
4.11.1. The Loader API.... 709
4.11.2. Import options.... 710
4.11.2.1. Throttling.... 710
4.11.2.2. Duplicate handling.... 712
4.11.2.3. Error handling.... 712
4.11.3. Import data sources.... 712
4.11.3.1. Importing CSV.... 712
4.11.3.2. Importing JSON.... 713
4.11.3.3. Importing records.... 714
4.11.3.4. Importing arrays.... 714
4.11.3.5. Importing XML.... 714
4.11.4. Import listeners.... 714
4.11.5. Import result and error handling.... 715
4.12. CRUD with UpdatableRecords.... 715
4.12.1. Simple CRUD.... 716
4.12.2. Records' internal flags.... 718
4.12.3. IDENTITY values.... 718
4.12.4. Navigation methods.... 719
© 2009 - 2025 by Data Geekery™ GmbH.

4.12.5. Non-updatable records 719
4.12.6. Optimistic locking 720
4.12.7. Batch execution 721
4.12.8. CRUD SPI: RecordListener 721
4.13. DAOs 722
4.14. Transaction management 723
4.15. Exception handling 726
4.16. ExecuteListeners 727
4.17. Database meta data 729
4.17.1. JDBC meta data 730
4.17.2. Interpreted meta data 730 (new)
4.17.3. XML meta data 731
4.17.4. Generated meta data 732 (new)
4.18. JDBC Connection 732
4.19. Mocking Connection 732
4.20. Mock File Database 734
4.21. Parsing Connection 736
4.22. Diagnostics 737
4.22.1. Too Many Rows 738
4.22.2. Too Many Columns 738
4.22.3. Duplicate Statements 739
4.22.4. Repeated statements 741
4.22.5. WasNull calls 742
4.23. Logging with LoggerListener 742
4.24. Logging Connection 743 (new)
4.25. Performance considerations 744
4.26. Alternative execution models 744
4.26.1. Using jOOQ with Spring's JdbcTemplate 745
4.26.2. Using jOOQ with JPA 745
4.26.2.1. Using jOOQ with JPA Native Query 746
4.26.2.2. Using jOOQ with JPA entities 747
4.26.2.3. Using jOOQ with JPA EntityResult 748
5. Code generation 751
5.1. Configuration and setup of the generator 751
5.2. Advanced generator configuration 759
5.2.1. Logging 759
5.2.2. Error handling 760 (new)
5.2.3. Jdbc 760
5.2.4. Generator 763
5.2.5. Database 765
5.2.5.1. Database name and properties 765
5.2.5.2. RegexFlags 767
5.2.5.3. Includes and Excludes 768
5.2.5.4. Include object types 772
5.2.5.5. Record Version and Timestamp Fields 773
5.2.5.6. Synthetic identities 774
5.2.5.7. Synthetic primary keys 775
5.2.5.8. Override primary keys 775
5.2.5.9. Date as timestamp 776
5.2.5.10. Ignore procedure return values (deprecated) 777
5.2.5.11. Unsigned types 778
5.2.5.12. Catalog and schema mapping 779
5.2.5.13. Catalog and schema version providers 785
5.2.5.14. Custom ordering of generated code 788
© 2009 - 2025 by Data Geekery™ GmbH.

5.2.5.15. Forced types.... 790
5.2.5.15.1. Matching of forced types.... 790
5.2.5.15.2. Data type rewriting.... 794
5.2.5.15.3. Qualified converters.... 795
5.2.5.15.4. Inline converters.... 797
5.2.5.15.5. Enum converters.... 798
5.2.5.15.6. Data type bindings.... 799
5.2.5.16. Table valued functions.... 802
5.2.6. Generate.... 803
5.2.6.1. Annotations.... 803
5.2.6.2. Covariant overrides.... 805
5.2.6.2.1. Overriding as().... 806
5.2.6.2.2. Overriding rename().... 806
5.2.6.3. Default catalog and schema.... 806
5.2.6.4. Fluent setters.... 807
5.2.6.5. Fully Qualified Types.... 807
5.2.6.6. Global Artefacts.... 808
5.2.6.7. Implicit JOIN paths.... 809
5.2.6.8. Java Time Types.... 810
5.2.6.9. Whitespace (newlines and indentation).... 811 (new)
5.2.6.10. Zero Scale Decimal Types.... 812
5.2.7. Output target configuration.... 813
5.3. Generated object types.... 814
5.3.1. Generated tables.... 814
5.3.2. Generated records.... 816
5.3.3. Generated POJOs.... 818
5.3.4. Generated Interfaces.... 820
5.3.5. Generated DAOs.... 821
5.3.6. Generated sequences.... 822
5.3.7. Generated procedures.... 823
5.3.8. Generated UDTs.... 824
5.3.9. Generated global artefacts.... 825
5.4. Class names, method names, identifiers.... 826
5.4.1. Custom generator strategies.... 827
5.4.2. Matcher strategies.... 831
5.4.2.1. MatcherRule.... 833
5.4.2.2. Matching schemas.... 836
5.4.2.3. Matching tables.... 837
5.4.2.4. Matching fields.... 839
5.4.2.5. Matching routines.... 841
5.4.2.6. Matching sequences.... 842
5.4.2.7. Matching enums.... 844
5.4.2.8. Matcher examples.... 845
5.5. Custom code sections.... 847
5.6. Mapping generated catalogs and schemas.... 849
5.7. Alternative meta data sources.... 851
5.7.1. JPADatabase: Code generation from entities.... 851
5.7.2. XMLDatabase: Code generation from XML files.... 855
5.7.3. DDLDatabase: Code generation from SQL files.... 858
5.8. Alternative output languages.... 865
5.8.1. XMLGenerator: Generating XML.... 865
5.9. Code generation execution.... 866
5.9.1. Running the code generator with Maven.... 866
5.9.2. Running the code generator with Ant.... 869
© 2009 - 2025 by Data Geekery™ GmbH.

5.9.3. Running the code generator with Gradle.... 870
5.9.4. Programmatic configuration and execution.... 872
5.10. System properties governing code generation.... 873
5.11. Code generation dependencies.... 873
5.12. Code generation for large schemas.... 875
5.13. Code generation and version control.... 875
5.14. Features requiring generated code.... 876
6. Coming from JPA.... 878
6.1. Set based thinking.... 878
6.2. Database first.... 879
6.3. Eager or lazy loading.... 880
6.4. First level cache and second level cache.... 880
6.5. AttributeConverter.... 880
6.6. User types.... 880
6.7. Implicit JOIN.... 881
7. Reference.... 882
7.1. Supported RDBMS.... 882
7.2. Commercial only features.... 882
7.3. Experimental features.... 883
7.4. Built-in data types.... 884
7.4.1. BLOBs and CLOBs.... 884
7.4.2. BOOLEAN data type.... 884
7.4.3. Unsigned integer types.... 885
7.4.4. INTERVAL data types.... 885
7.4.5. JSON data types.... 886 (new)
7.4.6. CURSOR data types.... 886
7.4.7. ARRAY and TABLE data types.... 886
7.4.8. Oracle DATE data type.... 887
7.5. SQL to DSL mapping rules.... 888
7.6. Quality Assurance.... 891
7.7. Security.... 893
7.7.1. SQL Injection.... 894
7.7.2. Debug logging.... 894
7.7.3. Exception message.... 894
7.7.4. Contact.... 894
7.8. Migrating to jOOQ 3.0.... 894
7.9. Don't do this.... 899
7.9.1. jOOQ: Implementing the DSL types.... 899
7.9.2. jOOQ: Referencing the Step types.... 900
7.9.3. Schema: NULL columns.... 900
7.9.4. Schema: Unnamed constraints.... 901
7.9.5. Schema: Unnecessary surrogate keys.... 901
7.9.6. Schema: Wrong data types.... 902
7.9.7. SQL: COUNT(*) instead of EXISTS().... 902
7.9.8. SQL: N+1.... 902
7.9.9. SQL: NATURAL JOIN or JOIN USING.... 904
7.9.10. SQL: NOT IN predicate.... 905
7.9.11. SQL: ORDER BY [column index].... 905
7.9.12. SQL: Rely on implicit ordering.... 906
7.9.13. SQL: SELECT *.... 908
7.9.14. SQL: SELECT DISTINCT.... 909
7.9.15. SQL: Unnecessary UNION instead of UNION ALL.... 910
7.10. The most important jOOQ types.... 910
7.11. Credits.... 910
© 2009 - 2025 by Data Geekery™ GmbH.

1. Copyright, License, and Trademarks

This section lists the various licenses that apply to different versions of jOOQ. Prior to version 3.2, jOOQ was shipped for free under the terms of the Apache Software License 2.0. With jOOQ 3.2, jOOQ became dual-licensed: Apache Software License 2.0 (for use with Open Source databases) and commercial (for use with commercial databases).

This manual itself (as well as the www.jooq.org public website) is licensed to you under the terms of the CC BY-SA 4.0 license.

Please contact legal@datageekery.com, should you have any questions regarding licensing.

License for jOOQ 3.2 and later

```
This work is dual-licensed
- under the Apache Software License 2.0 (the "ASL")
- under the jOOQ License and Maintenance Agreement (the "jOOQ License")
==============================================================================
You may choose which license applies to you:

- If you're using this work with Open Source databases, you may choose
  either ASL or jOOQ License.
- If you're using this work with at least one commercial database, you must
  choose jOOQ License

For more information, please visit https://www.jooq.org/licenses

Apache Software License 2.0:
------------------------------------------------------------------------------
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

 https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

jOOQ License and Maintenance Agreement:
------------------------------------------------------------------------------
Data Geekery grants the Customer the non-exclusive, timely limited and
non-transferable license to install and use the Software under the terms of
the jOOQ License and Maintenance Agreement.

This library is distributed with a LIMITED WARRANTY. See the jOOQ License
and Maintenance Agreement for more details: https://www.jooq.org/licensing
```

Historic license for jOOQ 1.x, 2.x, 3.0, 3.1

```
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Trademarks owned by Data Geekery™ GmbH

- jOOλ™ is a trademark by Data Geekery™ GmbH
- jOOQ™ is a trademark by Data Geekery™ GmbH
- jOOR™ is a trademark by Data Geekery™ GmbH
- jOOU™ is a trademark by Data Geekery™ GmbH
- jOOX™ is a trademark by Data Geekery™ GmbH

Trademarks owned by database vendors with no affiliation to Data Geekery™ GmbH

- Access® is a registered trademark of Microsoft® Inc.
- Adaptive Server® Enterprise is a registered trademark of Sybase®, Inc.
- CUBRID™ is a trademark of NHN® Corp.
- DB2® is a registered trademark of IBM® Corp.
- Derby is a trademark of the Apache™ Software Foundation
- H2 is a trademark of the H2 Group
- HANA is a trademark of SAP SE
- HSQLDB is a trademark of The hsql Development Group
- Ingres is a trademark of Actian™ Corp.
- MariaDB is a trademark of Monty Program Ab
- MySQL® is a registered trademark of Oracle® Corp.
- Firebird® is a registered trademark of Firebird Foundation Inc.
- Oracle® database is a registered trademark of Oracle® Corp.
- PostgreSQL® is a registered trademark of The PostgreSQL Global Development Group
- Postgres Plus® is a registered trademark of EnterpriseDB® software
- SQL Anywhere® is a registered trademark of Sybase®, Inc.
- SQL Server® is a registered trademark of Microsoft® Inc.
- SQLite is a trademark of Hipp, Wyrick & Company, Inc.

Other trademarks by vendors with no affiliation to Data Geekery™ GmbH

- Java® is a registered trademark by Oracle® Corp. and/or its affiliates
- Liquibase is a trademark by Datical, Inc
- Flyway is a trademark by Red Gate Software Ltd
- Scala is a trademark of EPFL

Other trademark remarks

Other names may be trademarks of their respective owners.

Throughout the manual, the above trademarks are referenced without a formal ® (R) or ™ (TM) symbol. It is believed that referencing third-party trademarks in this manual or on the jOOQ website constitutes "fair use". Please contact us if you think that your trademark(s) are not properly attributed.

Contributions

The following are authors and contributors of jOOQ or parts of jOOQ in alphabetical order:

- Aaron Digulla
- Andreas Franzén
- Anuraag Agrawal
- Arnaud Roger
- Art O Cathain
- Artur Dryomov
- Ben Manes
- Brent Douglas
- Brett Meyer
- Christian Stein
- Christopher Deckers
- Dennis Neufeld
- Ed Schaller
- Eric Peters
- Ernest Mishkin
- Espen Stromsnes
- Eugeny Karpov
- Fabrice Le Roy
- Gonzalo Ortiz Jaureguizar
- Gregory Hlavac
- Henrik Sjöstrand
- Ivan Dugic
- Javier Durante
- Johannes Bühler
- Joseph B Phillips
- Joseph Pachod
- Knut Wannheden
- Laurent Pireyn
- Logan Hauspie
- Luc Marchaud
- Lukas Eder
- Matti Tahvonen
- Michael Doberenz
- Michael Simons
- Michał Kołodziejski
- Miguel Gonzalez Sanchez
- Mustafa Yücel
- Nathaniel Fischer
- Nicholas Chong W.B.
- Octavia Togami
- Oliver Flege
- Per Lundberg
- Peter Ertl
- Richard Bradley
- Robin Stocker
- Roland Weisleder
- Samy Deghou
- Sander Plas
- Sean Wellington
- Sergey Epik
- Sergey Zhuravlev
- Stanislas Nanchen
- Stephan Schroevers
- Sugiharto Lim
- Sven Jacobs
- Szymon Jachim
- Terence Zhang
- Thomas Darimont
- Timothy Wilson

See the following website for details about contributing to jOOQ:
https://www.jooq.org/legal/contributions

2. Getting started with jOOQ

These chapters contain a quick overview of how to get started with this manual and with jOOQ. While the subsequent chapters contain a lot of reference information, this chapter here just wraps up the essentials.

2.1. How to read this manual

This section helps you correctly interpret this manual in the context of jOOQ.

Code blocks

The following are code blocks:

```
-- A SQL code block
SELECT 1 FROM DUAL
```

```
// A Java code block
for (int i = 0; i < 10; i++);
```

```
<!-- An XML code block -->
<hello what="world"></hello>
```

```
# A config file code block
org.jooq.property=value
```

These are useful to provide examples in code. Often, with jOOQ, it is even more useful to compare SQL code with its corresponding Java/jOOQ code. When this is done, the blocks are aligned side-by-side, with SQL usually being on the left, and an equivalent jOOQ DSL query in Java usually being on the right:

```
-- In SQL:
SELECT 1 FROM DUAL
```

```
// Using jOOQ:
create.selectOne().fetch()
```

Code block contents

The contents of code blocks follow conventions, too. If nothing else is mentioned next to any given code block, then the following can be assumed:

```
-- SQL assumptions
------------------

-- If nothing else is specified, assume that the Oracle syntax is used
SELECT 1 FROM DUAL
```

© 2009 - 2025 by Data Geekery™ GmbH.

```
// Java assumptions
// ----------------

// Whenever you see "standalone functions", assume they were static imported from org.jooq.impl.DSL
// "DSL" is the entry point of the static query DSL
exists(); max(); min(); val(); inline(); // correspond to DSL.exists(); DSL.max(); DSL.min(); etc...

// Whenever you see BOOK/Book, AUTHOR/Author and similar entities, assume they were (static) imported from the generated schema
BOOK.TITLE, AUTHOR.LAST_NAME // com.example.generated.Tables.BOOK.TITLE, com.example.generated.Tables.AUTHOR.LAST_NAME
FK_BOOK_AUTHOR               // com.example.generated.Keys.FK_BOOK_AUTHOR

// Whenever you see "create" being used in Java code, assume that this is an instance of org.jooq.DSLContext.
// The reason why it is called "create" is the fact, that a jOOQ QueryPart is being created from the DSL object.
// "create" is thus the entry point of the non-static query DSL
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);
```

Your naming may differ, of course. For instance, you could name the "create" instance "db", instead.

Execution

When you're coding PL/SQL, T-SQL or some other procedural SQL language, SQL statements are always executed immediately at the semi-colon. This is not the case in jOOQ, because as an internal DSL, jOOQ can never be sure that your statement is complete until you call fetch() or execute(). The manual tries to apply fetch() and execute() as thoroughly as possible. If not, it is implied:

```
SELECT 1 FROM DUAL
UPDATE t SET v = 1
```

```
create.selectOne().fetch();
create.update(T).set(T.V, 1).execute();
```

Degree (arity)

jOOQ records (and many other API elements) have a degree N between 1 and 22. The variable degree of an API element is denoted as [N], e.g. Row[N] or Record[N]. The term "degree" is preferred over arity, as "degree" is the term used in the SQL standard, whereas "arity" is used more often in mathematics and relational theory.

Settings

jOOQ allows to override runtime behaviour using org.jooq.conf.Settings. If nothing is specified, the default runtime settings are assumed.

Sample database

jOOQ query examples run against the sample database. See the manual's section about the sample database used in this manual to learn more about the sample database.

2.2. The sample database used in this manual

For the examples in this manual, the same database will always be referred to. It essentially consists of these entities created using the Oracle dialect

```
CREATE TABLE language (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  cd              CHAR(2)       NOT NULL,
  description     VARCHAR2(50)
);

CREATE TABLE author (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  first_name      VARCHAR2(50),
  last_name       VARCHAR2(50)  NOT NULL,
  date_of_birth   DATE,
  year_of_birth   NUMBER(7),
  distinguished   NUMBER(1)
);

CREATE TABLE book (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  author_id       NUMBER(7)     NOT NULL,
  title           VARCHAR2(400) NOT NULL,
  published_in    NUMBER(7)     NOT NULL,
  language_id     NUMBER(7)     NOT NULL,

  CONSTRAINT fk_book_author     FOREIGN KEY (author_id)   REFERENCES author(id),
  CONSTRAINT fk_book_language   FOREIGN KEY (language_id) REFERENCES language(id)
);

CREATE TABLE book_store (
  name            VARCHAR2(400) NOT NULL UNIQUE
);

CREATE TABLE book_to_book_store (
  name            VARCHAR2(400) NOT NULL,
  book_id         INTEGER       NOT NULL,
  stock           INTEGER,

  PRIMARY KEY(name, book_id),
  CONSTRAINT fk_b2bs_book_store FOREIGN KEY (name)        REFERENCES book_store (name) ON DELETE CASCADE,
  CONSTRAINT fk_b2bs_book       FOREIGN KEY (book_id)     REFERENCES book (id)         ON DELETE CASCADE
);
```

More entities, types (e.g. UDT's, ARRAY types, ENUM types, etc), stored procedures and packages are introduced for specific examples

In addition to the above, you may assume the following sample data:

```
INSERT INTO language (id, cd, description) VALUES (1, 'en', 'English');
INSERT INTO language (id, cd, description) VALUES (2, 'de', 'Deutsch');
INSERT INTO language (id, cd, description) VALUES (3, 'fr', 'Français');
INSERT INTO language (id, cd, description) VALUES (4, 'pt', 'Português');

INSERT INTO author (id, first_name, last_name, date_of_birth    , year_of_birth)
  VALUES           (1 , 'George'  , 'Orwell' , DATE '1903-06-26', 1903         );
INSERT INTO author (id, first_name, last_name, date_of_birth    , year_of_birth)
  VALUES           (2 , 'Paulo'   , 'Coelho' , DATE '1947-08-24', 1947         );

INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (1 , 1        , '1984'        , 1948        , 1          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (2 , 1        , 'Animal Farm' , 1945        , 1          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (3 , 2        , 'O Alquimista', 1988        , 4          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (4 , 2        , 'Brida'       , 1990        , 2          );

INSERT INTO book_store VALUES ('Orell Füssli');
INSERT INTO book_store VALUES ('Ex Libris');
INSERT INTO book_store VALUES ('Buchhandlung im Volkshaus');

INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 1, 10);
INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 2, 10);
INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 3, 10);
INSERT INTO book_to_book_store VALUES ('Ex Libris'                , 1, 1 );
INSERT INTO book_to_book_store VALUES ('Ex Libris'                , 3, 2 );
INSERT INTO book_to_book_store VALUES ('Buchhandlung im Volkshaus', 3, 1 );
```

2.3. Different use cases for jOOQ

jOOQ has originally been created as a library for complete abstraction of JDBC and all database interaction. Various best practices that are frequently encountered in pre-existing software products are applied to this library. This includes:

- Typesafe database object referencing through generated schema, table, column, record, procedure, type, dao, pojo artefacts (see the chapter about code generation)
- Typesafe SQL construction / SQL building through a complete querying DSL API modelling SQL as a domain specific language in Java (see the chapter about the query DSL API)
- Convenient query execution through an improved API for result fetching (see the chapters about the various types of data fetching)
- SQL dialect abstraction and SQL clause emulation to improve cross-database compatibility and to enable missing features in simpler databases (see the chapter about SQL dialects)
- SQL logging and debugging using jOOQ as an integral part of your development process (see the chapters about logging)

Effectively, jOOQ was originally designed to replace any other database abstraction framework short of the ones handling connection pooling (and more sophisticated transaction management)

Use jOOQ the way you prefer

... but open source is community-driven. And the community has shown various ways of using jOOQ that diverge from its original intent. Some use cases encountered are:

- Using Hibernate for 70% of the queries (i.e. CRUD) and jOOQ for the remaining 30% where SQL is really needed
- Using jOOQ for SQL building and JDBC for SQL execution
- Using jOOQ for SQL building and Spring Data for SQL execution
- Using jOOQ without the source code generator to build the basis of a framework for dynamic SQL execution.

The following sections explain about various use cases for using jOOQ in your application.

2.3.1. jOOQ as a SQL builder without code generation

We strongly recommend to use jOOQ with its code generator to get the most out of jOOQ!

However, if you have a dynamic schema, you don't have to use the code generator. This is the most simple of all use cases, allowing for construction of valid SQL for any database. In this use case, you will not use jOOQ's code generator and maybe not even jOOQ's query execution facilities. Instead, you'll use jOOQ's query DSL API to wrap strings, literals and other user-defined objects into an object-oriented, type-safe AST modelling your SQL statements. An example is given here:

```
// Fetch a SQL string from a jOOQ Query in order to manually execute it with another tool.
// For simplicity reasons, we're using the API to construct case-insensitive object references, here.
Query query = create.select(field("BOOK.TITLE"), field("AUTHOR.FIRST_NAME"), field("AUTHOR.LAST_NAME"))
                    .from(table("BOOK"))
                    .join(table("AUTHOR"))
                    .on(field("BOOK.AUTHOR_ID").eq(field("AUTHOR.ID")))
                    .where(field("BOOK.PUBLISHED_IN").eq(1948));
String sql = query.getSQL();
List<Object> bindValues = query.getBindValues();
```

The SQL string built with the jOOQ query DSL can then be executed using JDBC directly, using Spring's JdbcTemplate, using Apache DbUtils and many other tools (note that since jOOQ uses

java.sql.PreparedStatement by default, this will generate a bind variable for "1948". Read more about bind variables here).

You can also avoid getting the SQL string and bind values separately:

```
String sql = query.getSQL(ParamType.INLINED);
```

If you wish to use jOOQ only as a SQL builder, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Plain SQL: This section contains information useful in particular to those that want to supply table expressions, column expressions, etc. as plain SQL to jOOQ, rather than through generated artefacts
- Bind values: This section explains how bind values are managed and/or inlined in jOOQ.

2.3.2. jOOQ as a SQL builder with code generation

In addition to using jOOQ as a standalone SQL builder, you can also use jOOQ's code generation features in order to compile your SQL statements using a Java compiler against an actual database schema. This adds a lot of power and expressiveness to just simply constructing SQL using the query DSL and custom strings and literals, as you can be sure that all database artefacts actually exist in the database, and that their type is correct. We strongly recommend using this approach. An example is given here:

```
// Fetch a SQL string from a jOOQ Query in order to manually execute it with another tool.
Query query = create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
                    .from(BOOK)
                    .join(AUTHOR)
                    .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                    .where(BOOK.PUBLISHED_IN.eq(1948));

String sql = query.getSQL();
List<Object> bindValues = query.getBindValues();
```

The SQL string built with the jOOQ query DSL can then be executed using JDBC directly, using Spring's JdbcTemplate, using Apache DbUtils and many other tools (note that since jOOQ uses java.sql.PreparedStatement by default, this will generate a bind variable for "1948". Read more about bind variables here).

You can also avoid getting the SQL string and bind values separately:

```
String sql = query.getSQL(ParamType.INLINED);
```

If you wish to use jOOQ only as a SQL builder with code generation, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database
- Bind values: This section explains how bind values are managed and/or inlined in jOOQ.

2.3.3. jOOQ as a SQL executor

Instead of any tool mentioned in the previous chapters, you can also use jOOQ directly to execute your jOOQ-generated SQL statements. This will add a lot of convenience on top of the previously discussed API for typesafe SQL construction, when you can re-use the information from generated classes to fetch records and custom data types. An example is given here:

```
// Typesafely execute the SQL statement directly with jOOQ
Result<Record3<String, String, String>> result =
create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .from(BOOK)
      .join(AUTHOR)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.PUBLISHED_IN.eq(1948))
      .fetch();
```

By having jOOQ execute your SQL, the jOOQ query DSL becomes truly embedded SQL.

jOOQ doesn't stop here, though! You can execute any SQL with jOOQ. In other words, you can use any other SQL building tool and run the SQL statements with jOOQ. An example is given here:

```
// Use your favourite tool to construct SQL strings:
String sql = "SELECT title, first_name, last_name FROM book JOIN author ON book.author_id = author.id " +
             "WHERE book.published_in = 1984";

// Fetch results using jOOQ
Result<Record> result = create.fetch(sql);

// Or execute that SQL with JDBC, fetching the ResultSet with jOOQ:
ResultSet rs = connection.createStatement().executeQuery(sql);
Result<Record> result = create.fetch(rs);
```

If you wish to use jOOQ as a SQL executor with (or without) code generation, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database
- SQL execution: This section contains a lot of information about executing SQL statements using the jOOQ API
- Fetching: This section contains some useful information about the various ways of fetching data with jOOQ

2.3.4. jOOQ for CRUD

Apart from jOOQ's fluent API for query construction, jOOQ can also help you execute everyday CRUD operations. An example is given here:

© 2009 - 2025 by Data Geekery™ GmbH.

```
// Fetch an author
AuthorRecord author = create.fetchOne(AUTHOR, AUTHOR.ID.eq(1));

// Create a new author, if it doesn't exist yet
if (author == null) {
    author = create.newRecord(AUTHOR);
    author.setId(1);
    author.setFirstName("Dan");
    author.setLastName("Brown");
}

// Mark the author as a "distinguished" author and store it
author.setDistinguished(1);

// Executes an update on existing authors, or insert on new ones
author.store();
```

If you wish to use all of jOOQ's features, the following sections of the manual will be of interest to you (including all sub-sections):

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database
- SQL execution: This section contains a lot of information about executing SQL statements using the jOOQ API

2.3.5. jOOQ for PROs

jOOQ isn't just a library that helps you build and execute SQL against your generated, compilable schema. jOOQ ships with a lot of tools. Here are some of the most important tools shipped with jOOQ:

- jOOQ's Execute Listeners: jOOQ allows you to hook your custom execute listeners into jOOQ's SQL statement execution lifecycle in order to centrally coordinate any arbitrary operation performed on SQL being executed. Use this for logging, identity generation, SQL tracing, performance measurements, etc.
- Logging: jOOQ has a standard DEBUG logger built-in, for logging and tracing all your executed SQL statements and fetched result sets
- Stored Procedures: jOOQ supports stored procedures and functions of your favourite database. All routines and user-defined types are generated and can be included in jOOQ's SQL building API as function references.
- Batch execution: Batch execution is important when executing a big load of SQL statements. jOOQ simplifies these operations compared to JDBC
- Exporting and Importing: jOOQ ships with an API to easily export/import data in various formats

If you're a power user of your favourite, feature-rich database, jOOQ will help you access all of your database's vendor-specific features, such as OLAP features, stored procedures, user-defined types, vendor-specific SQL, functions, etc. Examples are given throughout this manual.

2.4. Downloading jOOQ

jOOQ is distributed over 3 main channels:

- The website as downloadable ZIP files: https://www.jooq.org/download/versions
- The repository for jOOQ's commercial editions only: https://repo.jooq.org
- Maven Central for jOOQ's open source edition only: https://repo1.maven.org/maven2/org/jooq

The ZIP file

If you choose to download jOOQ over the website, you will be able to download a ZIP file with the following layout:

- maven-deploy.bat: A Windows batch script to deploy artifacts to a maven repository
- maven-deploy.sh: A bash script to deploy artifacts to a maven repository
- maven-install.bat: A Windows batch script to install artifacts to the local maven repository
- maven-install.sh: A bash script to install artifacts to the local maven repository

The website hosts the latest versions of the jOOQ Open Source Edition as well as all the historic versions of the commercial jOOQ editions including snapshot builds of all distributions that are available to paying customers only.

The commercial artifact repository

The commercial artifact repository hosts all the historic versions of the commercial jOOQ editions including snapshot builds of all distributions that are available to paying customers only.

Below is information regarding how to include these dependencies in Maven / Gradle:

Maven

settings.xml

```
<server>
    <id>jooq-pro</id>
    <username>[your licensee email]</username>
    <password>[your license key]</password>
</server>
```

pom.xml

```
<repositories>
    <repository>
        <id>central</id>
        <url>https://repo1.maven.org/maven2/</url>
    </repository>
    <!-- Other repositories ... -->

    <repository>
        <id>jooq-pro</id>
        <url>https://repo.jooq.org/repo</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>central</id>
        <url>https://repo1.maven.org/maven2/</url>
    </pluginRepository>
    <!-- Other repositories ... -->

    <pluginRepository>
        <id>jooq-pro</id>
        <url>https://repo.jooq.org/repo</url>
    </pluginRepository>
</pluginRepositories>
```

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)

Dependencies

Depending on the edition you're using, please declare the following dependencies in Maven or Gradle: Maven

```
<dependency>
    <!-- Use org.jooq              for the Open Source Edition
             org.jooq.pro          for commercial editions with Java 11 support,
             org.jooq.pro-java-8   for commercial editions with Java 8 support,
             org.jooq.pro-java-6   for commercial editions with Java 6 support,
             org.jooq.trial        for the free trial edition with Java 11 support,
             org.jooq.trial-java-8 for the free trial edition with Java 8 support,
             org.jooq.trial-java-6 for the free trial edition with Java 6 support

     Note: Only the Open Source Edition is hosted on Maven Central.
           Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
           See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

    <groupId>org.jooq</groupId>
    <artifactId>jooq</artifactId>
    <version>3.12.4</version>
</dependency>
```

Gradle (Kotlin)

```
dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    implementation("org.jooq:jooq:3.12.4")
}
```

Gradle (Groovy)

2.5. Tutorials

Don't have time to read the full manual? Here are a couple of tutorials that will get you into the most essential parts of jOOQ as quick as possible.

2.5.1. jOOQ in 7 easy steps

This manual section is intended for new users, to help them get a running application with jOOQ, quickly.

2.5.1.1. Step 1: Preparation

If you haven't already downloaded it, download jOOQ:
https://www.jooq.org/download

Alternatively, you can create a Maven dependency to download jOOQ artefacts:

Open Source Edition

```
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq</artifactId>
  <version>3.12.4</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.12.4</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.12.4</version>
</dependency>
```

Commercial Editions (Java 9+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq</artifactId>
  <version>3.12.4</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.12.4</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.12.4</version>
</dependency>
```

© 2009 - 2025 by Data Geekery™ GmbH.

Commercial Editions (Java 8+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro-java-8</groupId>
  <artifactId>jooq</artifactId>
  <version>3.12.4</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.pro-java-8</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.12.4</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-8</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.12.4</version>
</dependency>
```

Commercial Editions (Java 6+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq</artifactId>
  <version>3.12.4</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.12.4</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.12.4</version>
</dependency>
```

Commercial Editions (Free Trial)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq</artifactId>
  <version>3.12.4</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.12.4</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.12.4</version>
</dependency>
```

Note that only the jOOQ Open Source Edition is available from Maven Central. If you're using the jOOQ Professional Edition or the jOOQ Enterprise Edition, you will have to manually install jOOQ in your local Nexus, or in your local Maven cache. For more information, please refer to the licensing pages.

Please refer to the manual's section about Code generation configuration to learn how to use jOOQ's code generator with Maven.

For this example, we'll be using MySQL. If you haven't already downloaded MySQL Connector/J, download it here:
https://dev.mysql.com/downloads/connector/j/

If you don't have a MySQL instance up and running yet, get it from https://www.mysql.com or https://hub.docker.com/_/mysql now!

2.5.1.2. Step 2: Your database

We're going to create a database called "library" and a corresponding "author" table. Connect to MySQL via your command line client and type the following:

```
CREATE DATABASE `library`;

USE `library`;

CREATE TABLE `author` (
  `id` int NOT NULL,
  `first_name` varchar(255) DEFAULT NULL,
  `last_name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
);
```

2.5.1.3. Step 3: Code generation

In this step, we're going to use jOOQ's command line tools to generate classes that map to the Author table we just created. More detailed information about how to set up the jOOQ code generator can be found here:
jOOQ manual pages about setting up the code generator

The easiest way to generate a schema is to copy the jOOQ jar files (there should be 3) and the MySQL Connector jar file to a temporary directory. Then, create a library.xml that looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
  <!-- Configure the database connection here -->
  <jdbc>
    <driver>com.mysql.cj.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost:3306/library</url>
    <user>root</user>
    <password></password>
  </jdbc>

  <generator>
    <!-- The default code generator. You can override this one, to generate your own code style.
         Supported generators:
         - org.jooq.codegen.JavaGenerator
         - org.jooq.codegen.ScalaGenerator
         Defaults to org.jooq.codegen.JavaGenerator -->
    <name>org.jooq.codegen.JavaGenerator</name>

    <database>
      <!-- The database type. The format here is:
           org.jooq.meta.[database].[database]Database -->
      <name>org.jooq.meta.mysql.MySQLDatabase</name>

      <!-- The database schema (or in the absence of schema support, in your RDBMS this
           can be the owner, user, database name) to be generated -->
      <inputSchema>library</inputSchema>

      <!-- All elements that are generated from your schema
           (A Java regular expression. Use the pipe to separate several expressions)
           Watch out for case-sensitivity. Depending on your database, this might be important! -->
      <includes>.*</includes>

      <!-- All elements that are excluded from your schema
           (A Java regular expression. Use the pipe to separate several expressions).
           Excludes match before includes, i.e. excludes have a higher priority -->
      <excludes></excludes>
    </database>

    <target>
      <!-- The destination package of your generated classes (within the destination directory) -->
      <packageName>test.generated</packageName>

      <!-- The destination directory of your generated classes. Using Maven directory layout here -->
      <directory>C:/workspace/MySQLTest/src/main/java</directory>
    </target>
  </generator>
</configuration>
```

Replace the username (<username/> or <user/>) with whatever user has the appropriate privileges to query the database meta data. You'll also want to look at the other values and replace as necessary. Here are the two interesting properties:

<packageName/> - set this to the parent package you want to create for the generated classes. Setting the value to test.generated will cause the test.generated.tables.Author and test.generated.tables.records.AuthorRecord classes to be created

<directory/> - the directory to output the generated classes to.

Once you have the JAR files and library.xml in your temp directory, type this on a Windows machine:

```
java -classpath jooq-3.12.4.jar;^
jooq-meta-3.12.4.jar;^
jooq-codegen-3.12.4.jar;^
jaxb-api-2.3.1.jar;^
mysql-connector-java.jar;. ^
org.jooq.codegen.GenerationTool library.xml
```

... or type this on a UNIX / Linux / Mac system (colons instead of semi-colons):

```
java -classpath jooq-3.12.4.jar:\
jooq-meta-3.12.4.jar:\
jooq-codegen-3.12.4.jar:\
jaxb-api-2.3.1.jar:\
mysql-connector-java.jar:. \
org.jooq.codegen.GenerationTool library.xml
```

(!)

- *jOOQ will try loading the **library.xml** from your classpath. This is also why there is a trailing period (.) on the classpath. If the file cannot be found on the classpath, jOOQ will look on the file system from the current working directory.*
- *Replace the filenames with your actual filenames. In this example, jOOQ 3.12.4 is being used.*
- *If you're using a linux style shell on Windows, but a Windows JDK/JRE, you still need to use semi-colons in your classpath! (;) In git-bash, you might have to quote your classpath ("jooq-3.12.4.jar;jooq-meta-3.12.4.jar;...")*

If everything has worked, you should see this in your console output:

```
Nov 1, 2011 7:25:06 PM org.jooq.impl.JooqLogger info
INFO: Initialising properties  : /library.xml
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Database parameters
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ----------------------------------------------------------
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   dialect                : MYSQL
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   schema                 : library
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   target dir             : C:/workspace/MySQLTest/src
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   target package         : test.generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ----------------------------------------------------------
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Emptying                 : C:/workspace/MySQLTest/src/test/generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating classes in    : C:/workspace/MySQLTest/src/test/generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating schema        : Library.java
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Schema generated         : Total: 122.18ms
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Sequences fetched        : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Tables fetched           : 5 (5 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating tables        : C:/workspace/MySQLTest/src/test/generated/tables
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ARRAYs fetched           : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Enums fetched            : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: UDTs fetched             : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating table         : Author.java
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Tables generated         : Total: 680.464ms, +558.284ms
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating Keys          : C:/workspace/MySQLTest/src/test/generated/tables
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Keys generated           : Total: 718.621ms, +38.157ms
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Generating records       : C:/workspace/MySQLTest/src/test/generated/tables/records
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Generating record        : AuthorRecord.java
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Table records generated  : Total: 782.545ms, +63.924ms
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Routines fetched         : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Packages fetched         : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: GENERATION FINISHED!     : Total: 791.688ms, +9.143ms
```

2.5.1.4. Step 4: Connect to your database

Let's just write a vanilla main class in the project containing the generated classes:

```
// For convenience, always static import your generated tables and jOOQ functions to decrease verbosity:
import static test.generated.Tables.*;
import static org.jooq.impl.DSL.*;

import java.sql.*;

public class Main {
    public static void main(String[] args) {
        String userName = "root";
        String password = "";
        String url = "jdbc:mysql://localhost:3306/library";

        // Connection is the only JDBC resource that we need
        // PreparedStatement and ResultSet are handled by jOOQ, internally
        try (Connection conn = DriverManager.getConnection(url, userName, password)) {
            // ...
        }

        // For the sake of this tutorial, let's keep exception handling simple
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This is pretty standard code for establishing a MySQL connection.

2.5.1.5. Step 5: Querying

Let's add a simple query constructed with jOOQ's query DSL:

```
DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
Result<Record> result = create.select().from(AUTHOR).fetch();
```

First get an instance of DSLContext so we can write a simple SELECT query. We pass an instance of the MySQL connection to DSL. Note that the DSLContext doesn't close the connection. We'll have to do that ourselves.

We then use jOOQ's query DSL to return an instance of Result. We'll be using this result in the next step.

2.5.1.6. Step 6: Iterating

After the line where we retrieve the results, let's iterate over the results and print out the data:

```
for (Record r : result) {
    Integer id = r.getValue(AUTHOR.ID);
    String firstName = r.getValue(AUTHOR.FIRST_NAME);
    String lastName = r.getValue(AUTHOR.LAST_NAME);

    System.out.println("ID: " + id + " first name: " + firstName + " last name: " + lastName);
}
```

The full program should now look like this:

```
package test;

// For convenience, always static import your generated tables and
// jOOQ functions to decrease verbosity:
import static test.generated.Tables.*;
import static org.jooq.impl.DSL.*;

import java.sql.*;

import org.jooq.*;
import org.jooq.impl.*;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String userName = "root";
        String password = "";
        String url = "jdbc:mysql://localhost:3306/library";

        // Connection is the only JDBC resource that we need
        // PreparedStatement and ResultSet are handled by jOOQ, internally
        try (Connection conn = DriverManager.getConnection(url, userName, password)) {
            DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
            Result<Record> result = create.select().from(AUTHOR).fetch();

            for (Record r : result) {
                Integer id = r.getValue(AUTHOR.ID);
                String firstName = r.getValue(AUTHOR.FIRST_NAME);
                String lastName = r.getValue(AUTHOR.LAST_NAME);

                System.out.println("ID: " + id + " first name: " + firstName + " last name: " + lastName);
            }
        }

        // For the sake of this tutorial, let's keep exception handling simple
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2.5.1.7. Step 7: Explore!

jOOQ has grown to be a comprehensive SQL library. For more information, please consider the documentation:
https://www.jooq.org/learn

... explore the Javadoc:
https://www.jooq.org/javadoc/latest/

... or join the news group:
https://groups.google.com/forum/#!forum/jooq-user

This tutorial is the courtesy of Ikai Lan. See the original source here:
https://ikaisays.com/2011/11/01/getting-started-with-jooq-a-tutorial/

2.5.2. Using jOOQ with Flyway

When performing database migrations, we at Data Geekery recommend using jOOQ with Flyway - Database Migrations Made Easy. In this chapter, we're going to look into a simple way to get started with the two frameworks.

Philosophy

There are a variety of ways how jOOQ and Flyway could interact with each other in various development setups. In this tutorial we're going to show just one variant of such framework team play - a variant that we find particularly compelling for most use cases.

The general philosophy behind the following approach can be summarised as this:

- 1. Database increment
- 2. Database migration
- 3. Code re-generation
- 4. Development

The four steps above can be repeated time and again, every time you need to modify something in your database. More concretely, let's consider:

- 1. Database increment - You need a new column in your database, so you write the necessary DDL in a Flyway script
- 2. Database migration - This Flyway script is now part of your deliverable, which you can share with all developers who can migrate their databases with it, the next time they check out your change
- 3. Code re-generation - Once the database is migrated, you regenerate all jOOQ artefacts (see code generation), locally
- 4. Development - You continue developing your business logic, writing code against the updated, generated database schema

Maven Project Configuration - Properties

The following properties are defined in our pom.xml, to be able to reuse them between plugin configurations:

```
<properties>
    <db.url>jdbc:h2:~/flyway-test</db.url>
    <db.username>sa</db.username>
</properties>
```

0. Maven Project Configuration - Dependencies

While jOOQ and Flyway could be used in standalone migration scripts, in this tutorial, we'll be using Maven for the standard project setup.

These are the dependencies that we're using in our Maven configuration:

```
<!-- We'll add the latest version of jOOQ and our JDBC driver - in this case H2 -->
<dependency>
    <!-- Use org.jooq            for the Open Source Edition
             org.jooq.pro        for commercial editions,
             org.jooq.pro-java-8 for commercial editions with Java 8 support,
             org.jooq.pro-java-6 for commercial editions with Java 6 support,
             org.jooq.trial      for the free trial edition

         Note: Only the Open Source Edition is hosted on Maven Central.
               Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq</artifactId>
    <version>3.12.4</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
</dependency>

<!-- For improved logging, we'll be using log4j via slf4j to see what's going on during migration and code generation -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.11.0</version>
</dependency>

<!-- To ensure our code is working, we're using JUnit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
```

0. Maven Project Configuration - Plugins

After the dependencies, let's simply add the Flyway and jOOQ Maven plugins like so. The Flyway plugin:

```
<plugin>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-maven-plugin</artifactId>
    <version>3.0</version>

    <!-- Note that we're executing the Flyway plugin in the "generate-sources" phase -->
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>migrate</goal>
            </goals>
        </execution>
    </executions>

    <!-- Note that we need to prefix the db/migration path with filesystem: to prevent Flyway
         from looking for our migration scripts only on the classpath -->
    <configuration>
        <url>${db.url}</url>
        <user>${db.username}</user>
        <locations>
            <location>filesystem:src/main/resources/db/migration</location>
        </locations>
    </configuration>
</plugin>
```

The above Flyway Maven plugin configuration will read and execute all database migration scripts from src/main/resources/db/migration prior to compiling Java source code. While the official Flyway documentation may suggest that migrations be done in the compile phase, the jOOQ code generator relies on such migrations having been done *prior* to code generation.

After the Flyway plugin, we'll add the jOOQ Maven Plugin. For more details, please refer to the manual's section about the code generation configuration.

```
<plugin>
    <!-- Use org.jooq            for the Open Source Edition
             org.jooq.pro        for commercial editions,
             org.jooq.pro-java-8 for commercial editions with Java 8 support,
             org.jooq.pro-java-6 for commercial editions with Java 6 support,
             org.jooq.trial      for the free trial edition

         Note: Only the Open Source Edition is hosted on Maven Central.
               Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <version>${org.jooq.version}</version>

    <!-- The jOOQ code generation plugin is also executed in the generate-sources phase, prior to compilation -->
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>

    <!-- This is a minimal working configuration. See the manual's section about the code generator for more details -->
    <configuration>
        <jdbc>
            <url>${db.url}</url>
            <user>${db.username}</user>
        </jdbc>
        <generator>
            <database>
                <includes>.*</includes>
                <inputSchema>FLYWAY_TEST</inputSchema>
            </database>
            <target>
                <packageName>org.jooq.example.flyway.db.h2</packageName>
                <directory>target/generated-sources/jooq-h2</directory>
            </target>
        </generator>
    </configuration>
</plugin>
```

This configuration will now read the FLYWAY_TEST schema and reverse-engineer it into the target/generated-sources/jooq-h2 directory, and within that, into the org.jooq.example.flyway.db.h2 package.

1. Database increments

Now, when we start developing our database. For that, we'll create database increment scripts, which we put into the src/main/resources/db/migration directory, as previously configured for the Flyway plugin. We'll add these files:

- V1__initialise_database.sql
- V2__create_author_table.sql
- V3__create_book_table_and_records.sql

These three scripts model our schema versions 1-3 (note the capital V!). Here are the scripts' contents

```
-- V1__initialise_database.sql
DROP SCHEMA flyway_test IF EXISTS;

CREATE SCHEMA flyway_test;
```

```
-- V2__create_author_table.sql
CREATE SEQUENCE flyway_test.s_author_id START WITH 1;

CREATE TABLE flyway_test.author (
  id INT NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50) NOT NULL,
  date_of_birth DATE,
  year_of_birth INT,
  address VARCHAR(50),

  CONSTRAINT pk_author PRIMARY KEY (ID)
);
```

```
-- V3__create_book_table_and_records.sql
CREATE TABLE flyway_test.book (
  id INT NOT NULL,
  author_id INT NOT NULL,
  title VARCHAR(400) NOT NULL,

  CONSTRAINT pk_book PRIMARY KEY (id),
  CONSTRAINT fk_book_author_id FOREIGN KEY (author_id) REFERENCES flyway_test.author(id)
);


INSERT INTO flyway_test.author VALUES (next value for flyway_test.s_author_id, 'George', 'Orwell', '1903-06-25', 1903, null);
INSERT INTO flyway_test.author VALUES (next value for flyway_test.s_author_id, 'Paulo', 'Coelho', '1947-08-24', 1947, null);

INSERT INTO flyway_test.book VALUES (1, 1, '1984');
INSERT INTO flyway_test.book VALUES (2, 1, 'Animal Farm');
INSERT INTO flyway_test.book VALUES (3, 2, 'O Alquimista');
INSERT INTO flyway_test.book VALUES (4, 2, 'Brida');
```

2. Database migration and 3. Code regeneration

The above three scripts are picked up by Flyway and executed in the order of the versions. This can be seen very simply by executing:

```
mvn clean install
```

And then observing the log output from Flyway...

```
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] Database: jdbc:h2:~/flyway-test (H2 1.4)
[INFO] Validated 3 migrations (execution time 00:00.004s)
[INFO] Creating Metadata table: "PUBLIC"."schema_version"
[INFO] Current version of schema "PUBLIC": << Empty Schema >>
[INFO] Migrating schema "PUBLIC" to version 1
[INFO] Migrating schema "PUBLIC" to version 2
[INFO] Migrating schema "PUBLIC" to version 3
[INFO] Successfully applied 3 migrations to schema "PUBLIC" (execution time 00:00.073s).
```

... and from jOOQ on the console:

```
[INFO] --- jooq-codegen-maven:3.12.4:generate (default) @ jooq-flyway-example ---
[INFO] --- jooq-codegen-maven:3.12.4:generate (default) @ jooq-flyway-example ---
[INFO] Using this configuration:
...
[INFO] Generating schemata      : Total: 1
[INFO] Generating schema        : FlywayTest.java
[INFO] ---------------------------------------------------------
[....]
[INFO] GENERATION FINISHED!     : Total: 337.576ms, +4.299ms
```

4. Development

Note that all of the previous steps are executed automatically, every time someone adds new migration scripts to the Maven module. For instance, a team member might have committed a new migration script, you check it out, rebuild and get the latest jOOQ-generated sources for your own development or integration-test database.

Now, that these steps are done, you can proceed writing your database queries. Imagine the following test case

```
import org.jooq.Result;
import org.jooq.impl.DSL;
import org.junit.Test;

import java.sql.DriverManager;

import static java.util.Arrays.asList;
import static org.jooq.example.flyway.db.h2.Tables.*;
import static org.junit.Assert.assertEquals;

public class AfterMigrationTest {

    @Test
    public void testQueryingAfterMigration() throws Exception {
        try (Connection c = DriverManager.getConnection("jdbc:h2:~/flyway-test", "sa", "")) {
            Result<?> result =
            DSL.using(c)
               .select(
                   AUTHOR.FIRST_NAME,
                   AUTHOR.LAST_NAME,
                   BOOK.ID,
                   BOOK.TITLE
               )
               .from(AUTHOR)
               .join(BOOK)
               .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
               .orderBy(BOOK.ID.asc())
               .fetch();

            assertEquals(4, result.size());
            assertEquals(asList(1, 2, 3, 4), result.getValues(BOOK.ID));
        }
    }
}
```

Reiterate

The power of this approach becomes clear once you start performing database modifications this way. Let's assume that the French guy on our team prefers to have things his way:

```
-- V4__le_french.sql
ALTER TABLE flyway_test.book ALTER COLUMN title RENAME TO le_titre;
```

They check it in, you check out the new database migration script, run

```
mvn clean install
```

And then observing the log output:

```
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] Database: jdbc:h2:~/flyway-test (H2 1.4)
[INFO] Validated 4 migrations (execution time 00:00.005s)
[INFO] Current version of schema "PUBLIC": 3
[INFO] Migrating schema "PUBLIC" to version 4
[INFO] Successfully applied 1 migration to schema "PUBLIC" (execution time 00:00.016s).
```

So far so good, but later on:

```
[ERROR] COMPILATION ERROR :
[INFO] -------------------------------------------------------------
[ERROR] C:\...\jOOQ-flyway-example\src\test\java\AfterMigrationTest.java:[24,19] error: cannot find symbol
[INFO] 1 error
```

When we go back to our Java integration test, we can immediately see that the TITLE column is still being referenced, but it no longer exists:

```
public class AfterMigrationTest {

    @Test
    public void testQueryingAfterMigration() throws Exception {
        try (Connection c = DriverManager.getConnection("jdbc:h2:~/flyway-test", "sa", "")) {
            Result<?> result =
            DSL.using(c)
                .select(
                    AUTHOR.FIRST_NAME,
                    AUTHOR.LAST_NAME,
                    BOOK.ID,
                    BOOK.TITLE
                    //   ^^^^^ This column no longer exists. We'll have to rename it to LE_TITRE
                )
                .from(AUTHOR)
                .join(BOOK)
                .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
                .orderBy(BOOK.ID.asc())
                .fetch();

            assertEquals(4, result.size());
            assertEquals(asList(1, 2, 3, 4), result.getValues(BOOK.ID));
        }
    }
}
```

Automation

The above steps can be automated in your build using another third party called testcontainers. Please look at this article here for examples on how to do that: https://blog.jooq.org/using-testcontainers-to-generate-jooq-code/

Conclusion

This tutorial shows very easily how you can build a rock-solid development process using Flyway and jOOQ to prevent SQL-related errors very early in your development lifecycle - immediately at compile time, rather than in production!

Please, visit the Flyway website for more information about Flyway.

2.5.3. Using jOOQ with jbang

jbang allows for quickly working with all sorts of Java libraries without the hassle of setting up environments, dependencies, etc. This catalog allows for using jOOQ's code generator right away on an existing database.

For more information on jbang, see:

- Installation
- Usage

An example

In a shell, type

```
git clone https://github.com/jOOQ/jbang-example
cd jbang-example
jbang Example.java
```

In order to re-generate the example code, e.g. when your schema changes, just type:

```
jbang codegen@jooq db.xml
```

If you prefer working with a pre-existing database, just edit the db.xml file and point it to your database. Add the JDBC driver dependency like this:

```
jbang --deps org.postgresql:postgresql:RELEASE codegen@jooq db.xml
```

To override the jOOQ version from the default RELEASE to a specific version, use

```
jbang -Djooq.version=<version> codegen@jooq db.xml
```

2.6. jOOQ and Java 8

Java 8 has introduced a great set of enhancements, among which lambda expressions and the new java.util.stream.Stream. These new constructs align very well with jOOQ's fluent API as can be seen in the following examples:

jOOQ and lambda expressions

jOOQ's RecordMapper API is fully Java-8-ready, which basically means that it is a SAM (Single Abstract Method) type, which can be instanciated using a lambda expression. Consider this example:

```
try (Connection c = getConnection()) {
    String sql = "select schema_name, is_default " +
                 "from information_schema.schemata " +
                 "order by schema_name";

    DSL.using(c)
       .fetch(sql)

       // We can use lambda expressions to map jOOQ Records
       .map(rs -> new Schema(
           rs.getValue("SCHEMA_NAME", String.class),
           rs.getValue("IS_DEFAULT", boolean.class)
       ))

       // ... and then profit from the new Collection methods
       .forEach(System.out::println);
}
```

The above example shows how jOOQ's Result.map() method can receive a lambda expression that implements RecordMapper to map from jOOQ Records to your custom types.

jOOQ and the Streams API

jOOQ's Result type extends java.util.List, which opens up access to a variety of new Java features in Java 8. The following example shows how easy it is to transform a jOOQ Result containing INFORMATION_SCHEMA meta data to produce DDL statements:

```
DSL.using(c)
   .select(
       COLUMNS.TABLE_NAME,
       COLUMNS.COLUMN_NAME,
       COLUMNS.TYPE_NAME
   )
   .from(COLUMNS)
   .orderBy(
       COLUMNS.TABLE_CATALOG,
       COLUMNS.TABLE_SCHEMA,
       COLUMNS.TABLE_NAME,
       COLUMNS.ORDINAL_POSITION
   )
   .fetch()  // jOOQ ends here
   .stream() // JDK 8 Streams start here
   .collect(groupingBy(
       r -> r.getValue(COLUMNS.TABLE_NAME),
       LinkedHashMap::new,
       mapping(
           r -> new Column(
               r.getValue(COLUMNS.COLUMN_NAME),
               r.getValue(COLUMNS.TYPE_NAME)
           ),
           toList()
       )
   ))
   .forEach(
       (table, columns) -> {
            // Just emit a CREATE TABLE statement
            System.out.println(
                "CREATE TABLE " + table + " (");

            // Map each "Column" type into a String
            // containing the column specification,
            // and join them using comma and
            // newline. Done!
            System.out.println(
                columns.stream()
                        .map(col -> "  " + col.name +
                                    " " + col.type)
                        .collect(Collectors.joining(",\n"))
            );

           System.out.println(");");
       }
   );
```

2.7. jOOQ and Scala

As any other library, jOOQ can be easily used in Scala, taking advantage of the many Scala language features such as for example:

- Optional "." to dereference methods from expressions
- Optional "(" and ")" to delimit method argument lists
- Optional ";" at the end of a Scala statement
- Type inference using "var" and "val" keywords
- Lambda expressions and for-comprehension syntax for record iteration and data type conversion

But jOOQ also leverages other useful Scala features, such as

- implicit defs for operator overloading
- Scala Macros (soon to come)

All of the above heavily improve jOOQ's querying DSL API experience for Scala developers.

A short example jOOQ application in Scala might look like this:

```
import collection.JavaConversions._                                  // Import implicit defs for iteration over org.jooq.Result
                                                                     //
import java.sql.DriverManager                                        //
                                                                     //
import org.jooq._                                                    //
import org.jooq.impl._                                               //
import org.jooq.impl.DSL._                                           //
import org.jooq.examples.scala.h2.Tables._                           //
import org.jooq.scalaextensions.Conversions._                        // Import implicit defs for overloaded jOOQ/SQL operators
                                                                     //
object Test {                                                        //
  def main(args: Array[String]): Unit = {                            //
    val c = DriverManager.getConnection("jdbc:h2:~/test", "sa", ""); // Standard JDBC connection
    val e = DSL.using(c, SQLDialect.H2);                             //
    val x = AUTHOR as "x"                                            // SQL-esque table aliasing
                                                                     //
    for (r <- e                                                      // Iteration over Result. "r" is an org.jooq.Record3
        select (                                                     //
          BOOK.ID * BOOK.AUTHOR_ID,                                  // Using the overloaded "*" operator
          BOOK.ID + BOOK.AUTHOR_ID * 3 + 4,                          // Using the overloaded "+" operator
          BOOK.TITLE || " abc" || " xy"                              // Using the overloaded "||" operator
        )                                                            //
        from BOOK                                                    // No need to use parentheses or "." here
        leftOuterJoin (                                              //
          select (x.ID, x.YEAR_OF_BIRTH)                             // Dereference fields from aliased table
          from x                                                     //
          limit 1                                                    //
          asTable x.getName()                                        //
        )                                                            //
        on BOOK.AUTHOR_ID === x.ID                                   // Using the overloaded "===" operator
        where (BOOK.ID <> 2)                                         // Using the olerloaded "<>" operator
        or (BOOK.TITLE in ("O Alquimista", "Brida"))                 // Neat IN predicate expression
        fetch                                                        //
    ) {                                                              //
      println(r)                                                     //
    }                                                                //
  }                                                                  //
}
```

For more details about jOOQ's Scala integration, please refer to the manual's section about SQL building with Scala.

2.8. jOOQ and Groovy

As any other library, jOOQ can be easily used in Groovy, taking advantage of the many Groovy language features such as for example:

- Optional ";" at the end of a Groovy statement
- Type inference for local variables

A short example jOOQ application in Groovy might look like this:

Note that while Groovy supports some means of operator overloading, we think that these means should be avoided in a jOOQ integration. For instance, a + b in Groovy maps to a formal a.plus(b) method invocation, and jOOQ provides the required synonyms in its API to help you write such expressions. Nonetheless, Groovy only offers little typesafety, and as such, operator overloading can lead to many runtime issues.

Another caveat of Groovy operator overloading is the fact that operators such as == or >= map to a.equals(b), a.compareTo(b) == 0, a.compareTo(b) >= 0 respectively. This behaviour does not make sense in a fluent API such as jOOQ.

2.9. jOOQ and Kotlin

As any other library, jOOQ can be easily used in Kotlin, taking advantage of the many Kotlin language features such as for example:

- Optional ";" at the end of a Kotlin statement
- Type inference for local variables

A short example jOOQ application in Kotlin might look like this:

Note that Kotlin supports some means of operator overloading. For instance, a + b in Kotlin maps to a formal a.plus(b) method invocation, and jOOQ provides the required synonyms in its API to help you write such expressions.

One particularly nice language feature is the fact that [square brackets] allow for accessing any object's contents via get() and set() methods. Instead of using the above value1(), value2(), and value3() methods, we could also iterate as such:

A caveat of Kotlin operator overloading is the fact that operators such as == or >= map to a.equals(b), a.compareTo(b) == 0, a.compareTo(b) >= 0 respectively. This behaviour does not make sense in a fluent API such as jOOQ.

2.10. jOOQ and NoSQL

jOOQ users often get excited about jOOQ's intuitive API and would then wish for NoSQL support.

There are a variety of NoSQL databases that implement some sort of proprietary query language. Some of these query languages even look like SQL. Examples are JCR-SQL2, CQL (Cassandra Query Language), Cypher (Neo4j's Query Language), and many more.

Mapping the jOOQ API onto these alternative query languages would be a very poor fit and a leaky abstraction. We believe in the power and expressivity of the SQL standard and its various dialects. Databases that extend this standard too much, or implement it not thoroughly enough are often not suitable targets for jOOQ. It would be better to build a new, dedicated API for just that one particular query language. E.g. for Cypher, there's Cypher-DSL, which is a much better fit.

jOOQ is about SQL, and about SQL alone.

2.11. jOOQ and JPA

Just because you're using jOOQ doesn't mean you have to use it for everything!

When introducing jOOQ into an existing application that uses JPA, the common question is always: "Should we replace JPA by jOOQ?" and "How do we proceed doing that?"

Beware that jOOQ is not a replacement for JPA. Think of jOOQ as a complement. JPA (and ORMs in general) try to solve the ***object graph persistence*** problem. In short, this problem is about

- Loading an entity graph into client memory from a database
- Manipulating that graph in the client
- Storing the modification back to the database

As the above graph gets more complex, a lot of tricky questions arise like:

- What's the optimal order of SQL DML operations for loading and storing entities?
- How can we batch the commands more efficiently?
- How can we keep the transaction footprint as low as possible without compromising on ACID?
- How can we implement optimistic locking?

jOOQ only has *some* of the answers.

While jOOQ does offer updatable records that help running simple CRUD, a batch API, optimistic locking capabilities, jOOQ mainly focuses on executing actual SQL statements.

SQL is the preferred language of database interaction, when any of the following are given:

- You run reports and analytics on large data sets directly in the database
- You import / export data using ETL
- You run complex business logic as SQL queries

Whenever SQL is a good fit, jOOQ is a good fit. Whenever you're operating and persisting the *object graph*, JPA is a good fit.

And sometimes, it's best to combine both

2.12. Build your own

In order to build jOOQ (Open Source Edition) yourself, please download the sources from https://github.com/jOOQ/jOOQ and use Maven to build jOOQ, preferably in Eclipse. The jOOQ Open Source Edition requires Java 8+ to compile and run. The commercial jOOQ Editions require Java 8+ or Java 6+ to compile and run, depending on the distribution.

Some useful hints to build jOOQ yourself:

- Get the latest version of Git or EGit
- Get the latest version of Maven or M2E
- Check out the jOOQ sources from https://github.com/jOOQ/jOOQ
- Optionally, import Maven artefacts into an Eclipse workspace using the following command (see the maven-eclipse-plugin documentation for details):

  * mvn eclipse:eclipse

- Build the jooq-parent artefact by using any of these commands:

  * mvn clean package
    create .jar files in ${project.build.directory}
  * mvn clean install
    install the .jar files in your local repository (e.g. ~/.m2)
  * mvn clean {goal} -Dmaven.test.skip=true
    don't run unit tests when building artefacts

2.13. jOOQ and backwards-compatibility

Semantic versioning

jOOQ's understanding of backwards compatibility is inspired by the rules of semantic versioning according to https://semver.org. Those rules impose a versioning scheme [X].[Y].[Z] that can be summarised as follows:

- If a patch release includes bugfixes, performance improvements and API-irrelevant new features, [Z] is incremented by one.
- If a minor release includes backwards-compatible, API-relevant new features, [Y] is incremented by one and [Z] is reset to zero.
- If a major release includes backwards-incompatible, API-relevant new features, [X] is incremented by one and [Y], [Z] are reset to zero.

jOOQ's understanding of backwards-compatibility

Backwards-compatibility is important to jOOQ. You've chosen jOOQ as a strategic SQL engine and you don't want your SQL to break.

However, there are some elements of API evolution that would be considered backwards-incompatible in other APIs, but not in jOOQ. As discussed later on in the section about jOOQ's query DSL API, much of jOOQ's API is indeed an internal domain-specific language implemented mostly using Java interfaces. Adding language elements to these interfaces means any of these actions:

- Adding methods to the interface
- Overloading methods for convenience
- Changing the type hierarchy of interfaces (including raw type or binary compatibility implications)

It becomes obvious that it would be impossible to add new language elements (e.g. new SQL functions, new SELECT clauses) to the API without breaking any client code that actually implements those interfaces. Hence, the following rules should be observed:

- jOOQ's DSL interfaces should not be implemented by client code! Extend only those extension points that are explicitly documented as "extendable" (e.g. custom QueryParts).
- Generated code implements such interfaces and extends internal classes, and as such is recommended to be re-generated with a matching code generator version every time the runtime library is upgraded.
- Binary compatibility can be expected from patch releases, but not from minor releases as it is not practical to maintain binary compatibility in an internal DSL.
- Source compatibility can be expected from patch and minor releases, the exception being raw type compatibility (see #11879), and rare exceptions where API design is clearly lacking.
- Behavioural compatibility can be expected from patch and minor releases.
- Any jOOQ SPI XYZ that is meant to be implemented ships with a DefaultXYZ or AbstractXYZ, which can be used safely as a default implementation.

jOOQ-codegen and jOOQ-meta

While a reasonable amount of care is spent to maintain these two modules under the rules of semantic versioning, it may well be that minor releases introduce backwards-incompatible changes. This will be announced in the respective release notes and should be the exception.

© 2009 - 2025 by Data Geekery™ GmbH.

3. SQL building

SQL is a declarative language that is hard to integrate into procedural, object-oriented, functional or any other type of programming languages. jOOQ's philosophy is to give SQL the credit it deserves and integrate SQL itself as an "internal domain specific language" directly into Java.

With this philosophy in mind, SQL building is the main feature of jOOQ. All other features (such as SQL execution and code generation) are mere convenience built on top of jOOQ's SQL building capabilities.

This section explains all about the various syntax elements involved with jOOQ's SQL building capabilities. For a complete overview of all syntax elements, please refer to the manual's sections about SQL to DSL mapping rules.

3.1. The query DSL type

jOOQ exposes a lot of interfaces and hides most implementation facts from client code. The reasons for this are:

- Interface-driven design. This allows for modelling queries in a fluent API most efficiently
- Reduction of complexity for client code.
- API guarantee. You only depend on the exposed interfaces, not concrete (potentially dialect-specific) implementations.

The org.jooq.impl.DSL class is the main class from where you will create all jOOQ objects. It serves as a static factory for table expressions, column expressions (or "fields"), conditional expressions and many other QueryParts.

The static query DSL API

With jOOQ 2.0, static factory methods have been introduced in order to make client code look more like SQL. Ideally, when working with jOOQ, you will simply static import all methods from the DSL class:

```
import static org.jooq.impl.DSL.*;
```

Note, that when working with Eclipse, you could also add the DSL to your favourites. This will allow to access functions even more fluently:

```
concat(trim(FIRST_NAME), trim(LAST_NAME));

// ... which is in fact the same as:
DSL.concat(DSL.trim(FIRST_NAME), DSL.trim(LAST_NAME));
```

© 2009 - 2025 by Data Geekery™ GmbH.

3.2. The DSLContext API

DSLContext references a org.jooq.Configuration, an object that configures jOOQ's behaviour when executing queries (see SQL execution for more details). Unlike the static DSL, the DSLContext allow for creating SQL statements that are already "configured" and ready for execution.

Fluent creation of a DSLContext object

The DSLContext object can be created fluently from the DSL type:

```
// Create it from a pre-existing configuration
DSLContext create = DSL.using(configuration);

// Create it from ad-hoc arguments
DSLContext create = DSL.using(connection, dialect);
```

If you do not have a reference to a pre-existing Configuration object (e.g. created from org.jooq.impl.DefaultConfiguration), the various overloaded DSL.using() methods will create one for you.

Contents of a Configuration object

A Configuration can be supplied with these objects:

- org.jooq.SQLDialect : The dialect of your database. This may be any of the currently supported database types (see SQL Dialect for more details)
- org.jooq.conf.Settings : An optional runtime configuration (see Custom Settings for more details)
- org.jooq.ExecuteListenerProvider : An optional reference to a provider class that can provide execute listeners to jOOQ (see ExecuteListeners for more details)
- org.jooq.RecordListenerProvider : An optional reference to a provider class that can provide record listeners to jOOQ (see CRUD SPI: RecordListener for more details)
- org.jooq.RecordMapperProvider : An optional reference to a provider class that can provide record mappers to jOOQ (see RecordMapperProvider for more details)
- Any of these:
    * java.sql.Connection : An optional JDBC Connection that will be re-used for the whole lifecycle of your Configuration (see Connection vs. DataSource for more details). For simplicity, this is the use-case referenced from this manual, most of the time.
    * java.sql.DataSource : An optional JDBC DataSource that will be re-used for the whole lifecycle of your Configuration. If you prefer using DataSources over Connections, jOOQ will internally fetch new Connections from your DataSource, conveniently closing them again after query execution. This is particularly useful in Java EE or Spring contexts (see Connection vs. DataSource for more details)
    * org.jooq.ConnectionProvider : A custom abstraction that is used by jOOQ to "acquire" and "release" connections. jOOQ will internally "acquire" new Connections from your ConnectionProvider, conveniently "releasing" them again after query execution. (see Connection vs. DataSource for more details)

© 2009 - 2025 by Data Geekery™ GmbH.

Usage of DSLContext

Wrapping a Configuration object, a DSLContext can construct statements, for later execution. An example is given here:

```
// The DSLContext is "configured" with a Connection and a SQLDialect
DSLContext create = DSL.using(connection, dialect);

// This select statement contains an internal reference to the DSLContext's Configuration:
Select<?> select = create.selectOne();

// Using the internally referenced Configuration, the select statement can now be executed:
Result<?> result = select.fetch();
```

Note that you do not need to keep a reference to a DSLContext. You may as well inline your local variable, and fluently execute a SQL statement as such:

```
// Execute a statement from a single execution chain:
Result<?> result =
DSL.using(connection, dialect)
   .select()
   .from(BOOK)
   .where(BOOK.TITLE.like("Animal%"))
   .fetch();
```

3.2.1. SQL Dialect

While jOOQ tries to represent the SQL standard as much as possible, many features are vendor-specific to a given database and to its "SQL dialect". jOOQ models this using the org.jooq.SQLDialect enum type.

The SQL dialect is one of the main attributes of a Configuration. Queries created from DSLContexts will assume dialect-specific behaviour when rendering SQL and binding bind values.

Some parts of the jOOQ API are officially supported only by a given subset of the supported SQL dialects. For instance, the Oracle CONNECT BY clause, which is supported by the Oracle and CUBRID databases, is annotated with a org.jooq.Support annotation, as such:

```
/**
 * Add an Oracle-specific <code>CONNECT BY</code> clause to the query
 */
@Support({ SQLDialect.CUBRID, SQLDialect.ORACLE })
SelectConnectByConditionStep<R> connectBy(Condition condition);
```

jOOQ API methods which are not annotated with the org.jooq.Support annotation, or which are annotated with the Support annotation, but without any SQL dialects can be safely used in all SQL dialects. An example for this is the SELECT statement factory method:

```
/**
 * Create a new DSL select statement.
 */
@Support
SelectSelectStep<R> select(Field<?>... fields);
```

jOOQ's SQL clause emulation capabilities

The aforementioned Support annotation does not only designate, which databases natively support a feature. It also indicates that a feature is emulated by jOOQ for some databases lacking this feature. An

example of this is the DISTINCT predicate, a predicate syntax defined by SQL:1999 and implemented only by H2, HSQLDB, and Postgres:

```
A IS DISTINCT FROM B
```

Nevertheless, the IS DISTINCT FROM predicate is supported by jOOQ in all dialects, as its semantics can be expressed with an equivalent CASE expression. For more details, see the manual's section about the DISTINCT predicate.

jOOQ and the Oracle SQL dialect

Oracle SQL is much more expressive than many other SQL dialects. It features many unique keywords, clauses and functions that are out of scope for the SQL standard. Some examples for this are

- The CONNECT BY clause, for hierarchical queries
- The PIVOT keyword for creating PIVOT tables
- Packages, object-oriented user-defined types, member procedures as described in the section about stored procedures and functions
- Advanced analytical functions as described in the section about window functions

jOOQ has a historic affinity to Oracle's SQL extensions. If something is supported in Oracle SQL, it has a high probability of making it into the jOOQ API

3.2.2. SQL Dialect Family

In jOOQ 3.1, the notion of a SQLDialect.family() was introduced, in order to group several similar SQL dialects into a common family. An example for this is SQL Server, which is supported by jOOQ in various versions:

- SQL Server: The "version-less" SQL Server version. This always maps to the latest supported version of SQL Server
- SQL Server 2012: The SQL Server version 2012
- SQL Server 2008: The SQL Server version 2008

In the above list, SQLSERVER is both a dialect and a family of three dialects. This distinction is used internally by jOOQ to distinguish whether to use the OFFSET .. FETCH clause (SQL Server 2012), or whether to emulate it using ROW_NUMBER() OVER() (SQL Server 2008).

3.2.3. Connection vs. DataSource

Interact with JDBC Connections

While you can use jOOQ for SQL building only, you can also run queries against a JDBC java.sql.Connection. Internally, jOOQ creates java.sql.Statement or java.sql.PreparedStatement objects from such a Connection, in order to execute statements. The normal operation mode is to provide a

Configuration with a JDBC Connection, whose lifecycle you will control yourself. This means that jOOQ will not actively close connections, rollback or commit transactions.

Note, in this case, jOOQ will internally use a org.jooq.impl.DefaultConnectionProvider, which you can reference directly if you prefer that. The DefaultConnectionProvider exposes various transaction-control methods, such as commit(), rollback(), etc.

Interact with JDBC DataSources

If you're in a Java EE or Spring context, however, you may wish to use a javax.sql.DataSource instead. Connections obtained from such a DataSource will be closed after query execution by jOOQ. The semantics of such a close operation should be the returning of the connection into a connection pool, not the actual closing of the underlying connection. Typically, this makes sense in an environment using distributed JTA transactions.

Note, in this case, jOOQ will internally use a org.jooq.impl.DataSourceConnectionProvider, which you can reference directly if you prefer that.

Inject custom behaviour

If your specific environment works differently from any of the above approaches, you can inject your own custom implementation of a ConnectionProvider into jOOQ. This is the API contract you have to fulfil:

```
public interface ConnectionProvider {

    // Provide jOOQ with a connection
    Connection acquire() throws DataAccessException;

    // Get a connection back from jOOQ
    void release(Connection connection) throws DataAccessException;
}
```

3.2.4. Custom data

In advanced use cases of integrating your application with jOOQ, you may want to put custom data into your Configuration, which you can then access from your...

- Custom ExecuteListeners
- Custom QueryParts

Here is an example of how to use the custom data API. Let's assume that you have written an ExecuteListener, that prevents INSERT statements, when a given flag is set to true:

```
public class NoInsertListener extends DefaultExecuteListener {

    @Override
    public void start(ExecuteContext ctx) {

        // This listener is active only, when your custom flag is set to true
        if (Boolean.TRUE.equals(ctx.configuration().data("com.example.my-namespace.no-inserts"))) {

            // If active, fail this execution, if an INSERT statement is being executed
            if (ctx.query() instanceof Insert) {
                throw new DataAccessException("No INSERT statements allowed");
            }
        }
    }
}
```

See the manual's section about ExecuteListeners to learn more about how to implement an ExecuteListener.

Now, the above listener can be added to your Configuration, but you will also need to pass the flag to the Configuration, in order for the listener to work:

```
// Create your Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);

// Set a new execute listener provider onto the configuration:
configuration.set(new DefaultExecuteListenerProvider(new NoInsertListener()));

// Use any String literal to identify your custom data
configuration.data("com.example.my-namespace.no-inserts", true);

// Try to execute an INSERT statement
try {
    DSL.using(configuration)
       .insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
       .values(1, "Orwell")
       .execute();

    // You shouldn't get here
    Assert.fail();
}

// Your NoInsertListener should be throwing this exception here:
catch (DataAccessException expected) {
    Assert.assertEquals("No INSERT statements allowed", expected.getMessage());
}
```

Using the data() methods, you can store and retrieve custom data in your Configurations.

3.2.5. Custom ExecuteListeners

ExecuteListeners are a useful tool to...

- implement custom logging
- apply triggers written in Java
- collect query execution statistics

ExecuteListeners are hooked into your Configuration by returning them from an org.jooq.ExecuteListenerProvider:

```
// Create your Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);

// Hook your listener providers into the configuration:
configuration.set(
    new DefaultExecuteListenerProvider(new MyFirstListener()),
    new DefaultExecuteListenerProvider(new PerformanceLoggingListener()),
    new DefaultExecuteListenerProvider(new NoInsertListener())
);
```

See the manual's section about ExecuteListeners to see examples of such listener implementations.

3.2.6. Custom Unwrappers

JDBC knows the java.sql.Wrapper API, which is implemented by all JDBC types in order to be able to "unwrap" a native driver implementation for any given type. For example:

```
// This may be some proxy from a connection pool
Connection c = getConnection();

// Sometimes, we want the native driver connection instance
OracleConnection oc = c.unwrap(OracleConnection.class);
Array array = oc.createARRAY("ARRAY_TYPE", new Object[] { "a", "b" });
```

jOOQ internally makes similar calls occasionally. For this, it needs to unwrap the native java.sql.Connection or java.sql.PreparedStatement instance. Unfortunately, not all third party libraries correctly implement the Wrapper API contract, so this unwrapping might not work. The org.jooq.Unwrapper SPI is designed to allow for custom implementations to be injected into jOOQ configurations:

```
// Your jOOQ configuration
Configuration c1 = getConfiguration();
Configuration c2 = c.derive(new Unwrapper() {
    @Override
    public <T> T unwrap(Wrapper wrapper, Class<T> iface) {
        try {
            if (wrapper instanceof Connection)
                // ...
            else if (wrapper instanceof Statement)
                // ...
            else
                wrapper.unwrap(iface);
        }
        catch (SQLException e) {
            // ...
        }
    }
});

// Work with the derived configuration, where needed
DSL.using(c2).fetch("...");
```

3.2.7. Custom Settings

The jOOQ Configuration allows for some optional configuration elements to be used by advanced users. The org.jooq.conf.Settings class is a JAXB-annotated type, that can be provided to a Configuration in several ways:

- In the DSLContext constructor (DSL.using()). This will override default settings below
- in the org.jooq.impl.DefaultConfiguration constructor. This will override default settings below
- From a location specified by a JVM parameter: -Dorg.jooq.settings
- From the classpath at /jooq-settings.xml
- From the settings defaults, as specified in https://www.jooq.org/xsd/jooq-runtime-3.12.4.xsd

The most specific settings for a given context will apply.

If you wish to configure your settings through XML, but explicitly load them for a given Configuration, you can do so as well, using JAXB:

```
Settings settings = JAXB.unmarshal(new File("/path/to/settings.xml"), Settings.class);
```

Example

For example, if you want to indicate to jOOQ, that it should inline all bind variables, and execute static java.sql.Statement instead of binding its variables to java.sql.PreparedStatement, you can do so by creating the following DSLContext:

```
Settings settings = new Settings();
settings.setStatementType(StatementType.STATIC_STATEMENT);
DSLContext create = DSL.using(connection, dialect, settings);
```

More details

Please refer to the jOOQ runtime configuration XSD for more details:
https://www.jooq.org/xsd/jooq-runtime-3.12.4.xsd

3.2.7.1. Auto-attach Records

By default, all records fetched through jOOQ are "attached" to the configuration that created them. This allows for features like updatable records as can be seen here:

```
AuthorRecord author =
DSL.using(configuration) // This configuration will be attached to any record produced by the below query.
   .selectFrom(AUTHOR)
   .where(AUTHOR.ID.eq(1))
   .fetchOne();

author.setLastName("Smith");
author.store(); // This store call operates on the "attached" configuration.
```

In some cases (e.g. when serialising records), it may be desirable not to attach the Configuration that created a record to the record. This can be achieved with the attachRecords setting:

Example configuration

```
Settings settings = new Settings()
    .withAttachRecords(false); // Defaults to true
```

3.2.7.2. Backslash Escaping

Some databases (mainly MySQL and MariaDB) unfortunately chose to go an alternative, non-SQL-standard route when escaping string literals. Here's an example of how to escape a string containing apostrophes in different dialects:

```
SELECT 'I''m sure this is OK' AS val             -- Standard SQL escaping of apostrophe by doubling it.
SELECT 'I\'m certain this causes trouble' AS val -- Vendor-specific escaping of apostrophe by using a backslash.
```

As most databases don't support backslash escaping (and MySQL also allows for turning it off!), jOOQ by default also doesn't support it when inlining bind variables. However, this can lead to SQL injection vulnerabilities and syntax errors when not dealing with it carefully!

This feature is turned on by default and for historic reasons for MySQL and MariaDB.

- DEFAULT (the - surprise! - default): Turns the feature ON for MySQL and MariaDB and OFF for all other dialects
- ON: Turn the feature on.
- OFF: Turn the feature off.

Example configuration

```
Settings settings = new Settings()
    .withBackslashEscaping(BackslashEscaping.OFF); // Default to DEFAULT
```

3.2.7.3. Execute Logging

The executeLogging setting turns off the default logging implemented through org.jooq.tools.LoggerListener

Example configuration

```
Settings settings = new Settings()
    .withExecuteLogging(false); // Defaults to true
```

3.2.7.4. Fetch Warnings

Apart from JDBC exceptions, there is also the possibility to handle java.sql.SQLWarning, which are made available to jOOQ users through the java.sql.ExecuteListener SPI and the log. Users who do not wish to get these notifications (e.g. for performance reasons), may turn off fetching of warnings through the fetchWarnings setting:

Example configuration

```
Settings settings = new Settings()
    .withFetchWarnings(false); // Defaults to true
```

3.2.7.5. Identifier style

By default, jOOQ will always generate quoted names for all identifiers (even if this manual omits this for readability).

For instance:

```
SELECT "TABLE"."COLUMN" FROM "TABLE" -- SQL standard style
SELECT `TABLE`.`COLUMN` FROM `TABLE` -- MySQL style
SELECT [TABLE].[COLUMN] FROM [TABLE] -- SQL Server style
```

Quoting has the following effect on identifiers in most (but not all) databases:

- It allows for using reserved names as object names, e.g. a table called "FROM" is usually possible only when quoted.
- It allows for using special characters in object names, e.g. a column called "FIRST NAME" can be achieved only with quoting.
- It turns what are mostly case-insensitive identifiers into case-sensitive ones, e.g. "name" and "NAME" are different identifiers, whereas name and NAME are not. Please consider your database manual to learn what the proper default case and default case sensitivity is.

The renderQuotedNames and renderNameCase settings allow for overriding the name of all identifiers in jOOQ to a consistent style. The two flags are independent of one another. Possible options are:

RenderQuotedNames

- ALWAYS: This will quote all identifiers.
- EXPLICIT_DEFAULT_QUOTED: This will quote all identifiers, which are not explicitly unquoted using DSL.unquotedName().
- EXPLICIT_DEFAULT_UNQUOTED: This will not quote any identifiers, unless they are explicitly quoted using DSL.quotedName().
- NEVER: This will not quote any identifiers.

RenderNameCase

- AS_IS: This will generate all names in their proper case.
- LOWER: This will transform all names to lower case.
- LOWER_IF_UNQUOTED: This will transform all names to lower case if the name is unquoted.
- UPPER: This will transform all names to upper case.
- UPPER_IF_UNQUOTED: This will transform all names to upper case if the name is unquoted.

(!) In some database products, quoted identifiers only enable special characters. Other database products use quotes also to make identifiers case sensitive. Keep this in mind when working with the above flags.

Example configuration

```
Settings settings = new Settings()
    .withRenderQuotedNames(RenderQuotedNames.EXPLICIT_DEFAULT_UNQUOTED) // Defaults to EXPLICIT_DEFAULT_QUOTED
    .withRenderNameCase(RenderNameCase.LOWER_IF_UNQUOTED);              // Defaults to AS_IS
```

The behaviour of this setting is influenced by the renderLocale setting.

3.2.7.6. Inline Threshold

Previous sections showed how the SQL generation of bind values can be controlled, e.g. by forcing them to be inlined, or by running a static JDBC statement.

Sometimes, inlining needs to be enforced dynamically, depending on the query content. This is the case when there are a great number of bind variables. Known vendor-specific limits are:

- Access : 768
- Ingres : 1024
- Oracle : 32767
- PostgreSQL : 32767
- SQLite : 999
- SQL Server : 2100
- Sybase ASE : 2000
- Teradata : 2536

By default, jOOQ will automatically inline all bind variables in any SQL statement, once these thresholds have been reached. However, it is possible to override this default and provide a setting to re-define a global threshold for all dialects.

Example configuration

```
Settings settings = new Settings()
    .withInlineThreshold(100); // Defaults to 0, which means the default thresholds are applied
```

3.2.7.7. IN-list Padding

Databases that feature a cursor cache / statement cache (e.g. Oracle, SQL Server, DB2, etc.) are highly optimised for prepared statement re-use. When a client sends a prepared statement to the server, the server will go to the cache and look up whether there already exists a previously calculated execution plan for the statement (i.e. the SQL string). This is called a "soft-parse" (in Oracle). If not, the execution plan is calculated on the fly. This is called a "hard-parse" (in Oracle).

Preventing hard-parses is extremely important in high throughput OLTP systems where queries are usually not very complex but are run millions of times in a short amount of time. Using bind variables, this is usually not a problem, with the exception of the IN predicate, which generates different SQL strings even when using bind variables:

```
-- All of these are different SQL statements:
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
```

This problem may not be obvious to Java / jOOQ developers, as they are always produced from the same jOOQ statement:

```
// All of these are the same jOOQ statement
DSL.using(configuration)
   .select()
   .from(AUTHOR)
   .where(AUTHOR.ID.in(collection))
   .fetch();
```

Depending on the possible sizes of the collection, it may be worth exploring using arrays or temporary tables as a workaround, or to reuse the original query that produced the set of IDs in the first place (through a semi-join). But sometimes, this is not possible. In this case, users can opt in to a third workaround: enabling the inListPadding setting. If enabled, jOOQ will "pad" the IN list to a length that is

a power of two (configurable with Settings.inListPadBase). So, the original queries would look like this instead:

```
-- Original
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?)
```

```
-- Padded
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
```

This technique will drastically reduce the number of possible SQL strings without impairing too much the usual cases where the IN list is small. When padding, the last bind variable will simply be repeated many times.

Usually, there is a better way - use this as a last resort!

Example configuration

```
Settings settings = new Settings()
    .withInListPadding(true) // Default to false
    .withInListPadBase(4);   // Default to 2
```

3.2.7.8. JDBC Flags

JDBC statements feature a couple of flags that influence the execution of such a statement. Each of these flags can be configured through jOOQ's org.jooq.Query and org.jooq.ResultQuery on a statement-per-statement basis, but there's also the possibility to centrally specify a value for these flags. These are the three flags:

- queryTimeout: The JDBC statement timeout in seconds. Corresponds to Query.queryTimeout() or Statement.setQueryTimeout()
- maxRows: The maximum number of rows returned by the JDBC statement. Corresponds to ResultQuery.maxRows() or Statement.setMaxRows()
- fetchSize: The number of rows to be buffered by the JDBC ResultSet. Corresponds to ResultQuery.fetchSize() or Statement.setFetchSize()

All of these flags are JDBC-only features with no direct effect on jOOQ. jOOQ only passes them through to the underlying statement.

Example configuration

```
Settings settings = new Settings()
    .withQueryTimeout(5)
    .withQueryPoolable(DEFAULT)
    .withMaxRows(1000)
    .withFetchSize(20);
```

3.2.7.9. Keyword style

In all SQL dialects, keywords are case insensitive, and this is also the default in jOOQ, which mostly generates lower-case keywords. Users may wish to adapt this and they have these options for the renderKeywordCase setting:

- AS_IS (the default): Generate keywords as they are defined in the codebase (mostly lower case).
- LOWER: Generate keywords in lower case.
- UPPER: Generate keywords in upper case.
- PASCAL: Generate keywords in pascal case.

Example configuration

```
Settings settings = new Settings()
    .withRenderKeywordCase(RenderKeywordCase.UPPER); // Defaults to AS_IS
```

3.2.7.10. Listener Invocation Order

jOOQ offers a variety of SPIs in the Configuration object. Some of those SPIs are event listeners, that can listen to "start" and "end" events, such as for example the ExecuteListener that listens to the query execution lifecycle.

When registering multiple listeners of a type, the invocation order may be relevant as custom listeners might communicate with each other. In such a case, the following settings allow for overriding the invocation order of "start" and "end" events for each type of listener:

Example configuration

```
Settings settings = new Settings()
    .withTransactionListenerStartInvocationOrder(DEFAULT) // Defaults to DEFAULT
    .withTransactionListenerEndInvocationOrder(REVERSE)   // Defaults to DEFAULT
    .withVisitListenerStartInvocationOrder(DEFAULT)       // Defaults to DEFAULT
    .withVisitListenerEndInvocationOrder(REVERSE)         // Defaults to DEFAULT
    .withRecordListenerStartInvocationOrder(DEFAULT)      // Defaults to DEFAULT
    .withRecordListenerEndInvocationOrder(REVERSE)        // Defaults to DEFAULT
    .withExecuteListenerStartInvocationOrder(DEFAULT)     // Defaults to DEFAULT
    .withExecuteListenerEndInvocationOrder(REVERSE);      // Defaults to DEFAULT
```

3.2.7.11. Locales

When doing locale sensitive operations, such as upper casing or lower casing a name (see Name styles), then it may be important in some areas to be able to specify the java.util.Locale for the operation.

Example configuration

```
// All of these default to Locale.getDefault(), if not specified explicitly
Settings settings = new Settings()
    .withLocale(Locale.forLanguageTag("de"))             // The default locale if no more specific locales are specified
    .withRenderLocale(Locale.forLanguageTag("de"))       // The locale used when rendering SQL
    .withParseLocale(Locale.forLanguageTag("de"))        // The locale used when parsing SQL
    .withInterpreterLocale(Locale.forLanguageTag("de")); // The locale used when interpreting SQL
```

3.2.7.12. Map JPA Annotations

The org.jooq.impl.DefaultRecordMapper supports basic JPA mapping (mostly @Table and @Column annotations). Looking up these annotations costs a slight extra overhead (mostly taken care of through reflection caching). It can be turned off using the mapJPAAnnotations setting:

Example configuration

```
Settings settings = new Settings()
    .withMapJPAAnnotations(false); // Defaults to true
```

3.2.7.13. Object qualification

By default, jOOQ fully qualifies all objects with their catalog and schema names, if such qualification is made available by the code generator.

For instance, the following SQL statement containing full qualification may be produced by jOOQ code with seemingly no qualification:

```
-- Full qualification on columns and tables
SELECT catalog.schema.table.column
FROM catalog.schema.table
```

```
DSL.using(configuration)
   .select(TABLE.COLUMN) // Column only qualified with table
   .from(TABLE)          // No qualification on table
```

While the jOOQ code is also implicitly fully qualified (see implied imports), it may not be desireable to use fully qualified object names in SQL. The renderCatalog and renderSchema settings are used for this.

Example configuration

```
new Settings()
  .withRenderCatalog(false)  // Defaults to true
  .withRenderSchema(false);  // Defaults to true
```

More sophisticated multitenancy approaches are available through the render mapping feature.

3.2.7.14. Optimistic Locking

There are two settings governing the behaviour of the jOOQ optimistic locking feature:

- updateRecordVersion: Whether UpdatableRecord instances should modify the record version prior to storing the record. This feature is independent of, but related to optimistic locking.
- updateRecordTimestamp: Whether UpdatableRecord instances should modify the record timestamp prior to storing the record. This feature is independent of, but related to optimistic locking.
- executeWithOptimisticLocking: This allows for turning off the feature entirely.
- executeWithOptimisticLockingExcludeUnversioned: This allows for turning off the feature for updatable records who are not explicitly versioned.

© 2009 - 2025 by Data Geekery™ GmbH.

Example configuration

```
Settings settings = new Settings()
    .withUpdateRecordVersion(true)                          // Defaults to true
    .withUpdateRecordTimestamp(true)                        // Defaults to true
    .withExecuteWithOptimisticLocking(true)                 // Defaults to false
    .withExecuteWithOptimisticLockingExcludeUnversioned(false); // Defaults to false
```

For more details, please refer to the manual's section about the optimistic locking feature.

3.2.7.15. Parameter name prefix

When choosing a ParameterType.NAMED to produce named parameters, the default is to use a colon as a prefix to the parameter name.

For example:

```
-- NAMED
SELECT FIRST_NAME || :1 FROM AUTHOR WHERE ID = :x
```

Depending on how the named parameters are interpreted, this default is not optimal. A better character might be the $ sign, e.g. in PostgreSQL or R2DBC. For this, the renderNamedParamPrefix setting can be used:

Example configuration

```
Settings settings = new Settings()
    .withRenderNamedParamPrefix("$"); // Defaults to ":"
```

3.2.7.16. Parameter types

Bind values or bind parameters come in different flavours in different SQL databases. JDBC standardises on their syntax by allowing only ? (question mark) characters as placeholders for bind variables. Thus, jOOQ, by default, generates ? placeholders for JDBC consumptions. Users who wish to use jOOQ with a different backend than JDBC can specify that all jOOQ bind values, including indexed parameters and named parameters generate alternative strings, other than ?.

These are the current options:

- INDEXED (the default): Generates indexed parameter placeholders using ?.
- NAMED: Generates named parameter placeholders, such as :param for parameters that are named explicitly or :1 for unnamed, indexed parameters.
- NAMED_OR_INLINED: Generates named parameter placeholders for parameters that are named explicitly and inlines all unnamed parameters.
- INLINED: Inlines all parameters.

An example:

```
-- INDEXED
SELECT FIRST_NAME || ? FROM AUTHOR WHERE ID = ?
-- NAMED
SELECT FIRST_NAME || :1 FROM AUTHOR WHERE ID = :x
-- NAMED_OR_INLINED
SELECT FIRST_NAME || 'x' FROM AUTHOR WHERE ID = :x
-- INLINED
SELECT FIRST_NAME || 'x' FROM AUTHOR WHERE ID = 42
```

```
Param<String> x = val("x");
Param<Integer> i = param("x", 42);

DSL.using(configuration)
   .select(FIRST_NAME.concat(x))
   .from(AUTHOR)
   .where(ID.eq(i))
   .fetch();
```

Example configuration

```
Settings settings = new Settings()
    .withParamType(ParamType.NAMED); // Defaults to INDEXED
```

The following setting statementType may override this setting.

3.2.7.17. Parser Configuration

The SQL Parser API ships with a variety of settings that govern its behaviour. These settings include:

- parseDialect: The parser input dialect. This dialect is used to decide what vendor specific grammar should be applied in case of ambiguities that cannot be resolved from the context.
- parseIgnoreComments: Using this flag, the parser can ignore certain sections that would otherwise be executed by RDBMS. Everything between an parseIgnoreCommentStart and the parseIgnoreCommentStop token will be ignored.
- parseIgnoreCommentStart: The token that delimits the beginning of a section to be ignored by jOOQ. Ideally, this token is placed inside of a SQL comment.
- parseIgnoreCommentStop: The token that delimits the end of a section to be ignored by jOOQ. Ideally, this token is placed inside of a SQL comment.
- parseSearchPath: The search path to look up unqualified identifiers to be used when using parseWithMetaLookups. Most dialects support a single schema on their search path (the CURRENT_SCHEMA). PostgreSQL supports a 'search_path', which allows for listing multiple schemata to use to look up unqualified tables, procedures, etc. in.
- parseUnsupportedSyntax: The parser can parse some syntax that jOOQ does not support. By default, such syntax is ignored. Use this flag if you want to fail in such cases.
- parseUnknownFunctions: The parser only parses "known" (to jOOQ) built in functions, and fails otherwise. This flag allows for parsing any built in function using a standard func_name(arg1, arg2, ...) syntax.
- parseWithMetaLookups: Whether org.jooq.Meta should be used to look up meta information such as schemas, tables, columns, column types, etc.

An example of using the parseIgnoreComments feature:

```
-- What you execute
/* [jooq ignore start] */
CREATE SCHEMA s1;
SET SCHEMA s1;
/* [jooq ignore stop] */

/* [jooq ignore start] */ -- /* [jooq ignore stop] */ CREATE
 SCHEMA s2;
/* [jooq ignore start] */ -- /* [jooq ignore stop] */ SET SCHEMA
 s2;

CREATE TABLE t (i INTEGER);
```

```
-- What the jOOQ parser sees
/*


                        */

/*                                                      */ CREATE
 SCHEMA s2;
/*                                                      */ SET SCHEMA
 s2;

CREATE TABLE t (i INTEGER);
```

Example configuration

```
Settings settings = new Settings()
    .withParseDialect(SQLSERVER)                        // Defaults to DEFAULT
    .withParseWithMetaLookups(THROW_ON_FAILURE)         // Defaults to OFF
    .withParseSearchPath(
        new ParseSearchSchemata().withSchema("PUBLIC"),
        new ParseSearchSchemata().withSchema("TEST"))
    .withParseUnsupportedSyntax(FAIL)                   // Defaults to IGNORE
    .withParseUnknownFunctions(IGNORE)                  // Defaults to FAIL
    .withParseIgnoreComments(true)                      // Defaults to false
    .withParseIgnoreCommentStart("<ignore>")            // Defaults to "[jooq ignore start]"
    .withParseIgnoreCommentStop("</ignore>")            // Defaults to "[jooq ignore stop]"
```

3.2.7.18. Reflection caching

All operations of the DefaultRecordMapper are cached in the Configuration by default for improved mapping and reflection speed. Users who prefer to override this cache, or work with their own custom record mapper provider may wish to turn off the out-of-the-box caching feature.

Example configuration

```
Settings settings = new Settings()
    .withReflectionCaching(false); // Defaults to true
```

3.2.7.19. Return all columns on store

When using the updatable records feature, jOOQ always fetches the generated identity value, if such a value is availableand if the return identity on store feature is enabled (it is, by default).

The identity value is not the only value that is generated by default. Specifically, there may be triggers that are used for auditing or other reasons, which generate LAST_UPDATE or LAST_UPDATE_BY values in a record. Users who wish to also automatically fetch these values after all store(), insert(), or update() calls may do so by specifying the returnAllOnUpdatableRecord setting. This setting depends on the availability of INSERT .. RETURNING, UPDATE .. RETURNING, and DELETE .. RETURNING statements, which are not available from all databases, in case of which a refresh() call may be issued, creating a separate round trip to the server.

Example configuration

```
Settings settings = new Settings()
    .withReturnAllOnUpdatableRecord(true); // Defaults to false
```

3.2.7.20. Return Identity Value On Store

When using the updatable records feature, jOOQ by default fetches the generated identity value. In some situations, it is desirable for this feature to be turned off using the following flag:

Example configuration

```
Settings settings = new Settings()
    .withReturnIdentityOnUpdatableRecord(false); // Defaults to true
```

3.2.7.21. Runtime schema and table mapping

Most SQL object types are qualified with a org.jooq.Catalog and org.jooq.Schema. In multi-tenant application, users may want to map these identifier namespaces to something other than the default.

Mapping your DEV schema to a productive environment

You may wish to design your database in a way that you have several instances of your schema. This is useful when you want to cleanly separate data belonging to several customers / organisation units / branches / users and put each of those entities' data in a separate database or schema.

In our AUTHOR example this would mean that you provide a book reference database to several companies, such as My Book World and Books R Us. In that case, you'll probably have a schema setup like this:

- DEV: Your development schema. This will be the schema that you base code generation upon, with jOOQ
- MY_BOOK_WORLD: The schema instance for My Book World
- BOOKS_R_US: The schema instance for Books R Us

Mapping DEV to MY_BOOK_WORLD with jOOQ

When a user from My Book World logs in, you want them to access the MY_BOOK_WORLD schema using classes generated from DEV. This can be achieved with the org.jooq.conf.RenderMapping class, that you can equip your Configuration's settings with. Take the following example:

Example configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInput("DEV")
                          .withOutput("MY_BOOK_WORLD"),
        new MappedSchema().withInput("LOG")
                          .withOutput("MY_BOOK_WORLD_LOG")));
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.12.4.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <input>DEV</input>
        <output>MY_BOOK_WORLD</output>
      </schema>
      <schema>
        <input>LOG</input>
        <output>MY_BOOK_WORLD_LOG</output>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

© 2009 - 2025 by Data Geekery™ GmbH.

The query executed with a Configuration equipped with the above mapping will in fact produce this SQL statement:

```
SELECT *
FROM MY_BOOK_WORLD.AUTHOR
```

```
DSL.using(connection, dialect, settings)
   .selectFrom(DEV.AUTHOR)
```

This works because AUTHOR was generated from the DEV schema, which is mapped to the MY_BOOK_WORLD schema by the above settings.

Mapping of tables

Not only schemata can be mapped, but also tables. If you are not the owner of the database your application connects to, you might need to install your schema with some sort of prefix to every table. In our examples, this might mean that you will have to map DEV.AUTHOR to something MY_BOOK_WORLD.MY_APP__AUTHOR, where MY_APP__ is a prefix applied to all of your tables. This can be achieved by creating the following mapping:

Example configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInput("DEV")
                          .withTables(
         new MappedTable().withInput("AUTHOR")
                          .withOutput("MY_APP__AUTHOR"))));
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.12.4.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <input>DEV</input>
        <tables>
          <table>
            <input>AUTHOR</input>
            <output>MY_APP__AUTHOR</output>
          </table>
        </tables>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

The query executed with a Configuration equipped with the above mapping will in fact produce this SQL statement:

```
SELECT * FROM DEV.MY_APP__AUTHOR
```

Table mapping and schema mapping can be applied independently, by specifying several MappedSchema entries in the above configuration. jOOQ will process them in order of appearance and map at first match. Note that you can always omit a MappedSchema's output value, in case of which, only the table mapping is applied.

Using regular expressions

All of the above examples were using 1:1 constant name mappings where the input and output schema or table names are fixed by the configuration. With jOOQ 3.8, regular expression can be used as well for mapping, for example:

Example configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInputExpression(Pattern.compile("DEV_(.*)"))
                          .withOutput("PROD_$1")
                          .withTables(
         new MappedTable().withInputExpression(Pattern.compile("DEV_(.*)"))
                          .withOutput("PROD_$1"))));
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.12.4.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <inputExpression>DEV_(.*)</inputExpression>
        <output>PROD_$1</output>
        <tables>
          <table>
            <inputExpression>DEV_(.*)</inputExpression>
            <output>PROD_$1</output>
          </table>
        </tables>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

The only difference to the constant version is that the input field is replaced by the inputExpression field of type java.util.regex.Pattern, in case of which the meaning of the output field is a pattern replacement, not a constant replacement.

Hard-wiring mappings at code-generation time

Note that the manual's section about code generation schema mapping explains how you can hard-wire your catalog, schema and table mappings at code generation time.

Limitations

Mapped objects need to be known to the jOOQ org.jooq.RenderContext, which means that for example plain SQL templates and their contents cannot be mapped. See also features requiring code generation for more details.

3.2.7.22. Scalar subqueries for stored functions

This setting is useful mostly for the Oracle database, which implements a feature called scalar subquery caching, which is a good tool to avoid the expensive PL/SQL-to-SQL context switch when predicates make use of stored function calls.

With this setting in place, all stored function calls embedded in SQL statements will be wrapped in a scalar subquery:

```
SELECT
  (SELECT my_package.format(LANGUAGE_ID) FROM dual)
FROM BOOK
```

```
DSL.using(configuration)
   .select(MyPackage.format(BOOK.LANGUAGE_ID))
   .from(BOOK)
```

If our table contains thousands of books, but only a dozen of LANGUAGE_ID values, then with scalar subquery caching, we can avoid most of the function calls and cache the result per LANGUAGE_ID.

Example configuration

```
Settings settings = new Settings()
    .withRenderScalarSubqueriesForStoredFunctions(true);
```

3.2.7.23. Statement Type

JDBC knows two types of statements:

- java.sql.PreparedStatement: This allows for sending bind variables to the server. jOOQ uses prepared statements by default.
- java.sql.Statement: Also "static statement". These do not support bind variables and may be useful for one-shot commands like DDL statements.

The statementType setting allows for overriding the default of using prepared statements internally. There are two possible options for this setting:

- PREPARED_STATEMENT (the default): Use prepared statements.
- STATIC_STATEMENT: Use static statements. This enforces the paramType == INLINED. See parameter types

Example configuration

```
Settings settings = new Settings()
    .withStatementType(StatementType.STATIC_STATEMENT); // Defaults to PREPARED_STATEMENT
```

3.2.7.24. Updatable Primary Keys

In most database design guidelines, primary key values are expected to never change - an assumption that is essential to a normalised database.

As always, there are exceptions to these rules, and users may wish to allow for updatable primary key values in the updatable records feature (note: any value can always be updated through ordinary update statements). An example:

```
AuthorRecord author =
DSL.using(configuration) // This configuration will be attached to any record produced by the below query.
   .selectFrom(AUTHOR)
   .where(AUTHOR.ID.eq(1))
   .fetchOne();

author.setId(2);
author.store(); // The behaviour of this store call is governed by the updatablePrimaryKeys flag
```

The above store call depends on the value of the updatablePrimaryKeys flag:

- false (the default): Since immutability of primary keys is assumed, the store call will create a new record (a copy) with the new primary key value.
- true: Since mutablity of primary keys is allowed, the store call will change the primary key value from 1 to 2.

Example configuration

```
Settings settings = new Settings()
    .withUpdatablePrimaryKeys(true); // Defaults to false
```

3.2.8. Thread safety

org.jooq.Configuration, and by consequence org.jooq.DSLContext, make no thread safety guarantees, but by carefully observing a few rules, they can be shared in a thread safe way. We encourage sharing Configuration instances, because they contain caches for work not worth repeating, such as reflection field and method lookups for org.jooq.impl.DefaultRecordMapper. If you're using Spring or CDI for dependency injection, you will want to be able to inject a DSLContext instance everywhere you use it.

The following needs to be considered when attempting to share Configuration and DSLContext among threads:

- Configuration is mutable for historic reasons. Calls to various Configuration.set() methods must be avoided after initialisation, should a Configuration (and by consequence DSLContext) instance be shared among threads. If you wish to modify some elements of a Configuration for single use, use the Configuration.derive() methods instead, which create a copy.
- Configuration components, such as org.jooq.conf.Settings are mutable as well. The same rules for modification apply here.
- Configuration allows for supplying user-defined SPI implementations (see above for examples). All of these must be thread safe as well, for their wrapping Configuration to be thread safe. If you are using a org.jooq.impl.DataSourceConnectionProvider, for instance, you must make sure that your javax.sql.DataSource is thread safe as well. This is usually the case when you use a third party connection pool.

As can be seen above, Configuration was designed to work in a thread safe way, despite it not making any such guarantee.

3.3. SQL Statements (DML)

jOOQ currently supports 5 types of SQL statements. All of these statements are constructed from a DSLContext instance with an optional JDBC Connection or DataSource. If supplied with a Connection or DataSource, they can be executed. Depending on the query type, executed queries can return results.

3.3.1. jOOQ's DSL and model API

jOOQ ships with its own DSL (or Domain Specific Language) that emulates SQL in Java. This means, that you can write SQL statements almost as if Java natively supported it, just like .NET's C# does with LINQ to SQL.

Here is an example to illustrate what that means:

```
-- Select all books by authors born after 1920,
-- named "Paulo" from a catalogue:
SELECT *
  FROM author a
  JOIN book b ON a.id = b.author_id
 WHERE a.year_of_birth > 1920
   AND a.first_name = 'Paulo'
 ORDER BY b.title
```

```
Result<Record> result =
create.select()
      .from(AUTHOR.as("a"))
      .join(BOOK.as("b")).on(a.ID.eq(b.AUTHOR_ID))
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")))
      .orderBy(b.TITLE)
      .fetch();
```

We'll see how the aliasing works later in the section about aliased tables

jOOQ as an internal domain specific language in Java (a.k.a. the DSL API)

Many other frameworks have similar APIs with similar feature sets. Yet, what makes jOOQ special is its informal BNF notation modelling a unified SQL dialect suitable for many vendor-specific dialects, and implementing that BNF notation as a hierarchy of interfaces in Java. This concept is extremely powerful, when using jOOQ with IDE syntax auto completion. Not only can you code much faster, your SQL code will be compile-checked to a certain extent. An example of a DSL query equivalent to the previous one is given here:

```
DSLContext create = DSL.using(connection, dialect);
Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                         .fetch();
```

Unlike other, simpler frameworks that use "fluent APIs" or "method chaining", jOOQ's BNF-based interface hierarchy will not allow bad query syntax. The following will not compile, for instance:

```
DSLContext create = DSL.using(connection, dialect);
Result<?> result = create.select()
                         .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                      //  ^^^^ "join" is not possible here
                         .from(AUTHOR)
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK)
                         .fetch();
                      //  ^^^^^ "on" is missing here

Result<?> result = create.select(rowNumber())
                      //         ^^^^^^^^^ "over()" is missing here
                         .from(AUTHOR)
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .where(AUTHOR.ID.in(select(BOOK.TITLE).from(BOOK)))
                      //                     ^^^^^^^^^^^^^^^^^^
                      // AUTHOR.ID is of type Field<Integer> but subselect returns Record1<String>
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .where(AUTHOR.ID.in(select(BOOK.AUTHOR_ID, BOOK.ID).from(BOOK)))
                      //                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                      // AUTHOR.ID is of degree 1 but subselect returns Record2<Integer, Integer>
                         .fetch();
```

History of SQL building and incremental query building (a.k.a. the model API)

Historically, jOOQ started out as an object-oriented SQL builder library like any other. This meant that all queries and their syntactic components were modeled as so-called QueryParts, which delegate SQL rendering and variable binding to child components. This part of the API will be referred to as the model API (or non-DSL API), which is still maintained and used internally by jOOQ for incremental query building. An example of incremental query building is given here:

```
DSLContext create = DSL.using(connection, dialect);
SelectQuery<Record> query = create.selectQuery();
query.addFrom(AUTHOR);

// Join books only under certain circumstances
if (join) {
    query.addJoin(BOOK, BOOK.AUTHOR_ID.eq(AUTHOR.ID));
}

Result<?> result = query.fetch();
```

This query is equivalent to the one shown before using the DSL syntax. In fact, internally, the DSL API constructs precisely this SelectQuery object. Note, that you can always access the SelectQuery object to switch between DSL and model APIs:

```
DSLContext create = DSL.using(connection, dialect);
SelectFinalStep<?> select = create.select().from(AUTHOR);

// Add the JOIN clause on the internal QueryObject representation
SelectQuery<?> query = select.getQuery();
query.addJoin(BOOK, BOOK.AUTHOR_ID.eq(AUTHOR.ID));
```

Mutability

Note, that for historic reasons, the DSL API mixes mutable and immutable behaviour with respect to the internal representation of the QueryPart being constructed. While creating conditional expressions, column expressions (such as functions) assumes immutable behaviour, creating SQL statements does not. In other words, the following can be said:

```
// Conditional expressions (immutable)
// -----------------------------------
Condition a = BOOK.TITLE.eq("1984");
Condition b = BOOK.TITLE.eq("Animal Farm");

// The following can be said
a       != a.or(b); // or() does not modify a
a.or(b) != a.or(b); // or() always creates new objects

// Statements (mutable)
// --------------------
SelectFromStep<?> s1 = select();
SelectJoinStep<?> s2 = s1.from(BOOK);
SelectJoinStep<?> s3 = s1.from(AUTHOR);

// The following can be said
s1 == s2; // The internal object is always the same
s2 == s3; // The internal object is always the same
```

On the other hand, beware that you can always extract and modify bind values from any QueryPart.

3.3.2. The WITH clause

The SQL:1999 standard specifies the WITH clause to be an optional clause for the SELECT statement, in order to specify common table expressions (also: CTE). Many other databases (such as PostgreSQL, SQL

Server) also allow for using common table expressions also in other DML clauses, such as the INSERT statement, UPDATE statement, DELETE statement, or MERGE statement.

When using common table expressions with jOOQ, there are essentially two approaches:

- Declaring and assigning common table expressions explicitly to names
- Inlining common table expressions into a SELECT statement

Explicit common table expressions

The following example makes use of names to construct common table expressions, which can then be supplied to a WITH clause or a FROM clause of a SELECT statement:

```
-- Pseudo-SQL for a common table expression specification
"t1" ("f1", "f2") AS (SELECT 1, 'a')
```

```
// Code for creating a CommonTableExpression instance
name("t1").fields("f1", "f2").as(select(val(1), val("a")));
```

The above expression can be assigned to a variable in Java and then be used to create a full SELECT statement:

```
WITH "t1" ("f1", "f2") AS (SELECT 1, 'a'),
     "t2" ("f3", "f4") AS (SELECT 2, 'b')
SELECT
    "t1"."f1" + "t2"."f3" AS "add",
    "t1"."f2" || "t2"."f4" AS "concat"
FROM "t1", "t2"
;
```

```
CommonTableExpression<Record2<Integer, String>> t1 =
  name("t1").fields("f1", "f2").as(select(val(1), val("a")));
CommonTableExpression<Record2<Integer, String>> t2 =
  name("t2").fields("f3", "f4").as(select(val(2), val("b")));

Result<?> result2 =
create.with(t1)
      .with(t2)
      .select(
          t1.field("f1").add(t2.field("f3")).as("add"),
          t1.field("f2").concat(t2.field("f4")).as("concat"))
      .from(t1, t2)
      .fetch();
```

Note that the org.jooq.CommonTableExpression type extends the commonly used org.jooq.Table type, and can thus be used wherever a table can be used.

Inlined common table expressions

If you're just operating on plain SQL, you may not need to keep intermediate references to such common table expressions. An example of such usage would be this:

```
WITH "a" AS (SELECT
               1 AS "x",
               'a' AS "y"
            )
SELECT
FROM "a"
;
```

```
create.with("a").as(select(
                        val(1).as("x"),
                        val("a").as("y")
                    ))
      .select()
      .from(table(name("a")))
      .fetch();
```

3.3.3. The WITH RECURSIVE clause

The various SQL dialects do not agree on the use of RECURSIVE when writing recursive common table expressions. When using jOOQ, always use the DSLContext.withRecursive() or DSL.withRecursive() methods, and jOOQ will render the RECURSIVE keyword, if needed.

Assuming a table like this:

```
CREATE TABLE directory (
  id           INT NOT NULL,
  parent_id    INT,

  -- In PostgreSQL, use TEXT instead, to work around https://github.com/jOOQ/jOOQ/issues/12067
  label        VARCHAR(50),

  CONSTRAINT pk_directory PRIMARY KEY (id),
  CONSTRAINT fk_directory FOREIGN KEY (parent_id) REFERENCES directory (id)
);

INSERT INTO directory VALUES ( 1, null, 'C:');
INSERT INTO directory VALUES ( 2,    1, 'eclipse');
INSERT INTO directory VALUES ( 3,    2, 'configuration');
INSERT INTO directory VALUES ( 4,    2, 'dropins');
INSERT INTO directory VALUES ( 5,    2, 'features');
INSERT INTO directory VALUES ( 7,    2, 'plugins');
INSERT INTO directory VALUES ( 8,    2, 'readme');
INSERT INTO directory VALUES ( 9,    8, 'readme_eclipse.html');
INSERT INTO directory VALUES (10,    2, 'src');
INSERT INTO directory VALUES (11,    2, 'eclipse.exe');
```

Using WITH RECURSIVE, you can now query the structure of this directory as follows:

```
WITH RECURSIVE t (
  id,
  name,
  path
) AS (
  SELECT
    DIRECTORY.ID,
    DIRECTORY.LABEL,
    DIRECTORY.LABEL
  FROM
    DIRECTORY
  WHERE
    DIRECTORY.PARENT_ID IS NULL
  UNION ALL
  SELECT
    DIRECTORY.ID,
    DIRECTORY.LABEL,
    t.path
      || '\'
      || DIRECTORY.LABEL
  FROM
    t
  JOIN
    DIRECTORY
  ON t.id = DIRECTORY.PARENT_ID
)
SELECT *
FROM
  t;
```

```
CommonTableExpression<?> cte = name("t").fields(
  "id",
  "name",
  "path"
).as(
  select(
    DIRECTORY.ID,
    DIRECTORY.LABEL,
    DIRECTORY.LABEL)
  .from(DIRECTORY)
  .where(DIRECTORY.PARENT_ID.isNull())
  .unionAll(
  select(
    DIRECTORY.ID,
    DIRECTORY.LABEL,
    field(name("t", "path"), VARCHAR)
      .concat("\\")
      .concat(DIRECTORY.LABEL))
  .from(table(name("t")))
  .join(DIRECTORY)
  .on(field(name("t", "id"), INTEGER)
    .eq(DIRECTORY.PARENT_ID)))
);

System.out.println(
    create.withRecursive(cte)
          .selectFrom(cte)
          .fetch()
);
```

The output would look like this:

```
+----+--------------------+-------------------------------------+
| id | name               | path                                |
+----+--------------------+-------------------------------------+
| 1  | C:                 | C:                                  |
| 2  | eclipse            | C:\eclipse                          |
| 3  | configuration      | C:\eclipse\configuration            |
| 4  | dropins            | C:\eclipse\dropins                  |
| 11 | eclipse.exe        | C:\eclipse\eclipse.exe              |
| 5  | features           | C:\eclipse\features                 |
| 7  | plugins            | C:\eclipse\plugins                  |
| 8  | readme             | C:\eclipse\readme                   |
| 9  | readme_eclipse.html | C:\eclipse\readme\readme_eclipse.html |
| 10 | src                | C:\eclipse\src                      |
+----+--------------------+-------------------------------------+
```

Caveats

The SQL language expresses the recursion syntactically, meaning the table t in the above example is being referenced from within the declaration of t. This isn't possible in a language like Java. Hence, we must use the identifier API to construct identifier references for tables and columns. This technique usually appears a bit more verbose than ordinary jOOQ API usage that is based on generated code for your schema.

3.3.4. The SELECT statement

When you don't just perform CRUD (i.e. SELECT * FROM your_table WHERE ID = ?), you're usually generating new record types using custom projections. With jOOQ, this is as intuitive, as if using SQL directly. A more or less complete example of the "standard" SQL syntax, plus some extensions, is provided by a query like this:

SELECT from a complex table expression

```
-- get all authors' first and last names, and the number
-- of books they've written in German, if they have written
-- more than five books in German in the last three years
-- (from 2011), and sort those authors by last names
-- limiting results to the second and third row, locking
-- the rows for a subsequent update... whew!

  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, COUNT(*)
    FROM AUTHOR
    JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
   WHERE BOOK.LANGUAGE = 'DE'
     AND BOOK.PUBLISHED_IN > 2008
GROUP BY AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  HAVING COUNT(*) > 5
ORDER BY AUTHOR.LAST_NAME ASC NULLS FIRST
   LIMIT 2
  OFFSET 1
     FOR UPDATE
```

```
// And with jOOQ...



DSLContext create = DSL.using(connection, dialect);

create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
      .from(AUTHOR)
      .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED_IN.gt(2008))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
      .forUpdate()
      .fetch();
```

Details about the various clauses of this query will be provided in subsequent sections.

SELECT from single tables

A very similar, but limited API is available, if you want to select from single tables in order to retrieve TableRecords or even UpdatableRecords. The decision, which type of select to create is already made at the very first step, when you create the SELECT statement with the DSL or DSLContext types:

```
public <R extends Record> SelectWhereStep<R> selectFrom(Table<R> table);
```

As you can see, there is no way to further restrict/project the selected fields. This just selects all known TableFields in the supplied Table, and it also binds <R extends Record> to your Table's associated Record. An example of such a Query would then be:

```
BookRecord book = create.selectFrom(BOOK)
                        .where(BOOK.LANGUAGE.eq("DE"))
                        .orderBy(BOOK.TITLE)
                        .fetchAny();
```

The "reduced" SELECT API is limited in the way that it skips DSL access to any of these clauses:

- SELECT clause
- JOIN operator

In most parts of this manual, it is assumed that you do not use the "reduced" SELECT API. For more information about the simple SELECT API, see the manual's section about fetching strongly or weakly typed records.

© 2009 - 2025 by Data Geekery™ GmbH.

3.3.4.1. SELECT clause

The SELECT clause lets you project your own record types, referencing table fields, functions, arithmetic expressions, etc. The DSL type provides several methods for expressing a SELECT clause:

```
-- The SELECT clause
SELECT BOOK.ID, BOOK.TITLE
SELECT BOOK.ID, TRIM(BOOK.TITLE)
```

```
// Provide a varargs Fields list to the SELECT clause:
Select<?> s1 = create.select(BOOK.ID, BOOK.TITLE);
Select<?> s2 = create.select(BOOK.ID, trim(BOOK.TITLE));
```

The following sections illustrate various features and subclauses of the SELECT clause.

3.3.4.1.1. Projection type safety

Since jOOQ 3.0, records and row value expressions up to degree 22 are now generically typesafe. This is reflected by an overloaded SELECT (and SELECT DISTINCT) API in both DSL and DSLContext. An extract from the DSL type:

```
// Non-typesafe select methods:
public static SelectSelectStep<Record> select(Collection<? extends SelectField<?>> fields);
public static SelectSelectStep<Record> select(SelectField<?>... fields);

// Typesafe select methods:
public static <T1>         SelectSelectStep<Record1<T1>>         select(SelectField<T1> field1);
public static <T1, T2>     SelectSelectStep<Record2<T1, T2>>     select(SelectField<T1> field1, SelectField<T2> field2);
// [...]
```

The type that is being projected is the org.jooq.SelectField, see also the next section about SelectField. Since the generic R type is bound to some Record[N], the associated T type information can be used in various other contexts, e.g. the IN predicate. Such a SELECT statement can be assigned typesafely:

```
Select<Record2<Integer, String>> s1 = create.select(BOOK.ID, BOOK.TITLE);
Select<Record2<Integer, String>> s2 = create.select(BOOK.ID, trim(BOOK.TITLE));

// Alternatively, just use var to infer the type:
var s3 = create.select(BOOK.ID, trim(BOOK.TITLE));
```

For more information about typesafe record types with degree up to 22, see the manual's section about Record1 to Record22.

3.3.4.1.2. SelectField

The org.jooq.SelectField type is used by any projection of the SELECT clause and the INSERT .. RETURNING clause. It has numerous subtypes, which are allowed as projections in jOOQ:

- More subtypes are available from future jOOQ versions.

3.3.4.1.3. SELECT *

jOOQ supports the asterisk operator in projections both as a qualified asterisk (through Table.asterisk()) and as an unqualified asterisk (through DSL.asterisk()). It is also possible to omit the projection entirely, in case of which an asterisk may appear in generated SQL, if not all column names are known to jOOQ.

Whenever jOOQ generates an asterisk (explicitly, or because jOOQ doesn't know the exact projection), the column order, and the column set are defined by the database server, not jOOQ. If you're using generated code, this may lead to problems as there might be a different column order than expected, as well as too many or too few columns might be projected.

```
// Explicitly selects all columns available from BOOK - No asterisk
create.select().from(BOOK).fetch();

// Explicitly selects all columns available from BOOK and AUTHOR - No asterisk
create.select().from(BOOK, AUTHOR).fetch();
create.select().from(BOOK).crossJoin(AUTHOR).fetch();

// Renders a SELECT * statement, as columns are unknown to jOOQ - Implicit unqualified asterisk
create.select().from(table(name("BOOK"))).fetch();

// Renders a SELECT * statement - Explicit unqualified asterisk
create.select(asterisk()).from(BOOK).fetch();

// Renders a SELECT BOOK.* statement - Explicit qualified asterisk
create.select(BOOK.asterisk()).from(BOOK).fetch();
create.select(BOOK.asterisk(), AUTHOR.asterisk()).from(BOOK, AUTHOR).fetch();
```

With all of the above syntaxes, the row type (as discussed below) is unknown to jOOQ and to the Java compiler.

> *(!) Unlike the **Nesting tables syntax**, the asterisk is expanded by the SQL engine (or by jOOQ, if necessary) by flattening the table's fields into the surrounding projection. It does not nest tables.*

It is worth mentioning that in many cases, using an asterisk is a sign of an inefficient query because if not all columns are needed, too much data is transferred between client and server, plus some joins that could be eliminated otherwise, cannot. For more information check out this section.

3.3.4.1.4. SELECT * EXCEPT (...)

A useful extension to the previously mentioned standard SQL SELECT * syntax is the BigQuery inspired * EXCEPT (columns) syntax, which takes all of a projection's columns, except *some* columns. Just like the asterisk itself, this is mainly useful for ad-hoc querying, but it can also be useful for an occasional jOOQ query.

```
// Renders a SELECT * statement - Explicit unqualified asterisk
create.select(asterisk().except(BOOK.ID)).from(BOOK).fetch();

// Renders a SELECT BOOK.* statement - Explicit qualified asterisk
create.select(BOOK.asterisk().except(BOOK.ID))
      .from(BOOK)
      .fetch();

create.select(BOOK.asterisk().except(BOOK.ID), AUTHOR.asterisk().except(AUTHOR.ID))
      .from(BOOK, AUTHOR)
      .fetch();
```

If a dialect doesn't support this syntax natively, jOOQ will just expand the syntax for you, explicitly, given the knowledge about meta data in generated code.

© 2009 - 2025 by Data Geekery™ GmbH.

Dialect support

This example using jOOQ:

```
select(asterisk().except(LANGUAGE.ID)).from(LANGUAGE)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, DB2, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT LANGUAGE.CD, LANGUAGE.DESCRIPTION
FROM LANGUAGE
```

BigQuery

```
SELECT * EXCEPT (ID)
FROM LANGUAGE
```

Databricks, H2

```
SELECT * EXCEPT (LANGUAGE.ID)
FROM LANGUAGE
```

Redshift

```
SELECT * EXCLUDE (LANGUAGE.ID)
FROM LANGUAGE
```

Snowflake

```
SELECT * EXCLUDE (ID)
FROM LANGUAGE
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.1.5. SELECT DISTINCT

The DISTINCT keyword can be included in the method name, when constructing a SELECT clause, to remove duplicate tuples from the projection.

```
SELECT DISTINCT BOOK.TITLE FROM BOOK
```

```
create.selectDistinct(BOOK.TITLE).from(BOOK).fetch();
```

Dialect support

This example using jOOQ:

```
selectDistinct(BOOK.TITLE).from(BOOK)
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT DISTINCT BOOK.TITLE
FROM BOOK
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.1.6. SELECT DISTINCT ON

A useful, though perhaps a bit esoteric PostgreSQL specific extension to SELECT DISTINCT is the ON clause. Using this clause, PostgreSQL users can specify a distinctness criteria, but then produce other columns per distinct group from one of the group's tuples. This is normally not possible in SQL, but with ON, the first tuple in the group according to the ORDER BY clause can be accessed nonetheless. An example:

```
SELECT DISTINCT ON (BOOK.LANGUAGE_ID)
  BOOK.LANGUAGE_ID, BOOK.TITLE
FROM BOOK
ORDER BY BOOK.LANGUAGE_ID, BOOK.TITLE
```

```
Select<?> select1 = create.select(BOOK.LANGUAGE_ID, BOOK.TITLE)
      .distinctOn(BOOK.LANGUAGE_ID)
      .from(BOOK)
      .orderBy(BOOK.LANGUAGE_ID, BOOK.TITLE).fetch();
```

For syntactic reasons, the order of keywords had to be inversed as the PostgreSQL syntax cannot be easily reproduced in jOOQ's internal DSL. Quite likely, you might find jOOQ's syntax a bit more intuitive, though, as it more clearly separates the SELECT parts and the DISTINCT ON parts. Arguably, the DISTINCT ON clause should be positioned after ORDER BY, where it logically belongs.

© 2009 - 2025 by Data Geekery™ GmbH.

Standard SQL equivalence

The PostgreSQL extension isn't really necessary as there is a standard SQL equivalence using ROW_NUMBER filtering. In the below example, we're using an extension to the standard, the QUALIFY clause, to illustrate:

```
SELECT BOOK.LANGUAGE_ID, BOOK.TITLE
FROM BOOK
QUALIFY ROW_NUMBER() OVER (PARTITION BY BOOK.LANGUAGE_ID ORDER BY
 BOOK.TITLE) = 1
ORDER BY BOOK.LANGUAGE_ID, BOOK.TITLE
```

```
Select<?> select1 = create.select(BOOK.LANGUAGE_ID, BOOK.TITLE)
      .from(BOOK)

 .qualify(rowNumber().over(partitionBy(BOOK.LANGUAGE_ID).orderBy(BOOK.TITLE)).eq(
      .orderBy(BOOK.LANGUAGE_ID, BOOK.TITLE).fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.LANGUAGE_ID, BOOK.TITLE).distinctOn(BOOK.LANGUAGE_ID).from(BOOK).orderBy(BOOK.LANGUAGE_ID, BOOK.TITLE)
```

Translates to the following dialect specific expressions:

Aurora Postgres, ClickHouse, CockroachDB, DuckDB, H2, Postgres, YugabyteDB

```
SELECT DISTINCT ON (BOOK.LANGUAGE_ID) BOOK.LANGUAGE_ID, BOOK.TITLE
FROM BOOK
ORDER BY BOOK.LANGUAGE_ID, BOOK.TITLE
```

BigQuery, DB2, Databricks, Exasol, Firebird, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
SELECT t.LANGUAGE_ID, t.TITLE
FROM (
  SELECT
    BOOK.LANGUAGE_ID,
    BOOK.TITLE,
    row_number() OVER (
      PARTITION BY BOOK.LANGUAGE_ID
      ORDER BY BOOK.LANGUAGE_ID, BOOK.TITLE
    ) rn
  FROM BOOK
) t
WHERE rn = 1
ORDER BY LANGUAGE_ID, TITLE
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.1.7. Convenience methods

Some commonly used projections can be easily created using convenience methods:

```
-- Simple SELECTs
SELECT COUNT(*)
SELECT 0 -- Not a bind variable
SELECT 1 -- Not a bind variable
```

```
// Select commonly used values
Result<?> result1 = create.selectCount().fetch();
Result<?> result2 = create.selectZero().fetch();
Result<?> result3 = create.selectOne().fetch();
```

Which are short forms for creating Column expressions from the org.jooq.impl.DSL API

```
-- Simple SELECTs
SELECT COUNT(*)
SELECT 0 -- Not a bind variable
SELECT ? -- A bind variable
```

```
// Select commonly used values
Result<?> result1 = create.select(count()).fetch();
Result<?> result2 = create.select(inline(0)).fetch();
Result<?> result3 = create.select(val(1)).fetch();
```

3.3.4.2. FROM clause

The SQL FROM clause allows for specifying any number of table expressions to select data from. The following are examples of how to form normal FROM clauses:

```
SELECT 1 FROM BOOK
SELECT 1 FROM BOOK, AUTHOR
SELECT 1 FROM BOOK "b", AUTHOR "a"
```

```
create.selectOne().from(BOOK).fetch();
create.selectOne().from(BOOK, AUTHOR).fetch();
create.selectOne().from(BOOK.as("b"), AUTHOR.as("a")).fetch();
```

Read more about aliasing in the manual's section about aliased tables.

More advanced table expressions

Apart from simple tables, you can pass any arbitrary table expression to the jOOQ FROM clause. This may include unnested cursors in Oracle:

```
SELECT *
FROM TABLE(
    DBMS_XPLAN.DISPLAY_CURSOR(null, null, 'ALLSTATS')
);
```

```
create.select()
      .from(table(
          DbmsXplan.displayCursor(null, null, "ALLSTATS")
      ).fetch();
```

Note, in order to access the DbmsXplan package, you can use the code generator to generate Oracle's SYS schema.

Selecting FROM DUAL with jOOQ

In many SQL dialects, FROM is a mandatory clause, in some it isn't. jOOQ allows you to omit the FROM clause, returning just one record. An example:

```
SELECT 1 FROM DUAL
SELECT 1
```

```
DSL.using(SQLDialect.ORACLE).selectOne().fetch();
DSL.using(SQLDialect.POSTGRES).selectOne().fetch();
```

© 2009 - 2025 by Data Geekery™ GmbH.

Read more about dual or dummy tables in the manual's section about the DUAL table. The following are examples of how to form normal FROM clauses:

3.3.4.2.1. JOIN operator

jOOQ supports many different types of standard and non-standard SQL JOIN operations. All of these JOIN methods can be called on org.jooq.Table types the (more info in joined tables section), or directly after the FROM clause for convenience. The following example joins AUTHOR and BOOK

```
DSLContext create = DSL.using(connection, dialect);

// Call "join" directly on the AUTHOR table
Result<?> result = create.select()
                         .from(AUTHOR.join(BOOK)
                                     .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
                         .fetch();

// Call "join" on the type returned by "from"
Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK)
                         .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                         .fetch();
```

The two syntaxes will produce the same SQL statement. However, calling "join" on org.jooq.Table objects allows for more powerful, nested JOIN expressions (if you can handle the parentheses):

```
SELECT *
FROM AUTHOR
LEFT OUTER JOIN (
  BOOK JOIN BOOK_TO_BOOK_STORE
       ON BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
)
ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
// Nest joins and provide JOIN conditions only at the end
create.select()
      .from(AUTHOR
      .leftOuterJoin(BOOK
        .join(BOOK_TO_BOOK_STORE)
        .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK.ID)))
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
      .fetch();
```

- See the section about conditional expressions to learn more about the many ways to create org.jooq.Condition objects in jOOQ.
- See the section about table expressions to learn about the various ways of referencing org.jooq.Table objects in jOOQ

For more information about the different types of join, please refer to the joined tables section.

3.3.4.2.2. Implicit path JOIN

In SQL, a lot of explicit JOIN clauses are written simply to retrieve a parent table's column from a given child table. For example, we'll write:

```
-- Get all books, their authors, and their respective language
SELECT
  a.first_name,
  a.last_name,
  b.title,
  l.cd AS language
FROM book b
JOIN author a ON b.author_id = a.id
JOIN language l ON b.language_id = l.id;

-- Count the number of books by author and language
SELECT
  a.first_name,
  a.last_name,
  l.cd AS language,
  COUNT(*)
FROM book
JOIN author a ON b.author_id = a.id
JOIN language l ON b.language_id = l.id
GROUP BY a.id, a.first_name, a.last_name, l.cd
ORDER BY a.first_name, a.last_name, l.cd
```

There is quite a bit of syntactic ceremony (or we could even call it "noise") to get a relatively simple job done. A much simpler notation would be using implicit joins:

```
-- Get all books, their authors, and their respective language
SELECT
  b.author.first_name,
  b.author.last_name,
  b.title,
  b.language.cd AS language
FROM book b;

-- Count the number of books by author and language
SELECT
  b.author.first_name,
  b.author.last_name,
  b.language.cd AS language,
  COUNT(*)
FROM book b
GROUP BY
  b.author_id,
  b.author.first_name,
  b.author.last_name,
  b.language.cd
ORDER BY
  b.author.first_name,
  b.author.last_name,
  b.language.cd
```

Notice how this alternative notation (depending on your taste) may look more tidy and straightforward, as the semantics of accessing a table's parent table (or an entity's parent entity) is straightforward.

From jOOQ 3.11 onwards, this syntax is supported for to-one relationship navigation, and from jOOQ 3.19 also for to-many relationship navigation. The code generator produces relevant navigation methods on generated tables, which can be used in a type safe way. The navigation method names are:

- The parent table name, if there is only one foreign key between child table and parent table
- The foreign key name, if there are more than one foreign keys between child table and parent table

This default behaviour can be overridden by using a Code Generator Strategy.

The jOOQ version of the previous queries looks like this:

```
// Get all books, their authors, and their respective language
create.select(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.TITLE,
          BOOK.language().CD.as("language"))
      .from(BOOK)
      .fetch();

// Count the number of books by author and language
create.select(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD.as("language"),
          count())
      .from(BOOK)
      .groupBy(
          BOOK.AUTHOR_ID,
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD)
      .orderBy(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD)
      .fetch();
```

The generated SQL is almost identical to the original one - there is no performance penalty to this syntax.

Default JOIN type

The default type of join that is generated is:

- INNER JOIN for to-one path segments with non-nullable parent
- LEFT JOIN for to-one path segments with nullable parent

How it works

During the SQL generation phase, implicit join paths are replaced by generated aliases for the path's last table. The paths are translated to a join graph, which is always LEFT JOINed to the path's "root table". If two paths share a common prefix, that prefix is also shared in the join graph.

Known limitations

- Until jOOQ 3.17, implicit JOINs were only supported in SELECT statements (including any type of subquery), but not in the WHERE clause of UPDATE statements or DELETE statements, for instance.
- Implicit JOINs can currently only be used to access columns, not to produce joins. I.e. it is not possible to write things like FROM book IMPLICIT JOIN book.author
- Implicit JOINs are added to the SQL string after the entire SQL statement is available, for performance reasons. This means, that VisitListener SPI implementations cannot observe implicitly joined tables

3.3.4.3. WHERE clause

The WHERE clause can be used for JOIN or filter predicates, in order to restrict the data returned by the table expressions supplied to the previously specified from clause and join clause. Here is an example:

```
SELECT *
FROM BOOK
WHERE AUTHOR_ID = 1
AND TITLE = '1984'
```

```
create.select()
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(1))
      .and(BOOK.TITLE.eq("1984"))
      .fetch();
```

The above syntax is convenience provided by jOOQ, allowing you to connect the org.jooq.Condition supplied in the WHERE clause with another condition using an AND operator. You can of course also create a more complex condition and supply that to the WHERE clause directly (observe the different placing of parentheses). The results will be the same:

```
SELECT *
FROM BOOK
WHERE AUTHOR_ID = 1
AND TITLE = '1984'
```

```
create.select()
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(1).and(
             BOOK.TITLE.eq("1984")))
      .fetch();
```

You will find more information about creating conditional expressions later in the manual.

3.3.4.4. CONNECT BY clause

The Oracle database knows a very succinct syntax for creating hierarchical queries: the CONNECT BY clause, which is fully supported by jOOQ, including all related functions and pseudo-columns. A more or less formal definition of this clause is given here:

```
--   SELECT ..
--     FROM ..
--    WHERE ..
 CONNECT BY [ NOCYCLE ] condition [ AND condition, ... ] [ START WITH condition ]
-- GROUP BY ..
-- ORDER [ SIBLINGS ] BY ..
```

An example for an iterative query, iterating through values between 1 and 5 is this:

```
SELECT LEVEL
FROM DUAL
CONNECT BY LEVEL <= 5
```

```
// Get a table with elements 1, 2, 3, 4, 5
create.select(level())
      .connectBy(level().le(5))
      .fetch();
```

Here's a more complex example where you can recursively fetch directories in your database, and concatenate them to a path:

```
SELECT
  SUBSTR(SYS_CONNECT_BY_PATH(DIRECTORY.NAME, '/'), 2)
FROM DIRECTORY
CONNECT BY
  PRIOR DIRECTORY.ID = DIRECTORY.PARENT_ID
START WITH DIRECTORY.PARENT_ID IS NULL
ORDER BY 1
```

```
.select(
   substring(sysConnectByPath(DIRECTORY.NAME, "/"), 2))
.from(DIRECTORY)
.connectBy(
   prior(DIRECTORY.ID).eq(DIRECTORY.PARENT_ID))
.startWith(DIRECTORY.PARENT_ID.isNull())
.orderBy(1)
.fetch();
```

© 2009 - 2025 by Data Geekery™ GmbH.

The output might then look like this

```
+-----------------------------------------------+
|substring                                      |
+-----------------------------------------------+
|C:                                             |
|C:/eclipse                                     |
|C:/eclipse/configuration                       |
|C:/eclipse/dropins                             |
|C:/eclipse/eclipse.exe                         |
+-----------------------------------------------+
|...21 record(s) truncated...
```

Some of the supported functions, operators, and pseudo-columns are these (available from the DSL):

- CONNECT_BY_ISCYCLE function
- CONNECT_BY_ISLEAF function
- CONNECT_BY_ROOT operator
- LEVEL pseudo-column
- PRIOR operator
- SYS_CONNECT_BY_PATH function

ORDER SIBLINGS

The Oracle database allows for specifying a SIBLINGS keyword in the ORDER BY clause. Instead of ordering the overall result, this will only order siblings among each other, keeping the hierarchy intact. An example is given here:

```
SELECT DIRECTORY.NAME
FROM DIRECTORY
CONNECT BY
  PRIOR DIRECTORY.ID = DIRECTORY.PARENT_ID
START WITH DIRECTORY.PARENT_ID IS NULL
ORDER SIBLINGS BY 1
```

```
.select(DIRECTORY.NAME)
.from(DIRECTORY)
.connectBy(
   prior(DIRECTORY.ID).eq(DIRECTORY.PARENT_ID))
.startWith(DIRECTORY.PARENT_ID.isNull())
.orderSiblingsBy(1)
.fetch();
```

3.3.4.5. GROUP BY clause

GROUP BY can be used to create unique groups of data, to form aggregations, to remove duplicates and for other reasons. It will transform your previously defined set of table expressions, and return only one record per unique group as specified in this clause.

3.3.4.5.1. GROUP BY columns

The GROUP BY columns list specifies the columns whose values are used to form groups. The group columns can then be projected, whereas all the non-group columns can be aggregated. An example of such a grouped aggregation is this query:

```
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

The above example counts all books per author.

> *(!) Note: a different and more powerful way of grouping data is to use the **WINDOW clause and window functions**.*

Dialect support

This example using jOOQ:

```
select(BOOK.AUTHOR_ID, count()).from(BOOK).groupBy(BOOK.AUTHOR_ID)
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT
  BOOK.AUTHOR_ID,
  count(*)
FROM BOOK
GROUP BY BOOK.AUTHOR_ID
```

> *(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.3.4.5.2. GROUP BY ROLLUP

In reports, it may be useful to run multiple aggregations across multiple dimensions of the data in one go. ROLLUP is one way to do this.

```
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY ROLLUP (AUTHOR_ID, PUBLISHED_IN)
```

```
create.select(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN, count())
      .from(BOOK)
      .groupBy(rollup(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN))
      .fetch();
```

The above is a more concise (and possibly more performant) form of writing the following UNION ALL query:

```
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID, PUBLISHED_IN
UNION ALL
SELECT AUTHOR_ID, NULL, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
UNION ALL
SELECT NULL, NULL, COUNT(*)
FROM BOOK
GROUP BY ()
```

The ROLLUP function is just syntax sugar for a more complex GROUPING SETS specification. In general:

```
-- This
ROLLUP (A, B, C)

-- Is just short for this
GROUPING SETS ((A, B, C), (A, B), (A), ())
```

An example result set might look like this:

```
+-----------+--------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+--------------+----------+
|         1 |         1945 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         1948 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         2 |         1988 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         1990 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|      NULL |         NULL |        4 | <- GROUP BY ()
+-----------+--------------+----------+
```

Dialect support

This example using jOOQ:

```
select(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID, count()).from(BOOK).groupBy(rollup(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID))
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MySQL

```
SELECT
  BOOK.AUTHOR_ID,
  BOOK.LANGUAGE_ID,
  count(*)
FROM BOOK
GROUP BY BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID
WITH ROLLUP
```

Aurora Postgres, ClickHouse, DB2, Databricks, DuckDB, Hana, MemSQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Trino, Vertica

```
SELECT
  BOOK.AUTHOR_ID,
  BOOK.LANGUAGE_ID,
  count(*)
FROM BOOK
GROUP BY ROLLUP (BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID)
```

ASE, Access, BigQuery, CockroachDB, Derby, Exasol, Firebird, H2, HSQLDB, Informix, SQLite, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.5.3. GROUP BY CUBE

In reports, it may be useful to run multiple aggregations across multiple dimensions of the data in one go. CUBE is one way to do this.

```
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY CUBE (AUTHOR_ID, PUBLISHED_IN)
```

```
create.select(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN, count())
      .from(BOOK)
      .groupBy(cube(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN))
      .fetch();
```

The above is a more concise (and possibly more performant) form of writing the following UNION ALL query:

```
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID, PUBLISHED_IN
UNION ALL
SELECT AUTHOR_ID, NULL, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
SELECT NULL, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY LANGUAGE_ID
UNION ALL
SELECT NULL, NULL, COUNT(*)
FROM BOOK
GROUP BY ()
```

The CUBE function is just syntax sugar for a more complex GROUPING SETS specification. In general:

```
-- This
CUBE (A, B, C)

-- Is just short for this
GROUPING SETS ((A, B, C), (A, B), (A, C), (B, C), (A), (B), (C), ())
```

An example result set might look like this:

```
+-----------+--------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+--------------+----------+
|      NULL |         NULL |        4 | <- GROUP BY ()
|      NULL |         1945 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1948 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1988 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1990 |        1 | <- GROUP BY (PUBLISHED_IN)
|         1 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         1 |         1945 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         1948 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         2 |         1988 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         1990 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
+-----------+--------------+----------+
```

Dialect support

This example using jOOQ:

```
select(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID, count()).from(BOOK).groupBy(cube(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, ClickHouse, DB2, Databricks, DuckDB, Hana, Oracle, Postgres, Redshift, SQLServer, Snowflake, Sybase, Teradata, Trino, Vertica

```
SELECT
  BOOK.AUTHOR_ID,
  BOOK.LANGUAGE_ID,
  count(*)
FROM BOOK
GROUP BY CUBE (BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID)
```

ASE, Access, Aurora MySQL, BigQuery, CockroachDB, Derby, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.5.4. GROUP BY GROUPING SETS

In reports, it may be useful to run multiple aggregations across multiple dimensions of the data in one go. GROUPING SETS is one way to do this.

```
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY GROUPING SETS ((AUTHOR_ID), (PUBLISHED_IN))
```

```
create.select(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN, count())
      .from(BOOK)
      .groupBy(groupingSets(BOOK.AUTHOR_ID, BOOK.PUBLISHED_IN))
      .fetch();
```

The above is a more concise (and possibly more performant) form of writing the following UNION ALL query:

```
SELECT AUTHOR_ID, NULL AS PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
UNION ALL
SELECT NULL, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY LANGUAGE_ID
```

An example result set might look like this:

```
+-----------+--------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+--------------+----------+
|      NULL |         1945 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1948 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1988 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1990 |        1 | <- GROUP BY (PUBLISHED_IN)
|         1 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         2 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
+-----------+--------------+----------+
```

Note that the most common GROUPING SETS specifications have a dedicated, special syntax:

- ROLLUP
- CUBE

Dialect support

This example using jOOQ:

```
select(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID, count()).from(BOOK).groupBy(groupingSets(BOOK.AUTHOR_ID, BOOK.LANGUAGE_ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, ClickHouse, DB2, Databricks, DuckDB, Hana, Oracle, Postgres, Redshift, SQLServer, Snowflake, Sybase, Teradata, Trino, Vertica

```
SELECT
  BOOK.AUTHOR_ID,
  BOOK.LANGUAGE_ID,
  count(*)
FROM BOOK
GROUP BY GROUPING SETS (
  (BOOK.AUTHOR_ID),
  (BOOK.LANGUAGE_ID)
)
```

ASE, Access, Aurora MySQL, BigQuery, CockroachDB, Derby, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.5.5. GROUP BY empty grouping set

A special kind of GROUPING SET is the empty grouping set, which can be achieved in standard SQL and many SQL dialects using GROUP BY (). It is implicit, whenever an aggregate function is present in a query, but not an explicit GROUP BY clause.

```
SELECT COUNT(*)
FROM BOOK
GROUP BY ()
```

```
create.selectCount()
      .from(BOOK)
      .groupBy()
      .fetch();
```

Dialect support

This example using jOOQ:

```
selectCount().from(BOOK).groupBy()
```

Translates to the following dialect specific expressions:

Access

```
SELECT count(*)
FROM BOOK, (select count(*) dual from MSysResources) as empty_grouping_dummy_table
GROUP BY empty_grouping_dummy_table.dual
```

ASE, SQLDataWarehouse

```
SELECT count(*)
FROM BOOK, (select 1 as dual) as empty_grouping_dummy_table
GROUP BY empty_grouping_dummy_table.dual
```

Aurora MySQL, MemSQL

```
SELECT count(*)
FROM BOOK
GROUP BY (SELECT 1
FROM DUAL)
```

Aurora Postgres, BigQuery, DB2, DuckDB, Exasol, H2, Oracle, Postgres, SQLServer, Sybase, Teradata, Trino

```
SELECT count(*)
FROM BOOK
GROUP BY ()
```

© 2009 - 2025 by Data Geekery™ GmbH.

ClickHouse, CockroachDB, MariaDB, MySQL, Redshift, SQLite, Vertica, YugabyteDB

```
SELECT count(*)
FROM BOOK
GROUP BY (SELECT 1)
```

Databricks, Hana, Snowflake

```
SELECT count(*)
FROM BOOK
GROUP BY GROUPING SETS (())
```

Derby, HSQLDB

```
SELECT count(*)
FROM BOOK
GROUP BY 0
```

Firebird

```
SELECT count(*)
FROM BOOK
GROUP BY (SELECT 1
FROM RDB$DATABASE)
```

Informix

```
SELECT count(*)
FROM BOOK, (select 1 as dual from systables where tabid = 1) as empty_grouping_dummy_table
GROUP BY empty_grouping_dummy_table.dual
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.6. HAVING clause

The HAVING clause is commonly used to further restrict data resulting from a previously issued GROUP BY clause. An example, selecting only those authors that have written at least two books:

```
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
HAVING COUNT(*) >= 2
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(AUTHOR_ID)
      .having(count().ge(2))
      .fetch();
```

According to the SQL standard, you may omit the GROUP BY clause and still issue a HAVING clause. This will implicitly GROUP BY (). jOOQ also supports this syntax. The following example selects one record, only if there are at least 4 books in the books table:

```
SELECT COUNT(*)
FROM BOOK
HAVING COUNT(*) >= 4
```

```
create.select(count(*))
      .from(BOOK)
      .having(count().ge(4))
      .fetch();
```

3.3.4.7. WINDOW clause

The SQL:2003 standard supports a WINDOW clause that allows for specifying WINDOW frames for reuse in SELECT clauses and ORDER BY clauses.

```
SELECT
  LAG(first_name, 1) OVER w "prev",
  first_name,
  LEAD(first_name, 1) OVER w "next"
FROM author
WINDOW w AS (ORDER first_name)
ORDER BY first_name DESC
```

```
WindowDefinition w = name("w").as(
  orderBy(AUTHOR.FIRST_NAME));

create.select(
         lag(AUTHOR.FIRST_NAME, 1).over(w).as("prev"),
         AUTHOR.FIRST_NAME,
         lead(AUTHOR.FIRST_NAME, 1).over(w).as("next"))
      .from(AUTHOR)
      .window(w)
      .orderBy(AUTHOR.FIRST_NAME.desc())
      .fetch();
```

Note that in order to create such a window definition, we need to first create a name reference using DSL.name().

Even if only PostgreSQL and Sybase SQL Anywhere natively support this great feature, jOOQ can emulate it by expanding any org.jooq.WindowDefinition and org.jooq.WindowSpecification types that you pass to the window() method - if the database supports window functions at all.

Some more information about window functions and the WINDOW clause can be found on our blog: https://blog.jooq.org/probably-the-coolest-sql-feature-window-functions/

Dialect support

This example using jOOQ:

```
select(rowNumber().over("w")).from(AUTHOR).window(name("w").as(orderBy(AUTHOR.ID)))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, MySQL, Oracle, Postgres, SQLServer, SQLite, Sybase, Trino, YugabyteDB

```
SELECT row_number() OVER w
FROM AUTHOR
WINDOW w AS (ORDER BY AUTHOR.ID)
```

DB2, Hana, Informix, MariaDB, MemSQL, SQLDataWarehouse, Snowflake, Teradata, Vertica

```
SELECT row_number() OVER (ORDER BY AUTHOR.ID)
FROM AUTHOR
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Redshift

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.8. QUALIFY clause

A select few dialects support a very useful QUALIFY clause, which can be used to filter using window functions without having to nest the window function calculation in a derived table.

For example, if you do not have access to the WITH TIES clause, you could easily emulate it like this. The following query finds the top 5 author ***WITH TIES***, counting their books:

```
SELECT AUTHOR_ID, count(*)
FROM BOOK
GROUP BY AUTHOR_ID
QUALIFY rank() OVER (ORDER BY count(*) DESC) <= 5
ORDER BY count(*) DESC
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .qualify(rank().over(orderBy(count().desc())).le(5))
      .orderBy(count().desc())
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(AUTHOR.ID).from(AUTHOR).qualify(rank().over(orderBy(AUTHOR.ID)).le(10))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, MemSQL, Postgres, SQLite, Trino, Vertica, YugabyteDB

```
SELECT t.ID ID
FROM (
  SELECT
    *,
    (rank() OVER (ORDER BY AUTHOR.ID) <= 10) w0
  FROM AUTHOR
) t
WHERE w0
```

BigQuery

```
SELECT AUTHOR.ID
FROM AUTHOR
WHERE TRUE
QUALIFY rank() OVER (ORDER BY AUTHOR.ID) <= 10
```

ClickHouse, Databricks, DuckDB, Exasol, H2, Snowflake, Teradata

```
SELECT AUTHOR.ID
FROM AUTHOR
QUALIFY rank() OVER (ORDER BY AUTHOR.ID) <= 10
```

DB2, SQLDataWarehouse, SQLServer, Sybase

```
SELECT t.ID ID
FROM (
  SELECT
    *,
    CASE
      WHEN rank() OVER (ORDER BY AUTHOR.ID) <= 10 THEN 1
      WHEN NOT (rank() OVER (ORDER BY AUTHOR.ID) <= 10) THEN 0
    END w0
  FROM AUTHOR
) t
WHERE w0 = 1
```

Firebird

```
SELECT t.ID ID
FROM (
  SELECT
    AUTHOR.ID,
    AUTHOR.FIRST_NAME,
    AUTHOR.LAST_NAME,
    AUTHOR.DATE_OF_BIRTH,
    AUTHOR.YEAR_OF_BIRTH,
    AUTHOR.DISTINGUISHED,
    CASE
      WHEN rank() OVER (ORDER BY AUTHOR.ID) <= 10 THEN 1
      WHEN NOT (rank() OVER (ORDER BY AUTHOR.ID) <= 10) THEN 0
    END w0
  FROM AUTHOR
) t
WHERE w0 = 1
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
SELECT t.ID ID
FROM (
  SELECT
    *,
    CASE
      WHEN rank() OVER (ORDER BY AUTHOR.ID) <= 10 THEN TRUE
      WHEN NOT (rank() OVER (ORDER BY AUTHOR.ID) <= 10) THEN FALSE
    END w0
  FROM AUTHOR
) t
WHERE w0 = TRUE
```

Informix

```
SELECT t.ID ID
FROM (
  SELECT
    *,
    CASE
      WHEN rank() OVER (ORDER BY AUTHOR.ID) <= 10 THEN CAST('t' AS boolean)
      WHEN NOT (rank() OVER (ORDER BY AUTHOR.ID) <= 10) THEN CAST('f' AS boolean)
    END w0
  FROM AUTHOR
) t
WHERE w0
```

MariaDB, MySQL, Oracle

```
SELECT t.ID ID
FROM (
  SELECT
    AUTHOR.ID,
    AUTHOR.FIRST_NAME,
    AUTHOR.LAST_NAME,
    AUTHOR.DATE_OF_BIRTH,
    AUTHOR.YEAR_OF_BIRTH,
    AUTHOR.DISTINGUISHED,
    (rank() OVER (ORDER BY AUTHOR.ID) <= 10) w0
  FROM AUTHOR
) t
WHERE w0
```

Redshift

```
SELECT AUTHOR.ID
FROM AUTHOR
WHERE 1 = 1
QUALIFY rank() OVER (ORDER BY AUTHOR.ID) <= 10
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.9. ORDER BY clause

Databases are allowed to return data in any arbitrary order, unless you explicitly declare that order in the ORDER BY clause.

```
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY AUTHOR_ID ASC, TITLE DESC
```

```
create.select(BOOK.AUTHOR_ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(BOOK.AUTHOR_ID.asc(), BOOK.TITLE.desc())
      .fetch();
```

Any jOOQ column expression (or field) can be transformed into an org.jooq.SortField by calling the asc() and desc() methods.

jOOQ's understanding of SELECT .. ORDER BY

The SQL standard defines that a "query expression" can be ordered, and that query expressions can contain UNION, INTERSECT and EXCEPT clauses, whose subqueries cannot be ordered. While this is defined as such in the SQL standard, many databases allowing for the LIMIT clause in one way or another, do not adhere to this part of the SQL standard. Hence, jOOQ allows for ordering all SELECT statements, regardless whether they are constructed as a part of a UNION or not. Corner-cases are handled internally by jOOQ, by introducing synthetic subselects to adhere to the correct syntax, where this is needed.

3.3.4.9.1. Ordering by field index

The SQL standard allows for specifying integer literals (literals, not bind values!) to reference column indexes from the projection (SELECT clause). This may be useful if you do not want to repeat a lengthy expression, by which you want to order - although most databases also allow for referencing aliased column references in the ORDER BY clause.

An example of this is given here:

```
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY 1 ASC, 2 DESC
```

```
create.select(BOOK.AUTHOR_ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(inline(1).asc(), inline(2).desc())
      .fetch();
```

> *(!) This practice is generally discouraged as field indexes may shift in the SELECT clause, and developers might forget to update the indexes in ORDER BY. It is mainly useful for quick-and-dirty ad-hoc SQL. See also* ***the don't do this section about this topic****.*

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).orderBy(1)
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT BOOK.ID
FROM BOOK
ORDER BY 1
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.9.2. Ordering and NULLS

A few databases support the SQL standard "null ordering" clause in sort specification lists, to define whether NULL values should come first or last in an ordered result.

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
FROM AUTHOR
ORDER BY LAST_NAME ASC,
         FIRST_NAME ASC NULLS LAST
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .orderBy(AUTHOR.LAST_NAME.asc(),
               AUTHOR.FIRST_NAME.asc().nullsLast())
      .fetch();
```

If your database doesn't support this syntax, jOOQ emulates it using a CASE expression

Dialect support

This example using jOOQ:

```
select(AUTHOR.FIRST_NAME).from(AUTHOR).orderBy(AUTHOR.FIRST_NAME.asc().nullsLast())
```

Translates to the following dialect specific expressions:

Access, SQLServer

```
SELECT AUTHOR.FIRST_NAME
FROM AUTHOR
ORDER BY iif(AUTHOR.FIRST_NAME IS NOT NULL, 0, 1), AUTHOR.FIRST_NAME ASC
```

ASE, Aurora MySQL, MemSQL, MySQL, SQLDataWarehouse, Sybase

```
SELECT AUTHOR.FIRST_NAME
FROM AUTHOR
ORDER BY CASE
  WHEN AUTHOR.FIRST_NAME IS NOT NULL THEN 0
  ELSE 1
END, AUTHOR.FIRST_NAME ASC
```

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, SQLite, Snowflake, Teradata, Trino, YugabyteDB

```
SELECT AUTHOR.FIRST_NAME
FROM AUTHOR
ORDER BY AUTHOR.FIRST_NAME ASC NULLS LAST
```

DB2, MariaDB, Redshift, Vertica

```
SELECT AUTHOR.FIRST_NAME
FROM AUTHOR
ORDER BY nvl2(AUTHOR.FIRST_NAME, 0, 1), AUTHOR.FIRST_NAME ASC
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.9.3. Ordering using CASE expressions

Using CASE expressions in SQL ORDER BY clauses is a common pattern, if you want to introduce some sort indirection / sort mapping into your queries. As with SQL, you can add any type of column expression into your ORDER BY clause.

For instance, if you have two favourite books that you always want to appear on top, you could write:

```
SELECT *
FROM BOOK
ORDER BY CASE TITLE
         WHEN '1984' THEN 0
         WHEN 'Animal Farm' THEN 1
         ELSE 2 END ASC
```

```
create.select()
      .from(BOOK)
      .orderBy(case_(BOOK.TITLE)
               .when("1984", 0)
               .when("Animal Farm", 1)
               .else_(2).asc())
      .fetch();
```

But writing these things can become quite verbose. jOOQ supports a convenient syntax for specifying sort mappings. The same query can be written in jOOQ as such:

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sortAsc("1984", "Animal Farm"))
      .fetch();
```

More complex sort indirections can be provided using a Map:

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sort(Map.of(
          "1984", 1,
          "Animal Farm", 13,
          "The jOOQ book", 10
      )))
      .fetch();
```

Of course, you can combine this feature with the previously discussed NULLS FIRST / NULLS LAST feature. So, if in fact these two books are the ones you like least, you can put all NULLS FIRST (all the other books):

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sortAsc("1984", "Animal Farm").nullsFirst())
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).orderBy(BOOK.TITLE.sortAsc("1984", "Animal Farm"))
```

Translates to the following dialect specific expressions:

Access

```
SELECT BOOK.ID
FROM BOOK
ORDER BY SWITCH(BOOK.TITLE = '1984', 0, BOOK.TITLE = 'Animal Farm', 1) ASC
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
ORDER BY CASE BOOK.TITLE
  WHEN '1984' THEN 0
  WHEN 'Animal Farm' THEN 1
END ASC
```

Derby

```
SELECT BOOK.ID
FROM BOOK
ORDER BY CASE
  WHEN BOOK.TITLE = '1984' THEN 0
  WHEN BOOK.TITLE = 'Animal Farm' THEN 1
END ASC
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.9.4. Oracle's ORDER SIBLINGS BY clause

jOOQ also supports Oracle's SIBLINGS keyword to be used with ORDER BY clauses for hierarchical queries using CONNECT BY

```
create.select(sysConnectByPath(DIRECTORY.LABEL, "/").as("dir"))
      .from(DIRECTORY)
      .startWith(DIRECTORY.PARENT_ID.isNull())
      .connectBy(prior(DIRECTORY.ID).eq(DIRECTORY.PARENT_ID))
      .orderSiblingsBy(DIRECTORY.LABEL)
      .fetch();
```

3.3.4.10. LIMIT .. OFFSET clause

While being extremely useful for every application that does pagination, or just to limit result sets to reasonable sizes, this clause has not been standardised up until SQL:2008. Hence, there exist a variety of possible implementations in various SQL dialects, concerning this limit clause. jOOQ chose to implement the LIMIT .. OFFSET clause as understood and supported by MySQL, H2, HSQLDB, Postgres, and SQLite. Here is an example of how to apply limits with jOOQ:

```
create.select().from(BOOK).orderBy(BOOK.ID).limit(1).offset(2).fetch();
```

This will limit the result to 1 books skipping the first 2 books (offset 2). limit() is supported in all dialects, offset() in all but Sybase ASE, which has no reasonable means to emulate it. This is how jOOQ trivially emulates the above query in various SQL dialects with native OFFSET pagination support:

```
-- MySQL, H2, HSQLDB, and SQLite
SELECT * FROM BOOK ORDER BY ID LIMIT 1 OFFSET 2

-- Derby, SQL Server 2012, Oracle 12c, PostgreSQL, the SQL:2008 standard
SELECT * FROM BOOK ORDER BY ID OFFSET 2 ROWS FETCH NEXT 1 ROWS ONLY

-- Informix has SKIP .. FIRST support
SELECT SKIP 2 FIRST 1 * FROM BOOK ORDER BY ID

-- Ingres (almost the SQL:2008 standard)
SELECT * FROM BOOK ORDER BY ID OFFSET 2 FETCH FIRST 1 ROWS ONLY

-- Firebird
SELECT * FROM BOOK ORDER BY ID ROWS 2 TO 3

-- Sybase SQL Anywhere
SELECT TOP 1 START AT 3 * FROM BOOK ORDER BY ID

-- DB2 (almost the SQL:2008 standard, without OFFSET)
SELECT * FROM BOOK ORDER BY ID FETCH FIRST 1 ROWS ONLY

-- Sybase ASE, SQL Server 2008 (without OFFSET)
SELECT TOP 1 * FROM BOOK ORDER BY ID
```

Things get a little more tricky in those databases that have no native idiom for OFFSET pagination (actual queries may vary):

```
-- DB2 (with OFFSET), SQL Server 2008 (with OFFSET)
SELECT * FROM (
  SELECT BOOK.*,
    ROW_NUMBER() OVER (ORDER BY ID ASC) AS RN
  FROM BOOK
) AS X
WHERE RN > 2
AND RN <= 3

-- DB2 (with OFFSET), SQL Server 2008 (with OFFSET)
SELECT * FROM (
  SELECT DISTINCT BOOK.ID, BOOK.TITLE,
    DENSE_RANK() OVER (ORDER BY ID ASC, TITLE ASC) AS RN
  FROM BOOK
) AS X
WHERE RN > 2
AND RN <= 3

-- Oracle 11g and less
SELECT *
FROM (
  SELECT b.*, ROWNUM RN
  FROM (
    SELECT *
    FROM BOOK
    ORDER BY ID ASC
  ) b
  WHERE ROWNUM <= 3
)
WHERE RN > 2
```

As you can see, jOOQ will take care of the incredibly painful ROW_NUMBER() OVER() (or ROWNUM for Oracle) filtering in subselects for you, you'll just have to write limit(1).offset(2) in any dialect.

SQL Server's ORDER BY, TOP and subqueries

As can be seen in the above example, writing correct SQL can be quite tricky, depending on the SQL dialect. For instance, with SQL Server, you cannot have an ORDER BY clause in a subquery, unless you also have a TOP clause. This is illustrated by the fact that jOOQ renders a TOP 100 PERCENT clause for you. The same applies to the fact that ROW_NUMBER() OVER() needs an ORDER BY windowing clause, even if you don't provide one to the jOOQ query. By default, jOOQ adds ordering by the first column of your projection.

Keyset pagination

Note, the LIMIT clause can also be used with the SEEK clause for keyset pagination.

3.3.4.11. WITH TIES clause

The previous chapter talked about the LIMIT clause, which limits the result set to a certain number of rows. The SQL standard specifies the following syntax:

```
OFFSET m { ROW | ROWS }
FETCH { FIRST | NEXT } n { ROW | ROWS } { ONLY | WITH TIES }
```

By default, most users will use the semantics of the ONLY keyword, meaning a LIMIT 5 expression (or FETCH NEXT 5 ROWS ONLY expression) will result in at most 5 rows. The alternative clause WITH TIES will return at most 5 rows, except if the 5th row and the 6th row (and so on) are "tied" according to the ORDER BY clause, meaning that the ORDER BY clause does not deterministically produce a 5th or 6th row. For example, let's look at our book table:

```
SELECT *
FROM book
ORDER BY author_id
FETCH NEXT 1 ROWS WITH TIES
```

```
DSL.using(configuration)
   .selectFrom(BOOK)
   .orderBy(BOOK.AUTHOR_ID)
   .limit(1).withTies()
   .fetch();
```

Resulting in:

```
+----+----------+-------------+
| id | actor_id | title       |
+----+----------+-------------+
|  1 |        1 | 1984        |
|  2 |        1 | Animal Farm |
+----+----------+-------------+
```

We're now getting two rows because both rows "tied" when ordering them by ACTOR_ID. The database cannot really pick the next 1 row, so they're both returned. If we omit the WITH TIES clause, then only a random one of the rows would be returned.

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).orderBy(BOOK.AUTHOR_ID).limit(1).withTies()
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Firebird, Hana, MySQL, Sybase, Vertica, YugabyteDB

```
SELECT v0 ID
FROM (
  SELECT
    BOOK.ID v0,
    rank() OVER (ORDER BY BOOK.AUTHOR_ID) rn
  FROM BOOK
) x
WHERE rn <= (0 + 1)
ORDER BY rn
```

BigQuery, Databricks, DuckDB, Exasol, Snowflake

```
SELECT BOOK.ID
FROM BOOK
QUALIFY rank() OVER (ORDER BY BOOK.AUTHOR_ID) <= (0 + 1)
```

ClickHouse, H2, MariaDB, Oracle, Postgres, Trino

```
SELECT BOOK.ID
FROM BOOK
ORDER BY BOOK.AUTHOR_ID
FETCH NEXT 1 ROWS WITH TIES
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
SELECT v0 ID
FROM (
  SELECT *
  FROM (
    SELECT
      BOOK.ID v0,
      rank() OVER (ORDER BY BOOK.AUTHOR_ID) rn
    FROM BOOK
  ) x
) x
WHERE rn <= (0 + 1)
ORDER BY rn
```

Redshift

```
SELECT BOOK.ID
FROM BOOK
WHERE 1 = 1
QUALIFY rank() OVER (ORDER BY BOOK.AUTHOR_ID) <= (0 + 1)
```

SQLDataWarehouse, SQLServer, Teradata

```
SELECT TOP 1 WITH TIES BOOK.ID
FROM BOOK
ORDER BY BOOK.AUTHOR_ID
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, MemSQL, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.12. SEEK clause

One of the previous chapters talked about OFFSET pagination using LIMIT .. OFFSET, or OFFSET .. FETCH or some other vendor-specific variant of the same. This can lead to significant performance issues when reaching a high page number, as all unneeded records need to be skipped by the database.

A much faster and more stable way to perform pagination is the so-called ***keyset pagination method*** also called ***seek method***. jOOQ supports a synthetic seek() clause, that can be used to perform keyset pagination (learn about other synthetic sql syntaxes). Imagine we have these data:

```
+------+-------+---------------+
|   ID | VALUE | PAGE_BOUNDARY |
+------+-------+---------------+
|  ... |   ... |           ... |
|  474 |     2 |             0 |
|  533 |     2 |             1 | <-- Before page 6
|  640 |     2 |             0 |
|  776 |     2 |             0 |
|  815 |     2 |             0 |
|  947 |     2 |             0 |
|   37 |     3 |             1 | <-- Last on page 6
|  287 |     3 |             0 |
|  450 |     3 |             0 |
|  ... |   ... |           ... |
+------+-------+---------------+
```

Now, if we want to display page 6 to the user, instead of going to page 6 by using a record OFFSET, we could just fetch the record strictly after the last record on page 5, which yields the values (533, 2). This is how you would do it with SQL or with jOOQ:

```
SELECT id, value
FROM t
WHERE (value, id) > (2, 533)
ORDER BY value, id
LIMIT 5
```

```
DSL.using(configuration)
   .select(T.ID, T.VALUE)
   .from(T)
   .orderBy(T.VALUE, T.ID)
   .seek(lastValue, lastId) // from last page: value = 2, id =
 533
   .limit(5)
   .fetch();
```

As you can see, the jOOQ SEEK clause is a synthetic clause that does not really exist in SQL. However, the jOOQ syntax is far more intuitive for a variety of reasons:

- It replaces OFFSET where you would expect
- It doesn't force you to mix regular predicates with *"seek"* predicates
- It is typesafe
- It emulates row value expression predicates for you, in those databases that do not support them

This query now yields:

```
+-----+-------+
|  ID | VALUE |
+-----+-------+
| 640 |     2 |
| 776 |     2 |
| 815 |     2 |
| 947 |     2 |
|  37 |     3 |
+-----+-------+
```

Note that you cannot combine the SEEK clause with the OFFSET clause.

More information about this great feature can be found in the jOOQ blog:

- https://blog.jooq.org/faster-sql-paging-with-jooq-using-the-seek-method/
- https://blog.jooq.org/faster-sql-pagination-with-keysets-continued/

Further information about offset pagination vs. keyset pagination performance can be found on our partner page:

3.3.4.13. FOR UPDATE clause

For inter-process synchronisation and other reasons, you may choose to use the SELECT .. FOR UPDATE clause to indicate to the database, that a set of cells or records should be locked by a given transaction for subsequent updates. With jOOQ, this can be achieved as such:

```
SELECT *
FROM BOOK
WHERE ID = 3
FOR UPDATE
```

```
create.select()
      .from(BOOK)
      .where(BOOK.ID.eq(3))
      .forUpdate()
      .fetch();
```

The above example will produce a record-lock, locking the whole record for updates. Some databases also support cell-locks using FOR UPDATE OF ..

```
SELECT *
FROM BOOK
WHERE ID = 3
FOR UPDATE OF TITLE
```

```
create.select()
      .from(BOOK)
      .where(BOOK.ID.eq(3))
      .forUpdate().of(BOOK.TITLE)
      .fetch();
```

Oracle goes a bit further and also allows to specify the actual locking behaviour. It features these additional clauses, which are all supported by jOOQ:

- FOR UPDATE NOWAIT: This is the default behaviour. If the lock cannot be acquired, the query fails immediately
- FOR UPDATE WAIT n: Try to wait for [n] seconds for the lock acquisition. The query will fail only afterwards
- FOR UPDATE SKIP LOCKED: This peculiar syntax will skip all locked records. This is particularly useful when implementing queue tables with multiple consumers

With jOOQ, you can use those Oracle extensions as such:

```
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().nowait().fetch();
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().wait(5).fetch();
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().skipLocked().fetch();
```

FOR UPDATE in SQL Server

(!) This behaviour has been changed in jOOQ 3.14, where we generate hints instead of updatable cursors.

The SQL standard specifies a FOR UPDATE clause to be applicable for cursors. Most databases interpret this as being applicable for all SELECT statements. An exception to this rule are the SQL Server database, that do not allow for any FOR UPDATE clause in a regular SQL SELECT statement. jOOQ emulates the FOR UPDATE behaviour, by locking record by record with JDBC. JDBC allows for specifying the flags TYPE_SCROLL_SENSITIVE, CONCUR_UPDATABLE for any statement, and then using ResultSet.updateXXX() methods to produce a cell-lock / row-lock. Here's a simplified example in JDBC:

```
try (
    PreparedStatement stmt = connection.prepareStatement(
        "SELECT * FROM author WHERE id IN (3, 4, 5)",
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stmt.executeQuery()
) {
    while (rs.next()) {
        // UPDATE the primary key for row-locks, or any other columns for cell-locks
        rs.updateObject(1, rs.getObject(1));
        rs.updateRow();

        // Do more stuff with this record
    }
}
```

The main drawback of this approach is the fact that the database has to maintain a scrollable cursor, whose records are locked one by one. This can cause a major risk of deadlocks or race conditions if the JDBC driver can recover from the unsuccessful locking, if two Java threads execute the following statements:

```
-- thread 1
SELECT * FROM author ORDER BY id ASC;

-- thread 2
SELECT * FROM author ORDER BY id DESC;
```

So use this technique with care, possibly only ever locking single rows!

Pessimistic (shared) locking with the FOR SHARE clause

Some databases (MySQL, Postgres) also allow to issue a non-exclusive lock explicitly using a FOR SHARE clause. This is also supported by jOOQ

Optimistic locking in jOOQ

Note, that jOOQ also supports optimistic locking, if you're doing simple CRUD. This is documented in the section's manual about optimistic locking.

3.3.4.14. Set operations

SQL allows to perform set operations as understood in standard set theory on result sets. These operations include unions, intersections, subtractions. For two subselects to be combinable by such a set operator, each subselect must return a table expression of the same degree and type.

All of these set operations come with 2 flavours:

- DISTINCT (the default): Removing duplicates after applying the set operation
- ALL: Retaining duplicates after applying the set operation

3.3.4.14.1. Type safety

Two subselects of degree less than 22 that are combined by a set operator are required to be of the same degree and, in most databases, also of the same type. jOOQ 3.0's introduction of Typesafe Record[N] types helps compile-checking these constraints:

```
// Some sample SELECT statements
Select<Record2<Integer, String>>  s1 = select(BOOK.ID, BOOK.TITLE).from(BOOK);
Select<Record1<Integer>>          s2 = selectOne();
Select<Record2<Integer, Integer>> s3 = select(one(), zero());
Select<Record2<Integer, String>>  s4 = select(one(), inline("abc"));

// Let's try to combine them:
s1.union(s2); // Doesn't compile because of a degree mismatch. Expected: Record2<...>, got: Record1<...>
s1.union(s3); // Doesn't compile because of a type mismatch. Expected: <Integer, String>, got: <Integer, Integer>
s1.union(s4); // OK. The two Record[N] types match
```

3.3.4.14.2. Projection rowtype

Much like most dialects use only the first set operation subquery's column names and types for the resulting row type, so does jOOQ.

Since jOOQ does not know which row is produced by which union subquery, it cannot disambiguate these rows in case the projection row type isn't exactly identical. As such, the ad-hoc converter in the following example is ignored:

```
Result<Record1<Integer>> result =
create.select(BOOK.ID)
   .from(BOOK)
   .union(

    // This has no effect
    select(AUTHOR.ID.convertFrom(i -> -i))
   .from(AUTHOR))
   .fetch();
```

While this can lead to subtle bugs, it makes perfect sense, knowing that a Converter is always applied at the client side of the execution.

3.3.4.14.3. Differences to standard SQL

As previously mentioned in the manual's section about the ORDER BY clause, jOOQ has slightly changed the semantics of these set operators. While in SQL, a set operation subselect may not immediately contain any ORDER BY clause or LIMIT clause (unless you wrap the subselect into a derived table), jOOQ allows you to do so. In order to select both the youngest and the oldest author from the database, you can issue the following statement with jOOQ (rendered to the MySQL dialect):

```
  (SELECT * FROM AUTHOR
   ORDER BY DATE_OF_BIRTH ASC LIMIT 1)
UNION
  (SELECT * FROM AUTHOR
   ORDER BY DATE_OF_BIRTH DESC LIMIT 1)
ORDER BY 1
```

```
create.selectFrom(AUTHOR)
      .orderBy(AUTHOR.DATE_OF_BIRTH.asc()).limit(1)
      .union(
       selectFrom(AUTHOR)
      .orderBy(AUTHOR.DATE_OF_BIRTH.desc()).limit(1))
      .orderBy(1)
      .fetch();
```

In case your database doesn't support ordered UNION subselects, the subselects are nested in derived tables.

```
SELECT * FROM (
  SELECT * FROM AUTHOR
  ORDER BY DATE_OF_BIRTH ASC LIMIT 1
)
UNION
SELECT * FROM (
  SELECT * FROM AUTHOR
  ORDER BY DATE_OF_BIRTH DESC LIMIT 1
)
ORDER BY 1
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).orderBy(BOOK.ID).limit(1).union(select(AUTHOR.ID).from(AUTHOR).orderBy(AUTHOR.ID).limit(1)).orderBy(1)
```

Translates to the following dialect specific expressions:

Access, SQLDataWarehouse, Sybase

```
(
  SELECT TOP 1 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
)
UNION (
  SELECT TOP 1 AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
)
ORDER BY 1
```

ASE, SQLServer

```
SELECT *
FROM (
  SELECT TOP 1 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
) x
UNION
SELECT *
FROM (
  SELECT TOP 1 AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
) x
ORDER BY 1
```

© 2009 - 2025 by Data Geekery™ GmbH.

Aurora MySQL, Aurora Postgres, CockroachDB, DuckDB, Exasol, HSQLDB, Hana, MySQL, Redshift, Snowflake, Vertica, YugabyteDB

```
(
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  LIMIT 1
)
UNION (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  LIMIT 1
)
ORDER BY 1
```

BigQuery

```
(
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  LIMIT 1
)
UNION DISTINCT (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  LIMIT 1
)
ORDER BY 1
```

ClickHouse

```
SELECT
  t.*
FROM (
  (
    SELECT BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID
    LIMIT 1
  )
  UNION DISTINCT (
    SELECT AUTHOR.ID
    FROM AUTHOR
    ORDER BY AUTHOR.ID
    LIMIT 1
  )
) t
ORDER BY 1
```

Databricks

```
(
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  LIMIT CAST(1 AS int)
)
UNION (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  LIMIT CAST(1 AS int)
)
ORDER BY 1
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, Firebird, Oracle

```
SELECT *
FROM (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  FETCH NEXT 1 ROWS ONLY
) x
UNION
SELECT *
FROM (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  FETCH NEXT 1 ROWS ONLY
) x
ORDER BY 1
```

Derby, H2, MariaDB, Postgres, Trino

```
(
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  FETCH NEXT 1 ROWS ONLY
)
UNION (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  FETCH NEXT 1 ROWS ONLY
)
ORDER BY 1
```

Informix

```
SELECT *
FROM (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
) x
UNION
SELECT *
FROM (
  SELECT *
  FROM (
    SELECT FIRST 1 AUTHOR.ID
    FROM AUTHOR
    ORDER BY AUTHOR.ID
  ) x
) x
ORDER BY 1
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL

```
SELECT
  t.*
FROM (
  (
    SELECT BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID
    LIMIT 1
  )
  UNION (
    SELECT AUTHOR.ID
    FROM AUTHOR
    ORDER BY AUTHOR.ID
    LIMIT 1
  )
) t
ORDER BY 1
```

SQLite

```
SELECT *
FROM (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
  LIMIT 1
) x
UNION
SELECT *
FROM (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  LIMIT 1
) x
ORDER BY 1
```

Teradata

```
(
  SELECT TOP 1 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
)
UNION (
  SELECT *
  FROM (
    SELECT TOP 1 AUTHOR.ID
    FROM AUTHOR
    ORDER BY AUTHOR.ID
  ) x
)
ORDER BY 1
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.4. UNION

A UNION operation combines two subquery results of compatible row type into a single result. UNION DISTINCT semantics is implied, i.e. all duplicate records resulting from this combination are removed.

Typically, you should prefer UNION ALL over UNION, if you don't really need to remove duplicates, see also this section of the manual. The following example shows how to use such a UNION operation in jOOQ.

```
SELECT * FROM BOOK WHERE ID = 3
UNION
SELECT * FROM BOOK WHERE ID = 5
```

```
create.selectFrom(BOOK).where(BOOK.ID.eq(3))
      .union(
create.selectFrom(BOOK).where(BOOK.ID.eq(5)))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).union(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
UNION
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

BigQuery

```
SELECT BOOK.ID
FROM BOOK
UNION DISTINCT
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  UNION DISTINCT
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird

```
SELECT BOOK.ID
FROM BOOK
UNION
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY 1
```

MemSQL

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  UNION
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.5. UNION ALL

A UNION ALL operation combines two subquery results of compatible row type into a single result without removing duplicates. Typically, you should prefer UNION ALL over UNION, if you don't really need to remove duplicates, see also this section of the manual. The following example shows how to use such a UNION ALL operation in jOOQ.

```
SELECT * FROM BOOK WHERE ID = 3
UNION ALL
SELECT * FROM BOOK WHERE ID = 5
```

```
create.selectFrom(BOOK).where(BOOK.ID.eq(3))
      .unionAll(
create.selectFrom(BOOK).where(BOOK.ID.eq(5)))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).unionAll(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres,

Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
UNION ALL
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse, MemSQL

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  UNION ALL
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

Firebird

```
SELECT BOOK.ID
FROM BOOK
UNION ALL
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY 1
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.6. INTERSECT

INTERSECT is the operation that produces only those values that are returned by both subselects. By default, this removes duplicate rows. Use INTERSECT ALL in order to retain them, and require duplicates to appear in both subqueries.

```
SELECT ID FROM BOOK
INTERSECT
SELECT ID FROM AUTHOR
```

```
create.select(BOOK.ID).from(BOOK)
      .intersect(
create.select(AUTHOR.ID).from(AUTHOR))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).intersect(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

ASE, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
INTERSECT
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

BigQuery

```
SELECT BOOK.ID
FROM BOOK
INTERSECT DISTINCT
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  INTERSECT DISTINCT
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

MemSQL

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  INTERSECT
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

Access, Aurora MySQL, Firebird, Redshift

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.7. INTERSECT ALL

INTERSECT ALL is the operation that produces only those values that are returned by both subselects without removing duplicates.

```
SELECT ID FROM BOOK
INTERSECT ALL
SELECT ID FROM AUTHOR
```

```
create.select(BOOK.ID).from(BOOK)
      .intersectAll(
create.select(AUTHOR.ID).from(AUTHOR))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).intersectAll(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, HSQLDB, MariaDB, MySQL, Oracle, Postgres, Sybase, Teradata, Trino, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
INTERSECT ALL
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  INTERSECT ALL
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

ASE, Access, Aurora MySQL, BigQuery, Exasol, Firebird, H2, Hana, Informix, MemSQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.8. EXCEPT

EXCEPT (or MINUS in Oracle) is the operation that returns only those values that are returned exclusively in the first subselect. By default, this removes duplicate rows. Use EXCEPT ALL in order to retain them, and require duplicates to appear in both subqueries.

```
SELECT ID FROM BOOK
EXCEPT
SELECT ID FROM AUTHOR
```

```
create.select(BOOK.ID).from(BOOK)
      .except(
create.select(AUTHOR.ID).from(AUTHOR))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).except(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

ASE, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Postgres, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
EXCEPT
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

BigQuery

```
SELECT BOOK.ID
FROM BOOK
EXCEPT DISTINCT
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  EXCEPT DISTINCT
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  EXCEPT
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

Oracle

```
SELECT BOOK.ID
FROM BOOK
MINUS
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

Access, Aurora MySQL, Firebird, Redshift

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.14.9. EXCEPT ALL

EXCEPT ALL is the operation that returns only those values that are returned exclusively in the first subselect without removing duplicates.

```
SELECT ID FROM BOOK
EXCEPT ALL
SELECT ID FROM AUTHOR
```

```
create.select(BOOK.ID).from(BOOK)
      .exceptAll(
create.select(AUTHOR.ID).from(AUTHOR))
      .fetch();
```

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(BOOK).exceptAll(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, HSQLDB, MariaDB, MySQL, Oracle, Postgres, Sybase, Teradata, Trino, YugabyteDB

```
SELECT BOOK.ID
FROM BOOK
EXCEPT ALL
SELECT AUTHOR.ID
FROM AUTHOR
ORDER BY ID
```

ClickHouse

```
SELECT
  t.*
FROM (
  SELECT BOOK.ID
  FROM BOOK
  EXCEPT ALL
  SELECT AUTHOR.ID
  FROM AUTHOR
) t
ORDER BY ID
```

ASE, Access, Aurora MySQL, BigQuery, Exasol, Firebird, H2, Hana, Informix, MemSQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.4.15. Lexical and logical SELECT clause order

SQL has a lexical and a logical order of SELECT clauses. The lexical order of SELECT clauses is inspired by the English language. As SQL statements are commands for the database, it is natural to express a statement in an imperative tense, such as "SELECT this and that!".

Logical SELECT clause order

The logical order of SELECT clauses, however, does not correspond to the syntax. In fact, the logical order is this:

- The FROM clause: First, all data sources are defined and joined
- The WHERE clause: Then, data is filtered as early as possible
- The CONNECT BY clause: Then, data is traversed iteratively or recursively, to produce new tuples
- The GROUP BY clause: Then, data is reduced to groups, possibly producing new tuples if grouping functions like ROLLUP(), CUBE(), GROUPING SETS() are used
- The HAVING clause: Then, data is filtered again, based on aggregate functions
- The WINDOW clause: Then, window specifications and window functions are evaluated
- The QUALIFY clause: Then, data is filtered again, based on window functions
- The SELECT clause: Only now, the projection is evaluated.
- The DISTINCT clause: Duplicate projected rows are removed
- UNION, INTERSECT and EXCEPT clauses: Optionally, the above is repeated for several UNION-connected subqueries. Unless this is a UNION ALL clause, data is further reduced to remove duplicates
- The ORDER BY clause: Now, all remaining tuples are ordered
- The LIMIT clause: Then, a paginating view is created for the ordered tuples
- The FOR UPDATE clause: Finally, pessimistic locking is applied

The SQL Server documentation also explains this, with slightly different clauses:

- FROM
- ON
- JOIN
- WHERE
- GROUP BY
- WITH CUBE or WITH ROLLUP
- HAVING
- SELECT
- DISTINCT
- ORDER BY
- TOP

As can be seen, databases have to logically reorder a SQL statement in order to determine the best execution plan.

Alternative syntaxes: LINQ, SLICK

Some "higher-level" abstractions, such as C#'s LINQ or Scala's Slick try to inverse the lexical order of SELECT clauses to what appears to be closer to the logical order. The obvious advantage of moving the SELECT clause to the end is the fact that the projection type, which is the record type returned by the SELECT statement can be re-used more easily in the target environment of the internal domain specific language.

A LINQ example:

```
// LINQ-to-SQL looks somewhat similar to SQL
// AS clause    // FROM clause
From p          In db.Products

// WHERE clause
Where p.UnitsInStock <= p.ReorderLevel AndAlso Not p.Discontinued

// SELECT clause
Select p
```

A Slick example:

```
// "for" is the "entry-point" to the DSL
val q = for {

    // FROM clause   WHERE clause
    c <- Coffees     if c.supID === 101

// SELECT clause and projection to a tuple
} yield (c.name, c.price)
```

While this looks like a good and idiomatic idea at first, jOOQ's take here is that this only complicates translation to more advanced SQL statements while impairing readability for those users that are used to writing SQL. This is specifically true for Slick, which not only changed the SELECT clause order, but also heavily "integrated" SQL clauses with the Scala language.

jOOQ is designed to look just like SQL. For these reasons, the jOOQ DSL API is modelled in SQL's lexical order.

3.3.5. The INSERT statement

The INSERT statement is used to insert new records into a database table. The following sections describe the various operation modes of the jOOQ INSERT statement.

3.3.5.1. INSERT .. VALUES

INSERT .. VALUES with a single row

Records can either be supplied using a VALUES() constructor, or a SELECT statement. jOOQ supports both types of INSERT statements. An example of an INSERT statement using a VALUES() constructor is given here:

```
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse');
```

```
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(100, "Hermann", "Hesse")
      .execute();
```

Note that for explicit degrees up to 22, the VALUES() constructor provides additional typesafety. The following example illustrates this:

```
InsertValuesStep3<AuthorRecord, Integer, String, String> step =
  create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME);
    step.values("A", "B", "C");
         // ^^^ Doesn't compile, the expected type is Integer
```

INSERT .. VALUES with multiple rows

The SQL standard specifies that multiple rows can be supplied to the VALUES() constructor in an INSERT statement. Here's an example of a multi-record INSERT

```
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse'),
       (101, 'Alfred', 'Döblin');
```

```
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(100, "Hermann", "Hesse")
      .values(101, "Alfred", "Döblin")
      .execute()
```

jOOQ tries to stay close to actual SQL. In detail, however, Java's expressiveness is limited. That's why the values() clause is repeated for every record in multi-record inserts.

Some RDBMS do not support inserting several records in a single statement. In those cases, jOOQ emulates multi-record INSERTs using the following SQL:

```
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME)
SELECT 100, 'Hermann', 'Hesse' FROM DUAL UNION ALL
SELECT 101, 'Alfred', 'Döblin' FROM DUAL;
```

```
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(100, "Hermann", "Hesse")
      .values(101, "Alfred", "Döblin")
      .execute();
```

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR).columns(AUTHOR.ID, AUTHOR.LAST_NAME).values(100, "Hesse")
```

Translates to the following dialect specific expressions:

All dialects

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  100,
  'Hesse'
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.2. INSERT .. DEFAULT VALUES

A lesser-known syntactic feature of SQL is the INSERT .. DEFAULT VALUES statement, where a single record is inserted, containing only DEFAULT values for every row. It is written as such:

```
INSERT INTO AUTHOR
DEFAULT VALUES;
```

```
create.insertInto(AUTHOR)
      .defaultValues()
      .execute();
```

This can make a lot of sense in situations where you want to "reserve" a row in the database for an subsequent UPDATE statement within the same transaction. Or if you just want to send an event containing trigger-generated default values, such as IDs or timestamps.

The DEFAULT VALUES clause is not supported in all databases, but jOOQ can emulate it using the equivalent statement:

© 2009 - 2025 by Data Geekery™ GmbH.

```
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME, ...)
VALUES (
 DEFAULT,
 DEFAULT,
 DEFAULT, ...);
```

```
create.insertInto(
        AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME,
 AUTHOR.LAST_NAME, ...)
      .values(
       defaultValue(AUTHOR.ID),
       defaultValue(AUTHOR.FIRST_NAME),
       defaultValue(AUTHOR.LAST_NAME), ...)
      .execute();
```

The DEFAULT keyword (or DSL#defaultValue() method) can also be used for individual columns only, although that will have the same effect as leaving the column away entirely.

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR).defaultValues()
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, DB2, Databricks, Derby, MariaDB, MemSQL, MySQL, Oracle, Snowflake

```
INSERT INTO AUTHOR
VALUES (
  DEFAULT,
  DEFAULT,
  DEFAULT,
  DEFAULT,
  DEFAULT,
  DEFAULT
)
```

Aurora Postgres, ClickHouse, CockroachDB, DuckDB, Exasol, Firebird, H2, HSQLDB, Postgres, SQLDataWarehouse, SQLServer, Teradata, YugabyteDB

```
INSERT INTO AUTHOR
DEFAULT VALUES
```

Hana, Informix, Redshift, SQLite, Sybase, Vertica

```
INSERT INTO AUTHOR (FIRST_NAME, DATE_OF_BIRTH, YEAR_OF_BIRTH, DISTINGUISHED)
VALUES (
  NULL,
  NULL,
  NULL,
  NULL
)
```

Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.3. INSERT .. SET

MySQL (and some other RDBMS) allow for using a non-SQL-standard, UPDATE-like syntax for INSERT statements. This is also supported in jOOQ (and emulated for all databases), should you prefer that syntax. The above INSERT statement can also be expressed as follows:

```
create.insertInto(AUTHOR)
      .set(AUTHOR.ID, 100)
      .set(AUTHOR.FIRST_NAME, "Hermann")
      .set(AUTHOR.LAST_NAME, "Hesse")
      .execute();
```

Multi row variant:

```
create.insertInto(AUTHOR)
      .set(AUTHOR.ID, 100)
      .set(AUTHOR.FIRST_NAME, "Hermann")
      .set(AUTHOR.LAST_NAME, "Hesse")
      .newRecord()
      .set(AUTHOR.ID, 101)
      .set(AUTHOR.FIRST_NAME, "Alfred")
      .set(AUTHOR.LAST_NAME, "Döblin")
      .execute();
```

As you can see, this syntax is a bit more verbose, but also more readable, as every field can be matched with its value. Internally, the two syntaxes are strictly equivalent.

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR).set(AUTHOR.ID, 100).set(AUTHOR.LAST_NAME, "Hesse")
```

Translates to the following dialect specific expressions:

All dialects

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  100,
  'Hesse'
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.4. INSERT .. SELECT

In some occasions, you may prefer the INSERT SELECT syntax, for instance, when you copy records from one table to another:

```
create.insertInto(AUTHOR_ARCHIVE)
      .select(selectFrom(AUTHOR).where(AUTHOR.DECEASED.isTrue()))
      .execute();
```

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR).columns(AUTHOR.ID, AUTHOR.LAST_NAME).select(select(val(100), val("Hesse")))
```

Translates to the following dialect specific expressions:

Access

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM (
  SELECT count(*) dual
  FROM MSysResources
) AS dual
```

ASE, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, H2, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
```

Aurora MySQL, MemSQL

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM DUAL
```

DB2

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM SYSIBM.DUAL
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM SYSIBM.SYSDUMMY1
```

Firebird

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM RDB$DATABASE
```

Hana, Sybase

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM SYS.DUMMY
```

HSQLDB

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM (VALUES (1)) AS dual (dual)
```

Informix

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM (
  SELECT 1 AS dual
  FROM systables
  WHERE (tabid = 1)
) AS dual
```

Teradata

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 100, 'Hesse'
FROM (
  SELECT 1 AS "dual"
) AS "dual"
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.3.5.5. INSERT .. ON DUPLICATE KEY UPDATE

The MySQL database supports a very convenient way to INSERT or UPDATE a record. This is a non-standard extension to the SQL syntax, which is supported by jOOQ and emulated in other RDBMS, where this is possible (e.g. if they support the SQL standard MERGE statement). Here is an example how to use the ON DUPLICATE KEY UPDATE clause:

```
// Add a new author called "Koontz" with ID 3.
// If that ID is already present, update the author's name
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onDuplicateKeyUpdate()
      .set(AUTHOR.LAST_NAME, "Koontz")
      .execute();
```

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME).values(3, "X").onDuplicateKeyUpdate().set(AUTHOR.LAST_NAME, "X")
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MemSQL, MySQL

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON DUPLICATE KEY UPDATE
  AUTHOR.LAST_NAME = 'X'
```

Aurora Postgres, CockroachDB, DuckDB, Postgres, YugabyteDB

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON CONFLICT (ID)
DO UPDATE
SET
  LAST_NAME = 'X'
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYSIBM.DUAL
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Derby

```
MERGE INTO AUTHOR
USING SYSIBM.SYSDUMMY1
ON AUTHOR.ID = 3
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
```

Exasol

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Firebird

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM RDB$DATABASE
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

H2

```
MERGE INTO AUTHOR
USING (
  SELECT
    3 ID,
    'X' LAST_NAME
) t
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Hana

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
    FROM SYS.DUMMY
  )
) t
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

HSQLDB

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (VALUES (1)) AS dual (dual)
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Oracle

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
  )
) t
ON (AUTHOR.ID = t.ID)
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Redshift, Snowflake

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

SQLite

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON CONFLICT
DO UPDATE
SET
  LAST_NAME = 'X'
```

SQLServer

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
);
```

© 2009 - 2025 by Data Geekery™ GmbH.

Sybase

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYS.DUMMY
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Teradata

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

ASE, Access, BigQuery, ClickHouse, Databricks, Informix, SQLDataWarehouse, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.6. INSERT .. ON DUPLICATE KEY IGNORE

The MySQL database also supports an INSERT IGNORE INTO clause. This is supported by jOOQ using the more convenient SQL syntax variant of ON DUPLICATE KEY IGNORE:

```
// Add a new author called "Koontz" with ID 3.
// If that ID is already present, ignore the INSERT statement
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onDuplicateKeyIgnore()
      .execute();
```

If the underlying database doesn't have any way to "ignore" failing INSERT statements, (e.g. MySQL via INSERT IGNORE), jOOQ can emulate the statement using a MERGE statement, or using INSERT .. SELECT WHERE NOT EXISTS.

© 2009 - 2025 by Data Geekery™ GmbH.

(!) The MySQL INSERT IGNORE statement ignores more constraint violations than just duplicate keys, so the emulation isn't exactly equivalent, see #5211

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME).values(3, "X").onDuplicateKeyIgnore()
```

Translates to the following dialect specific expressions:

Access

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT t.v0, t.v1
FROM (
  SELECT
    3 v0,
    'X' v1
  FROM (
    SELECT count(*) dual
    FROM MSysResources
  ) AS dual
  WHERE NOT EXISTS (
    SELECT 1
    FROM AUTHOR
    WHERE AUTHOR.ID = 3
  )
) t
```

ASE, ClickHouse, SQLDataWarehouse, Trino, Vertica

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT t.v0, t.v1
FROM (
  SELECT
    3 v0,
    'X' v1
  WHERE NOT EXISTS (
    SELECT 1
    FROM AUTHOR
    WHERE AUTHOR.ID = 3
  )
) t
```

Aurora MySQL, MariaDB, MySQL

```
INSERT IGNORE INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
```

Aurora Postgres, CockroachDB, DuckDB, Postgres, SQLite, YugabyteDB

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON CONFLICT
DO NOTHING
```

BigQuery

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT t.v0, t.v1
FROM (
  SELECT
    3 v0,
    'X' v1
  FROM UNNEST([STRUCT(1 AS dual)]) AS dual
  WHERE NOT EXISTS (
    SELECT 1
    FROM AUTHOR
    WHERE AUTHOR.ID = 3
  )
) t
```

Databricks, Redshift, Snowflake

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

DB2

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYSIBM.DUAL
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
MERGE INTO AUTHOR
USING SYSIBM.SYSDUMMY1
ON AUTHOR.ID = 3
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
```

Exasol, MemSQL

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT t.v0, t.v1
FROM (
  SELECT
    3 v0,
    'X' v1
  FROM DUAL
  WHERE NOT EXISTS (
    SELECT 1
    FROM AUTHOR
    WHERE AUTHOR.ID = 3
  )
) t
```

Firebird

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM RDB$DATABASE
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

H2

```
MERGE INTO AUTHOR
USING (
  SELECT
    3 ID,
    'X' LAST_NAME
) t
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
    FROM SYS.DUMMY
  )
) t
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

HSQLDB

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (VALUES (1)) AS dual (dual)
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Informix

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT t.v0, t.v1
FROM (
  SELECT
    3 v0,
    'X' v1
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
  WHERE NOT EXISTS (
    SELECT 1
    FROM AUTHOR
    WHERE AUTHOR.ID = 3
  )
) t
```

Oracle

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
  )
) t
ON (AUTHOR.ID = t.ID)
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLServer

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
);
```

Sybase

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYS.DUMMY
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Teradata

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.7. INSERT .. ON CONFLICT

The PostgreSQL database offers an alternative syntax to MySQL's vendor specific INSERT .. ON DUPLICATE KEY syntax, which allows for specifying an explicit (reference by constraint name) or implicit (reference by column list) unique constraint for conflict resolution.

```
// Add a new author called "Koontz" with ID 3.
// If that ID is already present, update the author's name
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onConflict(AUTHOR.ID)
      .doUpdate()
      .set(AUTHOR.LAST_NAME, "Koontz")
      .execute();
```

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME).values(3, "X").onConflict(AUTHOR.ID).doUpdate().set(AUTHOR.LAST_NAME, "X")
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MemSQL, MySQL

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON DUPLICATE KEY UPDATE
  AUTHOR.LAST_NAME = 'X'
```

Aurora Postgres, CockroachDB, DuckDB, Postgres, SQLite, YugabyteDB

```
INSERT INTO AUTHOR (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
ON CONFLICT (ID)
DO UPDATE
SET
  LAST_NAME = 'X'
```

DB2

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYSIBM.DUAL
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Derby

```
MERGE INTO AUTHOR
USING SYSIBM.SYSDUMMY1
ON AUTHOR.ID = 3
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  3,
  'X'
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Exasol

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Firebird

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM RDB$DATABASE
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

H2

```
MERGE INTO AUTHOR
USING (
  SELECT
    3 ID,
    'X' LAST_NAME
) t
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Hana

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
    FROM SYS.DUMMY
  )
) t
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

HSQLDB

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (VALUES (1)) AS dual (dual)
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Oracle

```
MERGE INTO AUTHOR
USING (
  (
    SELECT
      3 ID,
      'X' LAST_NAME
  )
) t
ON (AUTHOR.ID = t.ID)
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Redshift, Snowflake

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

SQLServer

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
);
```

© 2009 - 2025 by Data Geekery™ GmbH.

Sybase

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM SYS.DUMMY
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  AUTHOR.LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

Teradata

```
MERGE INTO AUTHOR
USING (
  SELECT 3, 'X'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
) t (ID, LAST_NAME)
ON AUTHOR.ID = t.ID
WHEN MATCHED THEN UPDATE SET
  LAST_NAME = 'X'
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
VALUES (
  t.ID,
  t.LAST_NAME
)
```

ASE, Access, BigQuery, ClickHouse, Databricks, Informix, SQLDataWarehouse, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.5.8. INSERT .. RETURNING

The Postgres database has native support for an INSERT .. RETURNING clause. This is a very powerful concept that is emulated for all other dialects using JDBC's getGeneratedKeys() method. Take this example:

```
// Add another author, with a generated ID
Record record =
create.insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values("Charlotte", "Roche")
      .returningResult(AUTHOR.ID)
      .fetchOne();

System.out.println(record.getValue(AUTHOR.ID));

// For some RDBMS, this also works when inserting several values
// The following should return a 2x2 table
Result<?> result =
create.insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values("Johann Wolfgang", "von Goethe")
      .values("Friedrich", "Schiller")
      // You can request any field. Also trigger-generated values
      .returningResult(AUTHOR.ID, AUTHOR.CREATION_DATE)
      .fetch();
```

Some databases have poor support for returning generated keys after INSERTs. In those cases, jOOQ might need to issue another SELECT statement in order to fetch an @@identity value. Be aware, that this can lead to race-conditions in those databases that cannot properly return generated ID values. For more information, please consider the jOOQ Javadoc for the returningResult() clause.

Dialect support

This example using jOOQ:

```
insertInto(AUTHOR, AUTHOR.LAST_NAME).values("Doe").returningResult(AUTHOR.ID)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, Exasol, HSQLDB, Hana, Informix, MemSQL, MySQL, Oracle, Redshift, Snowflake, Sybase, Teradata, Trino, Vertica

```
INSERT INTO AUTHOR (LAST_NAME)
VALUES ('Doe')
```

Aurora Postgres, CockroachDB, DuckDB, Firebird, Postgres, SQLite, YugabyteDB

```
INSERT INTO AUTHOR (LAST_NAME)
VALUES ('Doe')
RETURNING AUTHOR.ID
```

DB2, H2

```
SELECT ID
FROM FINAL TABLE (
  INSERT INTO AUTHOR (LAST_NAME)
  VALUES ('Doe')
) AUTHOR
```

MariaDB

```
INSERT INTO AUTHOR (LAST_NAME)
VALUES ('Doe')
RETURNING ID
```

SQLDataWarehouse, SQLServer

```
INSERT INTO AUTHOR (LAST_NAME)
OUTPUT inserted.ID
VALUES ('Doe')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6. The UPDATE statement

The UPDATE statement is used to modify one or several pre-existing records in a database table. UPDATE statements are only possible on single tables. Support for multi-table updates might be implemented in the future. An example update query is given here:

```
UPDATE AUTHOR
   SET FIRST_NAME = 'Hermann',
       LAST_NAME = 'Hesse'
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(AUTHOR.FIRST_NAME, "Hermann")
      .set(AUTHOR.LAST_NAME, "Hesse")
      .where(AUTHOR.ID.eq(3))
      .execute();
```

The following subsections discuss the various subclauses of the UPDATE statement.

3.3.6.1. UPDATE .. SET

The SET clause allows for setting new values on updated records in a table.

Dialect support

This example using jOOQ:

```
update(BOOK).set(BOOK.TITLE, "New Title")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, DB2, Derby, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Oracle, SQLDataWarehouse, SQLServer, Sybase, Trino

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Postgres, Redshift, SQLite, Snowflake, Teradata, Vertica, YugabyteDB

```
UPDATE BOOK
SET
  TITLE = 'New Title'
```

BigQuery

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE TRUE
```

ClickHouse

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE TRUE
```

Hana

```
UPDATE BOOK
FROM BOOK
SET
  BOOK.TITLE = 'New Title'
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6.2. UPDATE .. SET ROW

The SET clause allows for setting ROW value expressions on updated records in a table.

```
UPDATE AUTHOR
   SET (FIRST_NAME, LAST_NAME) =
       ('Hermann',  'Hesse')
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME),
           row("Herman",          "Hesse"))
      .where(AUTHOR.ID.eq(3))
      .execute();
```

This can be particularly useful when using correlated subqueries in the SET clause, in case of which multiple columns can be updated with a single subquery, instead of only 1. See also UPDATE .. FROM for an alternative syntax for this scenario.

```
UPDATE AUTHOR
   SET (FIRST_NAME, LAST_NAME) = (
         SELECT PERSON.FIRST_NAME, PERSON.LAST_NAME
         FROM PERSON
         WHERE PERSON.ID = AUTHOR.ID
       )
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME),
           select(PERSON.FIRST_NAME, PERSON.LAST_NAME)
          .from(PERSON)
          .where(PERSON.ID.eq(AUTHOR.ID))
      )
      .where(AUTHOR.ID.eq(3))
      .execute();
```

The above row value expressions usages are completely typesafe.

Dialect support

This example using jOOQ:

```
update(BOOK).set(row(BOOK.TITLE, BOOK.LANGUAGE_ID), row("New Title", 1))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Derby, Exasol, Firebird, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, Sybase

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title',
  BOOK.LANGUAGE_ID = 1
```

Aurora Postgres, CockroachDB, DB2, H2, HSQLDB, Trino

```
UPDATE BOOK
SET
  (TITLE, LANGUAGE_ID) = ('New Title', 1)
```

BigQuery

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title',
  BOOK.LANGUAGE_ID = 1
WHERE TRUE
```

ClickHouse

```
UPDATE BOOK
SET
  TITLE = 'New Title',
  LANGUAGE_ID = 1
WHERE TRUE
```

Databricks, DuckDB, Redshift, SQLite, Snowflake, Teradata, Vertica

```
UPDATE BOOK
SET
  TITLE = 'New Title',
  LANGUAGE_ID = 1
```

Hana

```
UPDATE BOOK
FROM BOOK
SET
  (TITLE, LANGUAGE_ID) = ('New Title', 1)
```

Oracle

```
UPDATE BOOK
SET
  (TITLE, LANGUAGE_ID) = (
    SELECT 'New Title', 1
  )
```

Postgres, YugabyteDB

```
UPDATE BOOK
SET
  (TITLE, LANGUAGE_ID) = ROW ('New Title', 1)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6.3. UPDATE .. FROM

Some databases, including for example PostgreSQL and SQL Server, support joining additional tables to an UPDATE statement using a vendor-specific FROM clause. This is supported as well by jOOQ:

```
UPDATE BOOK_ARCHIVE
SET
  BOOK_ARCHIVE.TITLE = BOOK.TITLE
FROM BOOK
WHERE BOOK_ARCHIVE.ID = BOOK.ID
```

```
create.update(BOOK_ARCHIVE)
      .set(BOOK_ARCHIVE.TITLE, BOOK.TITLE)
      .from(BOOK)
      .where(BOOK_ARCHIVE.ID.eq(BOOK.ID))
      .execute();
```

In many cases, such a joined update statement can be emulated using a correlated subquery, or using updatable views. For example, most databases allow for using scalar subselects in UPDATE statements in one way or another. jOOQ models this through a set(Field<T>, Select<? extends Record1<T>>) method in the UPDATE DSL API, for convenience (see the section about scalar subqueries for more details):

```
UPDATE AUTHOR
   SET FIRST_NAME = (
         SELECT FIRST_NAME
         FROM PERSON
         WHERE PERSON.ID = AUTHOR.ID
       ),
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(AUTHOR.FIRST_NAME,
         select(PERSON.FIRST_NAME)
        .from(PERSON)
        .where(PERSON.ID.eq(AUTHOR.ID))
      )
      .where(AUTHOR.ID.eq(3))
      .execute();
```

Dialect support

This example using jOOQ:

```
update(BOOK_TO_BOOK_STORE).set(BOOK_TO_BOOK_STORE.STOCK,
 0).from(BOOK).where(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK.ID)).and(BOOK.AUTHOR_ID.eq(1))
```

Translates to the following dialect specific expressions:

ASE, BigQuery, Oracle, SQLServer, Sybase

```
UPDATE BOOK_TO_BOOK_STORE
SET
  BOOK_TO_BOOK_STORE.STOCK = 0
FROM BOOK
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
```

Aurora MySQL, MariaDB, MemSQL, MySQL, Trino

```
UPDATE BOOK_TO_BOOK_STORE
  CROSS JOIN BOOK
SET
  BOOK_TO_BOOK_STORE.STOCK = 0
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Aurora Postgres, CockroachDB, DuckDB, Postgres, Redshift, SQLite, Snowflake, YugabyteDB

```
UPDATE BOOK_TO_BOOK_STORE
SET
  STOCK = 0
FROM BOOK
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
```

ClickHouse

```
UPDATE BOOK_TO_BOOK_STORE
SET
  STOCK = 0
WHERE (NAME, BOOK_ID) IN (
  SELECT BOOK_TO_BOOK_STORE.NAME, BOOK_TO_BOOK_STORE.BOOK_ID
  FROM BOOK_TO_BOOK_STORE, BOOK
  WHERE (
    BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
    AND BOOK.AUTHOR_ID = 1
  )
)
```

Databricks

```
UPDATE BOOK_TO_BOOK_STORE
SET
  STOCK = 0
WHERE EXISTS (
  SELECT alias_1.v0, alias_1.v1
  FROM (
    SELECT
      BOOK_TO_BOOK_STORE.NAME v0,
      BOOK_TO_BOOK_STORE.BOOK_ID v1
    FROM BOOK_TO_BOOK_STORE, BOOK
    WHERE (
      BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
      AND BOOK.AUTHOR_ID = 1
    )
  ) alias_1
  WHERE (
    coalesce(BOOK_TO_BOOK_STORE.NAME),
    coalesce(BOOK_TO_BOOK_STORE.BOOK_ID)
  ) = (
    coalesce(alias_1.v0),
    coalesce(alias_1.v1)
  )
)
```

DB2, Derby, Exasol, Firebird, H2, HSQLDB

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = 0
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
UPDATE BOOK_TO_BOOK_STORE
FROM BOOK_TO_BOOK_STORE, BOOK
SET
  BOOK_TO_BOOK_STORE.STOCK = 0
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
```

SQLDataWarehouse

```
UPDATE BOOK_TO_BOOK_STORE
SET
  BOOK_TO_BOOK_STORE.STOCK = 0
WHERE EXISTS (
  SELECT alias_1.v0, alias_1.v1
  FROM (
    SELECT
      BOOK_TO_BOOK_STORE.NAME v0,
      BOOK_TO_BOOK_STORE.BOOK_ID v1
    FROM BOOK_TO_BOOK_STORE, BOOK
    WHERE (
      BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
      AND BOOK.AUTHOR_ID = 1
    )
  ) alias_1
  WHERE (
    BOOK_TO_BOOK_STORE.NAME = alias_1.v0
    AND BOOK_TO_BOOK_STORE.BOOK_ID = alias_1.v1
  )
)
```

Teradata, Vertica

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
  AND BOOK.AUTHOR_ID = 1
)
WHEN MATCHED THEN UPDATE SET
  STOCK = 0
```

Access, Informix

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6.4. UPDATE .. WHERE

The WHERE clause allows for adding a conditional expressions to the UPDATE statement, which restricts the rows to be updated.

Dialect support

This example using jOOQ:

```
update(BOOK).set(BOOK.TITLE, "New Title").where(BOOK.ID.eq(1))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, DB2, Derby, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Oracle, SQLDataWarehouse, SQLServer, Sybase, Trino

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID = 1
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Postgres, Redshift, SQLite, Snowflake, Teradata, Vertica, YugabyteDB

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE BOOK.ID = 1
```

ClickHouse

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE ID = 1
```

Hana

```
UPDATE BOOK
FROM BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID = 1
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6.5. UPDATE .. ORDER BY .. LIMIT

The ORDER BY and LIMIT clauses allow for updating only a subset of the data in a table, based on their ordering. This can be useful to implement queue semantics, e.g. to update only the top row, and possibly return it in one go.

Dialect support

This example using jOOQ:

```
update(BOOK).set(BOOK.TITLE, "New Title").orderBy(BOOK.ID.asc()).limit(1)
```

Translates to the following dialect specific expressions:

Aurora MySQL, DB2, MariaDB, MySQL, Trino

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
ORDER BY BOOK.ID ASC
LIMIT 1
```

Aurora Postgres, Databricks, DuckDB, Redshift, SQLite, Snowflake, Vertica, YugabyteDB

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

BigQuery, Exasol, HSQLDB

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

ClickHouse

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

CockroachDB

```
UPDATE BOOK
SET
  TITLE = 'New Title'
ORDER BY BOOK.ID ASC
LIMIT 1
```

Derby, H2, Oracle

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  FETCH NEXT 1 ROWS ONLY
)
```

Firebird

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
ORDER BY BOOK.ID ASC
ROWS 1
```

Hana

```
UPDATE BOOK
FROM BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT FIRST 1 BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
  ) x
)
```

MemSQL

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
    LIMIT 1
  ) t
)
```

Postgres

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  FETCH NEXT 1 ROWS ONLY
)
```

SQLDataWarehouse, SQLServer, Sybase

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT TOP 1 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
UPDATE BOOK
SET
  TITLE = 'New Title'
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT TOP 1 BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
  ) x
)
```

ASE, Access

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.6.6. UPDATE .. RETURNING

Various dialect support a RETURNING clause or something similar on their UPDATE statements, similar as the RETURNING clause in INSERT statements. This is useful to fetch trigger-generated values in one go. An example is given here:

```
-- Fetch a trigger-generated value
UPDATE BOOK
SET TITLE = 'Animal Farm'
WHERE ID = 5
RETURNING TITLE
```

```
String title = create.update(BOOK)
                   .set(BOOK.TITLE, "Animal Farm")
                   .where(BOOK.ID.eq(5))
                   .returningResult(BOOK.TITLE)
                   .fetchOne().getValue(BOOK.TITLE);
```

Dialect support

This example using jOOQ:

```
update(BOOK).set(BOOK.TITLE, "New Title").returningResult(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, SQLite, YugabyteDB

```
UPDATE BOOK
SET
  TITLE = 'New Title'
RETURNING BOOK.ID
```

DB2, H2

```
SELECT ID
FROM FINAL TABLE (
  UPDATE BOOK
  SET
    BOOK.TITLE = 'New Title'
) BOOK
```

Firebird

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
RETURNING BOOK.ID
```

MariaDB

```
INSERT INTO BOOK (
  ID,
  AUTHOR_ID,
  TITLE,
  PUBLISHED_IN,
  LANGUAGE_ID
)
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
ON DUPLICATE KEY UPDATE
  BOOK.TITLE = 'New Title'
RETURNING ID
```

Oracle

```
DECLARE
  o0 DBMS_SQL.NUMBER_TABLE;
  c0 sys_refcursor;
BEGIN
  UPDATE BOOK
  SET
    BOOK.TITLE = 'New Title'
  RETURNING BOOK.ID
  BULK COLLECT INTO o0;
  ? := SQL%ROWCOUNT;
  OPEN c0 FOR SELECT * FROM TABLE(o0);
  ? := c0;
END;
```

SQLServer

```
UPDATE BOOK
SET
  BOOK.TITLE = 'New Title'
OUTPUT inserted.ID
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MemSQL, MySQL, Redshift, SQLDataWarehouse, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.7. The DELETE statement

The DELETE statement removes records from a database table. DELETE statements are only possible on single tables. Support for multi-table deletes might be implemented in the future. An example delete query is given here:

```
DELETE AUTHOR
 WHERE ID = 100;
```

```
create.delete(AUTHOR)
      .where(AUTHOR.ID.eq(100))
      .execute();
```

The following subsections discuss the various subclauses of the DELETE statement.

3.3.7.1. DELETE .. WHERE

The WHERE clause allows for adding a conditional expressions to the DELETE statement, which restricts the rows to be deleted.

Dialect support

This example using jOOQ:

```
deleteFrom(BOOK).where(BOOK.ID.in(1, 2, 3))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  1, 2, 3
)
```

ClickHouse

```
DELETE FROM BOOK
WHERE ID IN (
  1, 2, 3
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.7.2. DELETE .. ORDER BY .. LIMIT

The ORDER BY and LIMIT clauses allow for deleting only a subset of the data in a table, based on their ordering. This can be useful to implement queue semantics, e.g. to delete only the top row, and possibly return it in one go.

Dialect support

This example using jOOQ:

```
deleteFrom(BOOK).orderBy(BOOK.ID.asc()).limit(1)
```

Translates to the following dialect specific expressions:

Aurora MySQL, CockroachDB, DB2, MariaDB, MySQL, Trino

```
DELETE FROM BOOK
ORDER BY BOOK.ID ASC
LIMIT 1
```

Aurora Postgres, BigQuery, Databricks, DuckDB, Exasol, HSQLDB, Hana, Redshift, SQLite, Snowflake, Vertica, YugabyteDB

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

ClickHouse

```
DELETE FROM BOOK
WHERE ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  LIMIT 1
)
```

Derby, H2, Oracle, Postgres

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
  FETCH NEXT 1 ROWS ONLY
)
```

Firebird

```
DELETE FROM BOOK
ORDER BY BOOK.ID ASC
ROWS 1
```

Informix

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT FIRST 1 BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
  ) x
)
```

MemSQL

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
    LIMIT 1
  ) t
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse, SQLServer, Sybase

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT TOP 1 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID ASC
)
```

Teradata

```
DELETE FROM BOOK
WHERE BOOK.ID IN (
  SELECT *
  FROM (
    SELECT TOP 1 BOOK.ID
    FROM BOOK
    ORDER BY BOOK.ID ASC
  ) x
)
```

ASE, Access

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.7.3. DELETE .. RETURNING

The RETURNING clause allows for returning expressions based on the deleted rows.

Dialect support

This example using jOOQ:

```
deleteFrom(BOOK).where(BOOK.ID.eq(1)).returningResult(BOOK.TITLE)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Firebird, Postgres, SQLite, YugabyteDB

```
DELETE FROM BOOK
WHERE BOOK.ID = 1
RETURNING BOOK.TITLE
```

DB2, H2

```
SELECT TITLE
FROM OLD TABLE (
  DELETE FROM BOOK
  WHERE BOOK.ID = 1
) BOOK
```

MariaDB

```
DELETE FROM BOOK
WHERE BOOK.ID = 1
RETURNING TITLE
```

Oracle

```
DECLARE
  o0 DBMS_SQL.VARCHAR2_TABLE;
  c0 sys_refcursor;
BEGIN
  DELETE FROM BOOK
  WHERE BOOK.ID = 1
  RETURNING BOOK.TITLE
  BULK COLLECT INTO o0;
  ? := SQL%ROWCOUNT;
  OPEN c0 FOR SELECT * FROM TABLE(o0);
  ? := c0;
END;
```

SQLServer

```
DELETE FROM BOOK
OUTPUT deleted.TITLE
WHERE BOOK.ID = 1
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MemSQL, MySQL, Redshift, SQLDataWarehouse, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.8. The MERGE statement

The MERGE statement is one of the most advanced standardised SQL features. While it can be used for UPSERT semantics (similar to INSERT .. ON DUPLICATE KEY UPDATE or INSERT .. ON CONFLICT), it is much more than that.

The point of the standard MERGE statement is to take a TARGET table, and merge (INSERT, UPDATE, or DELETE) data from a SOURCE table into it.

Here is an example assuming that we have a BOOK_TO_BOOK_STORE_STAGING table that we use to load data into the "actual" BOOK_TO_BOOK_STORE data table when doing ETL:

```
-- Merge staging data into the real table
MERGE INTO BOOK_TO_BOOK_STORE
USING (
  SELECT * FROM BOOK_TO_BOOK_STORE_STAGING
)
ON BOOK_TO_BOOK_STORE.BOOK_ID
   = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
AND BOOK_TO_BOOK_STORE.NAME
   = BOOK_TO_BOOK_STORE_STAGING.NAME
WHEN MATCHED THEN UPDATE
  SET STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
WHEN NOT MATCHED THEN INSERT (
  BOOK_ID,
  NAME,
  STOCK
)
VALUES (
  BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
  BOOK_TO_BOOK_STORE_STAGING.NAME,
  BOOK_TO_BOOK_STORE_STAGING.STOCK
);
```

```
create.mergeInto(BOOK_TO_BOOK_STORE)
      .using(BOOK_TO_BOOK_STORE_STAGING)
      .on(BOOK_TO_BOOK_STORE.BOOK_ID
          .eq(BOOK_TO_BOOK_STORE_STAGING.BOOK_ID)
      .and(BOOK_TO_BOOK_STORE.NAME
          .eq(BOOK_TO_BOOK_STORE_STAGING.NAME)))
      .whenMatchedThenUpdate().set(
          BOOK_TO_BOOK_STORE.STOCK,
          BOOK_TO_BOOK_STORE_STAGING.STOCK
      )
      .whenNotMatchedThenInsert(
           BOOK_TO_BOOK_STORE.BOOK_ID,
           BOOK_TO_BOOK_STORE.NAME,
           BOOK_TO_BOOK_STORE.STOCK
      )
      .values(
           BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
           BOOK_TO_BOOK_STORE_STAGING.NAME,
           BOOK_TO_BOOK_STORE_STAGING.STOCK
       )
      .execute();
```

Think of the MERGE statement as a RIGHT JOIN between TARGET and SOURCE tables, where the ON clause defines whether there is a MATCH between TARGET and SOURCE rows:

- If there's a MATCH, an UPDATE or DELETE operation on the TARGET row is possible
- If there's no MATCH, then the SOURCE row can be used to perform an INSERT operation on the TARGET table

The following sections explain the MERGE statement in detail.

3.3.8.1. USING .. ON

The USING .. ON clause specifies the MERGE statement SOURCE table and JOIN condition with the TARGET table, based on which the remaining clauses will be applied. By default, the SOURCE table is RIGHT JOIN-ed to the TARGET table, meaning that actions take place for every row in the SOURCE table.

Some dialects also support a BY SOURCE clause, which allows for LEFT JOIN or FULL JOIN semantics instead, allowing to act also on rows in the TARGET table for which there is no matching SOURCE row.

Dialect support

This example using jOOQ:

```
mergeInto(BOOK_TO_BOOK_STORE)
    .using(BOOK_TO_BOOK_STORE_STAGING)
    .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK_TO_BOOK_STORE_STAGING.BOOK_ID)
        .and(BOOK_TO_BOOK_STORE.NAME.eq(BOOK_TO_BOOK_STORE_STAGING.NAME)))
    .whenMatchedThenUpdate().set(BOOK_TO_BOOK_STORE.STOCK, BOOK_TO_BOOK_STORE_STAGING.STOCK)
```

Translates to the following dialect specific expressions:

Databricks, Postgres, Snowflake, Teradata, Vertica

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

DB2, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Sybase

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

Oracle

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON ((
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
))
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

Redshift

```
UPDATE BOOK_TO_BOOK_STORE
SET
  STOCK = CASE
    WHEN 1 = 1 THEN BOOK_TO_BOOK_STORE_STAGING.STOCK
    ELSE BOOK_TO_BOOK_STORE.STOCK
  END
FROM BOOK_TO_BOOK_STORE_STAGING
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLServer

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK;
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DuckDB, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.8.2. WHEN MATCHED THEN UPDATE

When there's a MATCH between SOURCE row and TARGET row, then the TARGET row can be updated with values from the SOURCE row, similar to what the UPDATE .. FROM clause would allow.

Dialect support

This example using jOOQ:

```
mergeInto(BOOK_TO_BOOK_STORE)
    .using(BOOK_TO_BOOK_STORE_STAGING)
    .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK_TO_BOOK_STORE_STAGING.BOOK_ID)
        .and(BOOK_TO_BOOK_STORE.NAME.eq(BOOK_TO_BOOK_STORE_STAGING.NAME)))
    .whenMatchedThenUpdate().set(BOOK_TO_BOOK_STORE.STOCK, BOOK_TO_BOOK_STORE_STAGING.STOCK)
```

Translates to the following dialect specific expressions:

Databricks, Postgres, Snowflake, Teradata, Vertica

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Sybase

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

Oracle

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON ((
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
))
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK
```

Redshift

```
UPDATE BOOK_TO_BOOK_STORE
SET
  STOCK = CASE
    WHEN 1 = 1 THEN BOOK_TO_BOOK_STORE_STAGING.STOCK
    ELSE BOOK_TO_BOOK_STORE.STOCK
  END
FROM BOOK_TO_BOOK_STORE_STAGING
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
```

SQLServer

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.STOCK = BOOK_TO_BOOK_STORE_STAGING.STOCK;
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DuckDB, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.8.3. WHEN MATCHED THEN DELETE

When there's a MATCH between SOURCE row and TARGET row, then the TARGET row can be deleted, similar to what the DELETE .. USING clause would allow.

Dialect support

This example using jOOQ:

```
mergeInto(BOOK_TO_BOOK_STORE)
    .using(BOOK_TO_BOOK_STORE_STAGING)
    .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK_TO_BOOK_STORE_STAGING.BOOK_ID)
        .and(BOOK_TO_BOOK_STORE.NAME.eq(BOOK_TO_BOOK_STORE_STAGING.NAME)))
    .whenMatchedThenDelete()
```

Translates to the following dialect specific expressions:

DB2, Databricks, Derby, Firebird, H2, HSQLDB, Postgres, Snowflake, Sybase

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN DELETE
```

Exasol

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE.NAME DELETE
```

Oracle

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON ((
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
))
WHEN MATCHED THEN UPDATE SET
  BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE.NAME DELETE
```

© 2009 - 2025 by Data Geekery™ GmbH.

Redshift

```
DELETE FROM BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
WHERE (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
```

SQLServer

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN DELETE;
```

Teradata

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN MATCHED THEN UPDATE SET
  NAME = BOOK_TO_BOOK_STORE.NAME DELETE
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DuckDB, Hana, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.3.8.4. WHEN NOT MATCHED THEN INSERT

When there's no MATCH between SOURCE row and TARGET table, then data derived from the SOURCE row can be inserted into the TARGET table, similar to what INSERT .. SELECT would allow.

Dialect support

This example using jOOQ:

```
mergeInto(BOOK_TO_BOOK_STORE)
    .using(BOOK_TO_BOOK_STORE_STAGING)
    .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK_TO_BOOK_STORE_STAGING.BOOK_ID)
        .and(BOOK_TO_BOOK_STORE.NAME.eq(BOOK_TO_BOOK_STORE_STAGING.NAME)))
    .whenNotMatchedThenInsert(
        BOOK_TO_BOOK_STORE.BOOK_ID,
        BOOK_TO_BOOK_STORE.NAME,
        BOOK_TO_BOOK_STORE.STOCK
    )
    .values(
        BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
        BOOK_TO_BOOK_STORE_STAGING.NAME,
        BOOK_TO_BOOK_STORE_STAGING.STOCK
    )
```

Translates to the following dialect specific expressions:

DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Postgres, Redshift, Snowflake, Sybase, Teradata, Vertica

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN NOT MATCHED THEN INSERT (BOOK_ID, NAME, STOCK)
VALUES (
  BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
  BOOK_TO_BOOK_STORE_STAGING.NAME,
  BOOK_TO_BOOK_STORE_STAGING.STOCK
)
```

Oracle

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON ((
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
))
WHEN NOT MATCHED THEN INSERT (BOOK_ID, NAME, STOCK)
VALUES (
  BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
  BOOK_TO_BOOK_STORE_STAGING.NAME,
  BOOK_TO_BOOK_STORE_STAGING.STOCK
)
```

SQLServer

```
MERGE INTO BOOK_TO_BOOK_STORE
USING BOOK_TO_BOOK_STORE_STAGING
ON (
  BOOK_TO_BOOK_STORE.BOOK_ID = BOOK_TO_BOOK_STORE_STAGING.BOOK_ID
  AND BOOK_TO_BOOK_STORE.NAME = BOOK_TO_BOOK_STORE_STAGING.NAME
)
WHEN NOT MATCHED THEN INSERT (BOOK_ID, NAME, STOCK)
VALUES (
  BOOK_TO_BOOK_STORE_STAGING.BOOK_ID,
  BOOK_TO_BOOK_STORE_STAGING.NAME,
  BOOK_TO_BOOK_STORE_STAGING.STOCK
);
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DuckDB, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4. SQL Statements (DDL)

The Data Definition Language (DDL) is used to CREATE, ALTER, and DROP various object types in the database catalog. jOOQ supports an increasing number of these operations natively, and also adds synthetic operation support for convenience.

While many DDL statements are supported natively, and have a 1:1 correspondence to the jOOQ API's representation, dialects differ in many subtle ways when it comes to DDL statement support. These differences may include:

- Different keywords to mean the same thing. For example, the keywords ALTER, CHANGE, or MODIFY may be used when altering columns or other attributes in a table.
- Different statements instead of subclauses. For example, some dialects may choose to support RENAME [object type] .. TO .. statements instead of making the rename operation a subclause of ALTER [object type] .. RENAME TO ..
- Some syntax may not be supported, or not be supported consistently, such as the various IF EXISTS and IF NOT EXISTS clauses. Emulations are possible using the dialect's procedural language

Because of these many differences, the jOOQ manual will not list each individual native SQL representation of each jOOQ API call. Also, some optional clauses may exist, such as the IF EXISTS or OR REPLACE clauses, which can easily be discovered from the API. The manual will omit documenting these clauses in every example.

Commercial support for emulations

A lot of DDL queries come with syntax that requires emulation using anonymous blocks. While basic anonymous blocks are supported in the jOOQ Open Source Edition as well, more sophisticated blocks and other procedural logic is a commercial only feature.

3.4.1. The ALTER statement

ALTER statements are used to alter properties of existing objects in the database catalog.

3.4.1.1. ALTER INDEX

The only property of an index that can be changed, currently, is its name.

3.4.1.1.1. ALTER INDEX .. RENAME

Like most object types, indexes can be renamed:

```
// Renaming an index with schema-scoped naming
create.alterIndex("old_index").renameTo("new_index").execute();

// Renaming an index with table-scoped naming
create.alterTable("t").renameIndex("old_index").renameTo("new_index").execute();
```

(!) Some RDBMS work with table-scoped index names, in case of which it is recommended to use the ALTER TABLE statement.

Dialect support

This example using jOOQ:

```
alterTable("t").renameIndex("i").to("j")
```

Translates to the following dialect specific expressions:

ASE

```
EXEC sp_rename 't.i', j, 'index'
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
ALTER TABLE t RENAME INDEX i TO j
```

Aurora Postgres, CockroachDB, H2, HSQLDB, Oracle, Postgres

```
ALTER INDEX i RENAME TO j
```

DB2, Derby, Hana

```
RENAME INDEX i TO j
```

SQLDataWarehouse, SQLServer

```
EXEC sp_rename 't.i', j, 'INDEX'
```

Access, BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Firebird, Informix, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.1.2. ALTER INDEX IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

```
// Renaming the index
create.alterIndexIfExists("old_index").renameTo("new_index").execute();
```

Dialect support

This example using jOOQ:

```
alterIndexIfExists("i").renameTo("j")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, H2, Postgres

```
ALTER INDEX IF EXISTS i RENAME TO j
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    RENAME INDEX i TO j
  ';
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 259 BEGIN END;
  EXECUTE IMMEDIATE '
    RENAME INDEX i TO j
  ';
END;
```

MariaDB

```
BEGIN NOT ATOMIC
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42000' BEGIN END;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' BEGIN END;
  ALTER TABLE  RENAME INDEX i TO j;
END
```

MySQL

```
CREATE PROCEDURE block_1751479554461_6662923()
MODIFIES SQL DATA
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42000' BEGIN END;
  ALTER TABLE  RENAME INDEX i TO j;
END;
CALL block_1751479554461_6662923();
DROP PROCEDURE block_1751479554461_6662923;
```

Oracle

```
BEGIN
  EXECUTE IMMEDIATE '
    ALTER INDEX i RENAME TO j
  ';
EXCEPTION
  WHEN others THEN
    IF sqlerrm LIKE 'ORA-01418%' THEN NULL;
    ELSE RAISE;
    END IF;
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse

```
BEGIN TRY
  EXEC sp_rename 'i', j, 'INDEX'
END TRY
BEGIN CATCH
  IF error_number() NOT IN (2714) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  EXEC sp_rename 'i', j, 'INDEX'
END TRY
BEGIN CATCH
  IF error_number() NOT IN (2714) THROW;
END CATCH
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Informix, MemSQL, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.2. ALTER SCHEMA

The only property of a schema that can be changed, currently, is its name.

3.4.1.2.1. ALTER SCHEMA .. RENAME

Like most object types, schemas can be renamed:

```
// Renaming the schema
create.alterSchema("old_schema").renameTo("new_schema").execute();
```

Dialect support

This example using jOOQ:

```
alterSchema("s").renameTo("t")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, H2, HSQLDB, Postgres, Redshift, Snowflake, Vertica

```
ALTER SCHEMA s RENAME TO t
```

ClickHouse

```
RENAME DATABASE s TO t
```

Hana

```
RENAME SCHEMA s TO t
```

ASE, Access, Aurora MySQL, BigQuery, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, Informix, MariaDB, MemSQL, MySQL, Oracle, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.2.2. ALTER SCHEMA IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

```
// Renaming the schema
create.alterSchemaIfExists("old_schema").renameTo("new_schema").execute();
```

Dialect support

This example using jOOQ:

```
alterSchemaIfExists("s").renameTo("t")
```

Translates to the following dialect specific expressions:

H2, Snowflake

```
ALTER SCHEMA IF EXISTS s RENAME TO t
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 362 BEGIN END;
  EXECUTE IMMEDIATE '
    RENAME SCHEMA s TO t
  ';
END;
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.3. ALTER SEQUENCE

Most sequence properties can be altered after a sequence has been created.

3.4.1.3.1. ALTER SEQUENCE .. RENAME

Like most object types, sequences can be renamed:

```
// Renaming the sequence
create.alterSequence("old_sequence").renameTo("new_sequence").execute();
```

Dialect support

This example using jOOQ:

```
alterSequence("s").renameTo("t")
```

Translates to the following dialect specific expressions:

Aurora Postgres, HSQLDB, Postgres, Snowflake, Vertica

```
ALTER SEQUENCE s RENAME TO t
```

MariaDB

```
ALTER TABLE s RENAME TO t
```

Oracle

```
RENAME s TO t
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, Hana, Informix, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.3.2. ALTER SEQUENCE .. RESTART

A sequence can be manually restarted using the RESTART clause, in case of which it will be restarted at the START WITH value, which can be altered as follows:

```
// Restart the sequence
create.alterSequence("s").restart().execute();
```

Alternatively, an explicit value can be provide to restart the sequence with:

```
// Restart the sequence
create.alterSequence("s").restartWith(10).execute();
```

Dialect support

This example using jOOQ:

```
alterSequence("s").restart()
```

Translates to the following dialect specific expressions:

Aurora Postgres, DB2, HSQLDB, MariaDB, Oracle, Postgres, SQLServer, YugabyteDB

```
ALTER SEQUENCE s RESTART
```

Hana, Informix, Vertica

```
ALTER SEQUENCE s RESTART WITH 1
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Snowflake, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.3.3. ALTER SEQUENCE IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

```
// Renaming the schema
create.alterSequenceIfExists("old_sequence").renameTo("new_sequence").execute();
```

Dialect support

This example using jOOQ:

```
alterSequenceIfExists("s").renameTo("t")
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, Snowflake

```
ALTER SEQUENCE IF EXISTS s RENAME TO t
```

MariaDB

```
ALTER TABLE s RENAME TO t
```

Oracle

```
BEGIN
  EXECUTE IMMEDIATE '
    RENAME s TO t
  ';
EXCEPTION
  WHEN others THEN
    IF sqlerrm LIKE 'ORA-04043%' THEN NULL;
    ELSE RAISE;
    END IF;
END;
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4. ALTER TABLE

The ALTER TABLE statement is certainly the most powerful among DDL statements, as tables are the most important object type in a database catalog.

3.4.1.4.1. ALTER TABLE .. ADD COLUMN

Most dialects support altering a table to add a single column per statement.

```
// Adding a single column to a table. These statements are equivalent
create.alterTable("table").add("column", INTEGER).execute();
create.alterTable("table").addColumn("column", INTEGER).execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").add("c", INTEGER)
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Informix, Teradata

```
ALTER TABLE t ADD c integer
```

ASE, Sybase

```
ALTER TABLE t ADD c int NULL
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Vertica, YugabyteDB

```
ALTER TABLE t ADD c int
```

BigQuery

```
ALTER TABLE t ADD COLUMN c int64
```

ClickHouse

```
ALTER TABLE t ADD COLUMN c Nullable(integer)
```

© 2009 - 2025 by Data Geekery™ GmbH.

CockroachDB

```
ALTER TABLE t ADD c int4
```

Databricks

```
ALTER TABLE t ADD COLUMN (c int)
```

Hana

```
ALTER TABLE t ADD (c integer)
```

Oracle, Snowflake

```
ALTER TABLE t ADD c number(10)
```

Trino

```
ALTER TABLE t ADD COLUMN c int
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.2. ALTER TABLE .. ADD COLUMNS

If multiple columns should be added atomically, and to save server round trips, some RDBMS support adding multiple columns to a table using a single ALTER TABLE statement.

```
// Adding several columns to a table in one go
create.alterTable("table").add(field(name("column1"), INTEGER), field(name("column2"), INTEGER)).execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").add(field("c1", INTEGER), field("c2", INTEGER))
```

Translates to the following dialect specific expressions:

ASE

```
ALTER TABLE t ADD
  c1 int NULL,
  c2 int NULL
```

Aurora MySQL, Aurora Postgres, MariaDB, MemSQL, MySQL, Postgres, YugabyteDB

```
ALTER TABLE t ADD c1 int,
ADD c2 int
```

BigQuery

```
ALTER TABLE t ADD COLUMN c1 int64,
ADD COLUMN c2 int64
```

ClickHouse

```
ALTER TABLE t ADD COLUMN c1 Nullable(integer),
ADD COLUMN c2 Nullable(integer)
```

CockroachDB

```
ALTER TABLE t ADD c1 int4,
ADD c2 int4
```

Databricks

```
ALTER TABLE t ADD COLUMN (
  c1 int,
  c2 int
)
```

DB2

```
ALTER TABLE t ADD c1 integer
ADD c2 integer
```

Firebird, Teradata

```
ALTER TABLE t ADD c1 integer,
ADD c2 integer
```

H2

```
ALTER TABLE t ADD (
  c1 int,
  c2 int
)
```

Hana, Informix

```
ALTER TABLE t ADD (
  c1 integer,
  c2 integer
)
```

Oracle, Snowflake

```
ALTER TABLE t ADD (
  c1 number(10),
  c2 number(10)
)
```

SQLDataWarehouse, SQLServer

```
ALTER TABLE t ADD
  c1 int,
  c2 int
```

Access, Derby, DuckDB, Exasol, HSQLDB, Redshift, SQLite, Sybase, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.3. ALTER TABLE .. ADD COLUMN IF NOT EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF NOT EXISTS clause:

Dialect support

This example using jOOQ:

```
alterTable("t").addIfNotExists("c", INTEGER)
```

Translates to the following dialect specific expressions:

Aurora Postgres, DuckDB, Exasol, H2, MariaDB, Postgres, YugabyteDB

```
ALTER TABLE t ADD IF NOT EXISTS c int
```

BigQuery

```
ALTER TABLE t ADD COLUMN IF NOT EXISTS c int64
```

ClickHouse

```
ALTER TABLE t ADD COLUMN IF NOT EXISTS c Nullable(integer)
```

CockroachDB

```
ALTER TABLE t ADD IF NOT EXISTS c int4
```

Databricks

```
ALTER TABLE t ADD COLUMN IF NOT EXISTS (c int)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42711' BEGIN END;
  EXECUTE IMMEDIATE '
    ALTER TABLE t ADD c integer
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    ALTER TABLE t ADD c integer
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 308 BEGIN END;
  EXECUTE IMMEDIATE '
    ALTER TABLE t ADD (c integer)
  ';
END;
```

Oracle

```
BEGIN
  EXECUTE IMMEDIATE '
    ALTER TABLE t ADD c number(10)
  ';
EXCEPTION
  WHEN others THEN
    IF sqlerrm LIKE 'ORA-01430%' THEN NULL;
    ELSE RAISE;
    END IF;
END;
```

SQLDataWarehouse

```
BEGIN TRY
  EXEC sp_executesql N'
    ALTER TABLE t ADD c int
  ';
END TRY
BEGIN CATCH
  IF error_number() NOT IN (2705) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  EXEC sp_executesql N'
    ALTER TABLE t ADD c int
  ';
END TRY
BEGIN CATCH
  IF error_number() NOT IN (2705) THROW;
END CATCH
```

Teradata

```
ALTER TABLE t ADD IF NOT EXISTS c integer
```

Trino

```
ALTER TABLE t ADD COLUMN IF NOT EXISTS c int
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Informix, MemSQL, MySQL, Redshift, SQLite, Snowflake, Sybase, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.4. ALTER TABLE .. ADD PRIMARY KEY

All constraint types can be added with the ALTER TABLE statement. This includes PRIMARY KEY constraints:

```
// Adding an unnamed constraint to a table
create.alterTable("table").add(primaryKey("id")).execute();

// Adding a named constraint to a table
create.alterTable("table").add(constraint("pk").primaryKey("id")).execute();
```

(!) It is recommended to always provide a name with a constraint to simplify schema management. ***See also this section of the manual***

Dialect support

This example using jOOQ:

```
alterTable("t").add(constraint("pk").primaryKey("id"))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t ADD CONSTRAINT pk PRIMARY KEY (id)
```

BigQuery

```
ALTER TABLE t ADD PRIMARY KEY (id) NOT ENFORCED
```

ClickHouse

```
ALTER TABLE t ADD PRIMARY KEY (id)
```

Informix

```
ALTER TABLE t ADD CONSTRAINT PRIMARY KEY (id) CONSTRAINT pk
```

SQLDataWarehouse

```
ALTER TABLE t ADD CONSTRAINT pk PRIMARY KEY NONCLUSTERED (id) NOT ENFORCED
```

DuckDB, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.5. ALTER TABLE .. ADD UNIQUE

All constraint types can be added with the ALTER TABLE statement. This includes UNIQUE constraints:

```
// Adding an unnamed constraint to a table
create.alterTable("table").add(unique("user_name")).execute();

// Adding a named constraint to a table
create.alterTable("table").add(constraint("uk").unique("user_name")).execute();
```

(!) It is recommended to always provide a name with a constraint to simplify schema management. ***See also this section of the manual***

Dialect support

This example using jOOQ:

```
alterTable("t").add(constraint("uk").unique("user_name"))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t ADD CONSTRAINT uk UNIQUE (user_name)
```

Informix

```
ALTER TABLE t ADD CONSTRAINT UNIQUE (user_name) CONSTRAINT uk
```

SQLDataWarehouse

```
ALTER TABLE t ADD CONSTRAINT uk UNIQUE (user_name) NOT ENFORCED
```

DuckDB, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.6. ALTER TABLE .. ADD FOREIGN KEY

All constraint types can be added with the ALTER TABLE statement. This includes FOREIGN KEY constraints:

```
// Adding an unnamed constraint to a table
create.alterTable("table").add(foreignKey("author_id").references("author")).execute();

// Adding a named constraint to a table
create.alterTable("table").add(constraint("fk").foreignKey("author_id").references("author")).execute();
```

(!) It is recommended to always provide a name with a constraint to simplify schema management. ***See also this section of the manual***

Dialect support

This example using jOOQ:

```
alterTable("t").add(constraint("pk").foreignKey("author_id").references("author"))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t ADD CONSTRAINT pk FOREIGN KEY (author_id) REFERENCES author
```

BigQuery, ClickHouse, SQLDataWarehouse

```
ALTER TABLE t ADD CONSTRAINT pk FOREIGN KEY (author_id) REFERENCES author NOT ENFORCED
```

Informix

```
ALTER TABLE t ADD CONSTRAINT FOREIGN KEY (author_id) REFERENCES author CONSTRAINT pk
```

DuckDB, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.7. ALTER TABLE .. ADD CHECK

All constraint types can be added with the ALTER TABLE statement. This includes CHECK constraints:

```
// Adding an unnamed constraint to a table
create.alterTable("table").add(check(length(field(name("c"), INTEGER)).gt(0))).execute();

// Adding a named constraint to a table
create.alterTable("table").add(constraint("ck").check(length(field(name("c"), INTEGER)).gt(0))).execute();
```

(!) It is recommended to always provide a name with a constraint to simplify schema management. ***See also this section of the manual***

Dialect support

This example using jOOQ:

```
alterTable("t").add(constraint("ck").check(field("c", INTEGER).gt(0)))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora Postgres, ClickHouse, CockroachDB, DB2, Derby, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, SQLServer, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t ADD CONSTRAINT ck CHECK (c > 0)
```

Informix

```
ALTER TABLE t ADD CONSTRAINT CHECK (c > 0) CONSTRAINT ck
```

© 2009 - 2025 by Data Geekery™ GmbH.

Aurora MySQL, BigQuery, Databricks, DuckDB, Exasol, MemSQL, Redshift, SQLDataWarehouse, SQLite, Snowflake, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.8. ALTER TABLE .. RENAME

Like most object types, tables can be renamed:

```
// Rename a table
create.alterTable("old_table").renameTo("new_table").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").renameTo("u")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, H2, HSQLDB, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
ALTER TABLE t RENAME TO u
```

ASE, SQLServer

```
EXEC sp_rename 't', u
```

ClickHouse, DB2, Derby, Exasol, Hana, Informix, Teradata

```
RENAME TABLE t TO u
```

SQLDataWarehouse

```
RENAME OBJECT t TO u
```

Sybase

```
ALTER TABLE t RENAME u
```

Firebird

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.9. ALTER TABLE .. COMMENT

For convenience, jOOQ supports MySQL's COMMENT syntax also on ALTER TABLE, which corresponds to the more standard COMMENT ON TABLE statement

```
// Specify a new comment on a table
create.alterTable("table").comment("a comment describing the table").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").comment("comment")
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MemSQL, MySQL

```
ALTER TABLE t COMMENT 'comment'
```

Aurora Postgres, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
COMMENT ON TABLE t IS 'comment'
```

BigQuery

```
ALTER TABLE t SET OPTIONS (DESCRIPTION = 'comment')
```

ClickHouse

```
ALTER TABLE t MODIFY COMMENT 'comment'
```

Snowflake

```
ALTER TABLE t SET COMMENT = 'comment'
```

SQLServer

```
BEGIN TRY
  DECLARE @u varchar(max) = schema_name();
  EXEC sp_addextendedproperty 'MS_Description', 'comment', 'schema', @u, 'table', 't', DEFAULT, DEFAULT
END TRY
BEGIN CATCH
  EXEC sp_updateextendedproperty 'MS_Description', 'comment', 'schema', @u, 'table', 't', DEFAULT, DEFAULT
END CATCH
```

ASE, Access, Derby, Informix, SQLDataWarehouse, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.10. ALTER TABLE .. ALTER COLUMN .. SET DEFAULT

A column DEFAULT value can be added to a column using the ALTER TABLE's SET DEFAULT clause on a column:

```
// Specify a new default value for a column
create.alterTable("table").alter("column").setDefault(1).execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").alter("c").setDefault(1)
```

Translates to the following dialect specific expressions:

Access, Aurora Postgres, CockroachDB, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Postgres, Snowflake, Sybase, YugabyteDB

```
ALTER TABLE t ALTER c SET DEFAULT 1
```

ASE

```
ALTER TABLE t REPLACE c DEFAULT 1
```

Aurora MySQL, BigQuery, Databricks, MariaDB, MySQL, Vertica

```
ALTER TABLE t ALTER COLUMN c SET DEFAULT 1
```

ClickHouse

```
ALTER TABLE t MODIFY COLUMN c DEFAULT 1
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL, Oracle

```
ALTER TABLE t MODIFY c DEFAULT 1
```

SQLServer

```
DECLARE @constraint NVARCHAR(max);
DECLARE @command NVARCHAR(max);

SELECT @constraint = name
FROM sys.default_constraints
WHERE parent_object_id = object_id('t')
AND parent_column_id = columnproperty(object_id('t'), 'c', 'ColumnId');

IF @constraint IS NOT NULL
BEGIN
  SET @command = 'ALTER TABLE ' + 't' + ' DROP CONSTRAINT ' + @constraint
  EXECUTE sp_executesql @command

  SET @command = 'ALTER TABLE ' + 't' + ' ADD CONSTRAINT ' + @constraint + ' DEFAULT ' + '1' + ' FOR ' + 'c'
  EXECUTE sp_executesql @command
END
ELSE
BEGIN
  SET @command = 'ALTER TABLE ' + 't' + ' ADD DEFAULT ' + '1' + ' FOR ' + 'c'
  EXECUTE sp_executesql @command
END
```

Hana, Informix, Redshift, SQLDataWarehouse, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.11. ALTER TABLE .. ALTER COLUMN .. SET NOT NULL

A column can be set to NOT NULL using the ALTER TABLE's NOT NULL clause on a column:

```
// Specify the not null constraint on a column
// Note that in some but not all dialects, the data type of the column needs to be known to jOOQ and the RDBMS
create.alterTable("table").alter(field("column", VARCHAR(10))).setNotNull().execute();
```

The existing data in the column must not contain any nulls or an error will be raised.

Dialect support

This example using jOOQ:

```
alterTable("t").alter(field("c", VARCHAR(10))).setNotNull()
```

Translates to the following dialect specific expressions:

ASE, Oracle

```
ALTER TABLE t MODIFY c NOT NULL
```

Aurora MySQL, MariaDB, MySQL

```
ALTER TABLE t CHANGE COLUMN c c varchar(10) NOT NULL
```

Aurora Postgres, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Postgres, Snowflake, YugabyteDB

```
ALTER TABLE t ALTER c SET NOT NULL
```

Databricks, Vertica

```
ALTER TABLE t ALTER COLUMN c SET NOT NULL
```

Exasol

```
ALTER TABLE t MODIFY c SET NOT NULL
```

MemSQL

```
ALTER TABLE t MODIFY c varchar(10) NOT NULL
```

Access, BigQuery, ClickHouse, Hana, Informix, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.12. ALTER TABLE .. ALTER COLUMN .. DROP NOT NULL

An existing column's NOT NULL constraint can be dropped using the ALTER TABLE's NOT NULL clause on a column:

```
// Drop the not null constraint on a column
// Note that in some but not all dialects, the data type of the column needs to be known to jOOQ and the RDBMS
create.alterTable("table").alter(field("column", VARCHAR(10))).dropNotNull().execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").alter(field("c", VARCHAR(10))).dropNotNull()
```

Translates to the following dialect specific expressions:

ASE, Oracle

```
ALTER TABLE t MODIFY c NULL
```

Aurora MySQL, MariaDB, MySQL

```
ALTER TABLE t CHANGE COLUMN c c varchar(10) NULL
```

Aurora Postgres, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Postgres, Snowflake, YugabyteDB

```
ALTER TABLE t ALTER c DROP NOT NULL
```

Databricks, Vertica

```
ALTER TABLE t ALTER COLUMN c DROP NOT NULL
```

Exasol

```
ALTER TABLE t MODIFY c DROP NOT NULL
```

MemSQL

```
ALTER TABLE t MODIFY c varchar(10) NULL
```

Access, BigQuery, ClickHouse, Hana, Informix, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.13. ALTER TABLE .. ALTER COLUMN .. SET TYPE

The type of a column can be changed using the ALTER TABLE's SET TYPE clause on a column:

```
// Set a new data type on the column
create.alterTable("table").alter("column").set(VARCHAR(50)).execute();
```

Whether this operation is supported for any given pair of existing/new types is vendor specific. If the existing data doesn't conform to the new type, an error is raised by the database.

Dialect support

This example using jOOQ:

```
alterTable("t").alter("c").set(VARCHAR(50))
```

Translates to the following dialect specific expressions:

Access

```
ALTER TABLE t ALTER c text(50)
```

ASE, Exasol, MemSQL

```
ALTER TABLE t MODIFY c varchar(50)
```

Aurora MySQL, MariaDB, MySQL

```
ALTER TABLE t CHANGE COLUMN c c varchar(50)
```

Aurora Postgres, Firebird, Postgres, YugabyteDB

```
ALTER TABLE t ALTER c TYPE varchar(50)
```

ClickHouse

```
ALTER TABLE t MODIFY COLUMN c Nullable(String(50))
```

CockroachDB

```
ALTER TABLE t ALTER c TYPE string(50)
```

Databricks

```
ALTER TABLE t ALTER COLUMN c TYPE varchar(50)
```

DB2, Derby, DuckDB

```
ALTER TABLE t ALTER c SET DATA TYPE varchar(50)
```

H2, HSQLDB, Snowflake, Sybase

```
ALTER TABLE t ALTER c varchar(50)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
ALTER TABLE t ALTER(c varchar(50))
```

Informix

```
ALTER TABLE t MODIFY c lvarchar(50)
```

Oracle

```
ALTER TABLE t MODIFY c varchar2(50)
```

SQLDataWarehouse, SQLServer

```
ALTER TABLE t ALTER COLUMN c varchar(50)
```

Trino

```
ALTER TABLE t ALTER COLUMN c SET DATA TYPE varchar(50)
```

BigQuery, Redshift, SQLite, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.14. ALTER TABLE .. RENAME COLUMN

Like most object types, columns can be renamed:

```
// Rename a column
create.alterTable("table").renameColumn("old_column").to("new_column").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").renameColumn("c").to("d")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
ALTER TABLE t RENAME COLUMN c TO d
```

ASE

```
EXEC sp_rename 't.c', d, 'column'
```

Derby, Hana, Informix

```
RENAME COLUMN c TO d
```

Firebird

```
ALTER TABLE t ALTER COLUMN c TO d
```

H2, HSQLDB

```
ALTER TABLE t ALTER COLUMN c RENAME TO d
```

MemSQL

```
ALTER TABLE t CHANGE c d
```

SQLServer

```
EXEC sp_rename 't.c', d, 'COLUMN'
```

Sybase, Teradata

```
ALTER TABLE t RENAME c TO d
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.15. ALTER TABLE .. RENAME CONSTRAINT

Like most object types, constraints can be renamed:

```
// Rename a constraint
create.alterTable("table").renameConstraint("old_constraint").to("new_constraint").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").renameConstraint("c").to("d")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, CockroachDB, DB2, H2, Informix, Oracle, Postgres, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Vertica, YugabyteDB

```
ALTER TABLE t RENAME CONSTRAINT c TO d
```

ASE

```
EXEC sp_rename 't.c', d, 'index'
```

HSQLDB

```
ALTER CONSTRAINT c RENAME TO d
```

Teradata

```
ALTER TABLE t RENAME c TO d
```

BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, Hana, MariaDB, MemSQL, MySQL, Redshift, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.16. ALTER TABLE .. RENAME INDEX

Like most object types, indexes can be renamed. As some RDBMS treat indexes as objects contained in tables, just like constraints, this syntax is offered also as an alternative to the more popular variant of the ALTER INDEX .. RENAME statement:

```
// Rename a index (as a convenience for the ALTER INDEX statement)
create.alterTable("table").renameIndex("old_index").to("new_index").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").renameIndex("i").to("j")
```

Translates to the following dialect specific expressions:

ASE

```
EXEC sp_rename 't.i', j, 'index'
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
ALTER TABLE t RENAME INDEX i TO j
```

Aurora Postgres, CockroachDB, H2, HSQLDB, Oracle, Postgres

```
ALTER INDEX i RENAME TO j
```

DB2, Derby, Hana

```
RENAME INDEX i TO j
```

SQLDataWarehouse, SQLServer

```
EXEC sp_rename 't.i', j, 'INDEX'
```

Access, BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Firebird, Informix, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.17. ALTER TABLE .. DROP COLUMN

A single column can be dropped from a table using the ALTER TABLE's DROP COLUMN clause:

```
// Drop a single column
create.alterTable("table").drop("column").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").drop("c")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLite, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t DROP c
```

ASE

```
ALTER TABLE t DROP c WITH NO DATACOPY
```

BigQuery, ClickHouse, Databricks, Oracle, SQLDataWarehouse, SQLServer, Trino

```
ALTER TABLE t DROP COLUMN c
```

Hana

```
ALTER TABLE t DROP (c)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.18. ALTER TABLE .. DROP COLUMN RESTRICT

An optional RESTRICT clause can be added to a DROP COLUMN clause to hint at the operation failing when there are objects depending on the column.

```
// Add a RESTRICT clause when dropping columns
create.alterTable("table").drop("column").restrict().execute();
```

In most RDBMS, RESTRICT semantics is the default when dropping columns, although implementations do not agree on what dependent objects are included in the RESTRICT check.

Dialect support

This example using jOOQ:

```
alterTable("t").drop("c").restrict()
```

Translates to the following dialect specific expressions:

CockroachDB, DB2, Derby, DuckDB, HSQLDB, Postgres, Redshift, YugabyteDB

```
ALTER TABLE t DROP c RESTRICT
```

Databricks

```
ALTER TABLE t DROP COLUMN c RESTRICT
```

Oracle

```
ALTER TABLE t DROP COLUMN c
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.19. ALTER TABLE .. DROP COLUMN CASCADE

An optional CASCADE clause can be added to a DROP COLUMN clause to cascade the DROP operatoin to dependent objects.

```
// Add a CASCADE clause when dropping columns
create.alterTable("table").drop("column").cascade().execute();
```

Note: Implementations do not agree on what dependent objects are included in the CASCADE operation.

Dialect support

This example using jOOQ:

```
alterTable("t").drop("c").cascade()
```

Translates to the following dialect specific expressions:

CockroachDB, DB2, Derby, HSQLDB, Postgres, Redshift, YugabyteDB

```
ALTER TABLE t DROP c CASCADE
```

Oracle

```
ALTER TABLE t DROP COLUMN c CASCADE CONSTRAINTS
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.20. ALTER TABLE .. DROP COLUMNS

Multiple columns can be dropped from a table using the ALTER TABLE's DROP COLUMNS clause:

```
// Drop several columns in one go
create.alterTable("table").drop("column1", "column2").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").drop("c1", "c2")
```

Translates to the following dialect specific expressions:

ASE

```
ALTER TABLE t DROP c1, c2 WITH NO DATACOPY
```

Aurora MySQL, Aurora Postgres, CockroachDB, Firebird, MariaDB, MemSQL, MySQL, Postgres, Teradata, YugabyteDB

```
ALTER TABLE t DROP c1, DROP c2
```

BigQuery, ClickHouse

```
ALTER TABLE t DROP COLUMN c1, DROP COLUMN c2
```

Databricks, SQLDataWarehouse, SQLServer

```
ALTER TABLE t DROP COLUMN c1, c2
```

DB2

```
ALTER TABLE t DROP c1 DROP c2
```

H2, Snowflake

```
ALTER TABLE t DROP c1, c2
```

Hana, Informix, Oracle

```
ALTER TABLE t DROP (c1, c2)
```

Access, Derby, DuckDB, Exasol, HSQLDB, Redshift, SQLite, Sybase, Trino, Vertica

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.21. ALTER TABLE .. DROP COLUMN IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

Dialect support

This example using jOOQ:

```
alterTable("t").dropIfExists("c")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, Exasol, H2, MariaDB, Postgres, Teradata, YugabyteDB

```
ALTER TABLE t DROP IF EXISTS c
```

BigQuery, ClickHouse, Databricks, Trino

```
ALTER TABLE t DROP COLUMN IF EXISTS c
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42703' BEGIN END;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    ALTER TABLE t DROP c
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    ALTER TABLE t DROP c
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 257 BEGIN END;
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 260 BEGIN END;
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 397 BEGIN END;
  EXECUTE IMMEDIATE '
    ALTER TABLE t DROP (c)
  ';
END;
```

Oracle

```
BEGIN
  EXECUTE IMMEDIATE '
    ALTER TABLE t DROP COLUMN c
  ';
EXCEPTION
  WHEN others THEN
    IF sqlerrm LIKE 'ORA-00904%' THEN NULL;
    ELSIF sqlerrm LIKE 'ORA-02443%' THEN NULL;
    ELSE RAISE;
    END IF;
END;
```

SQLDataWarehouse

```
BEGIN TRY
  EXEC sp_executesql N'
    ALTER TABLE t DROP COLUMN c
  ';
END TRY
BEGIN CATCH
  IF error_number() NOT IN (4924, 3728, 3727, 1750, 15248) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLServer

```
BEGIN TRY
  EXEC sp_executesql N'
    ALTER TABLE t DROP COLUMN c
  ';
END TRY
BEGIN CATCH
  IF error_number() NOT IN (4924, 3728, 3727, 1750, 15248) THROW;
END CATCH
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Informix, MemSQL, MySQL, Redshift, SQLite, Snowflake, Sybase, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.22. ALTER TABLE .. DROP CONSTRAINT

A constraint can be dropped from the table using the ALTER TABLE's DROP CONSTRAINT clause:

```
// Drop a constraint
create.alterTable("table").dropConstraint("uk").execute();
```

Alternative syntaxes are available to restrict dropping only to specific types of constraints:

- DROP PRIMARY KEY
- DROP FOREIGN KEY

Dialect support

This example using jOOQ:

```
alterTable("t").dropConstraint("c")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t DROP CONSTRAINT c
```

BigQuery, ClickHouse

```
ALTER TABLE t DROP PRIMARY KEY
```

Aurora MySQL, DuckDB, MemSQL, MySQL, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.23. ALTER TABLE .. DROP PRIMARY KEY

A PRIMARY KEY constraint can be dropped from the table using the ALTER TABLE's DROP PRIMARY KEY clause:

```
// Drop specific types of constraints
create.alterTable("table").dropPrimaryKey().execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").dropPrimaryKey()
```

Translates to the following dialect specific expressions:

Aurora MySQL, BigQuery, DB2, Databricks, Derby, Exasol, H2, HSQLDB, Hana, MariaDB, MySQL, Snowflake

```
ALTER TABLE t DROP PRIMARY KEY
```

Aurora Postgres, Postgres

```
DO $$
DECLARE
  n varchar;
BEGIN
  n := (
    SELECT constraint_name
    FROM information_schema.table_constraints
    WHERE (
      (table_name = 't')
      AND (constraint_type = 'PRIMARY KEY')
    )
  );
  EXECUTE ('alter table "t" drop constraint ' || n);
END;
$$
```

ASE, Access, ClickHouse, CockroachDB, DuckDB, Firebird, Informix, MemSQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.24. ALTER TABLE .. DROP FOREIGN KEY

A FOREIGN KEY constraint can be dropped from the table using the ALTER TABLE's DROP FOREIGN KEY clause:

```
// Drop specific types of constraints
create.alterTable("table").dropForeignKey("fk").execute();
```

Dialect support

This example using jOOQ:

```
alterTable("t").dropForeignKey("c")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora Postgres, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ALTER TABLE t DROP CONSTRAINT c
```

Aurora MySQL, MariaDB, MySQL

```
ALTER TABLE t DROP FOREIGN KEY c
```

BigQuery

```
ALTER TABLE t DROP PRIMARY KEY
```

ClickHouse, DuckDB, MemSQL, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.4.25. ALTER TABLE IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

```
// Renaming the table
create.alterTableIfExists("old_table").renameTo("new_table").execute();
```

Dialect support

This example using jOOQ:

```
alterTableIfExists("old").renameTo("new")
```

Translates to the following dialect specific expressions:

Aurora MySQL

```
ALTER TABLE old RENAME TO new
```

Aurora Postgres, BigQuery, CockroachDB, DuckDB, H2, Oracle, Postgres, Snowflake, Trino, YugabyteDB

```
ALTER TABLE IF EXISTS old RENAME TO new
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    RENAME TABLE old TO new
  ';
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 259 BEGIN END;
  EXECUTE IMMEDIATE '
    RENAME TABLE old TO new
  ';
END;
```

MariaDB

```
BEGIN NOT ATOMIC
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' BEGIN END;
  ALTER TABLE old RENAME TO new;
END
```

MySQL

```
CREATE PROCEDURE block_1751479554478_2048956()
MODIFIES SQL DATA
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' BEGIN END;
  ALTER TABLE old RENAME TO new;
END;
CALL block_1751479554478_2048956();
DROP PROCEDURE block_1751479554478_2048956;
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLServer

```
BEGIN TRY
  EXEC sp_executesql N'
    EXEC sp_rename ''old'', new
  ';
END TRY
BEGIN CATCH
  IF error_number() NOT IN (4902, 15165, 15225) THROW;
END CATCH
```

ASE, Access, ClickHouse, Databricks, Derby, Exasol, Firebird, HSQLDB, Informix, MemSQL, Redshift, SQLDataWarehouse, SQLite, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.5. ALTER VIEW

The ALTER VIEW statement allows of altering existing views. It supports the following subclauses:

3.4.1.5.1. ALTER VIEW .. COMMENT

This statement allows for changing the comment associated with a view. It is an alias for the COMMENT ON VIEW statement

Dialect support

This example using jOOQ:

```
alterView("v").comment("new comment")
```

Translates to the following dialect specific expressions:

Aurora Postgres, Firebird, Hana, Postgres, Redshift, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
COMMENT ON VIEW v IS 'new comment'
```

BigQuery

```
ALTER VIEW v SET OPTIONS (DESCRIPTION = 'new comment')
```

DB2, DuckDB, H2, HSQLDB, Oracle

```
COMMENT ON TABLE v IS 'new comment'
```

Snowflake

```
ALTER VIEW v SET COMMENT = 'new comment'
```

SQLServer

```
BEGIN TRY
  DECLARE @u varchar(max) = schema_name();
  EXEC sp_addextendedproperty 'MS_Description', 'new comment', 'schema', @u, 'view', 'v', DEFAULT, DEFAULT
END TRY
BEGIN CATCH
  EXEC sp_updateextendedproperty 'MS_Description', 'new comment', 'schema', @u, 'view', 'v', DEFAULT, DEFAULT
END CATCH
```

ASE, Access, Aurora MySQL, ClickHouse, CockroachDB, Databricks, Derby, Exasol, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.5.2. ALTER VIEW .. RENAME

This statement allows for renaming a view.

Dialect support

This example using jOOQ:

```
alterView("v").renameTo("new_name")
```

Translates to the following dialect specific expressions:

ASE, SQLServer

```
EXEC sp_rename v, new_name
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, H2, Postgres, Snowflake, Trino, Vertica

```
ALTER VIEW v RENAME TO new_name
```

Exasol, Teradata

```
RENAME VIEW v TO new_name
```

HSQLDB, Redshift, YugabyteDB

```
ALTER TABLE v RENAME TO new_name
```

Oracle

```
RENAME v TO new_name
```

Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Derby, Firebird, Hana, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Sybase

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.1.5.3. ALTER VIEW IF EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF EXISTS clause:

Dialect support

This example using jOOQ:

```
alterViewIfExists("v").renameTo("new_name")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Databricks, DuckDB, H2, Postgres

```
ALTER VIEW IF EXISTS v RENAME TO new_name
```

Oracle

```
BEGIN
  EXECUTE IMMEDIATE '
    RENAME v TO new_name
  ';
EXCEPTION
  WHEN others THEN
    IF sqlerrm LIKE 'ORA-00942%' THEN NULL;
    ELSIF sqlerrm LIKE 'ORA-04043%' THEN NULL;
    ELSE RAISE;
    END IF;
END;
```

Snowflake

```
ALTER VIEW v RENAME TO new_name
```

SQLServer

```
BEGIN TRY
  EXEC sp_rename v, new_name
END TRY
BEGIN CATCH
  IF error_number() NOT IN (15225) THROW;
END CATCH
```

YugabyteDB

```
ALTER TABLE IF EXISTS v RENAME TO new_name
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.2. The COMMENT statement

The COMMENT statement can be used to store a description for an object from the database catalog. It is available for the following types of object:

3.4.2.1. COMMENT ON COLUMN

This statement is used to comment on a column.

Dialect support

This example using jOOQ:

```
commentOnColumn(name("t", "col")).is("the comment")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
COMMENT ON COLUMN t.col IS 'the comment'
```

ClickHouse

```
ALTER TABLE t COMMENT COLUMN col 'the comment'
```

Databricks, Snowflake

```
ALTER TABLE t ALTER COLUMN col COMMENT 'the comment'
```

SQLServer

```
BEGIN TRY
  DECLARE @u varchar(max) = schema_name();
  EXEC sp_addextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'table', 't', 'column', 'col'
END TRY
BEGIN CATCH
  EXEC sp_updateextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'table', 't', 'column', 'col'
END CATCH
```

ASE, Access, Aurora MySQL, BigQuery, Derby, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.2.2. COMMENT ON TABLE

This statement is used to comment on a table.

Dialect support

This example using jOOQ:

```
commentOnTable("t").is("the comment")
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MemSQL, MySQL

```
ALTER TABLE t COMMENT = 'the comment'
```

Aurora Postgres, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
COMMENT ON TABLE t IS 'the comment'
```

BigQuery

```
ALTER TABLE t SET OPTIONS (DESCRIPTION = 'the comment')
```

ClickHouse

```
ALTER TABLE t MODIFY COMMENT 'the comment'
```

Snowflake

```
ALTER TABLE t SET COMMENT = 'the comment'
```

SQLServer

```
BEGIN TRY
  DECLARE @u varchar(max) = schema_name();
  EXEC sp_addextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'table', 't', DEFAULT, DEFAULT
END TRY
BEGIN CATCH
  EXEC sp_updateextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'table', 't', DEFAULT, DEFAULT
END CATCH
```

ASE, Access, Derby, Informix, SQLDataWarehouse, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.2.3. COMMENT ON VIEW

This statement is used to comment on a view.

Dialect support

This example using jOOQ:

```
commentOnView("v").is("the comment")
```

Translates to the following dialect specific expressions:

Aurora Postgres, Firebird, Hana, Postgres, Redshift, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
COMMENT ON VIEW v IS 'the comment'
```

BigQuery

```
ALTER VIEW v SET OPTIONS (DESCRIPTION = 'the comment')
```

ClickHouse

```
ALTER TABLE v MODIFY COMMENT 'the comment'
```

DB2, DuckDB, H2, HSQLDB, Oracle

```
COMMENT ON TABLE v IS 'the comment'
```

Snowflake

```
ALTER VIEW v SET COMMENT = 'the comment'
```

SQLServer

```
BEGIN TRY
  DECLARE @u varchar(max) = schema_name();
  EXEC sp_addextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'view', 'v', DEFAULT, DEFAULT
END TRY
BEGIN CATCH
  EXEC sp_updateextendedproperty 'MS_Description', 'the comment', 'schema', @u, 'view', 'v', DEFAULT, DEFAULT
END CATCH
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, CockroachDB, Databricks, Derby, Exasol, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3. The CREATE statement

The CREATE statement is the most important DDL statement. It allows for creating new objects in the database catalog.

3.4.3.1. CREATE INDEX

The CREATE INDEX statement allows for creating indexes on table columns.

CREATE INDEX

In its simplest form, the statement can be used like this:

```
// Create an index on a single column
create.createIndex("index").on("table", "column").execute();

// Create an index on several columns
create.createIndex("index").on("table", "column1", "column2").execute();
```

CREATE UNIQUE INDEX

In many dialects, there is a possibility of creating a unique index, which acts like a constraint (see ALTER TABLE or CREATE TABLE), but is not really a constraint. Most dialects will create an index automatically to enforce a UNIQUE constraint, so using a constraint instead may seem a bit cleaner. A UNIQUE INDEX is created like this:

```
// Create an index on a single column
create.createUniqueIndex("index").on("table", "column").execute();

// Create an index on several columns
create.createUniqueIndex("index").on("table", "column1", "column2").execute();
```

Sorted indexes

In most dialects, indexes have their columns sorted ascendingly by default. If you wish to create an index with a differing sort order, you can do so by providing the order explicitly:

```
// Create a sorted index on several columns
create.createIndex("index").on(
  table(name("table")),
  field(name("column1")).asc(),
  field(name("column2")).desc()
).execute();
```

Covering indexes (with INCLUDE clause)

A few dialects support an INCLUDE clause when creating an index. This can be useful to create covering indexes. These are indexes that "cover" the needs of an entire query, such that no secondary lookup needs to be done in a heap table or clustered index, after finding only parts of the projection in the index data structure. The data from the columns of the INCLUDE clause will be located only in the index leaf nodes (useful for projections), not in the index tree structure (useful for searches), which reduces index maintenance overhead, and index size.

If a dialect does not support this clause, jOOQ will simply add the INCLUDE columns into the ordinary index column list.

```
// Create a covering index with included columns
create.createIndex("index").on("table", "search_column").include("projection_column").execute();
```

Partial indexes (with WHERE clause)

A few dialects support a WHERE clause when creating an index. This is very useful to drastically reduce the size of an index, and thus index maintenance, if only parts of the data of a column need to be included in the index.

```
// Create a partial index
create.createIndex("index").on("table", "column").where(field(name("column")).gt(0)).execute();
```

3.4.3.2. CREATE SCHEMA

The CREATE SCHEMA statement is used to create a new schema in the database catalog.

```
// Create a schema
create.createSchema("new_schema").execute();
```

Dialect support

This example using jOOQ:

```
createSchema("s")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Vertica, YugabyteDB

```
CREATE SCHEMA s
```

ClickHouse

```
CREATE DATABASE s
```

Oracle

```
CREATE USER s NO AUTHENTICATION QUOTA UNLIMITED ON USERS
```

ASE, Access, Aurora MySQL, BigQuery, Firebird, Informix, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.3. CREATE SEQUENCE

The CREATE SEQUENCE statement is used to create a new sequence in the database catalog.

```
// Create a sequence with default flags
create.createSequence("sequence").execute();
```

Dialect support

This example using jOOQ:

```
createSequence("s")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, DuckDB, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, Oracle, Postgres, Snowflake, Sybase, Vertica, YugabyteDB

```
CREATE SEQUENCE s
```

Derby, SQLServer

```
CREATE SEQUENCE s START WITH 1
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Exasol, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.3.1. CREATE SEQUENCE IF NOT EXISTS

A popular subclause of DDL statements that jOOQ can usually emulate, is the IF NOT EXISTS clause:

Dialect support

This example using jOOQ:

```
createSequenceIfNotExists("s")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, H2, HSQLDB, Informix, MariaDB, Oracle, Postgres, Snowflake, Vertica, YugabyteDB

```
CREATE SEQUENCE IF NOT EXISTS s
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42710' BEGIN END;
  EXECUTE IMMEDIATE '
    CREATE SEQUENCE s
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    CREATE SEQUENCE s
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 324 BEGIN END;
  EXECUTE IMMEDIATE '
    CREATE SEQUENCE s
  ';
END;
```

SQLServer

```
BEGIN TRY
  CREATE SEQUENCE s START WITH 1
END TRY
BEGIN CATCH
  IF error_number() NOT IN (2714) THROW;
END CATCH
```

Sybase

```
BEGIN
  CREATE SEQUENCE s;
EXCEPTION
  WHEN others THEN
END;
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, Exasol, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4. CREATE TABLE

Arguably the most used DDL statement is the CREATE TABLE statement.

The following subsections discuss various usages of CREATE TABLE, as well as the relevant bits of meta data that can be added to a table.

3.4.3.4.1. Columns

All tables contain at least one column (except for some esoteric cases in PostgreSQL), and all SQL dialects support creating such tables:

```
// Create a new table with a column
create.createTable("table")
      .column("col1", INTEGER)
      .execute();
```

Dialect support

This example using jOOQ:

```
createTable("table").column("col1", INTEGER)
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Hana, Informix, Teradata

```
CREATE TABLE table (
  col1 integer
)
```

ASE, Sybase

```
CREATE TABLE table (
  col1 int NULL
)
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Trino, Vertica, YugabyteDB

```
CREATE TABLE table (
  col1 int
)
```

BigQuery

```
CREATE TABLE table (
  col1 int64
)
```

ClickHouse

```
CREATE TABLE table (
  col1 Nullable(integer)
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  col1 int4
)
```

Databricks

```
CREATE TABLE table (
  col1 int
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

Oracle, Snowflake

```
CREATE TABLE table (
  col1 number(10)
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.2. Nullability

Nullability is a property of a data type, and as such can be attached to the data type using various methods. The default nullability is RDBMS specific, so if you want to be vendor agnostic about nullability in your DDL, better always state it explicitly, for example:

```
// Specify nullability on columns
create.createTable("table")
      .column("vendor_specific_default", INTEGER)
      .column("explicit_nullable", INTEGER.nullable(false))
      .column("explicit_not_nullable", INTEGER.nullable(true))
      .execute();
```

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("vendor_specific_default", INTEGER)
      .column("explicit_nullable", INTEGER.null_())
      .column("explicit_not_nullable", INTEGER.notNull())
```

Translates to the following dialect specific expressions:

Access, DB2, Hana, Informix, Teradata

```
CREATE TABLE table (
  vendor_specific_default integer,
  explicit_nullable integer NULL,
  explicit_not_nullable integer NOT NULL
)
```

ASE, Sybase

```
CREATE TABLE table (
  vendor_specific_default int NULL,
  explicit_nullable int NULL,
  explicit_not_nullable int NOT NULL
)
```

Aurora MySQL, Aurora Postgres, DuckDB, Exasol, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Vertica, YugabyteDB

```
CREATE TABLE table (
  vendor_specific_default int,
  explicit_nullable int NULL,
  explicit_not_nullable int NOT NULL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

BigQuery

```
CREATE TABLE table (
  vendor_specific_default int64,
  explicit_nullable int64,
  explicit_not_nullable int64 NOT NULL
)
```

ClickHouse

```
CREATE TABLE table (
  vendor_specific_default Nullable(integer),
  explicit_nullable Nullable(integer),
  explicit_not_nullable integer
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  vendor_specific_default int4,
  explicit_nullable int4 NULL,
  explicit_not_nullable int4 NOT NULL
)
```

Databricks

```
CREATE TABLE table (
  vendor_specific_default int,
  explicit_nullable int,
  explicit_not_nullable int NOT NULL
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

Derby, H2, HSQLDB

```
CREATE TABLE table (
  vendor_specific_default int,
  explicit_nullable int,
  explicit_not_nullable int NOT NULL
)
```

Firebird

```
CREATE TABLE table (
  vendor_specific_default integer,
  explicit_nullable integer,
  explicit_not_nullable integer NOT NULL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Oracle, Snowflake

```
CREATE TABLE table (
  vendor_specific_default number(10),
  explicit_nullable number(10) NULL,
  explicit_not_nullable number(10) NOT NULL
)
```

Trino

```
CREATE TABLE table (
  vendor_specific_default int,
  explicit_nullable int,
  explicit_not_nullable int
)
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.4.3.4.3. Defaults

The DEFAULT expression on a column definition defines what value the column should contain if it is omitted in an INSERT statement, or if an explicit DEFAULT expression is used in INSERT or UPDATE. By default, this is NULL in most dialects

```
// Create a new table with a column with a default expression
create.createTable("table")
      .column("column1", INTEGER.default_(1))
      .execute();
```

To trigger this DEFAULT expression, you can run this, for example:

```
// Insert a row using the default expression
create.insertInto(table(name("table"))).defaultValues().execute();
```

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER.default_(1))
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Hana, Informix, Teradata

```
CREATE TABLE table (
  column1 integer DEFAULT 1
)
```

ASE

```
CREATE TABLE table (
  column1 int DEFAULT 1 NULL
)
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Vertica, YugabyteDB

```
CREATE TABLE table (
  column1 int DEFAULT 1
)
```

BigQuery

```
CREATE TABLE table (
  column1 int64 DEFAULT 1
)
```

ClickHouse

```
CREATE TABLE table (
  column1 Nullable(integer) DEFAULT 1
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  column1 int4 DEFAULT 1
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Databricks

```
CREATE TABLE table (
  column1 int DEFAULT 1
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

Oracle, Snowflake

```
CREATE TABLE table (
  column1 number(10) DEFAULT 1
)
```

SQLite

```
CREATE TABLE table (
  column1 int DEFAULT (1)
)
```

Sybase

```
CREATE TABLE table (
  column1 int NULL DEFAULT 1
)
```

Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.4. Identities

An IDENTITY is a special type of DEFAULT on a column, which is computed only on INSERT, and should usually not be replaced by user content. It computes a new value for a surrogate key. Most dialects default to using some system sequence based IDENTITY, though a UUID or some other unique value might work as well.

In jOOQ, it is currently only possible to specify whether a column is an IDENTITY at all, not to influence the value generation algorithm.

```
// Create a new table with a column with a default expression
create.createTable("table")
      .column("column1", INTEGER.identity(true))
      .execute();
```

Whether an IDENTITY also needs to be explicitly NOT NULL or a PRIMARY KEY is vendor specific. Ideally, both of these properties are set as well on identities.

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER.identity(true))
```

Translates to the following dialect specific expressions:

Access

```
CREATE TABLE table (
  column1 AUTOINCREMENT NOT NULL
)
```

ASE, Exasol

```
CREATE TABLE table (
  column1 int IDENTITY NOT NULL
)
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
CREATE TABLE table (
  column1 int NOT NULL AUTO_INCREMENT
)
```

Aurora Postgres

```
CREATE TABLE table (
  column1 SERIAL4 NOT NULL
)
```

CockroachDB

```
CREATE TABLE table (
  column1 int4 GENERATED BY DEFAULT AS IDENTITY NOT NULL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, Firebird

```
CREATE TABLE table (
  column1 integer GENERATED BY DEFAULT AS IDENTITY NOT NULL
)
```

Derby, Postgres, YugabyteDB

```
CREATE TABLE table (
  column1 int GENERATED BY DEFAULT AS IDENTITY NOT NULL
)
```

H2

```
CREATE TABLE table (
  column1 int NOT NULL GENERATED BY DEFAULT AS IDENTITY
)
```

Hana, Teradata

```
CREATE TABLE table (
  column1 integer NOT NULL GENERATED BY DEFAULT AS IDENTITY
)
```

HSQLDB

```
CREATE TABLE table (
  column1 int GENERATED BY DEFAULT AS IDENTITY(START WITH 1) NOT NULL
)
```

Informix

```
CREATE TABLE table (
  column1 SERIAL NOT NULL
)
```

Oracle

```
CREATE TABLE table (
  column1 number(10) GENERATED BY DEFAULT AS IDENTITY(START WITH 1) NOT NULL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Redshift, SQLDataWarehouse, SQLServer

```
CREATE TABLE table (
  column1 int IDENTITY(1, 1) NOT NULL
)
```

Snowflake

```
CREATE TABLE table (
  column1 number(10) IDENTITY NOT NULL
)
```

SQLite

```
CREATE TABLE table (
  column1 integer PRIMARY KEY AUTOINCREMENT NOT NULL
)
```

Sybase

```
CREATE TABLE table (
  column1 int NOT NULL IDENTITY
)
```

Vertica

```
CREATE TABLE table (
  column1 IDENTITY(1, 1) NOT NULL
)
```

BigQuery, ClickHouse, Databricks, DuckDB, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.5. Primary key

In a normalised database, all tables should have a PRIMARY KEY. In jOOQ, numerous features are enabled by tables that have one, including for example UpdatableRecords. To create a table with a primary key, write any of these:

```
// Create a new table with columns and unnamed constraints
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          primaryKey("column1")
      )
      .execute();

// Create a new table with columns and named constraints (recommended if you want to alter the constraint)
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("pk").primaryKey("column1")
      )
      .execute();
```

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("pk").primaryKey("column1")
      )
```

Translates to the following dialect specific expressions:

Access, Firebird, Hana

```
CREATE TABLE table (
  column1 integer,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

ASE, MariaDB, MemSQL, MySQL, SQLServer

```
CREATE TABLE table (
  column1 int NOT NULL,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, Postgres, Redshift, SQLite, YugabyteDB

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

BigQuery

```
CREATE TABLE table (
  column1 int64,
  PRIMARY KEY (column1) NOT ENFORCED
)
```

ClickHouse

```
CREATE TABLE table (
  column1 integer,
  PRIMARY KEY (column1)
)
ENGINE MergeTree()
```

CockroachDB

```
CREATE TABLE table (
  column1 int4,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

Databricks

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT pk PRIMARY KEY (column1)
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

DB2, Teradata

```
CREATE TABLE table (
  column1 integer NOT NULL,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
CREATE TABLE table (
  column1 integer,
  PRIMARY KEY (column1) CONSTRAINT pk
)
```

Oracle, Snowflake

```
CREATE TABLE table (
  column1 number(10),
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

SQLDataWarehouse

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT pk PRIMARY KEY NONCLUSTERED (column1) NOT ENFORCED
)
```

Sybase

```
CREATE TABLE table (
  column1 int NULL,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

Trino

```
CREATE TABLE table (
  column1 int
)
```

Vertica

```
CREATE TABLE table (
  column1 int,
  dummy int,
  CONSTRAINT pk PRIMARY KEY (column1)
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.6. Unique constraints

A candidate key that is not ideal for a Primary key should still be declared UNIQUE to enforce uniqueness, as well as for query performance reasons. In jOOQ, this can be done with the following approaches:

```
// Create a new table with columns and unnamed constraints
create.createTable("table")
      .column("column1", INTEGER)
      .column("column2", INTEGER)
      .column("column3", INTEGER)
      .constraints(
          unique("column1"),
          unique("column2", "column3")
      )
      .execute();

// Create a new table with columns and named constraints (recommended if you want to alter the constraint)
create.createTable("table")
      .column("column1", INTEGER)
      .column("column2", INTEGER)
      .column("column3", INTEGER)
      .constraints(
          constraint("uk1").unique("column1"),
          constraint("uk2").unique("column2", "column3")
      )
      .execute();
```

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("uk").unique("column1")
      )
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Hana, Teradata

```
CREATE TABLE table (
  column1 integer,
  CONSTRAINT uk UNIQUE (column1)
)
```

ASE, Sybase

```
CREATE TABLE table (
  column1 int NULL,
  CONSTRAINT uk UNIQUE (column1)
)
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLServer, SQLite, Vertica, YugabyteDB

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT uk UNIQUE (column1)
)
```

BigQuery

```
CREATE TABLE table (
  column1 int64
)
```

ClickHouse

```
CREATE TABLE table (
  column1 Nullable(integer)
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  column1 int4,
  CONSTRAINT uk UNIQUE (column1)
)
```

Databricks

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT uk UNIQUE (column1)
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

Informix

```
CREATE TABLE table (
  column1 integer,
  UNIQUE (column1) CONSTRAINT uk
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Oracle, Snowflake

```
CREATE TABLE table (
  column1 number(10),
  CONSTRAINT uk UNIQUE (column1)
)
```

SQLDataWarehouse

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT uk UNIQUE (column1) NOT ENFORCED
)
```

Trino

```
CREATE TABLE table (
  column1 int
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.7. Foreign keys

A foreign key is a tool that helps further normalise your database by guaranteeing that a referenced value exists in a parent table. In our sample database, it enforces the integrity of the BOOK.AUTHOR_ID reference. Besides integrity, it can be a very useful tool for optimising more sophisticated execution plans, e.g. to support JOIN elimination. In jOOQ, create foreign keys like this:

```
// Create a new table with columns and unnamed constraints
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          foreignKey("column1").references("other_table", "other_column1")
      )
      .execute();

// Create a new table with columns and named constraints (recommended if you want to alter the constraint)
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("fk").foreignKey("column1").references("other_table", "other_column1")
      )
      .execute();
```

jOOQ's code generator will pick up foreign keys for a variety of purposes, including navigational methods, the ON KEY joinsand most prominently, the very powerful implicit joins.

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("fk").foreignKey("column1").references("other_table", "other_column1")
      )
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Hana, Teradata

```
CREATE TABLE table (
  column1 integer,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
```

ASE, Sybase

```
CREATE TABLE table (
  column1 int NULL,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
```

Aurora MySQL, Aurora Postgres, Derby, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLServer, SQLite, Vertica, YugabyteDB

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
```

BigQuery

```
CREATE TABLE table (
  column1 int64,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1) NOT ENFORCED
)
```

ClickHouse

```
CREATE TABLE table (
  column1 Nullable(integer)
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  column1 int4,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
```

Databricks

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
```

Informix

```
CREATE TABLE table (
  column1 integer,
  FOREIGN KEY (column1) REFERENCES other_table (other_column1) CONSTRAINT fk
)
```

Oracle, Snowflake

```
CREATE TABLE table (
  column1 number(10),
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1)
)
```

SQLDataWarehouse

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT fk FOREIGN KEY (column1) REFERENCES other_table (other_column1) NOT ENFORCED
)
```

Trino

```
CREATE TABLE table (
  column1 int
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.8. Check constraints

A CHECK constraint is a simple, yet very effective means of enforcing data integrity on a row basis. Want to ensure a number is only ever positive? Use a CHECK constraint.

```
// Create a new table with columns and unnamed constraints
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          check(field(name("column1"), INTEGER).gt(0))
      )
      .execute();

// Create a new table with columns and named constraints (recommended if you want to alter the constraint)
create.createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("chk").check(field(name("column1"), INTEGER).gt(0))
      )
      .execute();
```

Just like the previous constraints, this one can be used by the optimiser to remove some redundant predicates, see e.g. this blog post.

Dialect support

This example using jOOQ:

```
createTable("table")
      .column("column1", INTEGER)
      .constraints(
          constraint("chk").check(field(name("column1"), INTEGER).gt(0))
      )
```

Translates to the following dialect specific expressions:

Access, DB2, Firebird, Hana, Teradata

```
CREATE TABLE table (
  column1 integer,
  CONSTRAINT chk CHECK (column1 > 0)
)
```

ASE, Sybase

```
CREATE TABLE table (
  column1 int NULL,
  CONSTRAINT chk CHECK (column1 > 0)
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Aurora Postgres, Derby, DuckDB, H2, HSQLDB, MariaDB, MySQL, Postgres, SQLServer, SQLite, Vertica, YugabyteDB

```
CREATE TABLE table (
  column1 int,
  CONSTRAINT chk CHECK (column1 > 0)
)
```

ClickHouse

```
CREATE TABLE table (
  column1 Nullable(integer),
  CONSTRAINT chk CHECK (column1 > 0)
)
ENGINE Log()
```

CockroachDB

```
CREATE TABLE table (
  column1 int4,
  CONSTRAINT chk CHECK (column1 > 0)
)
```

Informix

```
CREATE TABLE table (
  column1 integer,
  CHECK (column1 > 0) CONSTRAINT chk
)
```

Oracle

```
CREATE TABLE table (
  column1 number(10),
  CONSTRAINT chk CHECK (column1 > 0)
)
```

Aurora MySQL, BigQuery, Databricks, Exasol, MemSQL, Redshift, SQLDataWarehouse, Snowflake, Trino

```
/* UNSUPPORTED */
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.4.3.4.9. From a SELECT

Occasionally, creating a table from a SELECT statement is very useful, copying the source table's data types and data.

```
// Create a new table from a source SELECT statement
create.createTable("book_archive")
      .as(select(BOOK.ID, BOOK.TITLE).from(BOOK))
      .execute();

// Create a new table from a source SELECT statement and specify that data should be included, explicitly
create.createTable("book_archive")
      .as(select(BOOK.ID, BOOK.TITLE).from(BOOK))
      .withData()
      .execute();

// Create a new table from a source SELECT statement and specify that data should be excluded, explicitly
create.createTable("book_archive")
      .as(select(BOOK.ID, BOOK.TITLE).from(BOOK))
      .withNoData()
      .execute();
```

Dialect support

This example using jOOQ:

```
createTable("book_archive")
      .as(select(BOOK.ID, BOOK.TITLE).from(BOOK))
      .withNoData()
```

Translates to the following dialect specific expressions:

ASE, Access, SQLDataWarehouse, SQLServer

```
SELECT BOOK.ID, BOOK.TITLE
INTO book_archive
FROM BOOK
WHERE 1 = 0
```

Aurora MySQL, MemSQL, Oracle, Redshift, SQLite, Vertica

```
CREATE TABLE book_archive
AS
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WHERE 1 = 0
```

Aurora Postgres, Derby, Exasol, Postgres, YugabyteDB

```
CREATE TABLE book_archive
AS
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WITH NO DATA
```

© 2009 - 2025 by Data Geekery™ GmbH.

BigQuery

```
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
  WHERE FALSE
)
```

CockroachDB, DuckDB, H2, MariaDB, MySQL, Snowflake

```
CREATE TABLE book_archive
AS
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WHERE FALSE
```

Databricks

```
CREATE TABLE book_archive
TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.feature.allowColumnDefaults' = 'supported'
)
AS
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WHERE FALSE
```

DB2

```
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
) WITH NO DATA
```

HSQLDB, Hana

```
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
)
WITH NO DATA
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
  WHERE 1 = 0
)
WITH DATA
```

ClickHouse, Firebird, Informix, Sybase, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.4.10. Global temporary tables

A GLOBAL TEMPORARY table is a table whose meta data is shared among all sessions or transaction, but where each session or transaction contains its own data. It acts like a table with a session or transaction based policy enabled.

```
// Create a new temporary table
create.createGlobalTemporaryTable("book_archive")
      .column("column1", INTEGER)
      .execute();
```

Historically, jOOQ also supported the CREATE TEMPORARY TABLE syntax via DSLContext.createTemporaryTable(Table) (without explicit GLOBAL or LOCAL keyword):

```
// Create a new temporary table
create.createTemporaryTable("book_archive")
      .column("column1", INTEGER)
      .execute();
```

- GLOBAL TEMPORARY if it's supported by a dialect
- LOCAL TEMPORARY otherwise

(!) This alternative syntax is still supported, but no longer recommended with jOOQ 3.21, as the two types of temporary tables work quite differently.

Dialect support

This example using jOOQ:

```
createGlobalTemporaryTable("book_archive")
      .column("column1", INTEGER)
```

Translates to the following dialect specific expressions:

Aurora Postgres, H2, Vertica

```
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 int
)
```

DB2, Firebird, Hana, Teradata

```
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 integer
)
```

MemSQL

```
CREATE ROWSTORE GLOBAL TEMPORARY TABLE book_archive (
  column1 int
)
```

Oracle

```
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 number(10)
)
```

SQLServer

```
CREATE TABLE ##book_archive (
  column1 int
)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, HSQLDB, Informix, MariaDB, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLite, Snowflake, Sybase, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.5. CREATE TYPE

The only type currently supported by jOOQ is the PostgreSQL ENUM type:

```
// Create a new ENUM type
create.createType("weekday")
      .asEnum("mon", "tue", "wed", "thu", "fri", "sat", "sun")
      .execute();
```

Dialect support

This example using jOOQ:

```
createType("t").asEnum("a", "b", "c")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, H2, Postgres, YugabyteDB

```
CREATE TYPE t AS ENUM ('a', 'b', 'c')
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.6. CREATE VIEW

This statement allows for creating a VIEW in the database catalog:

```
// Create a new view
create.createView("books_and_authors", "author_id", "first_name", "last_name", "book_id", "title")
      .as(select(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, BOOK.ID, BOOK.TITLE)
          .from(AUTHOR)
          .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID)))
      .execute();
```

© 2009 - 2025 by Data Geekery™ GmbH.

Dialect support

This example using jOOQ:

```
createView("a", "id").as(select(AUTHOR.ID).from(AUTHOR))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
```

ClickHouse, MemSQL

```
CREATE VIEW a
AS
SELECT t.id
FROM (
  SELECT AUTHOR.ID id
  FROM AUTHOR
) t
```

Trino

```
CREATE VIEW a
AS
SELECT t.id
FROM (
  SELECT AUTHOR.ID
  FROM AUTHOR
) t (id)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.6.1. CREATE OR REPLACE VIEW

Many dialects support a convenient OR REPLACE clause that allows for dropping any pre-existing view by the same name in a single statement.

```
// Create a new view
create.createOrReplaceView("early_authors", "author_id", "first_name", "last_name")
      .as(select(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
          .from(AUTHOR)

          // Any inserted or updated authors must continue to satisfy this condition
          .where(AUTHOR.ID.lt(200)))
      .execute();
```

Dialect support

This example using jOOQ:

```
createOrReplaceView("a", "id").as(select(AUTHOR.ID).from(AUTHOR))
```

Translates to the following dialect specific expressions:

ASE, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, H2, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Vertica, YugabyteDB

```
CREATE OR REPLACE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
```

ClickHouse

```
CREATE OR REPLACE VIEW a
AS
SELECT t.id
FROM (
  SELECT AUTHOR.ID id
  FROM AUTHOR
) t
```

Firebird, SQLServer

```
CREATE OR ALTER VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
```

Teradata

```
REPLACE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
```

Trino

```
CREATE OR REPLACE VIEW a
AS
SELECT t.id
FROM (
  SELECT AUTHOR.ID
  FROM AUTHOR
) t (id)
```

Access, Aurora MySQL, Derby, HSQLDB, Informix, MemSQL, SQLDataWarehouse, SQLite, Snowflake, Sybase

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.6.2. WITH CHECK OPTION

A CREATE VIEW statement of an updatable view can have a WITH CHECK OPTION clause appended to it, to make sure that any INSERT or UPDATE statement will produce rows that are also visible through this view.

```
// Create a new view
create.createView("early_authors", "author_id", "first_name", "last_name")
      .as(select(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
          .from(AUTHOR)

          // Any inserted or updated authors must continue to satisfy this condition
          .where(AUTHOR.ID.lt(200))

          // The flag is set on the Select object, not the view
          .withCheckOption())
      .execute();
```

(!) The flag is set on the SELECT object, not the CREATE VIEW statement, as it is also made available to inline views.

Dialect support

This example using jOOQ:

```
createView("a", "id").as(select(AUTHOR.ID).from(AUTHOR).withCheckOption())
```

Translates to the following dialect specific expressions:

ASE, DB2, Firebird, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, SQLServer, Sybase, Teradata

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
WITH CHECK OPTION
```

Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, MemSQL, Redshift, SQLDataWarehouse, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.3.6.3. WITH READ ONLY

A CREATE VIEW statement of an updatable view can have a WITH READ ONLY clause appended to it, to make sure that it cannot be updated.

```
// Create a new view
create.createView("authors", "author_id", "first_name", "last_name")
      .as(select(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
          .from(AUTHOR)
          .withReadOnly())
      .execute();
```

(!) The flag is set on the SELECT object, not the CREATE VIEW statement, as it is also made available to inline views.

Dialect support

This example using jOOQ:

```
createView("a", "id").as(select(AUTHOR.ID).from(AUTHOR).withReadOnly())
```

Translates to the following dialect specific expressions:

Access

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM (
  SELECT count(*) dual
  FROM MSysResources
) AS dual
WHERE 1 = 0
```

ASE, Redshift, SQLDataWarehouse, SQLServer, SQLite, Vertica

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
WHERE 1 = 0
```

Aurora MySQL

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM DUAL
WHERE 1 = 0
```

Aurora Postgres, Postgres, YugabyteDB

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT CAST(NULL AS int)
WHERE FALSE
```

BigQuery

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION DISTINCT
SELECT NULL
FROM UNNEST([STRUCT(1 AS dual)]) AS dual
WHERE FALSE
```

© 2009 - 2025 by Data Geekery™ GmbH.

CockroachDB, H2, MariaDB, MySQL, Snowflake

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
WHERE FALSE
```

DB2

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM SYSIBM.DUAL
WHERE 1 = 0
```

Derby

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT CAST(NULL AS int)
FROM SYSIBM.SYSDUMMY1
WHERE FALSE
```

Exasol

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM DUAL
WHERE FALSE
```

Firebird

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM RDB$DATABASE
WHERE 1 = 0
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana, Oracle

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
WITH READ ONLY
```

HSQLDB

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT CAST(NULL AS int)
FROM (VALUES (1)) AS dual (dual)
WHERE FALSE
```

Informix

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM (
  SELECT 1 AS dual
  FROM systables
  WHERE (tabid = 1)
) AS dual
WHERE 1 = 0
```

MemSQL

```
CREATE VIEW a
AS
SELECT t.id
FROM (
  SELECT
    t.*
  FROM (
    SELECT AUTHOR.ID id
    FROM AUTHOR
    UNION
    SELECT NULL
    FROM DUAL
    WHERE 1 = 0
  ) t
) t
```

Sybase

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM SYS.DUMMY
WHERE 1 = 0
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
FROM (
  SELECT 1 AS "dual"
) AS "dual"
WHERE 1 = 0
```

ClickHouse, Databricks, DuckDB, Trino

```
/* UNSUPPORTED */
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.4.4. The DROP statement

The DROP statement is used to drop objects from the database catalog.

3.4.4.1. DROP INDEX

This statement is used to drop an INDEX from the database catalog.

```
// Drop an index (for indexes stored in the schema namespace, i.e. most dialects)
create.dropIndex("index").execute();

// Drop an index (for indexes stored in the table namespace, e.g. MySQL, SQL Server)
create.dropIndex("index").on("table").execute();
```

CASCADE

It is possible to supply a CASCADE or RESTRICT clause, explicitly

```
// Specify the CASCADE / RESTRICT clauses explicitly
create.dropIndex("index").cascade().execute();
create.dropIndex("index").restrict().execute();
```

Dialect support

This example using jOOQ:

```
dropIndex("i")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, YugabyteDB

```
DROP INDEX i
```

BigQuery, Databricks, Exasol, Redshift, Snowflake, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.1.1. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop an index
create.dropIndexIfExists("index").execute();
```

Dialect support

This example using jOOQ:

```
dropIndexIfExists("index")
```

Translates to the following dialect specific expressions:

Access

```
DROP INDEX index
```

Aurora Postgres, ClickHouse, CockroachDB, DuckDB, H2, HSQLDB, Informix, MariaDB, Oracle, Postgres, SQLite, Sybase, YugabyteDB

```
DROP INDEX IF EXISTS index
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    DROP INDEX index
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    DROP INDEX index
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 261 BEGIN END;
  EXECUTE IMMEDIATE '
    DROP INDEX index
  ';
END;
```

MySQL

```
CREATE PROCEDURE block_1751479554496_6670064()
MODIFIES SQL DATA
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42000' BEGIN END;
  DROP INDEX index;
END;
CALL block_1751479554496_6670064();
DROP PROCEDURE block_1751479554496_6670064;
```

SQLDataWarehouse

```
BEGIN TRY
  DROP INDEX index
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  DROP INDEX index
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) THROW;
END CATCH
```

ASE, Aurora MySQL, BigQuery, Databricks, Derby, Exasol, MemSQL, Redshift, Snowflake, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.2. DROP SCHEMA

This statement is used to drop a SCHEMA from the database catalog.

```
// Drop a schema
create.dropSchema("schema").execute();
```

CASCADE

It is possible to supply a CASCADE or RESTRICT clause, explicitly

```
// Specify the CASCADE / RESTRICT clauses explicitly
create.dropSchema("schema").cascade().execute();
create.dropSchema("schema").restrict().execute();
```

Dialect support

This example using jOOQ:

```
dropSchema("s")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Vertica, YugabyteDB

```
DROP SCHEMA s
```

ClickHouse

```
DROP DATABASE s
```

DB2, Derby

```
DROP SCHEMA s RESTRICT
```

Oracle

```
DROP USER s
```

ASE, Access, Aurora MySQL, BigQuery, Firebird, Informix, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.2.1. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop a schema
create.dropSchemaIfExists("schema").execute();
```

Dialect support

This example using jOOQ:

```
dropSchemaIfExists("schema")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, Snowflake, Vertica, YugabyteDB

```
DROP SCHEMA IF EXISTS schema
```

ClickHouse

```
DROP DATABASE IF EXISTS schema
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    DROP SCHEMA schema RESTRICT
  ';
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 362 BEGIN END;
  EXECUTE IMMEDIATE '
    DROP SCHEMA schema
  ';
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

Oracle

```
DROP USER IF EXISTS schema
```

SQLDataWarehouse

```
BEGIN TRY
  DROP SCHEMA schema
END TRY
BEGIN CATCH
  IF error_number() NOT IN (15151) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  DROP SCHEMA schema
END TRY
BEGIN CATCH
  IF error_number() NOT IN (15151) THROW;
END CATCH
```

ASE, Access, Aurora MySQL, BigQuery, Derby, Firebird, Informix, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.3. DROP SEQUENCE

This statement is used to drop a SEQUENCE from the database catalog.

```
// Drop a sequence
create.dropSequence("sequence").execute();
```

Dialect support

This example using jOOQ:

```
dropSequence("s")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, DuckDB, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, Oracle, Postgres, SQLServer, Snowflake, Sybase, Vertica, YugabyteDB

```
DROP SEQUENCE s
```

Derby

```
DROP SEQUENCE s RESTRICT
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Exasol, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.3.1. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop a sequence
create.dropSequenceIfExists("sequence").execute();
```

Dialect support

This example using jOOQ:

```
dropSequenceIfExists("sequence")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, H2, HSQLDB, Informix, MariaDB, Oracle, Postgres, Snowflake, Vertica, YugabyteDB

```
DROP SEQUENCE IF EXISTS sequence
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    DROP SEQUENCE sequence
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    DROP SEQUENCE sequence
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 313 BEGIN END;
  EXECUTE IMMEDIATE '
    DROP SEQUENCE sequence
  ';
END;
```

SQLServer

```
BEGIN TRY
  DROP SEQUENCE sequence
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) THROW;
END CATCH
```

Sybase

```
BEGIN
  DROP SEQUENCE sequence;
EXCEPTION
  WHEN others THEN
END;
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, Exasol, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.4. DROP TABLE

This statement is used to drop a TABLE from the database catalog.

```
// Drop a table
create.dropTable("table").execute();
```

Dialect support

This example using jOOQ:

```
dropTable("t")
```

Translates to the following dialect specific expressions:

All dialects

```
DROP TABLE t
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.4.1. CASCADE

When dropping a table, the CASCADE clause extends the dropping to all dependent objects, including views, etc. The following two syntactic elements can be added to a DROP TABLE statement:

- RESTRICT (the default): The DROP operation should fail if there are any dependent objects, such as views.
- CASCADE: The DROP operation should succeed if there are any dependent objects, and drop those as well.

(!) This is a potentially very dangerous operation. While dropping dependent views seems reasonable, some RDMS may also drop dependent child tables if their FOREIGN KEY constraints point to the dropped table. Use the CASCADE clause with great care!

Dialect support

This example using jOOQ:

```
dropTable("t").cascade()
```

Translates to the following dialect specific expressions:

CockroachDB, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, Postgres, Redshift, Snowflake, Vertica, YugabyteDB

```
DROP TABLE t CASCADE
```

Oracle

```
DROP TABLE t CASCADE CONSTRAINTS
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Databricks, Derby, Firebird, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.4.2. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop a table
create.dropTableIfExists("table").execute();
```

Dialect support

This example using jOOQ:

```
dropTableIfExists("table")
```

Translates to the following dialect specific expressions:

Access

```
DROP TABLE table
```

Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, Exasol, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLite, Snowflake, Sybase, Vertica, YugabyteDB

```
DROP TABLE IF EXISTS table
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    DROP TABLE table
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    DROP TABLE table
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 259 BEGIN END;
  EXECUTE IMMEDIATE '
    DROP TABLE table
  ';
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse

```
BEGIN TRY
  DROP TABLE table
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  DROP TABLE table
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) THROW;
END CATCH
```

ASE, ClickHouse, Derby, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.5. DROP TYPE

This statement is used to drop a TYPE from the database catalog.

```
// Drop a type
create.dropType("type").execute();
```

CASCADE

It is possible to supply a CASCADE or RESTRICT clause, explicitly

```
// Specify the CASCADE / RESTRICT clauses explicitly
create.dropType("type").cascade().execute();
create.dropType("type").restrict().execute();
```

Dialect support

This example using jOOQ:

```
dropType("t")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, H2, Oracle, Postgres, YugabyteDB

```
DROP TYPE t
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.5.1. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop a type
create.dropTypeIfExists("type").execute();
```

Dialect support

This example using jOOQ:

```
dropTypeIfExists("type")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DuckDB, H2, Oracle, Postgres, YugabyteDB

```
DROP TYPE IF EXISTS type
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.6. DROP VIEW

This statement is used to drop a VIEW from the database catalog.

```
// Drop a view
create.dropView("view").execute();
```

Dialect support

This example using jOOQ:

```
dropView("v")
```

Translates to the following dialect specific expressions:

All dialects

```
DROP VIEW v
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.4.6.1. IF EXISTS

For idempotent execution of DDL scripts, the useful IF EXISTS clause is supported by jOOQ, and emulated using an anonymous, procedural block if possible.

```
// Drop a view
create.dropViewIfExists("view").execute();
```

Dialect support

This example using jOOQ:

```
dropViewIfExists("v")
```

Translates to the following dialect specific expressions:

Access

```
DROP VIEW v
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLite, Sybase, Trino, Vertica, YugabyteDB

```
DROP VIEW IF EXISTS v
```

DB2

```
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704' BEGIN END;
  EXECUTE IMMEDIATE '
    DROP VIEW v
  ';
END
```

Firebird

```
EXECUTE BLOCK
AS
BEGIN
  EXECUTE STATEMENT '
    DROP VIEW v
  ';
  WHEN sqlcode -607 DO
    BEGIN END
END
```

Hana

```
DO BEGIN
  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 321 BEGIN END;
  EXECUTE IMMEDIATE '
    DROP VIEW v
  ';
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse

```
BEGIN TRY
  DROP VIEW v
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) BEGIN
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
  END;
END CATCH
```

SQLServer

```
BEGIN TRY
  DROP VIEW v
END TRY
BEGIN CATCH
  IF error_number() NOT IN (3701) THROW;
END CATCH
```

ASE, Derby, Snowflake, Teradata

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.5. The GRANT statement

Databases that implement access control for their database catalog allow for using GRANT and REVOKE privileges from org.jooq.User and org.jooq.Role objects. In jOOQ, this can be done as follows:

```
// Define privileges
Privilege select = privilege("select");
Privilege insert = privilege("insert");
User user = user("user");

// Grant privileges to a given user or role
create.grant(select, insert).on(BOOK).to(user).execute();

// Grant privileges to a given user or role with the grant option
create.grant(select, insert).on(BOOK).to(user).withGrantOption().execute();

// Grant privileges to everyone
create.grant(select, insert).on(BOOK).toPublic().execute();
```

3.4.6. The REVOKE statement

Databases that implement access control for their database catalog allow for using GRANT and REVOKE privileges from org.jooq.User and org.jooq.Role objects. In jOOQ, this can be done as follows:

```
// Define privileges
Privilege select = privilege("select");
Privilege insert = privilege("insert");
User user = user("user");

// Revoke privileges from a given user or role
create.revoke(select, insert).on(BOOK).from(user).execute();

// Revoke the grant option from a given user or role
create.revokeGrantOptionFor(select, insert).on(BOOK).from(user).execute();

// Revoke privileges from everyone
create.revoke(select, insert).on(BOOK).fromPublic().execute();
```

3.4.7. The SET statement

Most databases support a variety of SET statements to set session specific environment variables.

3.4.7.1. SET CATALOG

Depending on whether your database product supports catalogs and schemas, the below SET statement may be supported to set the current session's catalog (e.g. the database).

```
SET CATALOG catalogname;
```

```
create.setCatalog("catalogname").execute();
```

Dialect support

This example using jOOQ:

```
setCatalog("c")
```

Translates to the following dialect specific expressions:

Aurora MySQL, ClickHouse, DuckDB, MariaDB, MemSQL, MySQL, SQLServer

```
USE c
```

Databricks

```
USE CATALOG c
```

Snowflake

```
USE DATABASE c
```

Teradata

```
DATABASE c
```

ASE, Access, Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLite, Sybase, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.7.2. SET SCHEMA

Depending on whether your database product supports catalogs and schemas, the below SET statement may be supported to set the current session's schema.

```
SET SCHEMA schemaname;
```

```
create.setSchema("schemaname").execute();
```

Dialect support

This example using jOOQ:

```
setSchema("c")
```

Translates to the following dialect specific expressions:

Aurora MySQL, ClickHouse, DuckDB, MariaDB, MemSQL, MySQL

```
USE c
```

Aurora Postgres, CockroachDB, Postgres, Vertica, YugabyteDB

```
SET SEARCH_PATH = c
```

Databricks, Snowflake

```
USE SCHEMA c
```

DB2, Derby, H2, HSQLDB

```
SET SCHEMA c
```

Exasol

```
OPEN SCHEMA c
```

Oracle

```
ALTER SESSION SET CURRENT_SCHEMA = c
```

Teradata

```
DATABASE c
```

ASE, Access, BigQuery, Firebird, Hana, Informix, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Trino

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.8. The TRUNCATE statement

Even if the TRUNCATE statement mainly modifies data, it is generally considered to be a DDL statement. It is popular in many databases when you want to bypass constraints for table truncation. Databases may behave differently, when a truncated table is referenced by other tables. For instance, they may fail if records from a truncated table are referenced, even with ON DELETE CASCADE clauses in place. Please, consider your database manual to learn more about its TRUNCATE implementation.

The TRUNCATE syntax is trivial:

```
create.truncate(AUTHOR).execute();
```

TRUNCATE is not supported by all dialects. jOOQ will execute a DELETE FROM AUTHOR statement instead, which is roughly equivalent.

Dialect support

This example using jOOQ:

```
truncate(AUTHOR)
```

Translates to the following dialect specific expressions:

Access, Firebird, SQLite, Teradata

```
DELETE FROM AUTHOR
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Trino, Vertica, YugabyteDB

```
TRUNCATE TABLE AUTHOR
```

DB2

```
TRUNCATE TABLE AUTHOR IMMEDIATE
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.4.9. Generating DDL from objects

When using jOOQ's code generator, a whole set of meta data is generated with the generated artefacts, such as schemas, tables, columns, data types, constraints, default values, etc.

This meta data can be used to generate DDL CREATE statements in any SQL dialect, in order to partially restore the original schema again on a new database instance. This is particularly useful, for instance, when working with an Oracle production database, and an H2 in-memory test database. The following code produces the DDL for a schema:

```
// SCHEMA is the generated schema that contains a reference to all generated tables
Queries ddl =
DSL.using(configuration)
   .ddl(SCHEMA);

for (Query query : ddl.queries()) {
    System.out.println(query);
}
```

When executing the above, you should see something like the following:

```
create table "PUBLIC"."AUTHOR"(
  "ID" int not null,
  "FIRST_NAME" varchar(50) null,
  "LAST_NAME" varchar(50) not null,
  ...
  constraint "PK_AUTHOR"
    primary key ("ID")
)
create table "PUBLIC"."BOOK"(
  "ID" int not null,
  "AUTHOR_ID" int not null,
  "TITLE" varchar(400) not null,
  ...
  constraint "PK_BOOK"
    primary key ("ID")
)
...
alter table "PUBLIC"."BOOK"
  add constraint "FK_BOOK_AUTHOR_ID"
    foreign key ("AUTHOR_ID")
    references "AUTHOR" ("ID")
```

Do note that these features only restore *parts* of the original schema. For instance, vendor-specific storage clauses that are not available to jOOQ's generated meta data cannot be reproduced this way.

3.5. Procedural statements

Most RDBMS support some sort of procedural language that allows for running imperative code inside of the database. The syntax of these languages is often similar, and hence it is supported and standardised by jOOQ as well.

In some databases, the procedural language can exist in the form of anonymous blocks, i.e. ad-hoc programs that are interpreted (or compiled) on the fly. In most RDBMS, however, the main approach to using the procedural languages is to store the procedural logic in stored procedures or functions, such that they can be pre-compiled and translated to native code for performance reasons.

© 2009 - 2025 by Data Geekery™ GmbH.

The jOOQ API currently supports anonymous blocks only. In some dialects where anonymous blocks are not supported, a procedure is stored, called, and dropped again as an emulation.

An example of an anonymous block that inserts values 1 - 10 into a table:

```
-- PL/SQL syntax
BEGIN
  FOR i IN 1 .. 10 LOOP
    INSERT INTO t (col) VALUES (i);
  END LOOP;
END;
```

```
Variable<Integer> i = var(name("i"), INTEGER);
create.begin(
  for_(i).in(1, 10).loop(
    insertInto(T).columns(T.COL).values(i)
  )
).execute();
```

The entire loop is executed on the server, which may greatly help reduce client server round trips.

Apart from the block statement, this feature set is available only in our commercial distributions.

3.5.1. Block statement

The most basic building block in procedural languages is the block statement, which allows for creating scope (except for T-SQL, which has no block scope), and for logically grouping related statement together. Just like in Java, where any set of statements can be grouped using curly braces: { statment1; statement2; }, in procedural languages, usually, the keywords BEGIN and END are used to delimit a block. For example:

```
BEGIN
  INSERT INTO t (col) VALUES (1);
  INSERT INTO t (col) VALUES (2);
END;
```

```
create.begin(
  insertInto(T).columns(T.COL).values(1),
  insertInto(T).columns(T.COL).values(2)
).execute();
```

Notice how jOOQ's DSLContext.begin(Statement...) takes an ordinary varargs array (or collection) of org.jooq.Statement as an argument. As such, the statements are comma separated, not semi colon separated. Also, it is important that statements passed to the procedural API do not call the Query.execute() method, as that would execute a statement in the client, rather than embedding a statement expression in a block.

Just like in SQL, such blocks can be nested with any depth, e.g.

```
BEGIN
  BEGIN
    INSERT INTO t (col) VALUES (1);
    INSERT INTO t (col) VALUES (2);
  END;
  BEGIN
    INSERT INTO t (col) VALUES (3);
    INSERT INTO t (col) VALUES (4);
  END;
END;
```

```
create.begin(
  begin(
    insertInto(T).columns(T.COL).values(1),
    insertInto(T).columns(T.COL).values(2)
  ),
  begin(
    insertInto(T).columns(T.COL).values(3),
    insertInto(T).columns(T.COL).values(4)
  )
).execute();
```

Client side "blocks"

In some cases, it may be desireable to group several statements in a "block" in the client only, without producing the BEGIN and END keywords on the server, in case it is not needed. This can be done using DSLContext.statements(Statement...).

```
INSERT INTO t (col) VALUES (1);
INSERT INTO t (col) VALUES (2);
```

```
statements(
  insertInto(T).columns(T.COL).values(1),
  insertInto(T).columns(T.COL).values(2))
```

This API is useful whenever you want to group several statements into one logical org.jooq.Statement and let jOOQ figure out if BEGIN .. END block syntax is required or not. If it is required, then they are added - e.g. when the block is executed on the top level, or nested inside an IF statement, in case the IF statement doesn't already have its own THEN keyword to delimit multi-statement content.

Block execution

org.jooq.Block extends org.jooq.Query, which in turn extends org.jooq.Statement. A Query is a statement that can be executed on its own, as a standalone executable.

All other org.jooq.Statement types (as explained in the following sections) cannot be executed on their own. For example, it makes no sense to execute a GOTO statement outside of a statement block.

Dialect support

This example using jOOQ:

```
begin(deleteFrom(BOOK), deleteFrom(AUTHOR))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, YugabyteDB

```
DO $$
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
$$
```

BigQuery

```
BEGIN
  DELETE FROM BOOK
  WHERE TRUE;
  DELETE FROM AUTHOR
  WHERE TRUE;
END;
```

DB2

```
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END
```

Exasol, Informix, Oracle, SQLDataWarehouse, SQLServer, Snowflake, Teradata, Vertica

```
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
```

Firebird

```
EXECUTE BLOCK AS
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END
```

H2

```
CREATE ALIAS block_1751479548385_8214865 AS $$
  void x(Connection c) throws SQLException {
    try (PreparedStatement s = c.prepareStatement(
      "DELETE FROM BOOK"
    )) {
      s.execute();
    }
    try (PreparedStatement s = c.prepareStatement(
      "DELETE FROM AUTHOR"
    )) {
      s.execute();
    }
  }
$$;
CALL block_1751479548385_8214865();
DROP ALIAS block_1751479548385_8214865;
```

Hana

```
DO BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
```

HSQLDB

```
BEGIN ATOMIC
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
```

MariaDB

```
BEGIN NOT ATOMIC
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

MySQL

```
CREATE PROCEDURE block_1751479554504_1066125()
MODIFIES SQL DATA
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
CALL block_1751479554504_1066125();
DROP PROCEDURE block_1751479554504_1066125;
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.2. CONTINUE statement

A safer way to jump to labels than via GOTO is to use EXIT (jumping out of a LOOP, or block, or other statement) or CONTINUE (skipping a LOOP iteration).

Just like Java's continue, the CONTINUE statement skips the rest of the current iteration in a loop and continues the next iteration. Some dialects use the ITERATE statement for this.

Without a label

```
-- PL/SQL
LOOP
  i := i + 1;
  CONTINUE WHEN i <= 10;
END LOOP;
```

```
// All dialects
loop(
  i.set(i.plus(1)),
  continueWhen(i.gt(10))
)
```

With a label

```
-- PL/SQL
<<label>>
LOOP
  i := i + 1;
  CONTINUE label WHEN i <= 10;
END LOOP;
```

```
// All dialects
Label label = label("label");
label.label(loop(
  i.set(i.plus(1)),
  continue(label).when(i.le(10))
))
```

Notice that continue is a reserved keyword in the Java language, so the jOOQ API cannot use it as a method name. We've suffixed such conflicts with an underscore: continue_().

Dialect support

This example using jOOQ:

```
loop(i.set(i.plus(1)), continueWhen(i.gt(10)))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Oracle, Postgres, YugabyteDB

```
LOOP
  i := (i + 1);
  CONTINUE WHEN i > 10;
END LOOP
```

BigQuery

```
LOOP
  SET i = (i + 1);
  IF i > 10 THEN
    CONTINUE;
  END IF;
END LOOP
```

DB2, HSQLDB, MariaDB, MySQL

```
alias_1:
LOOP
  SET i = (i + 1);
  IF i > 10 THEN
    ITERATE alias_1;
  END IF;
END LOOP
```

H2

```
for (;;) {
  i = (i + 1);
  if (i > 10) {
    continue;
  }
}
```

Hana

```
WHILE 1 = 1 DO
  i = (i + 1);
  IF i > 10 THEN
    CONTINUE;
  END IF;
END WHILE
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
LOOP
  LET i = (i + 1);
  IF i > 10 THEN
    CONTINUE;
  END IF;
END LOOP
```

Snowflake

```
LOOP
  i := (:i + 1);
  IF (:i > 10) THEN
    CONTINUE;
  END IF;
END LOOP
```

SQLDataWarehouse, SQLServer

```
WHILE 1 = 1 BEGIN
  SET @i = (@i + 1);
  IF @i > 10
    CONTINUE;
END
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.3. EXECUTE statement

Many dialects support some way of running dynamic SQL from procedural code. For this, the EXECUTE or EXECUTE IMMEDIATE statements can be used.

In some dialects (e.g. Oracle PL/SQL), using EXECUTE is the only way to run DDL from procedural code.

For example:

```
-- PL/SQL
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE t (col int)';
END;
```

```
// All dialects
create.begin(
  execute(createTable("t").column("col", INTEGER).getSQL())
).excute();
```

You could obviously just pass an arbitrary string to the EXECUTE statement, as in PL/SQL, but the above example shows how to use this approach also with dynamically created jOOQ statements, by calling Query.getSQL().

Dialect support

This example using jOOQ:

```
execute("create table t (i int)")
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, YugabyteDB

```
EXECUTE 'create table t (i int)'
```

BigQuery, DB2, Hana, MariaDB, Oracle, Snowflake

```
EXECUTE IMMEDIATE 'create table t (i int)'
```

Firebird

```
EXECUTE STATEMENT 'create table t (i int)'
```

MySQL

```
CREATE PROCEDURE block_1751479554507_4072423()
MODIFIES SQL DATA
BEGIN
  PREPARE s FROM 'create table t (i int)';
  EXECUTE s;
  DEALLOCATE PREPARE s;
END;
CALL block_1751479554507_4072423();
DROP PROCEDURE block_1751479554507_4072423;
```

SQLServer

```
EXECUTE ('create table t (i int)')
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Informix, MemSQL, Redshift, SQLDataWarehouse, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.4. EXIT statement

A safer way to jump to labels than via GOTO is to use EXIT (jumping out of a LOOP, or block, or other statement) or CONTINUE (skipping a LOOP iteration).

For example, in the absence of more sophisticated LOOP syntaxes, you may choose to exit a loop using EXIT (which translates to LEAVE or BREAK in some dialects, and works the same way as Java's break):

Without a label

```
-- PL/SQL
LOOP
  i := i + 1;
  EXIT WHEN i > 10;
END LOOP;
```

```
// All dialects
loop(
  i.set(i.plus(1)),
  exitWhen(i.gt(10))
)
```

With a label

```
-- PL/SQL
<<label>>
LOOP
  i := i + 1;
  EXIT label WHEN i > 10;
END LOOP;
```

```
// All dialects
Label label = label("label");
label.label(loop(
  i.set(i.plus(1)),
  exit(label).when(i.gt(10))
))
```

Dialect support

This example using jOOQ:

```
loop(i.set(i.plus(1)), exitWhen(i.gt(10)))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Oracle, Postgres, YugabyteDB

```
LOOP
  i := (i + 1);
  EXIT WHEN i > 10;
END LOOP
```

BigQuery

```
LOOP
  SET i = (i + 1);
  IF i > 10 THEN
    BREAK;
  END IF;
END LOOP
```

DB2, HSQLDB, MariaDB, MySQL

```
alias_1:
LOOP
  SET i = (i + 1);
  IF i > 10 THEN
    LEAVE alias_1;
  END IF;
END LOOP
```

Firebird

```
WHILE (1 = 1) DO BEGIN
  :i = (:i + 1);
  IF (:i > 10) THEN
    LEAVE;
END
```

H2

```
for (;;) {
  i = (i + 1);
  if (i > 10) {
    break;
  }
}
```

Hana

```
WHILE 1 = 1 DO
  i = (i + 1);
  IF i > 10 THEN
    BREAK;
  END IF;
END WHILE
```

Informix

```
LOOP
  LET i = (i + 1);
  EXIT WHEN i > 10;
END LOOP
```

© 2009 - 2025 by Data Geekery™ GmbH.

Snowflake

```
LOOP
  i := (:i + 1);
  IF (:i > 10) THEN
    BREAK;
  END IF;
END LOOP
```

SQLDataWarehouse, SQLServer

```
WHILE 1 = 1 BEGIN
  SET @i = (@i + 1);
  IF @i > 10
    BREAK;
END
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, Exasol, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.5. FOR statement

When iterating over a sequence of numeric values, the FOR loop provides useful syntax sugar over the previous types of loops, including the WHILE loop, despite being functional equivalent.

An example:

```
-- PL/SQL

FOR i IN 1 .. 10 LOOP
  INSERT INTO t (col) VALUES (i);
END LOOP;
```

```
// All dialects
Variable<Integer> i = var("i", INTEGER);
for_(i).in(1, 10).loop(
  insertInto(T).columns(T.COL).values(i)
)
```

In addition to simplifying the most common case, there are also options of traversing the arguments in a reversed way, and using an additional optional step variable, for example:

```
-- pgplsql

FOR i IN REVERSE 10 .. 1 BY 2 LOOP
  INSERT INTO t (col) VALUES (i);
END LOOP;
```

```
// All dialects
Variable<Integer> i = var("i", INTEGER);
for_(i).inReverse(10, 1).by(2).loop(
  insertInto(T).columns(T.COL).values(i)
)
```

Not all dialects support the entirety of this syntax, but luckily it is easy for jOOQ to emulate in all dialects using WHILE:

```
-- PL/SQL
WHILE i >= 1 LOOP
  INSERT INTO t (col) VALUES (i);
  i := i - 2;
END LOOP;
```

Notice that for is a reserved keyword in the Java language, so the jOOQ API cannot use it as a method name. We've suffixed such conflicts with an underscore: for_().

Dialect support

This example using jOOQ:

```
for_(i).in(1, 10).loop(insertInto(BOOK).columns(BOOK.ID).values(i))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Exasol, Oracle, Postgres, YugabyteDB

```
FOR i IN 1 .. 10 LOOP
  INSERT INTO BOOK (ID)
  VALUES (i);
END LOOP
```

BigQuery

```
BEGIN
  DECLARE i int64 DEFAULT 1;
  WHILE i <= 10 DO
    INSERT INTO BOOK (ID)
    VALUES (i);
    SET i = (i + 1);
  END WHILE;
END;
```

DB2

```
BEGIN
  DECLARE i integer;
  SET i = 1;
  WHILE i <= 10 DO
    INSERT INTO BOOK (ID)
    VALUES (i);
    SET i = (i + 1);
  END WHILE;
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird

```
DECLARE i integer DEFAULT 1;
WHILE (:i <= 10) DO BEGIN
  INSERT INTO BOOK (ID)
  VALUES (:i);
  :i = (:i + 1);
END
```

H2

```
for (Integer i = 1; i <= 10; i++) {
  try (PreparedStatement s = c.prepareStatement(
    "INSERT INTO BOOK (ID)\n" +
    "VALUES (?)"
  )) {
    s.setObject(1, i);
    s.execute();
  }
}
```

Hana

```
BEGIN
  DECLARE i integer;
  FOR i IN 1 .. 10 DO
    INSERT INTO BOOK (ID)
    VALUES (i);
  END FOR;
END;
```

HSQLDB

```
BEGIN ATOMIC
  DECLARE i int;
  SET i = 1;
  WHILE i <= 10 DO
    INSERT INTO BOOK (ID)
    VALUES (i);
    SET i = (i + 1);
  END WHILE;
END;
```

Informix

```
BEGIN
  DEFINE i integer;
  LET i = 1;
  FOR i IN (1 TO 10) LOOP
    INSERT INTO BOOK (ID)
    VALUES (i);
  END LOOP;
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

MariaDB

```
FOR i IN 1 .. 10 DO
  INSERT INTO BOOK (ID)
  VALUES (i);
END FOR
```

MySQL

```
BEGIN
  DECLARE i int;
  SET i = 1;
  WHILE i <= 10 DO
    INSERT INTO BOOK (ID)
    VALUES (i);
    SET i = (i + 1);
  END WHILE;
END;
```

Snowflake

```
FOR i IN 1 TO 10 DO
  INSERT INTO BOOK (ID)
  SELECT :i;
END FOR
```

SQLDataWarehouse

```
BEGIN
  DECLARE @i int DEFAULT 1;
  WHILE @i <= 10 BEGIN
    INSERT INTO BOOK (ID)
    SELECT @i;
    SET @i = (@i + 1);
  END;
END;
```

SQLServer

```
BEGIN
  DECLARE @i int = 1;
  WHILE @i <= 10 BEGIN
    INSERT INTO BOOK (ID)
    VALUES (@i);
    SET @i = (@i + 1);
  END;
END;
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.6. GOTO statement

Hey, we don't judge anyone. You have your reasons for using this statement.

In the previous section, we introduced labels. Now we make use of them. For instance, to conditionally skip a set of statements. Or to simplify the example, to inconditionally skip them:

```
-- PL/SQL
BEGIN
  INSERT INTO t (col) VALUES (1);
  GOTO l1;
  INSERT INTO t (col) VALUES (2);

  <<l1>>
  INSERT INTO t (col) VALUES (3);
END;
```

```
// All dialects
Label l1 = label("l1");

begin(
  insertInto(T).columns(T.COL).values(1),
  goto_(l1),
  insertInto(T).columns(T.COL).values(2),
  l1.label(insertInto(T).columns(T.COL).values(3))
)
```

Some dialects (e.g. T-SQL) may implement "full GOTO" in the ways that are generally not really helping readability or maintainability of code, namely the idea that GOTO can be used to jump into any arbitrary scope that should not be reachable through ordinary control flow. This is not possible in other languages, like PL/SQL, and currently cannot be emulated by jOOQ.

Notice that goto is a reserved keyword in the Java language, so the jOOQ API cannot use it as a method name. We've suffixed such conflicts with an underscore: goto_().

Dialect support

This example using jOOQ:

```
begin(l.label(insertInto(BOOK).columns(BOOK.ID).values(1)), goto_(l))
```

Translates to the following dialect specific expressions:

DB2

```
BEGIN
  l:
  INSERT INTO BOOK (ID)
  VALUES (1);
  GOTO l;
END
```

Informix, Oracle

```
BEGIN
  <<l>>
  INSERT INTO BOOK (ID)
  VALUES (1);
  GOTO l;
END;
```

SQLDataWarehouse, SQLServer

```
BEGIN
  l:
  INSERT INTO BOOK (ID)
  VALUES (1);
  GOTO l;
END;
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.7. IF statement

Conditional branching is an essential feature of all languages. Procedural languages support the IF statement.

There are different styles of IF statements in dialects, including:

- Requiring a THEN clause for the body of a branch, in case of which no BEGIN .. END block is required for multi-statement bodies.
- Allowing a dedicated ELSIF clause for alternative, nested branches, to avoid nesting. This is mostly a syntax sugar feature only.

In jOOQ, an IF statement might look as follows:

```
-- PL/SQL syntax
IF i = 0 THEN
  INSERT INTO a (col) VALUES (1);
ELSIF i = 1 THEN
  INSERT INTO b (col) VALUES (2);
ELSE
  INSERT INTO c (col) VALUES (3);
END IF;
```

```
// All dialects
if_(i.eq(0)).then(
  insertInto(A).columns(A.COL).values(1)
).elsif(i.eq(1)).then(
  insertInto(B).columns(B.COL).values(2)
).else_(
  insertInto(C).columns(C.COL).values(3)
)
```

Notice that both if and else are reserved keywords in the Java language, so the jOOQ API cannot use them as method names. We've suffixed such conflicts with an underscore: if_() and else_().

Dialect support

This example using jOOQ:

```
if_(i.eq(0)).then(deleteFrom(BOOK)).else_(deleteFrom(AUTHOR))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Exasol, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, YugabyteDB

```
IF i = 0 THEN
  DELETE FROM BOOK;
ELSE
  DELETE FROM AUTHOR;
END IF
```

BigQuery

```
IF i = 0 THEN
  DELETE FROM BOOK
  WHERE TRUE;
ELSE
  DELETE FROM AUTHOR
  WHERE TRUE;
END IF
```

Firebird

```
IF (:i = 0) THEN
  DELETE FROM BOOK;
ELSE
  DELETE FROM AUTHOR;
```

© 2009 - 2025 by Data Geekery™ GmbH.

H2

```
if (i = 0) {
  try (PreparedStatement s = c.prepareStatement(
    "DELETE FROM BOOK"
  )) {
    s.execute();
  }
} else {
  try (PreparedStatement s = c.prepareStatement(
    "DELETE FROM AUTHOR"
  )) {
    s.execute();
  }
}
```

Snowflake

```
IF (:i = 0) THEN
  DELETE FROM BOOK;
ELSE
  DELETE FROM AUTHOR;
END IF
```

SQLDataWarehouse, SQLServer

```
IF @i = 0
  DELETE FROM BOOK;
ELSE
  DELETE FROM AUTHOR;
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.8. Labels

In imperative languages, labels are essential with simpler cases of loops (e.g. the LOOP statement), with nested loops, or if you must, when using the GOTO statement.

Using jOOQ, you can label any org.jooq.Statement by using DSL.label():

```
-- PL/SQL
<<label>>
BEGIN NULL; END;
```

```
// All dialects
Label label = label("label");
label.label(begin())
```

That's a lot of labels.

The main usages of these labels will be discussed in the following sections about, EXIT, and CONTINUE

Dialect support

This example using jOOQ:

```
l.label(deleteFrom(BOOK))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Informix, Oracle, Postgres, YugabyteDB

```
<<l>>
DELETE FROM BOOK
```

DB2, Firebird, H2, HSQLDB, MariaDB, MySQL, SQLDataWarehouse, SQLServer

```
l:
DELETE FROM BOOK
```

Snowflake

```
DELETE FROM BOOK l
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Hana, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.9. LOOP statement

Many procedural languages support a condition-less loop, which in its pure form, just loops forever. In order to create an infinite number of records in a table, one might write the following:

```
-- PL/SQL syntax
LOOP
  INSERT INTO t (col) VALUES (1);
END LOOP;
```

```
// All dialects
loop(
  insertInto(T).columns(T.COL).values(1)
)
```

An "infinite" loop is usually exited using an EXIT statement.

Dialect support

This example using jOOQ:

```
loop(update(BOOK_TO_BOOK_STORE).set(BOOK_TO_BOOK_STORE.STOCK, BOOK_TO_BOOK_STORE.STOCK.plus(1)))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, Snowflake, YugabyteDB

```
LOOP
  UPDATE BOOK_TO_BOOK_STORE
  SET
    STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1);
END LOOP
```

BigQuery

```
LOOP
  UPDATE BOOK_TO_BOOK_STORE
  SET
    BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1)
  WHERE TRUE;
END LOOP
```

DB2, Exasol, HSQLDB, Informix, MariaDB, MySQL, Oracle

```
LOOP
  UPDATE BOOK_TO_BOOK_STORE
  SET
    BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1);
END LOOP
```

Firebird

```
WHILE (1 = 1) DO BEGIN
  UPDATE BOOK_TO_BOOK_STORE
  SET
    BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1);
END
```

© 2009 - 2025 by Data Geekery™ GmbH.

H2

```
for (;;) {
  try (PreparedStatement s = c.prepareStatement(
    "UPDATE BOOK_TO_BOOK_STORE\n" +
    "SET\n" +
    "  BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1)"
  )) {
    s.execute();
  }
}
```

Hana

```
WHILE 1 = 1 DO
  UPDATE BOOK_TO_BOOK_STORE
  FROM BOOK_TO_BOOK_STORE
  SET
    BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1);
END WHILE
```

SQLDataWarehouse, SQLServer

```
WHILE 1 = 1 BEGIN
  UPDATE BOOK_TO_BOOK_STORE
  SET
    BOOK_TO_BOOK_STORE.STOCK = (BOOK_TO_BOOK_STORE.STOCK + 1);
END
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.10. REPEAT statement

WHILE's lesser known little sibling is REPEAT, which works the same way as Java's do statement. It is mostly not as useful as WHILE, but can be, occasionally, when a loop must be iterated ***at least*** once.

An example:

```
-- MySQL syntax
REPEAT
  INSERT INTO t (col) VALUES (i);
  SET i = i + 1;
UNTIL i > 10 END REPEAT;
```

```
// All dialects
Variable<Integer> i = var("i", INTEGER);
repeat(
  insertInto(T).columns(T.COL).values(i),
  i.set(i.plus(1))
).until(i.gt(10))
```

Dialect support

This example using jOOQ:

```
repeat(deleteFrom(BOOK).where(BOOK.ID.eq(i)), i.set(i.plus(1))).until(i.gt(10))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Oracle, Postgres, YugabyteDB

```
<<alias_2>>
LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  i := (i + 1);
  EXIT alias_2 WHEN i > 10;
END LOOP
```

BigQuery, DB2, HSQLDB, MariaDB, MySQL

```
REPEAT
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  SET i = (i + 1);
UNTIL i > 10
END REPEAT
```

Exasol

```
REPEAT
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  i := (i + 1);
UNTIL i > 10
END REPEAT
```

Firebird

```
alias_2:
WHILE (1 = 1) DO BEGIN
  DELETE FROM BOOK
  WHERE BOOK.ID = :i;
  :i = (:i + 1);
  IF (:i > 10) THEN
    LEAVE alias_2;
END
```

© 2009 - 2025 by Data Geekery™ GmbH.

H2

```
do {
  try (PreparedStatement s = c.prepareStatement(
    "DELETE FROM BOOK\n" +
    "WHERE BOOK.ID = ?"
  )) {
    s.setObject(1, i);
    s.execute();
  }
  i = (i + 1);
}
while (!(i > 10))
```

Hana

```
WHILE 1 = 1 DO
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  i = (i + 1);
  IF i > 10 THEN
    BREAK;
  END IF;
END WHILE
```

Informix

```
<<alias_2>>
LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  LET i = (i + 1);
  EXIT alias_2 WHEN i > 10;
END LOOP
```

Snowflake

```
REPEAT
  DELETE FROM BOOK
  WHERE BOOK.ID = :i;
  i := (:i + 1);
UNTIL (:i > 10)
END REPEAT
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.11. Variables

In imperative languages, local variables are an essential way of temporarily storing data for further processing. All procedural languages have a way to declare, assign, and reference such local variables.

Declaration

In jOOQ, local variable expressions can be created using DSL.var() (not to be confused with DSL.val(T), which creates bind values!)

```
Variable<Integer> i = var("i", INTEGER);
```

This variable doesn't do anything on its own yet. But like many things in jOOQ, it has to be declared first, outside of an actual jOOQ expression, in order to be usable in jOOQ expressions.

We can now reference the variable in a declaration statement as follows:

```
-- MySQL syntax
DECLARE i INTEGER;
```

```
// All dialects
declare(i)
```

Notice that there are many different ways to declare a local variable in different dialects. There is

The Oracle PL/SQL, PostgreSQL pgplsql style

In these languages, the DECLARE statement is actually not an independent statement that can be used anywhere. It is part of a procedural block, prepended to BEGIN .. END:

```
-- PL/SQL syntax
DECLARE
  i INT;
BEGIN
  NULL;
END;
```

When using jOOQ, you can safely ignore this fact, and prepend that there is a DECLARE statement also in these dialects. jOOQ will add additional BEGIN .. END blocks to your surrounding block, to make sure the whole block becomes syntactically and semantically correct.

The T-SQL, MySQL style

In these languages, the DECLARE statement is really an independent statement that can be used anywhere. Just like in the Java language, variables can be declared at any position and used only "further down", lexically. Ignoring T-SQL's JavaScript-esque understanding of scope for a moment.

```
-- T-SQL syntax
DECLARE @i INTEGER;
```

Notice that you can safely ignore the @ sign that is required in some dialects, such as T-SQL. jOOQ will generate it for you.

© 2009 - 2025 by Data Geekery™ GmbH.

Assignment

A local variable needs a way to have a value assigned to it. Assignments are possible both on org.jooq.Variable, or on org.jooq.Declaration, directly. For example

```
-- T-SQL syntax
DECLARE @i INTEGER = 1;
```

```
// All dialects
declare(i).set(1)
```

Alternatively, you can split declaration and assignment, or re-assign new values to variables:

```
-- T-SQL syntax
DECLARE @i INTEGER;
SET @i = 1;
SET @i = 2;
```

```
// All dialects
declare(i),
i.set(1),
i.set(2)
```

Some dialects also support using subqueries in assignment expressions, and other expresions in their procedural languages. For example:

```
-- T-SQL syntax
SET @i = (SELECT MAX(col) FROM t);
```

```
// All dialects
i.set(select(max(T.COL)).from(T))
```

In PL/SQL, the above can be emulated as follows:

```
-- PL/SQL syntax
SELECT MAX(col)
INTO i
FROM t;
```

Referencing

Obviously, once we've assigned a value to a local variable, we want to reference it as well in arbitrary expressions, and queries.

For this purpose, org.jooq.Variable extends org.jooq.Field, and as such, can be used in arbitrary places where any other column expression can be used. Within the procedural language, a simple example would be to increment a local variable:

```
-- PL/SQL syntax
i := i + 1;
```

```
// All dialects
i.set(i.plus(1))
```

Or in a more complete example, use it in a SQL statement:

```
-- PL/SQL syntax
DECLARE
  i INT;
BEGIN
  i := 1;
  INSERT INTO t (col) VALUES (i);
END;
```

```
// All dialects
Variable<Integer> i = var("i", INTEGER);
create.begin(
  declare(i),
  i.set(1),
  insertInto(T).columns(T.COL).values(i)
).execute();
```

Dialect support

This example using jOOQ:

```
begin(declare(i), i.set(1))
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, YugabyteDB

```
DO $$
DECLARE
  i int;
BEGIN
  i := 1;
END;
$$
```

BigQuery

```
BEGIN
  DECLARE i int64;
  SET i = 1;
END;
```

CockroachDB

```
DO $$
DECLARE
  i int4;
BEGIN
  i := 1;
END;
$$
```

DB2

```
BEGIN
  DECLARE i integer;
  SET i = 1;
END
```

Exasol

```
BEGIN
  i int;
  i := 1;
END;
```

Firebird

```
EXECUTE BLOCK AS
  DECLARE i integer;
BEGIN
  :i = 1;
END
```

H2

```
CREATE ALIAS block_1751479548404_4144576 AS $$
  void x(Connection c) throws SQLException {
    Integer i = null;
    i = 1;
  }
$$;
CALL block_1751479548404_4144576();
DROP ALIAS block_1751479548404_4144576;
```

Hana

```
DO BEGIN
  DECLARE i integer;
  i = 1;
END;
```

HSQLDB

```
BEGIN ATOMIC
  DECLARE i int;
  SET i = 1;
END;
```

Informix

```
BEGIN
  DEFINE i integer;
  LET i = 1;
END;
```

MariaDB

```
BEGIN NOT ATOMIC
  DECLARE i int;
  SET i = 1;
END;
```

© 2009 - 2025 by Data Geekery™ GmbH.

MySQL

```
CREATE PROCEDURE block_1751479554516_6287440()
MODIFIES SQL DATA
BEGIN
  DECLARE i int;
  SET i = 1;
END;
CALL block_1751479554516_6287440();
DROP PROCEDURE block_1751479554516_6287440;
```

Oracle

```
DECLARE
  i number(10);
BEGIN
  i := 1;
END;
```

Snowflake

```
BEGIN
  LET i number(10);
  i := 1;
END;
```

SQLDataWarehouse, SQLServer

```
BEGIN
  DECLARE @i int;
  SET @i = 1;
END;
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.5.12. WHILE statement

One of the most commonly used loop types is the WHILE statement, which in its procedural form works just like a Java while statement with slightly different syntax.

Procedural dialects may or may not use the LOOP keyword to delimite the loop body, just as with the previous LOOP statement. The jOOQ API always uses it for DSL design reasons. For example, combining the WHILE loop with the variable assignment statement:

```
-- PL/SQL syntax
WHILE i <= 10
LOOP
  INSERT INTO t (col) VALUES (i);
END LOOP;
```

```
// All dialects
Variable<Integer> i = var("i", INTEGER);
while_(i.le(10)).loop(
  insertInto(T).columns(T.COL).values(i)
)
```

Notice that while is a reserved keyword in the Java language, so the jOOQ API cannot use it as a method name. We've suffixed such conflicts with an underscore: while_().

Dialect support

This example using jOOQ:

```
while_(i.le(10)).loop(deleteFrom(BOOK).where(BOOK.ID.eq(i)))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Exasol, Informix, Oracle, Postgres, YugabyteDB

```
WHILE i <= 10 LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
END LOOP
```

BigQuery, DB2, HSQLDB, Hana, MariaDB, MySQL

```
WHILE i <= 10 DO
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
END WHILE
```

Firebird

```
WHILE (:i <= 10) DO BEGIN
  DELETE FROM BOOK
  WHERE BOOK.ID = :i;
END
```

H2

```
while (i <= 10) {
  try (PreparedStatement s = c.prepareStatement(
    "DELETE FROM BOOK\n" +
    "WHERE BOOK.ID = ?"
  )) {
    s.setObject(1, i);
    s.execute();
  }
}
```

Snowflake

```
WHILE (:i <= 10) DO
  DELETE FROM BOOK
  WHERE BOOK.ID = :i;
END WHILE
```

SQLDataWarehouse, SQLServer

```
WHILE @i <= 10 BEGIN
  DELETE FROM BOOK
  WHERE BOOK.ID = @i;
END
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, MemSQL, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.6. Catalog and schema expressions

Most databases know some sort of namespace to group objects like tables, stored procedures, sequences and others into a common catalog or schema. jOOQ uses the types org.jooq.Catalog and org.jooq.Schema to model these groupings, following SQL standard naming.

The catalog

A catalog is a collection of schemas. In many databases, the catalog corresponds to the database, or the database instance. Most often, catalogs are completely independent and their tables cannot be joined or combined in any way in a single query. The exception here is SQL Server and Sybase ASE, which allow for fully referencing tables from multiple catalogs:

```
SELECT *
FROM [Catalog1].[Schema1].[Table1] AS [t1]
JOIN [Catalog2].[Schema2].[Table2] AS [t2] ON [t1].[ID] = [t2].[ID]
```

Some dialects, including MariaDB, MemSQL, MySQL, use catalogs (databases) and schemas as the same thing. jOOQ treats databases in those dialects as schemas instead.

By default, the Settings.renderCatalog flag is turned on. In case a database supports querying multiple catalogs, jOOQ will generate fully qualified object names, including catalog name. For more information about this setting, see the manual's section about settings

jOOQ's code generator generates subpackages for each catalog.

The schema

A schema is a collection of objects, such as tables. Most databases support some sort of schema (except for some embedded databases like Access, Firebird, SQLite). In most databases, the schema is an independent structural entity. In Oracle, the schema and the user / owner is mostly treated as the same thing. An example of a query that uses fully qualified tables including schema names is:

```
SELECT *
FROM "Schema1"."Table1" AS "t1"
JOIN "Schema2"."Table2" AS "t2" ON "t1"."ID" = "t2"."ID"
```

By default, the Settings.renderSettings flag is turned on. jOOQ will thus generate fully qualified object names, including the setting name. For more information about this setting, see the manual's section about settings

3.7. Table expressions

The following sections explain the various types of table expressions supported by jOOQ

3.7.1. Generated Tables

Most of the times, when thinking about a table expression you're probably thinking about an actual table in your database schema. If you're using jOOQ's code generator, you will have all tables from your database schema available to you as type safe Java objects. You can then use these tables in SQL FROM clauses, JOIN clauses or in other SQL statements, just like any other table expression. An example is given here:

```
SELECT *
FROM AUTHOR -- Table expression AUTHOR
JOIN BOOK   -- Table expression BOOK
ON (AUTHOR.ID = BOOK.AUTHOR_ID)
```

```
create.select()
      .from(AUTHOR) // Table expression AUTHOR
      .join(BOOK)   // Table expression BOOK
      .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .fetch();
```

The above example shows how AUTHOR and BOOK tables are joined in a SELECT statement. It also shows how you can access table columns by dereferencing the relevant Java attributes of their tables.

See the manual's section about generated tables for more information about what is really generated by the code generator

3.7.2. Aliased Tables

The following sections illustrate how to alias tables, their columns, and how to reference columns from aliased tables.

*(!) Regardless of how the aliased table is defined, the same aliased table instance is rendered differently depending on where it is placed in the jOOQ expression tree. See the manual's section about **rendering declarations vs references** for more details.*

3.7.2.1. Aliased generated tables

The strength of jOOQ's code generator becomes more obvious when you perform table aliasing and dereference fields from generated aliased tables. This can best be shown by example:

```
-- Select all books by authors born after 1920,
-- named "Paulo" from a catalogue:



SELECT *
  FROM author a
  JOIN book b ON a.id = b.author_id
 WHERE a.year_of_birth > 1920
   AND a.first_name = 'Paulo'
 ORDER BY b.title
```

```
// Declare your aliases before using them in SQL:
Author a = AUTHOR.as("a");
Book b = BOOK.as("b");

// Use aliased tables in your statement
create.select()
      .from(a)
      .join(b).on(a.ID.eq(b.AUTHOR_ID))
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")))
      .orderBy(b.TITLE)
      .fetch();
```

As you can see in the above example, calling as() on generated tables returns an object of the same type as the table. This means that the resulting object can be used to dereference fields from the aliased table. This is quite powerful in terms of having your Java compiler check the syntax of your SQL statements. If you remove a column from a table, dereferencing that column from that table alias will cause compilation errors.

Dialect support

This example using jOOQ:

```
select(a.ID).from(a)
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT a.ID
FROM AUTHOR a
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.7.2.2. Aliased table expressions

Only few types of table expressions can leverage code generation to provide the SQL syntax typesafety shown previously, where generated tables are used. All tables, however, allow for dereferencing their fields through Table::field methods:

```
// "Type-unsafe" aliased table:
Table<?> a = AUTHOR.as("a");

// Get fields from a:
Field<?> id = a.field("ID");
Field<?> firstName = a.field("FIRST_NAME");
```

The same is true for derived tables, including unnamed derived tables whose synthetic table name is generated by jOOQ:

```
Table<?> named = table(select(AUTHOR.ID).from(AUTHOR)).as("t");
Table<?> unnamed = table(select(AUTHOR.ID).from(AUTHOR));

Field<?> id = named.field("ID");   // Produces a t.ID reference
Field<?> id = unnamed.field("ID"); // Produces a <generated-alias>.ID reference
```

Note that if you *know* that the ID column is of the same type as the AUTHOR.ID column, you can use that again to dereference the column as is explained again in the section dereferencing table columns.

```
// Now with inferred Integer type
Field<Integer> id = named.field(AUTHOR.ID);   // Produces a t.ID reference
Field<Integer> id = unnamed.field(AUTHOR.ID); // Produces a <generated-alias>.ID reference
```

3.7.2.3. Derived column lists

The SQL standard specifies how a table can be renamed / aliased in one go along with its columns. It references the term "derived column list" for the following syntax:

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
) t(a, b)
```

This feature is useful in various use-cases where column names are not known in advance (but the table's degree is!). An example for this are unnested tables, or the VALUES() table constructor:

```
-- Unnested tables
SELECT t.a, t.b
FROM unnest(my_table_function()) t(a, b)

-- VALUES() constructor
SELECT t.a, t.b
FROM VALUES(1, 2),(3, 4) t(a, b)
```

Only few databases really support such a syntax, but fortunately, jOOQ can emulate it easily using UNION ALL and an empty dummy record specifying the new column names. The two statements are equivalent:

```
-- Using derived column lists
SELECT t.a, t.b
FROM (
  SELECT 1, 2
) t(a, b)

-- Using UNION ALL and a dummy record
SELECT t.a, t.b
FROM (
  SELECT null a, null b FROM DUAL WHERE 1 = 0
  UNION ALL
  SELECT 1, 2 FROM DUAL
) t
```

In jOOQ, you would simply specify a varargs list of column aliases as such:

```
// Unnested tables
create.select().from(unnest(myTableFunction()).as("t", "a", "b")).fetch();

// VALUES() constructor
create.select().from(values(
  row(1, 2),
  row(3, 4)
).as("t", "a", "b"))
.fetch();
```

Dialect support

This example using jOOQ:

```
selectFrom(values(row(1, 2)).as("t", "a", "b"))
```

Translates to the following dialect specific expressions:

Access

```
SELECT t.a, t.b
FROM (
  SELECT
    1 a,
    2 b
  FROM (
    SELECT count(*) dual
    FROM MSysResources
  ) AS dual
) t
```

ASE, Redshift, SQLDataWarehouse, Vertica

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
) t (a, b)
```

Aurora MySQL, MemSQL

```
SELECT t.a, t.b
FROM (
  SELECT
    1 a,
    2 b
  FROM DUAL
) t
```

Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Oracle, Postgres, SQLServer, Snowflake, Trino, YugabyteDB

```
SELECT t.a, t.b
FROM (
  VALUES (1, 2)
) t (a, b)
```

BigQuery

```
SELECT t.a, t.b
FROM (
  SELECT
    null a,
    null b
  FROM UNNEST([STRUCT(1 AS dual)]) AS dual
  WHERE FALSE
  UNION ALL
  SELECT *
  FROM UNNEST ([ STRUCT (1, 2)]) t
) t
```

ClickHouse, MariaDB

```
SELECT t.a, t.b
FROM (
  SELECT
    1 a,
    2 b
) t
```

Firebird

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
  FROM RDB$DATABASE
) t (a, b)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
SELECT t.a, t.b
FROM (
  SELECT
    1 a,
    2 b
  FROM SYS.DUMMY
) t
```

Informix

```
SELECT t.a, t.b
FROM (
  TABLE (MULTISET { ROW (1, 2)})
) t (a, b)
```

MySQL

```
SELECT t.a, t.b
FROM (
  VALUES ROW (1, 2)
) t (a, b)
```

SQLite

```
SELECT t.a, t.b
FROM (
  SELECT
    null a,
    null b
  WHERE 1 = 0
  UNION ALL
  SELECT *
  FROM (
    VALUES (1, 2)
  ) t
) t
```

Sybase

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
  FROM SYS.DUMMY
) t (a, b)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
) t (a, b)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.2.4. Unnamed derived tables

The org.jooq.Table type can reference a derived table:

```
-- Derived table
(SELECT 1 AS a)
```

```
// Derived table
table(select(inline(1).as("a")));
```

Most databases do not support unnamed derived tables, they require an explicit alias. If you do not provide jOOQ with such an explicit alias, an alias will be generated based on the derived table's content, to make sure the generated SQL will be syntactically correct. The generated alias is not specified and should not be referenced explicitly.

(!) While the actual alias shouldn't be relied upon, as the generation algorithm might change between jOOQ versions, the alias will remain stable per SQL content of the derived table, in order to prevent execution plan cache contention in dialects with an execution plan. In other words, two consecutive renderings of a structurally identical derived table should produce the same generated alias. Of course, it's usually better to provide an explicit alias nonetheless.

3.7.3. Joined tables

The JOIN operators that can be used in SQL SELECT statements are the most powerful and best supported means of creating new table expressions in SQL.

This section will explain the different types of join:

- CROSS JOIN: A cross product
- INNER JOIN: A cross product filtering on matches
- OUTER JOIN: A cross product filtering on matches, additionally producing some unmatched rows
- SEMI JOIN: A check for existence of rows from one table in another table (using EXISTS or IN)
- ANTI JOIN: A check for non-existence of rows from one table in another table (using NOT EXISTS or some conditions NOT IN)

... as well as the different types of forming join predicates:

- ON: Expressing join predicates explicitly
- ON KEY: Expressing join predicates explicitly or implicitly based on a FOREIGN KEY
- USING: Expressing join predicates implicitly based on an explicit set of shared column names in both tables
- NATURAL: Expressing join predicates implicitly based on an implicit set of shared column names in both tables

... and then, there are additional ways to enrich joins:

- APPLY or LATERAL: Ordering the join tree from left to right, allowing the right side to access rows from the left side
- PARTITION BY on OUTER JOIN: To fill the gaps in a report that uses OUTER JOIN

All of these approaches are available twice in the jOOQ API:

- On the org.jooq.Table API, where they form binary operators
- On the SELECT API, where they are offered as convenience in jOOQ's DSL, to tame the parentheses

3.7.3.1. CROSS JOIN

A CROSS JOIN creates a cartesian product or cross product between the two tables it joins. It does not allow for any join predicates to be specified.

It is an occasionally useful operator in reporting, when every element of one set need to be combined with every element of another set. For example, when you want to produce a report combining employees and weekdays, and then do something with the resulting table:

```
SELECT EMPLOYEE.NAME, WEEKDAY.NAME
FROM EMPLOYEE
CROSS JOIN WEEKDAY
```

```
create.select(EMPLOYEE.NAME, WEEKDAY.NAME)
      .from(EMPLOYEE)
      .crossJoin(WEEKDAY)
      .fetch();
```

Some example output might be:

```
+---------------+--------------+
| EMPLOYEE.NAME | WEEKDAY.NAME |
+---------------+--------------+
| Jon           | Monday       |
| Jon           | Tuesday      |
| Jon           | Wednesday    |
| Jon           | Thursday     |
| Jon           | Friday       |
| Jon           | Saturday     |
| Jon           | Sunday       |
| Jane          | Monday       |
| Jane          | Tuesday      |
| Jane          | Wednesday    |
| Jane          | Thursday     |
| Jane          | Friday       |
| Jane          | Saturday     |
| Jane          | Sunday       |
| ...           | ...          |
+---------------+--------------+
```

Table lists

Note that a CROSS JOIN is functionally (but not syntactically) equivalent to a table list that you can provide in the FROM clause:

```
SELECT EMPLOYEE.NAME, WEEKDAY.NAME
FROM EMPLOYEE, WEEKDAY
```

```
create.select(EMPLOYEE.NAME, WEEKDAY.NAME)
      .from(EMPLOYEE, WEEKDAY)
      .fetch();
```

It is usually recommended to prefer the CROSS JOIN syntax in order to clearly communicate intent.

Dialect support

This example using jOOQ:

```
select(BOOK.ID, AUTHOR.ID).from(BOOK.crossJoin(AUTHOR))
```

Translates to the following dialect specific expressions:

ASE

```
SELECT BOOK.ID, AUTHOR.ID
FROM BOOK
  JOIN AUTHOR
    ON 1 = 1
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID, AUTHOR.ID
FROM BOOK
  CROSS JOIN AUTHOR
```

Access, DuckDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.2. INNER JOIN

An INNER JOIN or just JOIN works like a CROSS JOIN, but adds a predicate of some sort filtering out unwanted combinations. This is the most popular way to join tables, as we hardly ever want to combine arbitrary rows from both tables, but the ones that have some relationship with each other, e.g. a FOREIGN KEY reference match.

```
SELECT *
FROM AUTHOR
JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The above query will return all authors and their books. True to the nature of an INNER JOIN, authors without books are excluded as well as books without authors (if the FOREIGN KEY is optional).

The result might look like this:

```
+------------+-----------+--------------+
| FIRST_NAME | LAST_NAME | TITLE        |
+------------+-----------+--------------+
| George     | Orwell    | 1984         |
| George     | Orwell    | Animal Farm  |
| Paulo      | Coelho    | O Alquimista |
| Paulo      | Coelho    | Brida        |
+------------+-----------+--------------+
```

In the example, we're using the ON clause to form the JOIN predicate, but other options will be discussed in later sections as well.

The INNER keyword is optional both in SQL and in jOOQ, and does not affect the query semantics at all.

Dialect support

This example using jOOQ:

```
select(BOOK.ID, AUTHOR.ID).from(BOOK.join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
```

Translates to the following dialect specific expressions:

Access

```
SELECT BOOK.ID, AUTHOR.ID
FROM BOOK
  INNER JOIN AUTHOR
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB,

MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT BOOK.ID, AUTHOR.ID
FROM BOOK
  JOIN AUTHOR
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.3. OUTER JOIN

OUTER JOIN allows for producing some additional rows when an INNER JOIN does not match. There are 3 types of OUTER JOIN:

- LEFT JOIN or LEFT OUTER JOIN: Always produce ***all*** rows from the left side of the join, and only matched rows from the right side of the join
- RIGHT JOIN or RIGHT OUTER JOIN: Always produce ***all*** rows from the right side of the join, and only matched rows from the left side of the join
- FULL JOIN or FULL OUTER JOIN: Always produce ***all*** rows from both left and right side of the join

The OUTER keyword is optional both in SQL and in jOOQ, and does not affect the query semantics at all.

This is best explained by example.

LEFT JOIN

LEFT JOIN is the most popular among the OUTER JOIN types.

The following query produces ***all*** authors, and possibly, their books:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  BOOK.TITLE
FROM AUTHOR
LEFT JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME,
         BOOK.TITLE)
      .from(AUTHOR)
      .leftJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The result might look like this:

```
+------------+-----------+--------------+
| FIRST_NAME | LAST_NAME | TITLE        |
+------------+-----------+--------------+
| George     | Orwell    | 1984         |
| George     | Orwell    | Animal Farm  |
| Paulo      | Coelho    | O Alquimista |
| Paulo      | Coelho    | Brida        | <-- Above rows are also produced by INNER JOIN
| Jane       | Austen    |              | <-- This row is only produced by LEFT JOIN or FULL JOIN
+------------+-----------+--------------+
```

As can be seen, ***all*** rows from the ***left*** side of the join (authors) are produced, including the ones that do not have any matches on the right side of the join (books). We don't have any books for Jane Austen yet, but Jane Austen is in the result set. She wouldn't be if this were an INNER JOIN.

© 2009 - 2025 by Data Geekery™ GmbH.

RIGHT JOIN

RIGHT JOIN is just the inverse of a LEFT JOIN, and is hardly ever used.

The following query produces ***all*** books, and possibly, their authors:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  BOOK.TITLE
FROM AUTHOR
RIGHT JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME,
         BOOK.TITLE)
      .from(AUTHOR)
      .rightJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The result might look like this:

```
+------------+-----------+--------------------+
| FIRST_NAME | LAST_NAME | TITLE              |
+------------+-----------+--------------------+
| George     | Orwell    | 1984               |
| George     | Orwell    | Animal Farm        |
| Paulo      | Coelho    | O Alquimista       |
| Paulo      | Coelho    | Brida              | <-- Above rows are also produced by INNER JOIN
|            |           | The Arabian Nights | <-- This row is only produced by RIGHT JOIN or FULL JOIN
+------------+-----------+--------------------+
```

As can be seen, ***all*** rows from the ***right*** side of the join (books) are produced, including the ones that do not have any matches on the left side of the join (authors). The Arabian Night does not have a specific author, but it is still in the result set. It wouldn't be if this were an INNER JOIN.

Not that a RIGHT JOIN is just an inversed LEFT JOIN, and you would be much more likely to write the same query like this, with no semantic difference:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  BOOK.TITLE
FROM BOOK
LEFT JOIN AUTHOR ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME,
         BOOK.TITLE)
      .from(BOOK)
      .leftJoin(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

There are complex join trees where a RIGHT JOIN may make things simpler, but in most cases, it only complicates readability and maintainability of your query.

FULL JOIN

FULL JOIN is an occasionally useful way to join two tables when no rows from either table should be omitted. This can be useful e.g. to compare two data sets.

The following query produces ***all*** authors and ***all*** books:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  BOOK.TITLE
FROM AUTHOR
FULL JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME,
         BOOK.TITLE)
      .from(AUTHOR)
      .fullJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The result might look like this:

```
+------------+-----------+--------------------+
| FIRST_NAME | LAST_NAME | TITLE              |
+------------+-----------+--------------------+
| George     | Orwell    | 1984               |
| George     | Orwell    | Animal Farm        |
| Paulo      | Coelho    | O Alquimista       |
| Paulo      | Coelho    | Brida              | <-- Above rows are also produced by INNER JOIN
| Jane       | Austen    |                    | <-- This row is only produced by LEFT JOIN or FULL JOIN
|            |           | The Arabian Nights | <-- This row is only produced by RIGHT JOIN or FULL JOIN
+------------+-----------+--------------------+
```

As can be seen, ***all*** rows from the ***left*** side of the join (authors) as well as from the ***right*** side of the join (books) are produced, including the ones that do not have any matches on the respective other side of the join.

Dialect support

This example using jOOQ:

```
select(BOOK.ID, AUTHOR.ID).from(BOOK.leftJoin(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT BOOK.ID, AUTHOR.ID
FROM BOOK
  LEFT OUTER JOIN AUTHOR
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.4. SEMI JOIN

Relational algebra defines a SEMI JOIN operation that regrettably didn't make it into standard SQL (yet), though it is easy to emulate using the EXISTS predicate or IN predicate, which is what most people are doing.

jOOQ offers a convenient LEFT SEMI JOIN operator to match the relational algebra semantics. The following query will produce all authors that have books (but doesn't produce any books):

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
FROM AUTHOR
WHERE EXISTS (
  SELECT * FROM BOOK WHERE BOOK.AUTHOR_ID = AUTHOR.ID
)
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME
      )
      .from(AUTHOR)
      .leftSemiJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The result might look like this:

```
+------------+-----------+
| FIRST_NAME | LAST_NAME |
+------------+-----------+
| George     | Orwell    |
| Paulo      | Coelho    |
+------------+-----------+
```

Of course, you can form an equivalent query using EXISTS or IN as well in jOOQ. It is also possible to achieve SEMI JOIN semantics by using an INNER JOIN, and possibly the SELECT DISTINCT clause, but chances are, that query is slower and incorrect (e.g. removing too many distinct rows). A SEMI JOIN both using jOOQ's convenience syntax or the equivalent SQL emulation using EXISTS or IN are semantically more precise and should be preferred.

SEMI JOIN is the inverse of the ANTI JOIN operator.

Dialect support

This example using jOOQ:

```
select(AUTHOR.ID).from(AUTHOR).leftSemiJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT AUTHOR.ID
FROM AUTHOR
WHERE EXISTS (
  SELECT 1
  FROM BOOK
  WHERE BOOK.AUTHOR_ID = AUTHOR.ID
)
```

ClickHouse, Databricks

```
SELECT AUTHOR.ID
FROM AUTHOR
  LEFT SEMI JOIN BOOK
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

DuckDB

```
SELECT AUTHOR.ID
FROM AUTHOR
  SEMI JOIN BOOK
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.5. ANTI JOIN

Relational algebra defines a ANTI JOIN operation that regrettably didn't make it into standard SQL (yet), though it is easy to emulate using the NOT EXISTS predicate. Unlike SEMI JOIN, it is not advised to use the NOT IN predicate to emulate ANTI JOIN, because that risks being incorrect in the presence of NULL values, a mistake that can be very subtle and thus hard to find.

jOOQ offers a convenient LEFT ANTI JOIN operator to match the relational algebra semantics. The following query will produce all authors that have no books:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
FROM AUTHOR
WHERE NOT EXISTS (
  SELECT * FROM BOOK WHERE BOOK.AUTHOR_ID = AUTHOR.ID
)
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME
      )
      .from(AUTHOR)
      .leftAntiJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

The result might look like this, i.e. we might have an author Jane Austen in our database, but we don't have any books for her yet:

```
+------------+-----------+
| FIRST_NAME | LAST_NAME |
+------------+-----------+
| Jane       | Austen    |
+------------+-----------+
```

Of course, you can form an equivalent query using NOT EXISTS as well in jOOQ. It is also possible to achieve ANTI JOIN semantics by using an LEFT JOIN and a NULL predicate on the anti joined table's primary key placed outside of the ON clause, though that might be a bit esoteric and hard to read:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
FROM AUTHOR
LEFT JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
WHERE BOOK.ID IS NULL
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .leftJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.ID.isNull())
      .fetch();
```

Think of the LEFT JOIN example result:

```
+------------+-----------+--------------+
| FIRST_NAME | LAST_NAME | TITLE        |
+------------+-----------+--------------+
| George     | Orwell    | 1984         |
| George     | Orwell    | Animal Farm  |
| Paulo      | Coelho    | O Alquimista |
| Paulo      | Coelho    | Brida        | <-- Reject all of the above where we have BOOK.ID IS NOT NULL
| Jane       | Austen    |              | <-- Keep only this row, where BOOK.ID IS NULL
+------------+-----------+--------------+
```

As can be seen, no DISTINCT is required to remove duplicates, because there's always only 1 row for an author without books.

ANTI JOIN is the inverse of the SEMI JOIN operator.

Dialect support

This example using jOOQ:

```
select(AUTHOR.ID).from(AUTHOR).leftAntiJoin(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
SELECT AUTHOR.ID
FROM AUTHOR
WHERE NOT EXISTS (
  SELECT 1
  FROM BOOK
  WHERE BOOK.AUTHOR_ID = AUTHOR.ID
)
```

ClickHouse, Databricks

```
SELECT AUTHOR.ID
FROM AUTHOR
  LEFT ANTI JOIN BOOK
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

DuckDB

```
SELECT AUTHOR.ID
FROM AUTHOR
  ANTI JOIN BOOK
    ON BOOK.AUTHOR_ID = AUTHOR.ID
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.6. ON clause

All of INNER JOIN, OUTER JOIN, SEMI JOIN, ANTI JOIN require a join predicate.

One way to supply this join predicate is the ON clause, which offers most flexibility. The following example shows how to "equi join" the author and books tables based on their FOREIGN KEY relationship:

```
SELECT *
FROM AUTHOR
JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

But in most dialects, any type of join predicate is possible in ON to specify what rows should be produced by the join operation. Note that while for INNER JOIN, the predicates in the ON clause and the predicates in the WHERE clause have the same effect, this isn't true for all the other join types, including OUTER

JOIN, SEMI JOIN, ANTI JOIN. For example, the following query will list ***all*** authors and their books, but only if the book was published before the year 1950:

```
SELECT *
FROM AUTHOR
LEFT JOIN BOOK
  ON BOOK.AUTHOR_ID = AUTHOR.ID
  AND BOOK.PUBLISHED_IN < 1950
```

```
create.select()
      .from(AUTHOR)
      .leftJoin(BOOK)
        .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
        .and(BOOK.PUBLISHED_IN.lt(1950))
      .fetch();
```

The result might look like this:

```
+------------+-----------+--------------+
| FIRST_NAME | LAST_NAME | TITLE        |
+------------+-----------+--------------+
| George     | Orwell    | 1984         |
| George     | Orwell    | Animal Farm  | <-- This author's books were all published before 1950
| Paulo      | Coelho    |              | <-- This author's books were published after 1950
+------------+-----------+--------------+
```

We still get ***all*** the authors, but only the books that fulfil the ON predicate. This is very different from putting that additional predicate in the WHERE clause:

```
SELECT *
FROM AUTHOR
LEFT JOIN BOOK
  ON BOOK.AUTHOR_ID = AUTHOR.ID
WHERE BOOK.PUBLISHED_IN < 1950
```

```
create.select()
      .from(AUTHOR)
      .leftJoin(BOOK)
        .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.PUBLISHED_IN.lt(1950))
      .fetch();
```

The result might now look like this:

```
+------------+-----------+--------------+
| FIRST_NAME | LAST_NAME | TITLE        |
+------------+-----------+--------------+
| George     | Orwell    | 1984         |
| George     | Orwell    | Animal Farm  | <-- This author's books were all published before 1950
+------------+-----------+--------------+
```

Now the predicate is applied ***after*** the join operator, not ***as a part of*** the join operator, so it's just an ordinary predicate.

3.7.3.7. ON KEY clause

> *(!) The ON KEY clause can quickly produce ambiguities as the implicit key path between two tables in a complex join tree isn't always unique. This can even happen for queries that have worked in the past, but due to new FOREIGN KEY constraints being added to tables, will stop working. Use this clause with caution!*

All of INNER JOIN, OUTER JOIN, SEMI JOIN, ANTI JOIN require a join predicate.

One way to supply this join predicate is the ON KEY clause, which allows for conveniently joining two tables based on their FOREIGN KEY relationship, assuming the relevant meta data is known to jOOQ via code generation:

```
SELECT *
FROM AUTHOR
JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).onKey()
      .fetch();
```

There are different overloads of this onKey() method. The above one is applicable when there are no ambiguous paths between the two joined tables. If there are several FOREIGN KEY declarations (e.g. a book has an AUTHOR_ID and a CO_AUTHOR_ID), then you can pass the org.jooq.ForeignKey reference to the method, instead, to resolve the ambiguity.

```
SELECT *
FROM AUTHOR
JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).onKey(Keys.FK_BOOK_AUTHOR)
      .fetch();
```

A similar way to join between tables by using the FOREIGN KEY information is implicit JOIN, which offers path-based navigational expressions from child table to parent table. Unlike the ON KEY syntax, implicit joins will never run into ambiguities.

3.7.3.8. USING clause

(!) The USING clause can quickly produce ambiguities as the column names between two tables in a complex join tree aren't always unique. This can even happen for queries that have worked in the past, but due to new columns being added to tables, will stop working. Use this clause with caution!

All of INNER JOIN, OUTER JOIN, SEMI JOIN, ANTI JOIN require a join predicate.

One way to supply this join predicate is the USING clause, which allows for specifying a set of column names that are common to both tables, based on which to form a join predicate. Assuming we called our AUTHOR.ID column AUTHOR.AUTHOR_ID instead:

```
SELECT *
FROM AUTHOR
JOIN BOOK USING (AUTHOR_ID)
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).using(AUTHOR.AUTHOR_ID)
      .fetch();
```

There is a certain risk of ambiguities as well in more complex join trees, but in simple cases, this can be a very convenient way to join tables if you design your schema accordingly. The is a good example where all FOREIGN KEY columns share the referenced PRIMARY KEY column's names.

3.7.3.9. NATURAL clause

(!) The NATURAL KEY operator can quickly produce ambiguities as the column names between two tables in a complex join tree aren't always unique, nor should they be included in a JOIN predicate (e.g. LAST_UPDATE or other technical columns, present on every table). This can even happen for queries that have worked in the past, but due to new columns being added to tables, will stop

working. In fact, it's very hard to design a schema to allow for using NATURAL JOIN. Use this clause with caution!

All of INNER JOIN, OUTER JOIN, SEMI JOIN, ANTI JOIN require a join predicate.

One way to supply this join predicate is the NATURAL clause, which works like USING clause, except that it discovers shared column names implicitly from the table metadata. Assuming we called our AUTHOR.ID column AUTHOR.AUTHOR_ID instead:

```
SELECT *
FROM AUTHOR
NATURAL JOIN BOOK
```

```
create.select()
      .from(AUTHOR)
      .naturalJoin(BOOK)
      .fetch();
```

There is a high risk of ambiguities even in simple join trees, which is why this syntax is hardly ever used. It can be very rarely useful combined with FULL JOIN to form a NATURAL FULL JOIN, which can create a sort of SQL-style untagged union type between two row types. A bit esoteric for every day usage.

3.7.3.10. LATERAL

LATERAL is a SQL standard table operator to wrap derived tables (or other table expressions, in some dialects), such that the tables and columns declared ***before*** the LATERAL derived table become in scope. See APPLY for an alternative, SQL Server specific syntax.

An example:

```
SELECT *
FROM
  AUTHOR,

  -- All previous objects (i.e. AUTHOR)
  -- are now in scope for the following subquery
  LATERAL (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID -- AUTHOR is in scope
  );
```

```
DSL.using(configuration)
   .select()
   .from(
      AUTHOR,
      lateral(
        select(count()
        .from(BOOK)
        .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
      )
   )
   .fetch();
```

This is most useful for:

- TOP N per category queries, which are harder to implement otherwise
- Local column variables
- Calling table valued functions on a row-by-row basis

Dialect support

This example using jOOQ:

```
select().from(AUTHOR, lateral(selectCount().from(BOOK).where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Databricks, Firebird, Hana, MySQL, Oracle, Postgres, Snowflake, Sybase, Trino, YugabyteDB

```
SELECT
  AUTHOR.ID,
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  AUTHOR.DATE_OF_BIRTH,
  AUTHOR.YEAR_OF_BIRTH,
  AUTHOR.DISTINGUISHED,
  alias_124651337.count
FROM
  AUTHOR,
  LATERAL (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) alias_124651337
```

SQLDataWarehouse, SQLServer

```
SELECT
  AUTHOR.ID,
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  AUTHOR.DATE_OF_BIRTH,
  AUTHOR.YEAR_OF_BIRTH,
  AUTHOR.DISTINGUISHED,
  alias_124651337.count
FROM AUTHOR
  CROSS APPLY (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) alias_124651337
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Derby, DuckDB, Exasol, H2, HSQLDB, Informix, MariaDB, MemSQL, Redshift, SQLite, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.11. APPLY

APPLY (specifically, CROSS APPLY or OUTER APPLY) is the SQL Server specific syntax for the SQL standard LATERAL derived table syntax.

An example:

```
SELECT *
FROM
  AUTHOR

  -- All previous objects (i.e. AUTHOR)
  -- are now in scope for the following subquery
  CROSS APPLY (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID -- AUTHOR is in scope
  );
```

```
DSL.using(configuration)
   .select()
   .from(
      AUTHOR
      .crossApply(
        select(count()
        .from(BOOK)
        .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
      )
   )
   .fetch();
```

This is most useful for:

- TOP N per category queries, which are harder to implement otherwise
- Local column variables
- Calling table valued functions on a row-by-row basis

Dialect support

This example using jOOQ:

```
selectFrom(AUTHOR.crossApply(selectCount().from(BOOK).where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))))
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Databricks, Firebird, Hana, Postgres, Snowflake, Trino, YugabyteDB

```
SELECT
  AUTHOR.ID,
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  AUTHOR.DATE_OF_BIRTH,
  AUTHOR.YEAR_OF_BIRTH,
  AUTHOR.DISTINGUISHED,
  alias_124651337.count
FROM AUTHOR
  CROSS JOIN LATERAL (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) alias_124651337
```

Oracle, SQLDataWarehouse, SQLServer, Sybase

```
SELECT
  AUTHOR.ID,
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  AUTHOR.DATE_OF_BIRTH,
  AUTHOR.YEAR_OF_BIRTH,
  AUTHOR.DISTINGUISHED,
  alias_124651337.count
FROM AUTHOR
  CROSS APPLY (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) alias_124651337
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Derby, DuckDB, Exasol, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLite, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.3.12. PARTITION BY

Standard SQL (e.g. implemented by Oracle) ships with a special syntax available for OUTER JOIN clauses. This can be used to fill gaps for simplified analytical calculations. jOOQ only supports putting the PARTITION BY clause to the right of the OUTER JOIN clause. The following example will create at least one record per AUTHOR and per existing value in BOOK.PUBLISHED_IN, regardless if an AUTHOR has actually published a book in that year.

```
SELECT *
FROM AUTHOR
LEFT OUTER JOIN BOOK
PARTITION BY (PUBLISHED_IN)
ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .leftOuterJoin(BOOK)
      .partitionBy(BOOK.PUBLISHED_IN)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

3.7.4. The VALUES() table constructor

Some databases allow for expressing in-memory temporary tables using a VALUES() constructor. This constructor usually works the same way as the VALUES() clause known from the INSERT statement or from the MERGE statement. With jOOQ, you can also use the VALUES() table constructor, to create tables that can be used in a SELECT statement's FROM clause:

```
SELECT a, b
FROM VALUES(1, 'a'),
           (2, 'b') t(a, b)
```

```
create.select()
      .from(values(row(1, "a"),
                   row(2, "b")).as("t", "a", "b"))
      .fetch();
```

Note, that it is usually quite useful to provide column aliases ("derived column lists") along with the table alias for the VALUES() constructor.

Dialect support

This example using jOOQ:

```
selectFrom(values(row(1, "a"), row(2, "b")).as("t", "a", "b"))
```

Translates to the following dialect specific expressions:

Access

```
SELECT t.a, t.b
FROM (
  SELECT
    t.*
  FROM (
    SELECT
      1 a,
      'a' b
    FROM (
      SELECT count(*) dual
      FROM MSysResources
    ) AS dual
    UNION ALL
    SELECT 2, 'b'
    FROM (
      SELECT count(*) dual
      FROM MSysResources
    ) AS dual
  ) t
) t
```

ASE, Redshift, SQLDataWarehouse, Vertica

```
SELECT t.a, t.b
FROM (
  SELECT 1, 'a'
  UNION ALL
  SELECT 2, 'b'
) t (a, b)
```

Aurora MySQL, MemSQL

```
SELECT t.a, t.b
FROM (
  SELECT
    t.*
  FROM (
    SELECT
      1 a,
      'a' b
    FROM DUAL
    UNION ALL
    SELECT 2, 'b'
    FROM DUAL
  ) t
) t
```

Aurora Postgres, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Oracle, Postgres, SQLServer, Snowflake, Trino, YugabyteDB

```
SELECT t.a, t.b
FROM (
  VALUES
    (1, 'a'),
    (2, 'b')
) t (a, b)
```

© 2009 - 2025 by Data Geekery™ GmbH.

BigQuery

```
SELECT t.a, t.b
FROM (
  SELECT
    null a,
    null b
  FROM UNNEST([STRUCT(1 AS dual)]) AS dual
  WHERE FALSE
  UNION ALL
  SELECT *
  FROM UNNEST ([
    STRUCT (1, 'a'),
    STRUCT (2, 'b')
  ]) t
) t
```

ClickHouse, MariaDB

```
SELECT t.a, t.b
FROM (
  SELECT
    t.*
  FROM (
    SELECT
      1 a,
      'a' b
    UNION ALL
    SELECT 2, 'b'
  ) t
) t
```

Firebird

```
SELECT t.a, t.b
FROM (
  SELECT 1, 'a'
  FROM RDB$DATABASE
  UNION ALL
  SELECT 2, 'b'
  FROM RDB$DATABASE
) t (a, b)
```

Hana

```
SELECT t.a, t.b
FROM (
  SELECT
    t.*
  FROM (
    SELECT
      1 a,
      'a' b
    FROM SYS.DUMMY
    UNION ALL
    SELECT 2, 'b'
    FROM SYS.DUMMY
  ) t
) t
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
SELECT t.a, t.b
FROM (
  TABLE (MULTISET {
    ROW (1, 'a'),
    ROW (2, 'b')
  })
) t (a, b)
```

MySQL

```
SELECT t.a, t.b
FROM (
  VALUES
    ROW (1, 'a'),
    ROW (2, 'b')
) t (a, b)
```

SQLite

```
SELECT t.a, t.b
FROM (
  SELECT
    null a,
    null b
  WHERE 1 = 0
  UNION ALL
  SELECT *
  FROM (
    VALUES
      (1, 'a'),
      (2, 'b')
  ) t
) t
```

Sybase

```
SELECT t.a, t.b
FROM (
  SELECT 1, 'a'
  FROM SYS.DUMMY
  UNION ALL
  SELECT 2, 'b'
  FROM SYS.DUMMY
) t (a, b)
```

Teradata

```
SELECT t.a, t.b
FROM (
  SELECT 1, 'a'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
  UNION ALL
  SELECT 2, 'b'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
) t (a, b)
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.5. Derived tables

A derived table is a nested SELECT in the FROM clause, i.e. it can be used as a table expression. As such, it works differently from a scalar subquery, which is a column expression.

```
SELECT nested.* FROM (
      SELECT AUTHOR_ID, count(*) books
        FROM BOOK
    GROUP BY AUTHOR_ID
) nested
ORDER BY nested.books DESC
```

```
Table<?> nested =
    create.select(BOOK.AUTHOR_ID, count().as("books"))
          .from(BOOK)
          .groupBy(BOOK.AUTHOR_ID).asTable("nested");

create.select(nested.fields())
      .from(nested)
      .orderBy(nested.field("books"))
      .fetch();
```

Dialect support

This example using jOOQ:

```
selectFrom(select(BOOK.ID).from(BOOK).asTable("t"))
```

Translates to the following dialect specific expressions:

All dialects

```
SELECT t.ID
FROM (
  SELECT BOOK.ID
  FROM BOOK
) t
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.6. Inline derived tables

An inline derived table is a Table expression with a WHERE clause, both of which can be inlined into the calling query in a lot of cases.

The main use-case for this feature is dynamic SQL, where you can add predicates to tables without call sites noticing, and without giving up on type safety.

*(!) Code generation support for inline derived tables was added in jOOQ 3.19, only. The following examples dereferenece columns in a type safe way from generated code. Earlier versions of jOOQ require using **Table.field(Field)**, or a similar method.*

```
-- In SQL, the derived table is inlined
SELECT
  BOOK.ID,
  BOOK.TITLE
FROM BOOK
WHERE BOOK.TITLE LIKE 'A%'
```

```
// This book reference can be supplied dynamically
Book aBooks = BOOK.where(BOOK.TITLE.like("A%"));

create.select(aBooks.ID, aBooks.TITLE)
      .from(aBooks)
      .fetch();
```

Whenever the context requires, the inline derived table generates an explicit derived table, e.g. when used in a LEFT JOIN:

```
-- In SQL, the derived table created explicitly
SELECT BOOK.ID, BOOK.TITLE
FROM AUTHOR
LEFT JOIN (
  SELECT * FROM BOOK WHERE BOOK.TITLE LIKE 'A%'
) BOOK
ON AUTHOR.ID = BOOK.AUTHOR_ID
```

```
// This book reference can be supplied dynamically
Book aBooks = BOOK.where(BOOK.TITLE.like("A%"));

create.select(AUTHOR.ID, aBooks.ID, aBooks.TITLE)
      .from(AUTHOR)
      .leftJoin(aBooks).on(AUTHOR.ID.eq(aBooks.AUTHOR_ID))
      .fetch();
```

3.7.7. The Oracle PIVOT clause

If you are closely coupling your application to an Oracle database, you can take advantage of some Oracle-specific features, such as the PIVOT clause, used for statistical analyses. The formal syntax definition is as follows:

```
-- SELECT ..
     FROM table PIVOT (aggregateFunction [, aggregateFunction] FOR column IN (expression [, expression]))
--  WHERE ..
```

The PIVOT clause is available from the org.jooq.Table type, as pivoting is done directly on a table. Currently, only Oracle's PIVOT clause is supported. Support for SQL Server's slightly different PIVOT clause will be added later. Also, jOOQ may emulate PIVOT for other dialects in the future.

3.7.8. Relational division

There is one operation in relational algebra that is not given a lot of attention, because it is rarely used in real-world applications. It is the relational division, the opposite operation of the cross product (or, relational multiplication). The following is an approximate definition of a relational division:

```
Assume the following cross join / cartesian product
C = A × B

Then it can be said that
A = C ÷ B
B = C ÷ A
```

With jOOQ, you can simplify using relational divisions by using the following syntax:

```
C.divideBy(B).on(C.ID.eq(B.C_ID)).returning(C.TEXT)
```

The above roughly translates to

```
SELECT DISTINCT C.TEXT FROM C "c1"
WHERE NOT EXISTS (
  SELECT 1 FROM B
  WHERE NOT EXISTS (
    SELECT 1 FROM C "c2"
    WHERE "c2".TEXT = "c1".TEXT
    AND "c2".ID = B.C_ID
  )
)
```

Or in plain text: Find those TEXT values in C whose ID's correspond to all ID's in B. Note that from the above SQL statement, it is immediately clear that proper indexing is of the essence. Be sure to have indexes on all columns referenced from the on(...) and returning(...) clauses.

For more information about relational division and some nice, real-life examples, see

- https://en.wikipedia.org/wiki/Relational_algebra#Division
- https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/divided-we-stand-the-sql-of-relational-division/

3.7.9. Array and cursor unnesting

The SQL standard specifies how SQL databases should implement ARRAY and TABLE types, as well as CURSOR types. Put simply, a CURSOR is a pointer to any materialised table expression. Depending on the cursor's features, this table expression can be scrolled through in both directions, records can be locked, updated, removed, inserted, etc. Often, CURSOR types contain s, whereas ARRAY and TABLE types contain simple scalar values, although that is not a requirement

ARRAY types in SQL are similar to Java's array types. They contain a "component type" or "element type" and a "dimension". This sort of ARRAY type is implemented in H2, HSQLDB and Postgres and supported by jOOQ as such. Oracle uses strongly-typed arrays, which means that an ARRAY type (VARRAY or TABLE type) has a name and possibly a maximum capacity associated with it.

Unnesting array and cursor types

The real power of these types become more obvious when you fetch them from stored procedures to unnest them as table expressions and use them in your FROM clause. An example is given here, where Oracle's DBMS_XPLAN package is used to fetch a cursor containing data about the most recent execution plan:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(null, null, 'ALLSTATS'));
```

```
create.select()
      .from(table(DbmsXplan.displayCursor(null, null,
 "ALLSTATS"))
      .fetch();
```

Note, in order to access the DbmsXplan package, you can use the code generator to generate Oracle's SYS schema.

3.7.10. Table-valued functions

Some databases support functions that can produce tables for use in arbitrary SELECT statements. jOOQ supports these functions out-of-the-box for such databases. For instance, in SQL Server, the following function produces a table of (ID, TITLE) columns containing either all the books or just one book by ID:

```
CREATE FUNCTION f_books (@id INTEGER)
RETURNS @out_table TABLE (
    id INTEGER,
    title VARCHAR(400)
)
AS
BEGIN
    INSERT @out_table
    SELECT id, title
    FROM book
    WHERE @id IS NULL OR id = @id
    ORDER BY id
    RETURN
END
```

The jOOQ code generator will now produce a generated table from the above, which can be used as a SQL function:

```
// Fetching all books records
Result<FBooksRecord> r1 = create.selectFrom(fBooks(null)).fetch();

// Lateral joining the table-valued function to another table using CROSS APPLY:
create.select(BOOK.ID, F_BOOKS.TITLE)
      .from(BOOK.crossApply(fBooks(BOOK.ID)))
      .fetch();
```

3.7.11. GENERATE_SERIES

A nice built-in table-valued function from the PostgreSQL dialect is the GENERATE_SERIES() function, which allows for creating a table for a range of numeric values. Many dialects have some way of generating such a table, and if not, it can be emulated using recursive SQL.

```
// Values from 1 to 10
Result<Record1<Integer>> r = create.selectFrom(generateSeries(1, 10)).fetch();
```

Dialect support

This example using jOOQ:

```
selectFrom(generateSeries(1, 10))
```

Translates to the following dialect specific expressions:

Aurora Postgres, DuckDB, Postgres, YugabyteDB

```
SELECT generate_series.generate_series
FROM generate_series(1, 10)
```

BigQuery

```
SELECT generate_series.generate_series
FROM (
  SELECT null generate_series
  FROM UNNEST([STRUCT(1 AS dual)]) AS dual
  WHERE FALSE
  UNION ALL
  SELECT *
  FROM unnest(generate_array(1, 10)) generate_series
) generate_series
```

ClickHouse

```
SELECT generate_series.generate_series
FROM (
  SELECT null generate_series
  WHERE FALSE
  UNION ALL
  SELECT *
  FROM (
    SELECT CAST(number AS Nullable(integer)) generate_series
    FROM numbers(1, (10 + 1))
  ) generate_series
) generate_series
```

CockroachDB

```
SELECT generate_series.generate_series
FROM generate_series(1, 10) generate_series (generate_series)
```

Databricks

```
SELECT generate_series.generate_series
FROM explode(sequence(1, 10)) generate_series (generate_series)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2

```
SELECT generate_series.generate_series
FROM (
  WITH
    generate_series(generate_series) AS (
      SELECT 1
      FROM SYSIBM.DUAL
      UNION ALL
      SELECT (generate_series + 1)
      FROM generate_series
      WHERE generate_series < 10
    )
  SELECT generate_series
  FROM generate_series
) generate_series
```

Exasol, Oracle

```
SELECT generate_series.generate_series
FROM (
  SELECT (level + (1 - 1)) generate_series
  FROM DUAL
  CONNECT BY level <= ((10 + 1) - 1)
) generate_series
```

Firebird

```
SELECT generate_series.generate_series
FROM (
  WITH RECURSIVE
    generate_series(generate_series) AS (
      SELECT 1
      FROM RDB$DATABASE
      UNION ALL
      SELECT (generate_series + 1)
      FROM generate_series
      WHERE generate_series < 10
    )
  SELECT generate_series
  FROM generate_series
) generate_series
```

H2

```
SELECT generate_series.generate_series
FROM system_range(1, 10) generate_series (generate_series)
```

HSQLDB

```
SELECT generate_series.generate_series
FROM (
  WITH RECURSIVE
    generate_series(generate_series) AS (
      SELECT 1
      FROM (VALUES (1)) AS dual (dual)
      UNION ALL
      SELECT (generate_series + 1)
      FROM generate_series
      WHERE generate_series < 10
    )
  SELECT generate_series
  FROM generate_series
) generate_series
```

Informix

```
SELECT generate_series.generate_series
FROM (
  SELECT (level + (1 - 1)) generate_series
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
  CONNECT BY level <= ((10 + 1) - 1)
) generate_series
```

MariaDB, MySQL, SQLite, Trino

```
SELECT generate_series.generate_series
FROM (
  WITH RECURSIVE
    generate_series(generate_series) AS (
      SELECT 1
      UNION ALL
      SELECT (generate_series + 1)
      FROM generate_series
      WHERE generate_series < 10
    )
  SELECT generate_series
  FROM generate_series
) generate_series
```

Snowflake

```
SELECT generate_series.generate_series
FROM (
  SELECT ((seq4() + 1) + (1 - 1)) generate_series
  FROM TABLE(generator(rowcount => (10 - (1 - 1))))
) generate_series (generate_series)
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse

```
WITH
  generate_series(generate_series) AS (
    SELECT 1
    UNION ALL
    SELECT (generate_series + 1)
    FROM generate_series
    WHERE generate_series < 10
  )
SELECT generate_series.generate_series
FROM (
  SELECT generate_series
  FROM generate_series
) generate_series
```

SQLServer

```
SELECT generate_series.generate_series
FROM (
  SELECT *
  FROM generate_series(1, 10)
) generate_series (generate_series)
```

Sybase

```
SELECT generate_series.generate_series
FROM (
  WITH RECURSIVE
    generate_series(generate_series) AS (
      SELECT 1
      FROM SYS.DUMMY
      UNION ALL
      SELECT (generate_series + 1)
      FROM generate_series
      WHERE generate_series < 10
    )
  SELECT generate_series
  FROM generate_series
) generate_series
```

Teradata

```
WITH RECURSIVE
  generate_series(generate_series) AS (
    SELECT 1
    FROM (
      SELECT 1 AS "dual"
    ) AS "dual"
    UNION ALL
    SELECT (generate_series + 1)
    FROM generate_series
    WHERE generate_series < 10
  )
SELECT generate_series.generate_series
FROM (
  SELECT generate_series
  FROM generate_series
) generate_series
```

© 2009 - 2025 by Data Geekery™ GmbH.

ASE, Access, Aurora MySQL, Derby, Hana, MemSQL, Redshift, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.7.12. The DUAL table

The SQL standard specifies that the FROM clause is mandatory in a SELECT statement. However, in the real world, there exist three types of databases:

- The ones that always require a FROM clause (as required by the SQL standard)
- The ones that never require a FROM clause (and still allow a WHERE clause)
- The ones that require a FROM clause only with a WHERE clause, GROUP BY clause, or HAVING clause

With jOOQ, you don't have to worry about the above distinction of SQL dialects. jOOQ never requires a FROM clause, but renders the necessary "DUAL" table, if needed. The following program shows how jOOQ renders "DUAL" tables

(!) Note, that some databases (H2, MySQL) can normally do without "DUAL". However, there exist some corner-cases with complex nested SELECT statements, where this will cause syntax errors (or parser bugs). To stay on the safe side, jOOQ will always render "dual" in those dialects.

Dialect support

This example using jOOQ:

```
select(inline(1))
```

Translates to the following dialect specific expressions:

Access

```
SELECT 1
FROM (
  SELECT count(*) dual
  FROM MSysResources
) AS dual
```

ASE, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, H2, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
SELECT 1
```

Aurora MySQL, MemSQL

```
SELECT 1
FROM DUAL
```

DB2

```
SELECT 1
FROM SYSIBM.DUAL
```

Derby

```
SELECT 1
FROM SYSIBM.SYSDUMMY1
```

Firebird

```
SELECT 1
FROM RDB$DATABASE
```

Hana, Sybase

```
SELECT 1
FROM SYS.DUMMY
```

HSQLDB

```
SELECT 1
FROM (VALUES (1)) AS dual (dual)
```

Informix

```
SELECT 1
FROM (
  SELECT 1 AS dual
  FROM systables
  WHERE (tabid = 1)
) AS dual
```

Teradata

```
SELECT 1
FROM (
  SELECT 1 AS "dual"
) AS "dual"
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8. Column expressions

Column expressions can be used in various SQL clauses in order to refer to one or several columns. This chapter explains how to form various types of column expressions with jOOQ. A particular type of column expression is given in the section about tuples or row value expressions, where an expression may have a degree of more than one.

Using column expressions in jOOQ

jOOQ allows you to freely create arbitrary column expressions using a fluent expression construction API. Many expressions can be formed as functions from DSL methods, other expressions can be formed based on a pre-existing column expression. For example:

```
// A regular table column expression
Field<String> field1 = BOOK.TITLE;

// A function created from the DSL
Field<String> field2 = trim(BOOK.TITLE);

// More complex function with advanced DSL syntax
Field<String> field4 = listAgg(BOOK.TITLE)
                          .withinGroupOrderBy(BOOK.ID.asc())
                          .over().partitionBy(AUTHOR.ID);
```

3.8.1. Table columns

Table columns are the most simple implementations of a column expression. They are mainly produced by jOOQ's code generator and can be dereferenced from the generated tables, but other ways of creating table column references are possible.

3.8.1.1. Generated table columns

The main type of column expression are the ones produced by jOOQ's code generator and can be dereferenced from the generated tables. This manual is full of examples involving table columns. Another example is given in this query:

```
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WHERE BOOK.TITLE LIKE '%SQL%'
ORDER BY BOOK.TITLE
```

```
create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .where(BOOK.TITLE.like("%SQL%"))
      .orderBy(BOOK.TITLE)
      .fetch();
```

Table columns implement a more specific interface called org.jooq.TableField, which is parameterised with its associated <R extends Record> record type, and provides access to the container org.jooq.Table instance.

See the manual's section about generated tables for more information about what is really generated by the code generator

3.8.1.2. Dereferenced table columns

Any org.jooq.Table instance that is constructed in a way to know its own columns can be used to dereference those columns. Examples include:

- Generated tables know their generated table columns
- Derived tables
- Table valued functions, including built-in ones, such as e.g. unnested arrays or GENERATE_SERIES

All of these table expressions, as well as some others, extend the org.jooq.Fields type, which allows for all of these field accessing types, like org.jooq.Table but also org.jooq.Record and others to share field accessing logic.

```
// Get fields from AUTHOR dynamically, without type safety:
Field<?> id = AUTHOR.field("ID");
Field<?> firstName = AUTHOR.field("FIRST_NAME");
```

3.8.1.3. Named table columns

When no org.jooq.Table instance is available to dereference a column from, the column expression can still be constructed on the fly, dynamically using a Name reference, and optionally a org.jooq.DataType reference:

```
// Get fields from AUTHOR dynamically, without type safety:
Field<?> id = field(name("AUTHOR", "ID"));
Field<String> firstName = field(name("AUTHOR", "FIRST_NAME"), INTEGER);
```

Note that by default, these names are quoted (among other reasons to prevent SQL injection), and thus case sensitive. For more details, please refer to the section about names and identifiers.

© 2009 - 2025 by Data Geekery™ GmbH.

3.8.2. Aliased columns

Just like tables, columns can be renamed using aliases. Here is an example:

```
  SELECT FIRST_NAME || ' ' || LAST_NAME author, COUNT(*) books
    FROM AUTHOR
    JOIN BOOK ON AUTHOR.ID = AUTHOR_ID
GROUP BY FIRST_NAME, LAST_NAME;
```

Here is how it's done with jOOQ:

```
Record record = create.select(
         concat(AUTHOR.FIRST_NAME, inline(" "), AUTHOR.LAST_NAME).as("author"),
         count().as("books"))
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .fetchAny();
```

When you alias Fields like above, you can access those Fields' values using the alias name:

```
System.out.println("Author : " + record.getValue("author"));
System.out.println("Books  : " + record.getValue("books"));
```

Unnamed column expressions

In most SQL databases, aliasing of column expressions in top level selects is optional. The database will generate a column name that is roughly based on the expression for documentation purposes (e.g. when running the query in a tool like SQL Developer), but applications cannot rely on the name explicitly. This is not a problem as columns can still be referenced by index.

In a similar fashion, jOOQ will assume an unspecified, generated column name for column expressions, based on their content.

```
-- Arithmetic expression
1 + 2

-- Correlated subquery
(SELECT 1 AS a)
```

```
// Arithmetic expression
inline(1).plus(inline(2));

// Correlated subquery
field(select(inline(1).as("a")));
```

These unnamed expressions can be used both in SQL as well as with jOOQ. However, do note that jOOQ will use Field.getName() to extract this column name from the field, when referencing the field or when nesting it in derived tables. In order to stay in full control of any such column names, it is always a good idea to provide explicit aliasing for column expressions, both in SQL as well as in jOOQ.

Rendering declarations or references

The same aliased column instance is rendered differently depending on where it is placed in the jOOQ expression tree. See the manual's section about rendering declarations vs references for more details.

© 2009 - 2025 by Data Geekery™ GmbH.

3.8.3. Cast expressions

jOOQ's source code generator tries to find the most accurate type mapping between your vendor-specific data types and a matching Java type. For instance, most VARCHAR, CHAR, CLOB types will map to String. Most BINARY, BYTEA, BLOB types will map to byte[]. NUMERIC types will default to java.math.BigDecimal, but can also be any of java.math.BigInteger, java.lang.Long, java.lang.Integer, java.lang.Short, java.lang.Byte, java.lang.Double, java.lang.Float.

Sometimes, this automatic mapping might not be what you needed, or jOOQ cannot know the type of a field. In those cases you would write SQL type CAST like this:

```
-- Let's say, your Postgres column LAST_NAME was VARCHAR(30)
-- Then you could do this:
SELECT CAST(AUTHOR.LAST_NAME AS TEXT) FROM DUAL
```

in jOOQ, you can write something like that:

```
create.select(AUTHOR.LAST_NAME.cast(VARCHAR(100))).fetch();
```

The complete CAST API in org.jooq.Field consists of these three methods:

```
public interface Field<T> {

    // Cast this field to the type of another field
    <Z> Field<Z> cast(Field<Z> field);

    // Cast this field to a given DataType
    <Z> Field<Z> cast(DataType<Z> type);

    // Cast this field to the default DataType for a given Class
    <Z> Field<Z> cast(Class<? extends Z> type);
}

// And additional convenience methods in the DSL:
public class DSL {
    <T> Field<T> cast(Object object, Field<T> field);
    <T> Field<T> cast(Object object, DataType<T> type);
    <T> Field<T> cast(Object object, Class<? extends T> type);
    <T> Field<T> castNull(Field<T> field);
    <T> Field<T> castNull(DataType<T> type);
    <T> Field<T> castNull(Class<? extends T> type);
}
```

*(!) The CAST expression converts between data types directly in SQL. If you wish to change a data type without any effect on rendered SQL, you may prefer to **coerce** your expressions.*

Dialect support

This example using jOOQ:

```
cast("1", VARCHAR(10))
```

Translates to the following dialect specific expressions:

Access

```
cstr('1')
```

ASE, Aurora Postgres, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
CAST('1' AS varchar(10))
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
CAST('1' AS char(10))
```

BigQuery

```
CAST('1' AS string)
```

ClickHouse

```
CAST('1' AS String(10))
```

CockroachDB

```
CAST('1' AS string(10))
```

Informix

```
CAST('1' AS lvarchar(10))
```

Oracle

```
CAST('1' AS varchar2(10))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.4. Datatype coercions

A slightly different use case than CAST expressions are data type coercions, which are not rendered through to generated SQL. Sometimes, you may want to pretend that a numeric value is really treated as a string value, for instance when binding a numeric bind value:

```
Field<String>  field1 = val(1).coerce(String.class);
Field<Integer> field2 = val("1").coerce(Integer.class);
```

In the above example, field1 will be treated by jOOQ as a Field<String>, binding the numeric literal 1 as a VARCHAR value. The same applies to field2, whose string literal "1" will be bound as an INTEGER value.

This technique is better than performing unsafe or rawtype casting in Java, if you cannot access the "right" field type from any given expression.

3.8.5. Collations

Many databases support "collations", which defines the sort order on character data types, such as VARCHAR.

Such databases usually allow for specifying:

- System-wide default collations
- Session-wide default collations
- Per-table specific default collations
- Per-column specific default collations
- Per-usage specific collation

The actual implementation is vendor-specific, including the way the above defaults override each other.

To accommodate most use-cases jOOQ 3.11 introduced the org.jooq.Collation type, which can be attached to a org.jooq.DataType through DataType.collate(Collation), or to a org.jooq.Field through Field.collate(Collation), for example:

```
SELECT *
FROM book
ORDER BY title COLLATE utf8_bin
```

```
create.selectFrom(BOOK)
      .orderBy(BOOK.TITLE.collate("utf8_bin"))
      .fetch();
```

3.8.6. Arithmetic expressions

Numeric arithmetic expressions

Your database can do the math for you. Arithmetic operations are implemented just like numeric functions, with similar limitations as far as type restrictions are concerned. You can use any of these operators:

```
+ - * / %
```

In order to express a SQL query like this one:

```
SELECT ((1 + 2) * (5 - 3) / 2) % 10 FROM DUAL
```

You can write something like this in jOOQ:

```
create.select(val(1).add(2).mul(val(5).sub(3)).div(2).mod(10)).fetch();
```

Operator precedence

jOOQ does not know any operator precedence (see also boolean operator precedence). All operations are evaluated from left to right, as with any object-oriented API. The two following expressions are the same:

```
   val(1).add(2) .mul(val(5).sub(3)) .div(2) .mod(10);
(((val(1).add(2)).mul(val(5).sub(3))).div(2)).mod(10);
```

Datetime arithmetic expressions

jOOQ also supports the Oracle-style syntax for adding days to a Field<? extends java.util.Date>

```
SELECT SYSDATE + 3 FROM DUAL;
```

```
create.select(currentTimestamp().add(3)).fetch();
```

For more advanced datetime arithmetic, use the DSL's timestampDiff() and dateDiff() functions, as well as jOOQ's built-in SQL standard INTERVAL data type support:

- INTERVAL YEAR TO MONTH: org.jooq.types.YearToMonth
- INTERVAL DAY TO SECOND: org.jooq.types.DayToSecond

© 2009 - 2025 by Data Geekery™ GmbH.

3.8.7. String concatenation

The SQL standard defines the concatenation operator to be an infix operator, similar to the ones we've seen in the chapter about arithmetic expressions. This operator looks like this: ||. Some other dialects do not support this operator, but expect a concat() function, instead. jOOQ renders the right operator / function, depending on your SQL dialect:

```
SELECT 'A' || 'B' || 'C' FROM DUAL
-- Or in MySQL:
SELECT concat('A', 'B', 'C') FROM DUAL
```

```
// For all RDBMS, including MySQL:
create.select(concat("A", "B", "C")).fetch();
```

3.8.8. Case sensitivity with strings

Most databases allow for specifying a COLLATION which allows for re-defining the ordering of string values. By default, ASCII, ISO, or Unicode encodings are applied to character data, and ordering is applied according to the respective encoding.

Sometimes, however, certain queries like to ignore parts of the encoding by treating upper-case and lower-case characters alike, such that ABC = abc, or such that ABC, jkl, XyZ are an ordered list of strings (case-insensitively).

For these ad-hoc ordering use-cases, most people resort to using LOWER() or UPPER() as follows:

```
-- Case-insensitive filtering:
SELECT * FROM BOOK
WHERE upper(TITLE) = 'ANIMAL FARM'

-- Case-insensitive ordering:
SELECT *
FROM AUTHOR
ORDER BY upper(FIRST_NAME), upper(LAST_NAME)
```

```
// Case-insensitive filtering:
create.selectFrom(BOOK)
      .where(upper(BOOK.TITLE).eq("ANIMAL FARM")).fetch();

// Case-insensitive ordering:
create.selectFrom(AUTHOR)
      .orderBy(upper(AUTHOR.FIRST_NAME), upper(AUTHOR.LAST_NAME))
      .fetch();
```

3.8.9. General functions

There are a variety of general functions supported by jOOQ. As discussed in the chapter about SQL dialects functions are mostly emulated in your database, in case they are not natively supported.

3.8.9.1. CHOOSE

The CHOOSE() function acts as a switch over an integer to return the nth argument. It is an abbreviated CASE expression

```
SELECT
  choose(1, 'a', 'b'),
  choose(2, 'a', 'b'),
  choose(3, 'a', 'b');
```

```
create.select(
  choose(val(1), val("a"), val("b")),
  choose(val(2), val("a"), val("b")),
  choose(val(3), val("a"), val("b"))).fetch();
```

The result being

```
+--------+--------+--------+
| choose | choose | choose |
+--------+--------+--------+
| a      | b      | {null} |
+--------+--------+--------+
```

Dialect support

This example using jOOQ:

```
choose(val(1), val("a"), val("b"))
```

Translates to the following dialect specific expressions:

Access

```
SWITCH(1 = 1, 'a', 1 = 2, 'b')
```

ASE, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
CASE 1
  WHEN 1 THEN 'a'
  WHEN 2 THEN 'b'
END
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
elt(1, 'a', 'b')
```

Databricks

```
elt(
  CASE
    WHEN 1 BETWEEN 1 AND 2 THEN 1
  END,
  'a',
  'b'
)
```

Derby

```
CASE
  WHEN 1 = 1 THEN 'a'
  WHEN 1 = 2 THEN 'b'
END
```

SQLServer

```
choose(1, 'a', 'b')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.2. COALESCE

The COALESCE() function produces the first non-NULL value from the variadic list of arguments.

```
SELECT coalesce(null, null, 1);
```

```
create.select(coalesce(null, null, 1)).fetch();
```

The result being

```
+----------+
| coalesce |
+----------+
|        1 |
+----------+
```

Dialect support

This example using jOOQ:

```
coalesce(null, null, 1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
coalesce(NULL, NULL, 1)
```

Derby

```
coalesce(?, ?, 1)
```

Informix

```
nvl(
  nvl(
    NULL,
    NULL
  ),
  1
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.3. DECODE

Some SQL dialects, including Db2, H2, Oracle know a more succinct, but maybe less readable DECODE() function with a variable number of arguments. This function works like a NULL safe CASE expression. jOOQ supports the DECODE() function and emulates it using CASE expressions in all dialects that do not have native support:

```
SELECT
  -- Oracle:
  DECODE(FIRST_NAME, 'Paulo', 'brazilian',
                     'George', 'english',
                     'unknown'),
  -- Other SQL dialects
  CASE
    WHEN FIRST_NAME IS NOT DISTINCT FROM 'Paulo'  THEN
 'brazilian'
    WHEN FIRST_NAME IS NOT DISTINCT FROM 'George' THEN 'english'
    ELSE 'unknown'
  END
FROM AUTHOR
```

```
// Use the Oracle-style DECODE() function with jOOQ.
// Note, that you will not be able to rely on type-safety
decode(
  AUTHOR.FIRST_NAME,
  "Paulo", "brazilian",
  "George", "english",
  "unknown"
);
```

See the DISTINCT predicate for details about the NULL safe semantics.

Dialect support

This example using jOOQ:

```
decode(AUTHOR.FIRST_NAME, "Paulo", "BR", "George", "EN", "unknown")
```

Translates to the following dialect specific expressions:

ASE, SQLDataWarehouse

```
CASE
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    INTERSECT
    SELECT 'Paulo' x
  ) THEN 'BR'
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    INTERSECT
    SELECT 'George' x
  ) THEN 'EN'
  ELSE 'unknown'
END
```

Aurora MySQL, MySQL

```
CASE
  WHEN (AUTHOR.FIRST_NAME <=> 'Paulo') THEN 'BR'
  WHEN (AUTHOR.FIRST_NAME <=> 'George') THEN 'EN'
  ELSE 'unknown'
END
```

Aurora Postgres, BigQuery, CockroachDB, DuckDB, Firebird, HSQLDB, Postgres, SQLServer, Snowflake, Trino, YugabyteDB

```
CASE
  WHEN AUTHOR.FIRST_NAME IS NOT DISTINCT FROM 'Paulo' THEN 'BR'
  WHEN AUTHOR.FIRST_NAME IS NOT DISTINCT FROM 'George' THEN 'EN'
  ELSE 'unknown'
END
```

ClickHouse

```
CASE
  WHEN arrayUniq(ARRAY(AUTHOR.FIRST_NAME, 'Paulo')) = 1 THEN 'BR'
  WHEN arrayUniq(ARRAY(AUTHOR.FIRST_NAME, 'George')) = 1 THEN 'EN'
  ELSE 'unknown'
END
```

DB2, Databricks, Exasol, H2, Informix, MemSQL, Oracle, Redshift, Teradata, Vertica

```
decode(
  AUTHOR.FIRST_NAME,
  'Paulo',
  'BR',
  'George',
  'EN',
  'unknown'
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
CASE
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    FROM SYSIBM.SYSDUMMY1
    INTERSECT
    SELECT 'Paulo' x
    FROM SYSIBM.SYSDUMMY1
  ) THEN 'BR'
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    FROM SYSIBM.SYSDUMMY1
    INTERSECT
    SELECT 'George' x
    FROM SYSIBM.SYSDUMMY1
  ) THEN 'EN'
  ELSE 'unknown'
END
```

Hana

```
map(
  AUTHOR.FIRST_NAME,
  'Paulo',
  'BR',
  'George',
  'EN',
  'unknown'
)
```

MariaDB

```
decode_oracle(
  AUTHOR.FIRST_NAME,
  'Paulo',
  'BR',
  'George',
  'EN',
  'unknown'
)
```

SQLite

```
CASE
  WHEN (AUTHOR.FIRST_NAME IS 'Paulo') THEN 'BR'
  WHEN (AUTHOR.FIRST_NAME IS 'George') THEN 'EN'
  ELSE 'unknown'
END
```

© 2009 - 2025 by Data Geekery™ GmbH.

Sybase

```
CASE
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    FROM SYS.DUMMY
    INTERSECT
    SELECT 'Paulo' x
    FROM SYS.DUMMY
  ) THEN 'BR'
  WHEN EXISTS (
    SELECT AUTHOR.FIRST_NAME x
    FROM SYS.DUMMY
    INTERSECT
    SELECT 'George' x
    FROM SYS.DUMMY
  ) THEN 'EN'
  ELSE 'unknown'
END
```

Access

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.4. IIF

The IIF() function checks if the first argument is TRUE to produce the second argument, or the third argument otherwise. It works in a similar way as the NVL2 function or the CASE expression

```
SELECT
  iif(1 = 1, 3, 4),
  iif(1 = 2, 3, 4);
```

```
create.select(
  iif(inline(1).eq(inline(1)), inline(3), inline(4))
  iif(inline(1).eq(inline(2)), inline(3), inline(4))).fetch();
```

The result being

```
+-----+-----+
| iif | iif |
+-----+-----+
|   3 |   4 |
+-----+-----+
```

Dialect support

This example using jOOQ:

```
iif(inline(1).eq(inline(2)), inline(3), inline(4))
```

Translates to the following dialect specific expressions:

Access, SQLServer

```
iif(1 = 2, 3, 4)
```

ASE, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
CASE
  WHEN 1 = 2 THEN 3
  ELSE 4
END
```

Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL

```
if(1 = 2, 3, 4)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.5. NULLIF

The NULLIF() function produces a NULL value if both its arguments are equal, otherwise it produces the first argument.

```
SELECT nullif(1, 1), nullif(1, 2);
```

```
create.select(nullif(1, 1), nullif(1, 2)).fetch();
```

The result being

```
+--------+--------+
| nullif | nullif |
+--------+--------+
|        |      1 |
+--------+--------+
```

Dialect support

This example using jOOQ:

```
nullif(1, 2)
```

Translates to the following dialect specific expressions:

Access

```
iif(1 = 2, NULL, 1)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
nullif(1, 2)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.6. NVL

The NVL() function (or also the ISNULL() or IFNULL() functions) produces the first argument if it is NOT NULL, otherwise the second argument. It is a special case of the COALESCE function, which takes any number of arguments.

```
SELECT nvl(null, 1);
```

```
create.select(nvl(null, 1)).fetch();
```

The result being

```
+-----+
| nvl |
+-----+
|   1 |
+-----+
```

Dialect support

This example using jOOQ:

```
nvl(null, 1)
```

Translates to the following dialect specific expressions:

Access

```
iif(NULL IS NULL, 1, NULL)
```

ASE, Aurora Postgres, CockroachDB, DuckDB, Exasol, Firebird, Hana, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
coalesce(
  NULL,
  1
)
```

Aurora MySQL, BigQuery, ClickHouse, MariaDB, MemSQL, MySQL, SQLite

```
ifnull(
  NULL,
  1
)
```

DB2, Databricks, H2, HSQLDB, Informix, Oracle

```
nvl(
  NULL,
  1
)
```

Derby

```
coalesce(
  ?,
  1
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.9.7. NVL2

The NVL2() function checks if the first argument is NOT NULL to produce the second argument, or the third argument otherwise. It works in a similar way as the IIF function or the CASE expression

```
SELECT
  nvl2(1,    2, 3),
  nvl2(null, 2, 3);
```

```
create.select(
  nvl2(val(1)             , 2, 3),
  nvl2(val((Integer) null), 2, 3)).fetch();
```

The result being

```
+------+------+
| nvl2 | nvl2 |
+------+------+
|    2 |    3 |
+------+------+
```

Dialect support

This example using jOOQ:

```
nvl2(val(1), 2, 3)
```

Translates to the following dialect specific expressions:

Access, SQLServer

```
iif(1 IS NOT NULL, 2, 3)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Derby, DuckDB, Firebird, Hana, MemSQL, MySQL, Postgres, SQLDataWarehouse, SQLite, Sybase, Trino, YugabyteDB

```
CASE
  WHEN 1 IS NOT NULL THEN 2
  ELSE 3
END
```

DB2, Databricks, Exasol, H2, HSQLDB, Informix, MariaDB, Oracle, Redshift, Snowflake, Teradata, Vertica

```
nvl2(1, 2, 3)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10. Numeric functions

In addition to the arithmetic expressions discussed previously, jOOQ also supports a variety of numeric functions. As discussed in the chapter about SQL dialects numeric functions (as any function type) are mostly emulated in your database, in case they are not natively supported.

3.8.10.1. ABS

The ABS() function produces the absolute value of a numeric value.

```
SELECT abs(-5), abs(0), abs(3);
```

```
create.select(abs(-5), abs(0), abs(3)).fetch();
```

The result being

```
+-----+-----+-----+
| abs | abs | abs |
+-----+-----+-----+
|   5 |   0 |   3 |
+-----+-----+-----+
```

Dialect support

This example using jOOQ:

```
abs(3)
```

Translates to the following dialect specific expressions:

All dialects

```
abs(3)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.2. ACOS

The ACOS() function calculates the arc cosine of a numeric value.

```
SELECT acos(0);
```

```
create.select(acos(0)).fetch();
```

The result being

```
+------------+
|       acos |
+------------+
| 1.57079633 |
+------------+
```

© 2009 - 2025 by Data Geekery™ GmbH.

Dialect support

This example using jOOQ:

```
acos(0)
```

Translates to the following dialect specific expressions:

Access

```
(atn((-0 / sqr(((-0 * 0) + 1)))) + (2 * atn(1)))
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Vertica, YugabyteDB

```
acos(0)
```

Snowflake, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.3. ASIN

The ASIN() function calculates the arc sine of a numeric value.

```
SELECT asin(1);
```

```
create.select(asin(1)).fetch();
```

The result being

```
+------------+
|       asin |
+------------+
| 1.57079633 |
+------------+
```

Dialect support

This example using jOOQ:

```
asin(1)
```

Translates to the following dialect specific expressions:

Access

```
atn((1 / sqr(((-1 * 1) + 1))))
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Vertica, YugabyteDB

```
asin(1)
```

Snowflake, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.4. ATAN

The ATAN() function calculates the arc tangent of a numeric value.

```
SELECT atan(1);
```

```
create.select(atan(1)).fetch();
```

The result being

```
+-------------+
|        atan |
+-------------+
| 0.785398163 |
+-------------+
```

Dialect support

This example using jOOQ:

```
atan(1)
```

Translates to the following dialect specific expressions:

Access

```
atn(1)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Vertica, YugabyteDB

```
atan(1)
```

Snowflake, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.5. ATAN2

The ATAN2() function calculates the ATAN2 of a numeric value.

```
SELECT atan2(1, 1);
```

```
create.select(atan2(1, 1)).fetch();
```

The result being

```
+---------------+
|         atan2 |
+---------------+
| 0.78539816339 |
+---------------+
```

Dialect support

This example using jOOQ:

```
atan2(1, 1)
```

Translates to the following dialect specific expressions:

ASE, SQLDataWarehouse, SQLServer

```
atn2(1, 1)
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLite, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
atan2(1, 1)
```

Access, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.6. CEIL

The CEIL() function rounds a numeric value to its nearest higher integer.

```
SELECT
  ceil(1.7),
  ceil(-1.7);
```

```
create.select(
  ceil(1.7),
  ceil(-1.7)).fetch();
```

The result being

```
+-------+-------+
| floor | floor |
+-------+-------+
|     2 |    -1 |
+-------+-------+
```

Dialect support

This example using jOOQ:

```
ceil(1.7)
```

Translates to the following dialect specific expressions:

Access

```
(CLNG(1.7E0) - (1.7E0 - clng(1.7E0) > 0))
```

ASE

```
ceiling(1.7E0)
```

Aurora MySQL

```
ceil(CAST(1.7E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
ceil(CAST(1.7E0 AS double precision))
```

BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
ceil(1.7E0)
```

DB2

```
ceil(CAST(1.7E0 AS double))
```

© 2009 - 2025 by Data Geekery™ GmbH.

H2

```
ceiling(CAST(1.7E0 AS double))
```

SQLDataWarehouse, SQLServer

```
ceiling(CAST(1.7E0 AS float))
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.10.7. COS

The COS() function calculates the cosine of a numeric value.

```
SELECT cos(3.14159265359);
```

```
create.select(cos(3.14159265359)).fetch();
```

The result being

```
+-----+
| cos |
+-----+
|  -1 |
+-----+
```

Dialect support

This example using jOOQ:

```
cos(3.14159265359)
```

Translates to the following dialect specific expressions:

ASE, Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
cos(3.14159265359E0)
```

Aurora MySQL

```
cos(CAST(3.14159265359E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
cos(CAST(3.14159265359E0 AS double precision))
```

DB2, H2

```
cos(CAST(3.14159265359E0 AS double))
```

SQLDataWarehouse, SQLServer

```
cos(CAST(3.14159265359E0 AS float))
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.10.8. COSH

The COSH() function calculates the hyperbolic cosine of a numeric value.

```
SELECT cosh(1);
```

```
create.select(cosh(1)).fetch();
```

The result being

```
+---------------+
|          cosh |
+---------------+
| 1.54308063482 |
+---------------+
```

Dialect support

This example using jOOQ:

```
cosh(1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, DuckDB, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Sybase, Vertica, YugabyteDB

```
((exp((1 * 2)) + 1) / (exp(1) * 2))
```

BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, H2, Hana, Informix, Oracle, SQLite, Snowflake, Teradata, Trino

```
cosh(1)
```

CockroachDB

```
((exp(CAST((1 * 2) AS decimal)) + 1) / (exp(CAST(1 AS decimal)) * 2))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.9. COT

The COT() function calculates the cotangent of a numeric value.

```
SELECT cot(1.5707963268);
```

```
create.select(cot(1.5707963268)).fetch();
```

The result being

```
+-----+
| cot |
+-----+
|   0 |
+-----+
```

Dialect support

This example using jOOQ:

```
cot(1.5707963268)
```

Translates to the following dialect specific expressions:

Access, BigQuery, ClickHouse, Informix, Oracle, SQLite, Teradata, Trino

```
(cos(1.5707963268E0) / sin(1.5707963268E0))
```

ASE, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Redshift, Snowflake, Sybase, Vertica

```
cot(1.5707963268E0)
```

Aurora MySQL

```
cot(CAST(1.5707963268E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
cot(CAST(1.5707963268E0 AS double precision))
```

DB2, H2

```
cot(CAST(1.5707963268E0 AS double))
```

SQLDataWarehouse, SQLServer

```
cot(CAST(1.5707963268E0 AS float))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.10. COTH

The COTH() function calculates the hyperbolic cotangent of a numeric value.

```
SELECT coth(1);
```

```
create.select(coth(1)).fetch();
```

The result being

```
+--------------+
|         coth |
+--------------+
| 1.3130352855 |
+--------------+
```

Dialect support

This example using jOOQ:

```
coth(1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
((exp((1 * 2)) + 1) / (exp((1 * 2)) - 1))
```

CockroachDB

```
((exp(CAST((1 * 2) AS decimal)) + 1) / (exp(CAST((1 * 2) AS decimal)) - 1))
```

Databricks

```
coth(1)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.11. DEG

The DEG() function calculates the degrees from a radian value (see also RAD).

```
SELECT deg(3.14159265359);
```

```
create.select(deg(3.14159265359)).fetch();
```

The result being

```
+-----+
| deg |
+-----+
| 180 |
+-----+
```

Dialect support

This example using jOOQ:

```
deg(3.14159265359)
```

Translates to the following dialect specific expressions:

Access

```
((3.14159265359E0 * 180) / 3.141592653589793)
```

ASE, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
degrees(3.14159265359E0)
```

Aurora MySQL

```
degrees(CAST(3.14159265359E0 AS decimal))
```

Aurora Postgres, CockroachDB, Postgres, YugabyteDB

```
degrees(CAST(3.14159265359E0 AS double precision))
```

BigQuery

```
((CAST(3.14159265359E0 AS decimal) * 180) / acos(-1))
```

DB2, H2

```
degrees(CAST(3.14159265359E0 AS double))
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird

```
((CAST(3.14159265359E0 AS numeric) * 180) / pi())
```

Hana

```
((CAST(3.14159265359E0 AS numeric) * 180) / acos(-1))
```

Oracle

```
((CAST(3.14159265359E0 AS number) * 180) / acos(-1))
```

SQLDataWarehouse, SQLServer

```
degrees(CAST(3.14159265359E0 AS float))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.12. E

The E() function produces the Euler constant e, which is around 2.71828182846

```
SELECT e();
```

```
create.select(e()).fetch();
```

The result being

```
+---------------+
| exp           |
+---------------+
| 2.71828182846 |
+---------------+
```

Dialect support

This example using jOOQ:

```
e()
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
exp(1)
```

ClickHouse, Databricks, Trino

```
e()
```

CockroachDB

```
exp(CAST(1 AS decimal))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.13. EXP

The EXP() function calculates e^x

```
SELECT exp(1);
```

```
create.select(exp(1)).fetch();
```

The result being

```
+---------------+
| exp           |
+---------------+
| 2.71828182846 |
+---------------+
```

Dialect support

This example using jOOQ:

```
exp(1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
exp(1)
```

CockroachDB

```
exp(CAST(1 AS decimal))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.14. FLOOR

The FLOOR() function rounds a numeric value to its nearest lower integer.

```
SELECT
  floor(1.7),
  floor(-1.7);
```

```
create.select(
  floor(1.7),
  floor(-1.7)).fetch();
```

The result being

```
+-------+-------+
| floor | floor |
+-------+-------+
|     1 |    -2 |
+-------+-------+
```

Dialect support

This example using jOOQ:

```
floor(1.7)
```

Translates to the following dialect specific expressions:

Access

```
(cdec(1.7E0) - (1.7E0 < cdec(1.7E0)))
```

ASE, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
floor(1.7E0)
```

Aurora MySQL

```
floor(CAST(1.7E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
floor(CAST(1.7E0 AS double precision))
```

DB2, H2

```
floor(CAST(1.7E0 AS double))
```

SQLDataWarehouse, SQLServer

```
floor(CAST(1.7E0 AS float))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.15. GREATEST

The GREATEST() function produces the greatest value among all the arguments.

```
SELECT greatest(2, 3);
```

```
create.select(greatest(2, 3)).fetch();
```

The result being

```
+----------+
| greatest |
+----------+
|        3 |
+----------+
```

Dialect support

This example using jOOQ:

```
greatest(2, 3)
```

Translates to the following dialect specific expressions:

Access

```
SWITCH(2 > 3, 2, TRUE, 3)
```

ASE, Derby, Informix, SQLDataWarehouse, Sybase

```
CASE
  WHEN 2 > 3 THEN 2
  ELSE 3
END
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
greatest(2, 3)
```

Firebird

```
maxvalue(2, 3)
```

SQLite

```
max(2, 3)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.16. LEAST

The LEAST() function produces the least value among all the arguments.

```
SELECT least(2, 3);
```

```
create.select(least(2, 3)).fetch();
```

The result being

```
+-------+
| least |
+-------+
|     2 |
+-------+
```

Dialect support

This example using jOOQ:

```
least(2, 3)
```

Translates to the following dialect specific expressions:

Access

```
SWITCH(2 < 3, 2, TRUE, 3)
```

ASE, Derby, Informix, SQLDataWarehouse, Sybase

```
CASE
  WHEN 2 < 3 THEN 2
  ELSE 3
END
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
least(2, 3)
```

Firebird

```
minvalue(2, 3)
```

SQLite

```
min(2, 3)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.17. LN

The LN() function calculates the natural logarithm of a numeric value.

```
SELECT ln(1);
```

```
create.select(ln(1)).fetch();
```

The result being

```
+----+
| ln |
+----+
|  0 |
+----+
```

Dialect support

This example using jOOQ:

```
ln(1)
```

Translates to the following dialect specific expressions:

ASE, Access, SQLDataWarehouse, SQLServer

```
log(1)
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
ln(1)
```

CockroachDB

```
ln(CAST(1 AS decimal))
```

Informix

```
logn(1)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.18. LOG

The LOG() function calculates the logarithm of a numeric value, given a base.

```
SELECT log(8, 2);
```

```
create.select(log(8, 2)).fetch();
```

The result being

```
+-----+
| log |
+-----+
|   3 |
+-----+
```

Dialect support

This example using jOOQ:

```
log(8, 2)
```

Translates to the following dialect specific expressions:

ASE, Access

```
(log(8) / log(2))
```

Aurora MySQL, Aurora Postgres, Databricks, Exasol, Firebird, H2, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, Snowflake, Trino, Vertica, YugabyteDB

```
log(2, 8)
```

BigQuery, SQLDataWarehouse, SQLServer

```
log(8, 2)
```

ClickHouse, DB2, Derby, DuckDB, HSQLDB, SQLite, Sybase, Teradata

```
(ln(8) / ln(2))
```

CockroachDB

```
(ln(CAST(8 AS decimal)) / ln(CAST(2 AS decimal)))
```

Informix

```
(logn(8) / logn(2))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.19. NEG

The NEG() function produces the negation of its argument.

```
SELECT neg(2);
```

```
create.select(neg(2)).fetch();
```

The result being

```
+-----+
| neg |
+-----+
|  -2 |
+-----+
```

Dialect support

This example using jOOQ:

```
neg(val(2))
```

Translates to the following dialect specific expressions:

All dialects

```
-2
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.20. PI

The PI() function produces the pi constant #, which is around 3.14159265359

```
SELECT pi();
```

```
create.select(pi()).fetch();
```

The result being

```
+---------------+
| pi            |
+---------------+
| 3.14159265359 |
+---------------+
```

Dialect support

This example using jOOQ:

```
pi()
```

Translates to the following dialect specific expressions:

Access

```
3.141592653589793
```

ASE, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Trino, Vertica, YugabyteDB

```
pi()
```

BigQuery, DB2, Hana, Informix, Oracle, Teradata

```
acos(-1)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.21. POWER

The POWER() function calculates the power of two numbers.

```
SELECT power(2, 3);
```

```
create.select(power(2, 3)).fetch();
```

The result being

```
+-------+
| power |
+-------+
|     8 |
+-------+
```

Dialect support

This example using jOOQ:

```
power(2, 3)
```

Translates to the following dialect specific expressions:

Access

```
(2 ^ 3)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
power(2, 3)
```

ClickHouse

```
pow(2, 3)
```

CockroachDB

```
power(
  CAST(2 AS decimal),
  CAST(3 AS decimal)
)
```

Derby

```
exp((ln(2) * 3))
```

Snowflake

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.22. RAD

The RAD() function calculates the radian value from degrees (see also DEG).

```
SELECT rad(180);
```

```
create.select(rad(180)).fetch();
```

The result being

```
+---------------+
|           rad |
+---------------+
| 3.14159265359 |
+---------------+
```

Dialect support

This example using jOOQ:

```
rad(180)
```

Translates to the following dialect specific expressions:

Access

```
((cdec(180) * 3.141592653589793) / 180)
```

ASE, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
radians(180)
```

BigQuery

```
((CAST(180 AS decimal) * acos(-1)) / 180)
```

Firebird

```
((CAST(180 AS numeric) * pi()) / 180)
```

Hana

```
((CAST(180 AS numeric) * acos(-1)) / 180)
```

Oracle

```
((CAST(180 AS number) * acos(-1)) / 180)
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.10.23. RAND

The RAND() function produces a random number.

```
SELECT rand();
```

```
create.select(rand()).fetch();
```

The result being

```
+------+
| rand |
+------+
|    4 | <-- chosen by fair dice roll
+------+
```

Dialect support

This example using jOOQ:

```
rand()
```

Translates to the following dialect specific expressions:

Access

```
rnd
```

ASE, Aurora MySQL, BigQuery, ClickHouse, DB2, Databricks, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, Sybase, Trino

```
rand()
```

Aurora Postgres, CockroachDB, Derby, DuckDB, Exasol, Postgres, Redshift, SQLite, Vertica, YugabyteDB

```
random()
```

Oracle

```
DBMS_RANDOM.RANDOM
```

Teradata

```
(CAST((random(-2147483648, 2147483647) + 2147483648) AS NUMERIC(38, 19)) / 4294967295)
```

Informix, Snowflake

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.24. ROUND

The ROUND() function rounds a numeric value to its nearest integer, or optionally, to the nearest decimal precision.

```
SELECT
  round(1.7),
  round(-1.7);
```

```
create.select(
  round(1.7),
  round(-1.7)).fetch();
```

The result being

```
+-------+-------+
| round | round |
+-------+-------+
|     2 |    -2 |
+-------+-------+
```

Dialect support

This example using jOOQ:

```
round(1.7)
```

Translates to the following dialect specific expressions:

Access, BigQuery, ClickHouse, Databricks, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Snowflake, Teradata, Trino, Vertica

```
round(1.7E0)
```

ASE, Sybase

```
round(1.7E0, 0)
```

Aurora MySQL

```
round(CAST(1.7E0 AS decimal))
```

Aurora Postgres, Firebird, Postgres, YugabyteDB

```
round(CAST(1.7E0 AS double precision))
```

CockroachDB

```
round(CAST(CAST(1.7E0 AS double precision) AS numeric))
```

DB2, H2

```
round(CAST(1.7E0 AS double))
```

Derby

```
CASE
  WHEN (1.7E0 - floor(1.7E0)) < 5E-1 THEN floor(1.7E0)
  ELSE ceil(1.7E0)
END
```

SQLDataWarehouse, SQLServer

```
round(CAST(1.7E0 AS float), 0)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.25. SIGN

The SIGN() function produces the sign of a numeric value, being any value of -1, 0, 1

```
SELECT sign(-5), sign(0), sign(3);
```

```
create.select(sign(-5), sign(0), sign(3)).fetch();
```

The result being

```
+------+------+------+
| sign | sign | sign |
+------+------+------+
|   -1 |    0 |    1 |
+------+------+------+
```

Dialect support

This example using jOOQ:

```
sign(3)
```

Translates to the following dialect specific expressions:

Access

```
sgn(3)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
sign(3)
```

SQLite

```
CASE
  WHEN 3 > 0 THEN 1
  WHEN 3 < 0 THEN -1
  WHEN 3 = 0 THEN 0
END
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.10.26. SIN

The SIN() function calculates the sine of a numeric value.

```
SELECT sin(3.14159265359);
```

```
create.select(sin(3.14159265359)).fetch();
```

The result being

```
+-----+
| sin |
+-----+
|   0 |
+-----+
```

Dialect support

This example using jOOQ:

```
sin(3.14159265359)
```

Translates to the following dialect specific expressions:

ASE, Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
sin(3.14159265359E0)
```

Aurora MySQL

```
sin(CAST(3.14159265359E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
sin(CAST(3.14159265359E0 AS double precision))
```

DB2, H2

```
sin(CAST(3.14159265359E0 AS double))
```

SQLDataWarehouse, SQLServer

```
sin(CAST(3.14159265359E0 AS float))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.27. SINH

The SINH() function calculates the hyperbolic sine of a numeric value.

```
SELECT sinh(1);
```

```
create.select(sinh(1)).fetch();
```

The result being

```
+---------------+
|          sinh |
+---------------+
| 1.17520119364 |
+---------------+
```

Dialect support

This example using jOOQ:

```
sinh(1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, DuckDB, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Sybase, Trino, Vertica, YugabyteDB

```
((exp((1 * 2)) - 1) / (exp(1) * 2))
```

BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, H2, Hana, Informix, Oracle, SQLite, Snowflake, Teradata

```
sinh(1)
```

CockroachDB

```
((exp(CAST((1 * 2) AS decimal)) - 1) / (exp(CAST(1 AS decimal)) * 2))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.28. SQRT

The SQRT() function calculates the square root of a numeric value.

```
SELECT sqrt(4);
```

```
create.select(sqrt(4)).fetch();
```

The result being

```
+------+
| sqrt |
+------+
|    2 |
+------+
```

Dialect support

This example using jOOQ:

```
sqrt(4)
```

Translates to the following dialect specific expressions:

Access

```
sqr(4)
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
sqrt(4)
```

CockroachDB

```
sqrt(CAST(4 AS decimal))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.29. TAN

The TAN() function calculates the tangent of a numeric value.

```
SELECT tan(3.14159265359);
```

```
create.select(tan(3.14159265359)).fetch();
```

The result being

```
+-----+
| tan |
+-----+
|   0 |
+-----+
```

Dialect support

This example using jOOQ:

```
tan(3.14159265359)
```

Translates to the following dialect specific expressions:

ASE, Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite, Sybase, Teradata, Trino, Vertica

```
tan(3.14159265359E0)
```

Aurora MySQL

```
tan(CAST(3.14159265359E0 AS decimal))
```

Aurora Postgres, CockroachDB, Firebird, Postgres, YugabyteDB

```
tan(CAST(3.14159265359E0 AS double precision))
```

DB2, H2

```
tan(CAST(3.14159265359E0 AS double))
```

SQLDataWarehouse, SQLServer

```
tan(CAST(3.14159265359E0 AS float))
```

Snowflake

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.30. TANH

The TANH() function calculates the hyperbolic tangent of a numeric value.

```
SELECT tanh(1);
```

```
create.select(tanh(1)).fetch();
```

The result being

```
+---------------+
|          tanh |
+---------------+
| 0.76159415595 |
+---------------+
```

Dialect support

This example using jOOQ:

```
tanh(1)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, DuckDB, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Sybase, Vertica, YugabyteDB

```
((exp((1 * 2)) - 1) / (exp((1 * 2)) + 1))
```

BigQuery, ClickHouse, DB2, Databricks, Derby, Exasol, Firebird, H2, Hana, Informix, Oracle, SQLite, Snowflake, Teradata, Trino

```
tanh(1)
```

CockroachDB

```
((exp(CAST((1 * 2) AS decimal)) - 1) / (exp(CAST((1 * 2) AS decimal)) + 1))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.31. TRUNC

The TRUNC() function rounds a numeric value to its nearest integer (or optionally, to a specific decimal precision) that is closer to zero.

```
SELECT
  trunc(1.7),
  trunc(-1.7);
```

```
create.select(
  trunc(1.7),
  trunc(-1.7)).fetch();
```

The result being

```
+-------+-------+
| trunc | trunc |
+-------+-------+
|     1 |    -1 |
+-------+-------+
```

Dialect support

This example using jOOQ:

```
trunc(1.7)
```

Translates to the following dialect specific expressions:

ASE

```
CASE
  WHEN sign(1.7E0) >= 0 THEN (floor((1.7E0 * 1)) / 1)
  ELSE (ceiling((1.7E0 * 1)) / 1)
END
```

Aurora MySQL

```
truncate(CAST(1.7E0 AS decimal), 0)
```

Aurora Postgres, Postgres

```
CAST(trunc(
  CAST(1.7E0 AS numeric),
  0
) AS double precision)
```

DB2

```
trunc(CAST(1.7E0 AS double), 0)
```

Derby

```
CASE
  WHEN sign(1.7E0) >= 0 THEN (floor((1.7E0 * 1)) / 1)
  ELSE (ceil((1.7E0 * 1)) / 1)
END
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird

```
trunc(CAST(1.7E0 AS double precision), 0)
```

H2

```
truncate(CAST(1.7E0 AS double), 0)
```

Hana

```
round(1.7E0, 0, round_down)
```

HSQLDB, Informix, Oracle, Teradata, Vertica

```
trunc(1.7E0, 0)
```

MariaDB, MemSQL, MySQL

```
truncate(1.7E0, 0)
```

SQLDataWarehouse, SQLServer

```
round(CAST(1.7E0 AS float), 0, 1)
```

Sybase

```
truncnum(1.7E0, 0)
```

Access, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, Redshift, SQLite, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.10.32. WIDTH_BUCKET

The WIDTH_BUCKET() function divides a numeric range into equally sized buckets and calculates which bucket number a value is in.

```
SELECT
  width_bucket(0 , 0, 100, 10),
  width_bucket(15, 0, 100, 10),
  width_bucket(99, 0, 100, 10);
```

```
create.select(
  widthBucket(val(0) , 0, 100, 10),
  widthBucket(val(15), 0, 100, 10),
  widthBucket(val(99), 0, 100, 10)).fetch();
```

The result being

```
+--------------+--------------+--------------+
| width_bucket | width_bucket | width_bucket |
+--------------+--------------+--------------+
|            1 |            2 |           10 |
+--------------+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
widthBucket(val(15), 0, 100, 10)
```

Translates to the following dialect specific expressions:

Access

```
SWITCH(15 < 0, 0, 15 >= 100, (10 + 1), TRUE, ((cdec((((15 - 0) * 10) / (100 - 0))) - ((((15 - 0) * 10) / (100 - 0)) < cdec((((15 - 0)
 * 10) / (100 - 0))))) + 1))
```

ASE, Aurora MySQL, BigQuery, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica

```
CASE
  WHEN 15 < 0 THEN 0
  WHEN 15 >= 100 THEN (10 + 1)
  ELSE (floor((((15 - 0) * 10) / (100 - 0))) + 1)
END
```

Aurora Postgres, ClickHouse, CockroachDB, Databricks, Oracle, Postgres, Snowflake, Teradata, Trino, YugabyteDB

```
width_bucket(15, 0, 100, 10)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11. Bitwise functions

Most databases only support a few bitwise operations, while others ship with the full set of operators. jOOQ's API includes most bitwise operations as listed below. In order to avoid ambiguities with conditional operators, most bitwise functions are prefixed with "bit"

3.8.11.1. BIT_AND

The BIT_AND() function produces the bitwise AND operation.

```
SELECT bit_and(5, 4);
```

```
create.select(bitAnd(5, 4)).fetch();
```

The result being

```
+---------+
| bit_and |
+---------+
|       4 |
+---------+
```

Dialect support

This example using jOOQ:

```
bitAnd(5, 4)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica, YugabyteDB

```
(5 & 4)
```

ClickHouse

```
bitAnd(5, 4)
```

DB2, H2, HSQLDB, Hana, Informix, Oracle, Snowflake, Teradata

```
bitand(5, 4)
```

Exasol

```
bit_and(5, 4)
```

Firebird

```
bin_and(5, 4)
```

Trino

```
bitwise_and(5, 4)
```

Access, Derby

```
/* UNSUPPORTED */
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.11.2. BIT_COUNT

The BIT_COUNT() function counts the number of bits in a value.

```
SELECT bit_count(5);
```

```
create.select(bitCount(5)).fetch();
```

The result being

```
+-----------+
| bit_count |
+-----------+
|         2 |
+-----------+
```

Dialect support

This example using jOOQ:

```
bitCount((byte) 5)
```

Translates to the following dialect specific expressions:

Aurora MySQL, Databricks, DuckDB, MariaDB, MemSQL, MySQL, SQLServer

```
bit_count(5)
```

Aurora Postgres, Postgres, Redshift, SQLite, Vertica, YugabyteDB

```
CAST(((5 & 1) + ((5 & 2) >> 1) + ((5 & 4) >> 2) + ((5 & 8) >> 3) + ((5 & 16) >> 4) + ((5 & 32) >> 5) + ((5 & 64) >> 6) + ((5 & -128)
 >> 7)) AS int)
```

BigQuery

```
CAST(((5 & 1) + ((5 & 2) >> 1) + ((5 & 4) >> 2) + ((5 & 8) >> 3) + ((5 & 16) >> 4) + ((5 & 32) >> 5) + ((5 & 64) >> 6) + ((5 & -128)
 >> 7)) AS int64)
```

ClickHouse

```
bitCount(5)
```

CockroachDB

```
CAST(((5 & 1) + ((5 & 2) >> 1) + ((5 & 4) >> 2) + ((5 & 8) >> 3) + ((5 & 16) >> 4) + ((5 & 32) >> 5) + ((5 & 64) >> 6) + ((5 & -128)
 >> 7)) AS int4)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird

```
CAST((bin_and(5, 1) + bin_shr(
  bin_and(5, 2),
  1
) + bin_shr(
  bin_and(5, 4),
  2
) + bin_shr(
  bin_and(5, 8),
  3
) + bin_shr(
  bin_and(5, 16),
  4
) + bin_shr(
  bin_and(5, 32),
  5
) + bin_shr(
  bin_and(5, 64),
  6
) + bin_shr(
  bin_and(5, -128),
  7
)) AS integer)
```

H2, HSQLDB

```
CAST((bitand(5, 1) + (bitand(5, 2) / 2) + (bitand(5, 4) / 4) + (bitand(5, 8) / 8) + (bitand(5, 16) / 16) + (bitand(5, 32) / 32) +
 (bitand(5, 64) / 64) + (bitand(5, -128) / -128)) AS int)
```

Hana

```
bitcount(5)
```

Informix

```
CAST((bitand(5, 1) + (bitand(5, 2) / 2) + (bitand(5, 4) / 4) + (bitand(5, 8) / 8) + (bitand(5, 16) / 16) + (bitand(5, 32) / 32) +
 (bitand(5, 64) / 64) + (bitand(5, -128) / -128)) AS integer)
```

Oracle

```
CAST((bitand(5, 1) + (bitand(5, 2) / 2) + (bitand(5, 4) / 4) + (bitand(5, 8) / 8) + (bitand(5, 16) / 16) + (bitand(5, 32) / 32) +
 (bitand(5, 64) / 64) + (bitand(5, -128) / -128)) AS number(10))
```

© 2009 - 2025 by Data Geekery™ GmbH.

Snowflake

```
CAST((bitand(5, 1) + bitshiftright(
  bitand(5, 2),
  1
) + bitshiftright(
  bitand(5, 4),
  2
) + bitshiftright(
  bitand(5, 8),
  3
) + bitshiftright(
  bitand(5, 16),
  4
) + bitshiftright(
  bitand(5, 32),
  5
) + bitshiftright(
  bitand(5, 64),
  6
) + bitshiftright(
  bitand(5, -128),
  7
)) AS number(10))
```

SQLDataWarehouse, Sybase

```
CAST(((5 & 1) + ((5 & 2) / 2) + ((5 & 4) / 4) + ((5 & 8) / 8) + ((5 & 16) / 16) + ((5 & 32) / 32) + ((5 & 64) / 64) + ((5 & -128) /
 -128)) AS int)
```

Teradata

```
countset(5, 1)
```

Trino

```
CAST((bitwise_and(5, 1) + bitwise_right_shift(
  bitwise_and(5, 2),
  1
) + bitwise_right_shift(
  bitwise_and(5, 4),
  2
) + bitwise_right_shift(
  bitwise_and(5, 8),
  3
) + bitwise_right_shift(
  bitwise_and(5, 16),
  4
) + bitwise_right_shift(
  bitwise_and(5, 32),
  5
) + bitwise_right_shift(
  bitwise_and(5, 64),
  6
) + bitwise_right_shift(
  bitwise_and(5, -128),
  7
)) AS int)
```

ASE, Access, DB2, Derby, Exasol

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.3. BIT_NAND

The BIT_NAND() function produces the bitwise NAND operation.

```
SELECT bit_nand(5, 4);
```

```
create.select(bitNand(5, 4)).fetch();
```

The result being

```
+----------+
| bit_nand |
+----------+
|       -5 |
+----------+
```

Dialect support

This example using jOOQ:

```
bitNand(5, 4)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica, YugabyteDB

```
~((5 & 4))
```

ClickHouse

```
bitNot(bitAnd(5, 4))
```

DB2, H2, Hana, Informix, Snowflake, Teradata

```
bitnot(bitand(5, 4))
```

Exasol

```
bit_not(bit_and(5, 4))
```

Firebird

```
bin_not(bin_and(5, 4))
```

HSQLDB, Oracle

```
((0 - bitand(5, 4)) - 1)
```

Trino

```
bitwise_not(bitwise_and(5, 4))
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.4. BIT_NOR

The BIT_NOR() function produces the bitwise NOR operation.

```
SELECT bit_nor(5, 2);
```

```
create.select(bitNor(5, 2)).fetch();
```

The result being

```
+---------+
| bit_nor |
+---------+
|      -8 |
+---------+
```

Dialect support

This example using jOOQ:

```
bitNor(5, 2)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica, YugabyteDB

```
~((5 | 2))
```

ClickHouse

```
bitNot(bitOr(5, 2))
```

DB2, H2, Hana, Informix, Snowflake, Teradata

```
bitnot(bitor(5, 2))
```

Exasol

```
bit_not(bit_or(5, 2))
```

Firebird

```
bin_not(bin_or(5, 2))
```

HSQLDB

```
((0 - bitor(5, 2)) - 1)
```

Oracle

```
((0 - ((5 + 2) - bitand(5, 2))) - 1)
```

Trino

```
bitwise_not(bitwise_or(5, 2))
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.5. BIT_NOT

The BIT_NOT() function inverts the bits in a number, producing the 2's complement of a signed number.

```
SELECT bit_not(5);
```

```
create.select(bitNot(5)).fetch();
```

The result being

```
+---------+
| bit_not |
+---------+
|      -6 |
+---------+
```

Dialect support

This example using jOOQ:

```
bitNot(5)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica, YugabyteDB

```
~5
```

ClickHouse

```
bitNot(5)
```

DB2, H2, Hana, Informix, Snowflake, Teradata

```
bitnot(5)
```

Exasol

```
bit_not(5)
```

Firebird

```
bin_not(5)
```

HSQLDB, Oracle

```
((0 - 5) - 1)
```

Trino

```
bitwise_not(5)
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.6. BIT_OR

The BIT_OR() function produces the bitwise OR operation.

```
SELECT bit_or(5, 2);
```

```
create.select(bitOr(5, 2)).fetch();
```

The result being

```
+--------+
| bit_or |
+--------+
|      7 |
+--------+
```

Dialect support

This example using jOOQ:

```
bitOr(5, 2)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Vertica, YugabyteDB

```
(5 | 2)
```

ClickHouse

```
bitOr(5, 2)
```

DB2, H2, HSQLDB, Hana, Informix, Snowflake, Teradata

```
bitor(5, 2)
```

Exasol

```
bit_or(5, 2)
```

Firebird

```
bin_or(5, 2)
```

Oracle

```
((5 + 2) - bitand(5, 2))
```

Trino

```
bitwise_or(5, 2)
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.7. BIT_XNOR

The BIT_XNOR() function produces the bitwise XNOR (exclusive NOR) operation.

```
SELECT bit_xnor(5, 3);
```

```
create.select(bitXNor(5, 3)).fetch();
```

The result being

```
+----------+
| bit_xnor |
+----------+
|       -7 |
+----------+
```

Dialect support

This example using jOOQ:

```
bitXNor(5, 3)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, BigQuery, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, Sybase

```
~((5 ^ 3))
```

Aurora Postgres, CockroachDB, Postgres, Redshift, Vertica, YugabyteDB

```
~((5 # 3))
```

ClickHouse

```
bitNot(bitXor(5, 3))
```

Databricks

```
~bit_xor(5, 3)
```

DB2, Hana, Informix, Snowflake, Teradata

```
bitnot(bitxor(5, 3))
```

DuckDB

```
~xor(5, 3)
```

Exasol

```
bit_not(bit_xor(5, 3))
```

Firebird

```
bin_not(bin_xor(5, 3))
```

H2

```
bitxnor(5, 3)
```

HSQLDB

```
((0 - bitxor(5, 3)) - 1)
```

Oracle

```
((0 - bitand(
  ((0 - bitand(5, 3)) - 1),
  ((5 + 3) - bitand(5, 3))
)) - 1)
```

SQLite

```
~((~((5 & 3)) & (5 | 3)))
```

Trino

```
bitwise_not(bitwise_xor(5, 3))
```

Access, Derby

```
/* UNSUPPORTED */
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.11.8. BIT_XOR

The BIT_XOR() function produces the bitwise XOR (exclusive OR) operation.

```
SELECT bit_xor(5, 3);
```

```
create.select(bitXor(5, 3)).fetch();
```

The result being

```
+---------+
| bit_xor |
+---------+
|       6 |
+---------+
```

Dialect support

This example using jOOQ:

```
bitXor(5, 3)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, BigQuery, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLServer, Sybase

```
(5 ^ 3)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, Vertica, YugabyteDB

```
(5 # 3)
```

ClickHouse

```
bitXor(5, 3)
```

Databricks, Exasol

```
bit_xor(5, 3)
```

DB2, H2, HSQLDB, Hana, Informix, Snowflake, Teradata

```
bitxor(5, 3)
```

DuckDB

```
xor(5, 3)
```

Firebird

```
bin_xor(5, 3)
```

Oracle

```
bitand(
  ((0 - bitand(5, 3)) - 1),
  ((5 + 3) - bitand(5, 3))
)
```

SQLite

```
(~((5 & 3)) & (5 | 3))
```

Trino

```
bitwise_xor(5, 3)
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.9. SHL

The SHL() function produces the bitwise shift left operation.

```
SELECT shl(1, 4);
```

```
create.select(shl(1, 4)).fetch();
```

The result being

```
+-----+
| shl |
+-----+
|  16 |
+-----+
```

Dialect support

This example using jOOQ:

```
shl(1, 4)
```

Translates to the following dialect specific expressions:

ASE, HSQLDB, SQLDataWarehouse, Sybase

```
(1 * CAST(power(2, 4) AS int))
```

Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLServer, SQLite, Vertica, YugabyteDB

```
(1 << 4)
```

ClickHouse

```
bitShiftLeft(1, 4)
```

Databricks, Teradata

```
shiftleft(1, 4)
```

DB2, Informix

```
(1 * CAST(power(2, 4) AS integer))
```

Exasol

```
bit_lshift(1, 4)
```

Firebird

```
bin_shl(1, 4)
```

H2

```
lshift(1, 4)
```

Oracle

```
(1 * CAST(power(2, 4) AS number(10)))
```

Snowflake

```
bitshiftleft(1, 4)
```

Trino

```
bitwise_left_shift(1, 4)
```

Access, Derby, Hana

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.11.10. SHR

The SR() function produces the bitwise shift right operation.

```
SELECT shr(16, 4);
```

```
create.select(shr(16, 4)).fetch();
```

The result being

```
+-----+
| shr |
+-----+
|   1 |
+-----+
```

Dialect support

This example using jOOQ:

```
shr(16, 4)
```

Translates to the following dialect specific expressions:

ASE, HSQLDB, SQLDataWarehouse, Sybase

```
(16 / CAST(power(2, 4) AS int))
```

Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLServer, SQLite, Vertica, YugabyteDB

```
(16 >> 4)
```

ClickHouse

```
bitShiftRight(16, 4)
```

Databricks, Teradata

```
shiftright(16, 4)
```

DB2, Informix

```
(16 / CAST(power(2, 4) AS integer))
```

Exasol

```
bit_rshift(16, 4)
```

Firebird

```
bin_shr(16, 4)
```

H2

```
rshift(16, 4)
```

Oracle

```
(16 / CAST(power(2, 4) AS number(10)))
```

Snowflake

```
bitshiftright(16, 4)
```

Trino

```
bitwise_right_shift(16, 4)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Access, Derby, Hana

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12. String functions

String formatting can be done efficiently in the database before returning results to your Java application. As discussed in the chapter about SQL dialects string functions (as any function type) are mostly emulated in your database, in case they are not natively supported.

3.8.12.1. ASCII

The ASCII() function calculates the ASCII code of a single character.

```
SELECT ascii('A');
```

```
create.select(ascii("A")).fetch();
```

The result being

```
+-------+
| ascii |
+-------+
|  65   |
+-------+
```

Dialect support

This example using jOOQ:

```
ascii("A")
```

Translates to the following dialect specific expressions:

Access

```
asc('A')
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL,

Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
ascii('A')
```

Firebird

```
ascii_val('A')
```

Derby, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.2. BIT_LENGTH

The BIT_LENGTH() function calculates the length of a given string in bits.

```
SELECT bit_length('hello');
```

```
create.select(bitLength("hello")).fetch();
```

The result being

```
+------------+
| bit_length |
+------------+
|         40 |
+------------+
```

Dialect support

This example using jOOQ:

```
bitLength("hello")
```

Translates to the following dialect specific expressions:

Access, SQLDataWarehouse, SQLServer

```
(8 * len('hello'))
```

ASE

```
(8 * datalength('hello'))
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MySQL, Postgres, Redshift, Snowflake, Vertica, YugabyteDB

```
bit_length('hello')
```

BigQuery

```
(8 * byte_length('hello'))
```

ClickHouse, DB2, Derby, Hana, MemSQL, SQLite, Sybase, Teradata, Trino

```
(8 * length('hello'))
```

Oracle

```
(8 * lengthb('hello'))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.3. CONCAT (|| operator)

The CONCAT() function concatenates several strings

```
SELECT concat('hello', ' ', 'world');
```

```
create.select(concat("hello", " ", "world")).fetch();
```

The result being

```
+-------------+
| concat      |
+-------------+
| hello world |
+-------------+
```

Dialect support

This example using jOOQ:

```
concat("hello", " ", "world")
```

Translates to the following dialect specific expressions:

Access

```
('hello' & ' ' & 'world')
```

ASE, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, Oracle, Postgres, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
('hello' || ' ' || 'world')
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
concat('hello', ' ', 'world')
```

SQLDataWarehouse, SQLServer

```
('hello' + ' ' + 'world')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.4. LEFT

The LEFT() function calculates the substring of a given string starting from the left end. See also SUBSTRING, RIGHT

```
SELECT left('hello world', 5);
```

```
create.select(left("hello world", 5)).fetch();
```

The result being

```
+-------+
| left  |
+-------+
| hello |
+-------+
```

Dialect support

This example using jOOQ:

```
left("hello world", 5)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
left('hello world', 5)
```

Derby, Oracle, SQLite

```
substr('hello world', 1, 5)
```

Trino

```
substring('hello world', 1, 5)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.5. LENGTH

The LENGTH() function calculates the length of a given string.

```
SELECT length('hello');
```

```
create.select(length("hello")).fetch();
```

The result being

```
+--------+
| length |
+--------+
|      5 |
+--------+
```

Dialect support

This example using jOOQ:

```
length("hello")
```

Translates to the following dialect specific expressions:

Access, SQLDataWarehouse, SQLServer

```
len('hello')
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, Vertica, YugabyteDB

```
char_length('hello')
```

ClickHouse, DB2, Derby, DuckDB, Hana, Oracle, SQLite, Snowflake, Sybase, Teradata, Trino

```
length('hello')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.6. LOWER

The LOWER() function transforms a string into lower case.

```
SELECT lower('HELLO');
```

```
create.select(lower("HELLO")).fetch();
```

The result being

```
+-------+
| lower |
+-------+
| hello |
+-------+
```

Dialect support

This example using jOOQ:

```
lower("HELLO")
```

Translates to the following dialect specific expressions:

Access

```
lcase('HELLO')
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
lower('HELLO')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.7. LPAD

The LPAD() pads a string at the left end. See also RPAD.

```
SELECT lpad('hello', 10, '.');
```

```
create.select(lpad(val("hello"), 10, '.')).fetch();
```

The result being

```
+------------+
| lpad       |
+------------+
| .....hello |
+------------+
```

Dialect support

This example using jOOQ:

```
lpad(val("hello"), 10, '.')
```

Translates to the following dialect specific expressions:

Access

```
(replace(space(10 - len('hello')), ' ', '.') & 'hello')
```

ASE

```
(replicate(
  '.',
  (10 - char_length('hello'))
) || 'hello')
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Teradata, Vertica

```
lpad('hello', 10, '.')
```

SQLDataWarehouse, SQLServer

```
(replicate(
  '.',
  (10 - len('hello'))
) + 'hello')
```

SQLite

```
substr(replace(hex(zeroblob(10)), '00', '.'), 1, 10 - length('hello')) || 'hello'
```

Sybase

```
(repeat(
  '.',
  (10 - length('hello'))
) || 'hello')
```

BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.8. LTRIM

The LTRIM() function trims a string from the left end, stripping it of whitespace. See also RTRIM and TRIM.

```
SELECT ltrim('  hello  ');
```

```
create.select(ltrim("  hello  ")).fetch();
```

The result being

```
+---------+
| ltrim   |
+---------+
| hello   |
+---------+
```

Dialect support

This example using jOOQ:

```
ltrim("  hello  ")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
ltrim('  hello  ')
```

Firebird

```
trim(LEADING FROM '  hello  ')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.9. MD5

The MD5() function calculates the MD5 hash of a given string.

```
SELECT md5('hello');
```

```
create.select(md5("hello")).fetch();
```

The result being

```
+--------------------------------+
| md5                            |
+--------------------------------+
| 5d41402abc4b2a76b9719d911017c592 |
+--------------------------------+
```

Dialect support

This example using jOOQ:

```
md5("hello")
```

Translates to the following dialect specific expressions:

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, MariaDB, MemSQL, MySQL, Postgres, Snowflake, Vertica, YugabyteDB

```
md5('hello')
```

Exasol

```
hash_md5('hello')
```

Oracle

```
lower(standard_hash('hello', 'MD5'))
```

SQLDataWarehouse

```
lower(convert(VARCHAR(32), hashbytes('MD5', CAST('hello' AS varchar(8000))), 2))
```

SQLServer

```
lower(convert(VARCHAR(32), hashbytes('MD5', CAST('hello' AS varchar(max))), 2))
```

ASE, Access, DB2, Derby, Firebird, H2, HSQLDB, Hana, Informix, Redshift, SQLite, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.10. MID

The MID() function is an alias for the substring function

3.8.12.11. OCTET_LENGTH

The OCTET_LENGTH() function calculates the length of a given string in bytes.

```
SELECT octet_length('hello');
```

```
create.select(octetLength("hello")).fetch();
```

The result being

```
+--------------+
| octet_length |
+--------------+
|            5 |
+--------------+
```

Dialect support

This example using jOOQ:

```
octetLength("hello")
```

Translates to the following dialect specific expressions:

Access, SQLDataWarehouse, SQLServer

```
len('hello')
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, Databricks, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, Snowflake, Vertica, YugabyteDB

```
octet_length('hello')
```

ClickHouse, DB2, Derby, DuckDB, Hana, SQLite, Sybase, Teradata, Trino

```
length('hello')
```

Oracle

```
lengthb('hello')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.12. POSITION

The POSITION() function finds the first position of a string within another string, starting with 1.

```
SELECT
  position('hello', 'e'),
  position('hello', 'l', 4);
```

```
create.select(
  position("hello", "e"),
  position("hello", "e", 4)).fetch();
```

The result being

```
+----------+----------+
| position | position |
+----------+----------+
|        2 |        4 |
+----------+----------+
```

Dialect support

This example using jOOQ:

```
position("hello", "e")
```

Translates to the following dialect specific expressions:

ASE, SQLDataWarehouse, SQLServer

```
charindex('e', 'hello')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, MariaDB, MemSQL, MySQL, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
position('e' IN 'hello')
```

BigQuery, Informix, Oracle, SQLite

```
instr('hello', 'e')
```

ClickHouse

```
position('hello', 'e')
```

DB2, Derby

```
locate('e', 'hello')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana, Sybase

```
locate('hello', 'e')
```

Access

```
/* UNSUPPORTED */
```

> *(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ.* ***Translate your own SQL on our website***

3.8.12.13. REPEAT

The REPEAT() function repeats a string a number of times.

```
SELECT repeat('abc', 3);
```

```
create.select(repeat("abc", 3)).fetch();
```

The result being

```
+-----------+
| repeat    |
+-----------+
| abcabcabc |
+-----------+
```

Dialect support

This example using jOOQ:

```
repeat("abc", 3)
```

Translates to the following dialect specific expressions:

ASE, SQLDataWarehouse, SQLServer

```
replicate('abc', 3)
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, H2, HSQLDB, MariaDB, MySQL, Postgres, Redshift, Snowflake, Sybase, Vertica, YugabyteDB

```
repeat('abc', 3)
```

Firebird, MemSQL

```
rpad(
  'abc',
  (char_length('abc') * 3),
  'abc'
)
```

Hana, Oracle, Teradata, Trino

```
rpad(
  'abc',
  (length('abc') * 3),
  'abc'
)
```

SQLite

```
replace(hex(zeroblob(3)), '00', 'abc')
```

Access, Derby, Informix

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.14. REPLACE

The REPLACE() function replaces a substring inside of a string by another string.

```
SELECT replace('hello world', 'llo', 'y');
```

```
create.select(replace(val("hello world"), "llo", "y")).fetch();
```

The result being

```
+-----------+
| replace   |
+-----------+
| hey world |
+-----------+
```

Dialect support

This example using jOOQ:

```
replace(val("hello world"), "llo", "y")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Trino, Vertica, YugabyteDB

```
replace('hello world', 'llo', 'y')
```

ASE

```
str_replace('hello world', 'llo', 'y')
```

ClickHouse

```
replaceAll('hello world', 'llo', 'y')
```

Teradata

```
oreplace('hello world', 'llo', 'y')
```

Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.15. REVERSE

The REVERSE() function reverses a string.

```
SELECT reverse('hello');
```

```
create.select(reverse("hello")).fetch();
```

The result being

```
+---------+
| reverse |
+---------+
| olleh   |
+---------+
```

Dialect support

This example using jOOQ:

```
reverse("hello")
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, HSQLDB, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Teradata, Trino, YugabyteDB

```
reverse('hello')
```

Access, DB2, Derby, Firebird, H2, Hana, Informix, MemSQL, SQLite, Sybase, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.16. RIGHT

The RIGHT() function calculates the substring of a given string starting from the right end. See also SUBSTRING, LEFT

```
SELECT right('hello world', 5);
```

```
create.select(right("hello world", 5)).fetch();
```

The result being

```
+-----------+
| right |
+-------+
| world |
+-------+
```

Dialect support

This example using jOOQ:

```
right("hello world", 5)
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica, YugabyteDB

```
right('hello world', 5)
```

Derby

```
substr(
  'hello world',
  (length('hello world') + (1 - 5))
)
```

Oracle, SQLite

```
substr(
  'hello world',
  -5
)
```

Trino

```
substring(
  'hello world',
  -5
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.17. RPAD

The RPAD() pads a string at the right end. See also LPAD.

```
SELECT rpad('hello', 10, '.');
```

```
create.select(rpad(val("hello"), 10, '.')).fetch();
```

The result being

```
+------------+
| rpad       |
+------------+
| hello..... |
+------------+
```

Dialect support

This example using jOOQ:

```
rpad(val("hello"), 10, '.')
```

Translates to the following dialect specific expressions:

Access

```
('hello' & replace(space(10 - len('hello')), ' ', '.'))
```

ASE

```
('hello' || replicate(
  '.',
  (10 - char_length('hello'))
))
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Teradata, Vertica

```
rpad('hello', 10, '.')
```

SQLDataWarehouse, SQLServer

```
('hello' + replicate(
  '.',
  (10 - len('hello'))
))
```

SQLite

```
'hello' || substr(replace(hex(zeroblob(10)), '00', '.'), 1, 10 - length('hello'))
```

Sybase

```
('hello' || repeat(
  '.',
  (10 - length('hello'))
))
```

BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.18. RTRIM

The RTRIM() function trims a string from the right end, stripping it of whitespace. See also LTRIM and TRIM.

```
SELECT rtrim('  hello  ');
```

```
create.select(rtrim("  hello  ")).fetch();
```

The result being

```
+---------+
| rtrim   |
+---------+
|   hello |
+---------+
```

Dialect support

This example using jOOQ:

```
rtrim("  hello  ")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
rtrim('  hello  ')
```

Firebird

```
trim(TRAILING FROM '  hello  ')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.19. SPACE

The SPACE() function repeats a space character a number of times. This is convenience for REPEAT, as available natively in SQL Server, for example.

```
SELECT 'a' || space(3) || 'b';
```

```
create.select(val("a").concat(space(3)).concat(val("b")).fetch();
```

The result being

```
+-------+
| space |
+-------+
| a   b |
+-------+
```

Dialect support

This example using jOOQ:

```
space(3)
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, ClickHouse, DB2, Databricks, Exasol, H2, MariaDB, MySQL, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Vertica

```
space(3)
```

Aurora Postgres, BigQuery, CockroachDB, DuckDB, HSQLDB, Postgres, Redshift, YugabyteDB

```
repeat(' ', 3)
```

Firebird, Hana, Informix, MemSQL, Oracle, Teradata, Trino

```
rpad(' ', 3, ' ')
```

SQLite

```
' ' || substr(replace(hex(zeroblob(3)), '00', ' '), 1, 3 - length(' '))
```

Access, Derby

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.20. SUBSTRING

The SUBSTRING() function calculates the substring of a string given a starting position and optionally, a length. See also LEFT, RIGHT.

```
SELECT
  substring('hello world', 7),
  substring('hello world', 7, 1);
```

```
create.select(
  substring("hello world", 7),
  substring("hello world", 7, 1)).fetch();
```

The result being

```
+-----------+-----------+
| substring | substring |
+-----------+-----------+
| world     | w         |
+-----------+-----------+
```

Dialect support

This example using jOOQ:

```
substring(val("hello world"), 7)
```

Translates to the following dialect specific expressions:

Access

```
mid('hello world', 7)
```

ASE, SQLDataWarehouse, SQLServer

```
substring('hello world', 7, 2147483647)
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, Snowflake, Sybase, Trino, Vertica, YugabyteDB

```
substring('hello world', 7)
```

DB2, Derby, Informix, Oracle, SQLite

```
substr('hello world', 7)
```

Firebird, Teradata

```
substring('hello world' FROM 7)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.21. TRANSLATE

The TRANSLATE() function translates a set of characters to another set of characters within a string, based on matching positions within the search and replacement string.

```
SELECT translate('1 * [2 + 3]', '[]', '()');
```

```
create.select(translate(val("1 * [2 + 3]"), "[]", "()")).fetch();
```

The result being

```
+-------------+
| translate   |
+-------------+
| 1 * (2 + 3) |
+-------------+
```

Dialect support

This example using jOOQ:

```
translate(val("1 * [2 + 3]"), "[]", "()")
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Exasol, H2, HSQLDB, Oracle, Postgres, Redshift, SQLServer, Snowflake, Trino, Vertica, YugabyteDB

```
translate('1 * [2 + 3]', '[]', '()')
```

DB2

```
translate('1 * [2 + 3]', '()', '[]')
```

Teradata

```
otranslate('1 * [2 + 3]', '[]', '()')
```

ASE, Access, Aurora MySQL, Derby, DuckDB, Firebird, Hana, Informix, MariaDB, MemSQL, MySQL, SQLDataWarehouse, SQLite, Sybase

```
/* UNSUPPORTED */
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.22. TRIM

The TRIM() function trims a string from both ends, stripping it of whitespace. See also LTRIM and RTRIM.

```
SELECT trim('  hello  ');
```

```
create.select(trim("  hello  ")).fetch();
```

The result being

```
+-------+
| trim  |
+-------+
| hello |
+-------+
```

Dialect support

This example using jOOQ:

```
trim("  hello  ")
```

Translates to the following dialect specific expressions:

Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
trim('  hello  ')
```

ASE, SQLDataWarehouse

```
ltrim(rtrim('  hello  '))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.12.23. UPPER

The UPPER() function transforms a string into upper case.

```
SELECT upper('hello');
```

```
create.select(upper("hello")).fetch();
```

The result being

```
+-------+
| upper |
+-------+
| HELLO |
+-------+
```

Dialect support

This example using jOOQ:

```
upper("hello")
```

Translates to the following dialect specific expressions:

Access

```
ucase('hello')
```

ASE, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
upper('hello')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13. Datetime functions

Datetime functions are useful to calculate date time arithmetic and formatting.

Many functions in this section come with two flavours supporting both the JDBC datetime data types, and the JSR 310 types. These include:

- SQL DATE modelled by java.time.LocalDate and JDBC's java.sql.Date
- SQL TIME modelled by java.time.LocalTime and JDBC's java.sql.Time
- SQL TIMESTAMP modelled by java.time.LocalDateTime and JDBC's java.sql.Timestamp

Some temporal SQL data types could not be represented canonically with historic JDBC types, but only with JSR 310 types. These include:

- SQL TIME WITH TIME ZONE modelled by java.time.OffsetTime
- SQL TIMESTAMP WITH TIME ZONE modelled by any of java.time.Instant (e.g. PostgreSQL), java.time.OffsetDateTime (JDBC and standard SQL), as well as java.time.ZonedDateTime (e.g. Oracle)

3.8.13.1. CENTURY

Extract the CENTURY value from a datetime value.

The CENTURY function is a short version of the EXTRACT, passing a DatePart.CENTURY value as an argument.

```
SELECT century(DATE '2020-02-03');
```

```
create.select(century(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+---------+
| century |
+---------+
|      21 |
+---------+
```

Dialect support

This example using jOOQ:

```
century(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
(cdec(((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 99)) / 100))
 - (((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 99)) / 100) <
 cdec(((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 99)) / 100))))
```

ASE, Sybase

```
floor(((sign(datepart(yy, '2020-02-03 00:00:00.0')) * (abs(datepart(yy, '2020-02-03 00:00:00.0')) + 99)) / 100))
```

Aurora MySQL, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Redshift, Snowflake, Teradata, Trino, Vertica

```
floor(((sign(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')) * (abs(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')) +
 99)) / 100))
```

Aurora Postgres, Postgres, YugabyteDB

```
extract(CENTURY FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
floor(((sign(extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0')) * (abs(extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0')) + 99)) /
 100))
```

ClickHouse

```
floor(((sign(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00')) * (abs(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00')) + 99)) /
 100))
```

DB2

```
floor(((sign(YEAR(TIMESTAMP '2020-02-03 00:00:00.0')) * (abs(YEAR(TIMESTAMP '2020-02-03 00:00:00.0')) + 99)) / 100))
```

Derby

```
floor(((sign(YEAR(TIMESTAMP('2020-02-03 00:00:00.0'))) * (abs(YEAR(TIMESTAMP('2020-02-03 00:00:00.0'))) + 99)) / 100))
```

Informix

```
floor(((sign(YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)) * (abs(YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)) +
 99)) / 100))
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL

```
floor(((sign(extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})) * (abs(extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})) + 99)) / 100))
```

SQLDataWarehouse, SQLServer

```
floor(((sign(datepart(yy, CAST('2020-02-03 00:00:00.0' AS DATETIME2))) * (abs(datepart(yy, CAST('2020-02-03 00:00:00.0' AS
 DATETIME2))) + 99)) / 100))
```

SQLite

```
floor(((CASE
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) > 0 THEN 1
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) < 0 THEN -1
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) = 0 THEN 0
END * (abs(CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int)) + 99)) / 100))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.2. CURRENT_DATE

Get the current server time as a SQL DATE type (represented by java.sql.Date).

```
SELECT current_date;
```

```
create.select(currentDate()).fetch();
```

The result being something like

```
+--------------+
| current_date |
+--------------+
| 2020-02-03   |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentDate()
```

Translates to the following dialect specific expressions:

Access

```
DATE()
```

ASE, Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL, Snowflake

```
current_date()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_DATE
```

Informix

```
CURRENT YEAR TO DAY
```

Oracle

```
trunc(current_date)
```

SQLDataWarehouse, SQLServer

```
convert(DATE, current_timestamp)
```

Sybase

```
CURRENT DATE
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.3. CURRENT_LOCALDATE

Get the current server time as a SQL DATE type (represented by java.time.LocalDate).

This does the same as CURRENT_DATE except that the client type representation uses JSR-310 types.

```
SELECT current_date;
```

```
create.select(currentLocalDate()).fetch();
```

The result being something like

```
+--------------+
| current_date |
+--------------+
| 2020-02-03   |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentLocalDate()
```

Translates to the following dialect specific expressions:

Access

```
DATE()
```

ASE, Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL, Snowflake

```
current_date()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_DATE
```

Informix

```
CURRENT YEAR TO DAY
```

Oracle

```
trunc(current_date)
```

SQLDataWarehouse, SQLServer

```
convert(DATE, current_timestamp)
```

Sybase

```
CURRENT DATE
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.4. CURRENT_LOCALDATETIME

Get the current server time as a SQL TIMESTAMP type (represented by java.time.LocalDateTime).

This does the same as CURRENT_TIMESTAMP except that the client type representation uses JSR-310 types.

```
SELECT current_timestamp;
```

```
create.select(currentLocalDateTime()).fetch();
```

The result being something like

```
+----------------------+
| current_timestamp    |
+----------------------+
|   2020-02-03 15:30:45 |
+----------------------+
```

Dialect support

This example using jOOQ:

```
currentLocalDateTime()
```

Translates to the following dialect specific expressions:

Access

```
now()
```

ASE

```
current_bigdatetime()
```

Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL, Snowflake

```
current_timestamp()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_TIMESTAMP
```

Informix

```
CURRENT YEAR TO FRACTION (5)
```

Sybase

```
CURRENT TIMESTAMP
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.5. CURRENT_LOCALTIME

Get the current server time as a SQL TIME type (represented by java.time.LocalTime).

This does the same as CURRENT_TIME except that the client type representation uses JSR-310 types.

```
SELECT current_time;
```

```
create.select(currentLocalTime()).fetch();
```

The result being something like

```
+--------------+
| current_time |
+--------------+
|     15:30:45 |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentLocalTime()
```

Translates to the following dialect specific expressions:

Access

```
TIME()
```

ASE, Aurora MySQL, MariaDB, MemSQL, MySQL, Snowflake

```
current_time()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_TIME
```

ClickHouse

```
current_timestamp()
```

Databricks, Exasol, Oracle

```
current_timestamp
```

Informix

```
CURRENT HOUR TO SECOND
```

SQLDataWarehouse, SQLServer

```
convert(TIME, current_timestamp)
```

Sybase

```
CURRENT TIME
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.6. CURRENT_OFFSETDATETIME

Get the current server time as a SQL TIMESTAMP WITH TIME ZONE type (represented by java.time.OffsetDateTime).

This does the same as CURRENT_TIMESTAMP except that a cast is added, and the client type representation uses JSR-310 types.

```
SELECT current_timestamp;
```

```
create.select(currentOffsetDateTime()).fetch();
```

The result being something like

```
+----------------------+
| current_timestamp    |
+----------------------+
|   2020-02-03 15:30:45 |
+----------------------+
```

Dialect support

This example using jOOQ:

```
currentOffsetDateTime()
```

Translates to the following dialect specific expressions:

Access

```
cstr(now())
```

ASE

```
CAST(current_bigdatetime() AS timestamp with time zone)
```

Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL

```
CAST(current_timestamp() AS timestamp with time zone)
```

Aurora Postgres, DB2, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CAST(CURRENT_TIMESTAMP AS timestamp with time zone)
```

BigQuery, Databricks

```
CAST(CURRENT_TIMESTAMP AS timestamp)
```

CockroachDB

```
CAST(CURRENT_TIMESTAMP AS timestamptz)
```

Informix

```
CAST(CURRENT YEAR TO FRACTION (5) AS timestamp with time zone)
```

Snowflake

```
CAST(current_timestamp() AS timestamp_tz)
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLDataWarehouse, SQLServer

```
CAST(CURRENT_TIMESTAMP AS datetimeoffset)
```

Sybase

```
CAST(CURRENT TIMESTAMP AS timestamp with time zone)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.7. CURRENT_OFFSETTIME

Get the current server time as a SQL TIME WITH TIME ZONE type (represented by java.time.OffsetTime).

This does the same as CURRENT_TIME except that a cast is added, and the client type representation uses JSR-310 types.

```
SELECT current_time;
```

```
create.select(currentOffsetTime()).fetch();
```

The result being something like

```
+--------------+
| current_time |
+--------------+
|     15:30:45 |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentOffsetTime()
```

Translates to the following dialect specific expressions:

Access

```
cstr(TIME())
```

ASE, Aurora MySQL, MariaDB, MemSQL, MySQL, Snowflake

```
CAST(current_time() AS time with time zone)
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CAST(CURRENT_TIME AS time with time zone)
```

ClickHouse

```
CAST(current_timestamp() AS time with time zone)
```

Databricks, Exasol

```
CAST(current_timestamp AS time with time zone)
```

Informix

```
CAST(CURRENT HOUR TO SECOND AS time with time zone)
```

Oracle

```
CAST(current_timestamp AS timestamp with time zone)
```

SQLDataWarehouse, SQLServer

```
CAST(convert(TIME, current_timestamp) AS time with time zone)
```

Sybase

```
CAST(CURRENT TIME AS time with time zone)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.8. CURRENT_TIME

Get the current server time as a SQL TIME type (represented by java.sql.Time).

```
SELECT current_time;
```

```
create.select(currentTime()).fetch();
```

The result being something like

```
+--------------+
| current_time |
+--------------+
|     15:30:45 |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentTime()
```

Translates to the following dialect specific expressions:

Access

```
TIME()
```

ASE, Aurora MySQL, MariaDB, MemSQL, MySQL, Snowflake

```
current_time()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, Postgres, Redshift, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_TIME
```

ClickHouse

```
current_timestamp()
```

Databricks, Exasol, Oracle

```
current_timestamp
```

Informix

```
CURRENT HOUR TO SECOND
```

SQLDataWarehouse, SQLServer

```
convert(TIME, current_timestamp)
```

Sybase

```
CURRENT TIME
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.9. CURRENT_TIMESTAMP

Get the current server time as a SQL TIMESTAMP type (represented by java.sql.Timestamp).

```
SELECT current_timestamp;
```

```
create.select(currentTimestamp()).fetch();
```

The result being something like

```
+-----------------------+
| current_timestamp     |
+-----------------------+
|   2020-02-03 15:30:45 |
+-----------------------+
```

Dialect support

This example using jOOQ:

```
currentTimestamp()
```

Translates to the following dialect specific expressions:

Access

```
now()
```

ASE

```
current_bigdatetime()
```

Aurora MySQL, ClickHouse, MariaDB, MemSQL, MySQL, Snowflake

```
current_timestamp()
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Teradata, Trino, Vertica, YugabyteDB

```
CURRENT_TIMESTAMP
```

Informix

```
CURRENT YEAR TO FRACTION (5)
```

Sybase

```
CURRENT TIMESTAMP
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.10. DATE

Convert an ISO 8601 DATE string literal into a SQL DATE type (represented by java.sql.Date).

```
SELECT CAST('2020-02-03' AS DATE);
```

```
create.select(date("2020-02-03")).fetch();
```

The result being

```
+------------+
| date       |
+------------+
| 2020-02-03 |
+------------+
```

Dialect support

This example using jOOQ:

```
date("2020-02-03")
```

Translates to the following dialect specific expressions:

Access

```
#2020/02/03#
```

ASE, SQLite, Sybase

```
'2020-02-03'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Teradata, Vertica

```
DATE '2020-02-03'
```

Derby

```
DATE('2020-02-03')
```

Informix

```
DATETIME(2020-02-03) YEAR TO DAY
```

MemSQL

```
{d '2020-02-03'}
```

SQLDataWarehouse, SQLServer

```
CAST('2020-02-03' AS date)
```

BigQuery, ClickHouse, Databricks, DuckDB, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.11. DATEADD

Add an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) to a date (represented by java.sql.Date).

```
SELECT DATE '2020-02-03' + 3;
```

```
create.select(dateAdd(Date.valueOf("2020-02-03"), 3)).fetch();
```

The result being

```
+------------+
| date_add   |
+------------+
| 2020-02-06 |
+------------+
```

Dialect support

This example using jOOQ:

```
dateAdd(Date.valueOf("2020-02-03"), 3)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', 3, #2020/02/03#)
```

ASE, Sybase

```
dateadd(DAY, 3, '2020-02-03')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(DATE '2020-02-03', INTERVAL 3 DAY)
```

Aurora Postgres, ClickHouse, CockroachDB, Databricks, Exasol, H2, Oracle, Postgres, Redshift, Vertica, YugabyteDB

```
(DATE '2020-02-03' + 3)
```

BigQuery

```
timestamp_add(DATE '2020-02-03', INTERVAL 3 DAY)
```

DB2, HSQLDB

```
(DATE '2020-02-03' + (3) day)
```

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, 3, DATE('2020-02-03')) } AS DATE)
```

DuckDB

```
CAST(date_add(DATE '2020-02-03', INTERVAL 1 DAY * 3) AS DATE)
```

Firebird, Snowflake

```
dateadd(DAY, 3, DATE '2020-02-03')
```

Hana

```
add_days(DATE '2020-02-03', 3)
```

Informix

```
(DATETIME(2020-02-03) YEAR TO DAY + 3 UNITS DAY)
```

MemSQL

```
date_add({d '2020-02-03'}, INTERVAL 3 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, 3, CAST('2020-02-03' AS date))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03', (CAST(3 AS varchar) || ' day'))
```

Teradata

```
DATE '2020-02-03' + CAST(3 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', 3, DATE '2020-02-03')
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.12. DATEDIFF

Subtract two SQL DATE types (represented by java.sql.Date).

This function comes in two flavours:

MySQL 2 argument version

In MySQL, there is a 2 argument verison of the DATEDIFF() function, where the result produces the number of days between the two dates. The argument order is in the order of the difference notation: end_date - start_date

```
SELECT DATEDIFF(
  DATE '2020-02-03',
  DATE '2020-02-01');
```

```
create.select(dateDiff(
  Date.valueOf("2020-02-03"),
  Date.valueOf("2020-02-01"))).fetch();
```

The result being

```
+------------+
| datediff   |
+------------+
|          2 |
+------------+
```

Dialect support

This example using jOOQ:

```
dateDiff(Date.valueOf("2020-02-03"), Date.valueOf("2020-02-01"))
```

Translates to the following dialect specific expressions:

Access

```
datediff('d', #2020/02/01#, #2020/02/03#)
```

ASE, Sybase

```
datediff(DAY, '2020-02-01', '2020-02-03')
```

Aurora MySQL, MariaDB, MySQL

```
datediff(DATE '2020-02-03', DATE '2020-02-01')
```

Aurora Postgres, CockroachDB, Oracle, Postgres, YugabyteDB

```
(DATE '2020-02-03' - DATE '2020-02-01')
```

BigQuery

```
date_diff(DATE '2020-02-03', DATE '2020-02-01', DAY)
```

ClickHouse, Teradata

```
CAST((DATE '2020-02-03' - DATE '2020-02-01') AS integer)
```

Databricks, Exasol

```
CAST((DATE '2020-02-03' - DATE '2020-02-01') AS int)
```

DB2

```
(days(DATE '2020-02-03') - days(DATE '2020-02-01'))
```

Derby

```
{fn timestampdiff(sql_tsi_day, DATE('2020-02-01'), DATE('2020-02-03')) }
```

DuckDB, Redshift

```
datediff('day', DATE '2020-02-01', DATE '2020-02-03')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird, H2, HSQLDB, Snowflake, Vertica

```
datediff(DAY, DATE '2020-02-01', DATE '2020-02-03')
```

Hana

```
days_between(DATE '2020-02-01', DATE '2020-02-03')
```

Informix

```
CAST((DATETIME(2020-02-03) YEAR TO DAY - DATETIME(2020-02-01) YEAR TO DAY) AS integer)
```

MemSQL

```
datediff({d '2020-02-03'}, {d '2020-02-01'})
```

SQLDataWarehouse, SQLServer

```
datediff(DAY, CAST('2020-02-01' AS date), CAST('2020-02-03' AS date))
```

SQLite

```
(strftime('%s', '2020-02-03') - strftime('%s', '2020-02-01')) / 86400
```

Trino

```
date_diff('day', DATE '2020-02-01', DATE '2020-02-03')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

Dialect support

This example using jOOQ:

```
dateDiff(DatePart.MONTH, Date.valueOf("2020-02-03"), Date.valueOf("2020-04-01"))
```

Translates to the following dialect specific expressions:

Aurora MySQL, Aurora Postgres, CockroachDB, Hana, MariaDB, MySQL, Oracle, Postgres, YugabyteDB

```
(((extract(YEAR FROM DATE '2020-04-01') - extract(YEAR FROM DATE '2020-02-03')) * 12) + (extract(MONTH FROM DATE '2020-04-01') -
 extract(MONTH FROM DATE '2020-02-03')))
```

BigQuery

```
date_diff(DATE '2020-04-01', DATE '2020-02-03', MONTH)
```

DB2

```
(((YEAR(DATE '2020-04-01') - YEAR(DATE '2020-02-03')) * 12) + (MONTH(DATE '2020-04-01') - MONTH(DATE '2020-02-03')))
```

Derby

```
(((YEAR(DATE('2020-04-01')) - YEAR(DATE('2020-02-03'))) * 12) + (MONTH(DATE('2020-04-01')) - MONTH(DATE('2020-02-03'))))
```

Firebird, H2, HSQLDB, Snowflake

```
datediff(MONTH, DATE '2020-02-03', DATE '2020-04-01')
```

MemSQL

```
(((extract(YEAR FROM {d '2020-04-01'}) - extract(YEAR FROM {d '2020-02-03'})) * 12) + (extract(MONTH FROM {d '2020-04-01'}) -
 extract(MONTH FROM {d '2020-02-03'})))
```

Redshift

```
datediff('month', DATE '2020-02-03', DATE '2020-04-01')
```

SQLDataWarehouse, SQLServer

```
datediff(MONTH, CAST('2020-02-03' AS date), CAST('2020-04-01' AS date))
```

ASE, Access, ClickHouse, Databricks, DuckDB, Exasol, Informix, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.13. DATESUB

Subtract an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) from a date (represented by java.sql.Date).

```
SELECT DATE '2020-02-03' - 2;
```

```
create.select(dateSub(Date.valueOf("2020-02-03"), 2)).fetch();
```

The result being

```
+------------+
| date_sub   |
+------------+
| 2020-02-01 |
+------------+
```

Dialect support

This example using jOOQ:

```
dateSub(Date.valueOf("2020-02-03"), 2)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', -2, #2020/02/03#)
```

ASE, Sybase

```
dateadd(DAY, -2, '2020-02-03')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(DATE '2020-02-03', INTERVAL -2 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(DATE '2020-02-03' + -2)
```

BigQuery

```
timestamp_sub(DATE '2020-02-03', INTERVAL 2 DAY)
```

ClickHouse, Databricks, Exasol, H2, Oracle, Vertica

```
(DATE '2020-02-03' - 2)
```

DB2, HSQLDB

```
(DATE '2020-02-03' - (2) day)
```

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, -2, DATE('2020-02-03')) } AS DATE)
```

DuckDB

```
CAST(date_add(DATE '2020-02-03', INTERVAL 1 DAY * -2) AS DATE)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Firebird, Snowflake

```
dateadd(DAY, -2, DATE '2020-02-03')
```

Hana

```
add_days(DATE '2020-02-03', -2)
```

Informix

```
(DATETIME(2020-02-03) YEAR TO DAY - 2 UNITS DAY)
```

MemSQL

```
date_add({d '2020-02-03'}, INTERVAL -2 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, -2, CAST('2020-02-03' AS date))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03', (CAST(-2 AS varchar) || ' day'))
```

Teradata

```
DATE '2020-02-03' - CAST(2 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', -2, DATE '2020-02-03')
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.14. DAY

Extract the DAY value from a datetime value.

The DAY function is a short version of the EXTRACT, passing a DatePart.DAY value as an argument.

```
SELECT day(DATE '2020-02-03');
```

```
create.select(day(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+-----+
| day |
+-----+
|   3 |
+-----+
```

Dialect support

This example using jOOQ:

```
day(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('d', #2020/02/03 00:00:00#)
```

ASE, Sybase

```
datepart(dd, '2020-02-03 00:00:00.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(DAY FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
extract(DAY FROM DATETIME '2020-02-03 00:00:00.0')
```

ClickHouse

```
extract(DAY FROM TIMESTAMP '2020-02-03 00:00:00')
```

DB2

```
DAY(TIMESTAMP '2020-02-03 00:00:00.0')
```

Derby

```
DAY(TIMESTAMP('2020-02-03 00:00:00.0'))
```

Informix

```
DAY(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)
```

MemSQL

```
extract(DAY FROM {ts '2020-02-03 00:00:00.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(dd, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%d', '2020-02-03 00:00:00.0') AS int)
```

© 2009 - 2025 by Data Geekery™ GmbH.

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.15. DAY_OF_YEAR

Extract the DAY_OF_YEAR value from a datetime value.

The DAY_OF_YEAR function is a short version of the EXTRACT, passing a DatePart.DAY_OF_YEAR value as an argument.

```
SELECT day_of_year(DATE '2020-02-03');
```

```
create.select(dayOfYear(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+-------------+
| day_of_year |
+-------------+
|          33 |
+-------------+
```

Dialect support

This example using jOOQ:

```
dayOfYear(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

ASE, Sybase

```
datepart(dy, '2020-02-03 00:00:00.0')
```

Aurora MySQL, DB2, Hana, MariaDB, MySQL

```
dayofyear(TIMESTAMP '2020-02-03 00:00:00.0')
```

Aurora Postgres, CockroachDB, Postgres

```
extract(DOY FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

H2, HSQLDB

```
extract(DAY_OF_YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

MemSQL

```
dayofyear({ts '2020-02-03 00:00:00.0'})
```

Oracle

```
to_number(to_char(TIMESTAMP '2020-02-03 00:00:00.0', 'DDD'))
```

SQLDataWarehouse, SQLServer

```
datepart(dy, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%j', '2020-02-03 00:00:00.0') AS int)
```

Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, Informix, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.16. DECADE

Extract the DECADE value from a datetime value.

The DECADE function is a short version of the EXTRACT, passing a DatePart.DECADE value as an argument.

```
SELECT decade(DATE '2020-02-03');
```

```
create.select(decade(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+--------+
| decade |
+--------+
|    202 |
+--------+
```

Dialect support

This example using jOOQ:

```
decade(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
(cdec((datepart('yyyy', #2020/02/03 00:00:00#) / 10)) - ((datepart('yyyy', #2020/02/03 00:00:00#) / 10) < cdec((datepart('yyyy',
 #2020/02/03 00:00:00#) / 10))))
```

ASE, Sybase

```
floor((datepart(yy, '2020-02-03 00:00:00.0') / 10))
```

Aurora MySQL, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Redshift, Snowflake, Teradata, Trino, Vertica

```
floor((extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0') / 10))
```

Aurora Postgres, Postgres, YugabyteDB

```
extract(DECADE FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
floor((extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0') / 10))
```

© 2009 - 2025 by Data Geekery™ GmbH.

ClickHouse

```
floor((extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00') / 10))
```

DB2

```
floor((YEAR(TIMESTAMP '2020-02-03 00:00:00.0') / 10))
```

Derby

```
floor((YEAR(TIMESTAMP('2020-02-03 00:00:00.0')) / 10))
```

Informix

```
floor((YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION) / 10))
```

MemSQL

```
floor((extract(YEAR FROM {ts '2020-02-03 00:00:00.0'}) / 10))
```

SQLDataWarehouse, SQLServer

```
floor((datepart(yy, CAST('2020-02-03 00:00:00.0' AS DATETIME2)) / 10))
```

SQLite

```
floor((CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) / 10))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.17. EPOCH

Extract the EPOCH value from a datetime value, i.e. the number of seconds since 1970-01-01 00:00:00 UTC.

The EPOCH function is a short version of the EXTRACT, passing a DatePart.EPOCH value as an argument.

```
SELECT epoch(TIMESTAMP '1970-01-01 00:00:15');
```

```
create.select(epoch(Timestamp.valueOf("1970-01-01
 00:00:15"))).fetch();
```

The result being

```
+-------+
| epoch |
+-------+
|    15 |
+-------+
```

Dialect support

This example using jOOQ:

```
epoch(Timestamp.valueOf("1970-01-01 00:00:15"))
```

Translates to the following dialect specific expressions:

ASE, Sybase

```
datediff(ss, '1970-01-01 00:00:00', '1970-01-01 00:00:15.0')
```

Aurora MySQL, HSQLDB, MariaDB, MySQL

```
UNIX_TIMESTAMP(TIMESTAMP '1970-01-01 00:00:15.0')
```

Aurora Postgres, CockroachDB, DB2, H2, Postgres

```
extract(EPOCH FROM TIMESTAMP '1970-01-01 00:00:15.0')
```

Hana

```
seconds_between('1970-01-01', TIMESTAMP '1970-01-01 00:00:15.0')
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL

```
UNIX_TIMESTAMP({ts '1970-01-01 00:00:15.0'})
```

Oracle

```
trunc((CAST(TIMESTAMP '1970-01-01 00:00:15.0' AS date) - DATE '1970-01-01') * 86400)
```

SQLDataWarehouse, SQLServer

```
datediff(ss, '1970-01-01 00:00:00', CAST('1970-01-01 00:00:15.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%s', '1970-01-01 00:00:15.0') AS int)
```

Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, Informix, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.18. EXTRACT

Extract a org.jooq.DatePart from a datetime value.

```
SELECT EXTRACT(MONTH FROM DATE '2020-02-03');
```

```
create.select(extract(Date.valueOf("2020-02-03"),
 DatePart.MONTH)).fetch();
```

The result being

```
+-------+
| month |
+-------+
|     2 |
+-------+
```

Dialect support

This example using jOOQ:

```
extract(Date.valueOf("2020-02-03"), DatePart.MONTH)
```

Translates to the following dialect specific expressions:

Access

```
datepart('m', #2020/02/03 00:00:00#)
```

ASE, Sybase

```
datepart(mm, '2020-02-03 00:00:00.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
extract(MONTH FROM DATETIME '2020-02-03 00:00:00.0')
```

ClickHouse

```
extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00')
```

DB2

```
MONTH(TIMESTAMP '2020-02-03 00:00:00.0')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
MONTH(TIMESTAMP('2020-02-03 00:00:00.0'))
```

Informix

```
MONTH(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)
```

MemSQL

```
extract(MONTH FROM {ts '2020-02-03 00:00:00.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(mm, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%m', '2020-02-03 00:00:00.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.19. HOUR

Extract the HOUR value from a datetime value.

The HOUR function is a short version of the EXTRACT, passing a DatePart.HOUR value as an argument.

```
SELECT hour(TIMESTAMP '2020-02-03 15:30:45');
```

```
create.select(hour(Timestamp.valueOf("2020-02-03
 15:30:45"))).fetch();
```

The result being

```
+------+
| hour |
+------+
|   15 |
+------+
```

Dialect support

This example using jOOQ:

```
hour(Timestamp.valueOf("2020-02-03 15:30:45"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('h', #2020/02/03 15:30:45#)
```

ASE, Sybase

```
datepart(hh, '2020-02-03 15:30:45.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(HOUR FROM TIMESTAMP '2020-02-03 15:30:45.0')
```

BigQuery

```
extract(HOUR FROM DATETIME '2020-02-03 15:30:45.0')
```

ClickHouse

```
extract(HOUR FROM TIMESTAMP '2020-02-03 15:30:45')
```

DB2

```
HOUR(TIMESTAMP '2020-02-03 15:30:45.0')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
HOUR(TIMESTAMP('2020-02-03 15:30:45.0'))
```

Informix

```
CAST(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION AS CAST(DATETIME HOUR TO HOUR AS CAST(CHAR(2) AS INT)))
```

MemSQL

```
extract(HOUR FROM {ts '2020-02-03 15:30:45.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(hh, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%H', '2020-02-03 15:30:45.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.20. ISO_DAY_OF_WEEK

Extract the ISO_DAY_OF_WEEK value from a datetime value.

The ISO_DAY_OF_WEEK function is a short version of the EXTRACT, passing a DatePart.ISO_DAY_OF_WEEK value as an argument.

```
SELECT iso_day_of_week(DATE '2020-02-03');
```

```
create.select(isoDayOfWeek(Date.valueOf("2020-02-03"))).fetch();
```

The result being (Monday = 1, ..., Sunday = 7)

```
+-----------------+
| iso_day_of_week |
+-----------------+
|               7 |
+-----------------+
```

Dialect support

This example using jOOQ:

```
isoDayOfWeek(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

ASE

```
(((DATEPART(dw, '2020-02-03 00:00:00.0') + @@datefirst + 5) % 7) + 1)
```

Aurora MySQL, MariaDB, MySQL

```
weekday(TIMESTAMP '2020-02-03 00:00:00.0') + 1
```

Aurora Postgres, CockroachDB, Postgres

```
extract(ISODOW FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

DB2

```
DAYOFWEEK_ISO(TIMESTAMP '2020-02-03 00:00:00.0')
```

H2

```
extract(ISO_DAY_OF_WEEK FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

Hana

```
(weekday(TIMESTAMP '2020-02-03 00:00:00.0') + 1)
```

© 2009 - 2025 by Data Geekery™ GmbH.

HSQLDB

```
(mod(
  (EXTRACT(DAY_OF_WEEK FROM TIMESTAMP '2020-02-03 00:00:00.0') + 5),
  7
) + 1)
```

MemSQL

```
weekday({ts '2020-02-03 00:00:00.0'}) + 1
```

Oracle

```
to_number(to_char(TIMESTAMP '2020-02-03 00:00:00.0', 'D'))
```

SQLDataWarehouse, SQLServer

```
(((DATEPART(dw, CAST('2020-02-03 00:00:00.0' AS DATETIME2)) + @@datefirst + 5) % 7) + 1)
```

SQLite

```
(((CAST(strftime('%w', '2020-02-03 00:00:00.0') AS int) + 6) % 7) + 1)
```

Sybase

```
(mod(
  (DATEPART(dw, '2020-02-03 00:00:00.0') + @@datefirst + 5),
  7
) + 1)
```

Access, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, Informix, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.21. LOCALDATE

Convert an ISO 8601 DATE string literal into a SQL DATE type (represented by java.time.LocalDate). This does the same as DATE except that the client type representation uses JSR-310 types.

```
SELECT CAST('2020-02-03' AS DATE);
```

```
create.select(localDate("2020-02-03")).fetch();
```

The result being

```
+------------+
| date       |
+------------+
| 2020-02-03 |
+------------+
```

Dialect support

This example using jOOQ:

```
localDate("2020-02-03")
```

Translates to the following dialect specific expressions:

Access

```
#2020/02/03#
```

ASE, SQLite, Sybase

```
'2020-02-03'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Teradata, Vertica

```
DATE '2020-02-03'
```

Derby

```
DATE('2020-02-03')
```

Informix

```
DATETIME(2020-02-03) YEAR TO DAY
```

MemSQL

```
{d '2020-02-03'}
```

SQLDataWarehouse, SQLServer

```
CAST('2020-02-03' AS date)
```

BigQuery, ClickHouse, Databricks, DuckDB, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.22. LOCALDATEADD

Add an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) to a date (represented by java.time.LocalDate).

This does the same as DATEADD except that the client type representation uses JSR-310 types.

```
SELECT DATE '2020-02-03' + 3;
```

```
create.select(localDateAdd(LocalDate.parse("2020-02-03"),
 3)).fetch();
```

The result being

```
+------------+
| date_add   |
+------------+
| 2020-02-06 |
+------------+
```

Dialect support

This example using jOOQ:

```
localDateAdd(LocalDate.parse("2020-02-03"), 3)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', 3, #2020/02/03#)
```

ASE, Sybase

```
dateadd(DAY, 3, '2020-02-03')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(DATE '2020-02-03', INTERVAL 3 DAY)
```

Aurora Postgres, ClickHouse, CockroachDB, Databricks, Exasol, H2, Oracle, Postgres, Redshift, Vertica, YugabyteDB

```
(DATE '2020-02-03' + 3)
```

BigQuery

```
timestamp_add(DATE '2020-02-03', INTERVAL 3 DAY)
```

DB2, HSQLDB

```
(DATE '2020-02-03' + (3) day)
```

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, 3, DATE('2020-02-03')) } AS DATE)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DuckDB

```
CAST(date_add(DATE '2020-02-03', INTERVAL 1 DAY * 3) AS DATE)
```

Firebird, Snowflake

```
dateadd(DAY, 3, DATE '2020-02-03')
```

Hana

```
add_days(DATE '2020-02-03', 3)
```

Informix

```
(DATETIME(2020-02-03) YEAR TO DAY + 3 UNITS DAY)
```

MemSQL

```
date_add({d '2020-02-03'}, INTERVAL 3 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, 3, CAST('2020-02-03' AS date))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03', (CAST(3 AS varchar) || ' day'))
```

Teradata

```
DATE '2020-02-03' + CAST(3 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Trino

```
date_add('day', 3, DATE '2020-02-03')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.23. LOCALDATESUB

Subtract an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) from a date (represented by java.time.LocalDate).

This does the same as DATESUB except that the client type representation uses JSR-310 types.

```
SELECT DATE '2020-02-03' + 2;
```

```
create.select(localDateSub(Date.valueOf("2020-02-03"),
 2)).fetch();
```

The result being

```
+------------+
| date_sub   |
+------------+
| 2020-02-01 |
+------------+
```

Dialect support

This example using jOOQ:

```
localDateSub(LocalDate.parse("2020-02-03"), 2)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', -2, #2020/02/03#)
```

ASE, Sybase

```
dateadd(DAY, -2, '2020-02-03')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(DATE '2020-02-03', INTERVAL -2 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(DATE '2020-02-03' + -2)
```

BigQuery

```
timestamp_sub(DATE '2020-02-03', INTERVAL 2 DAY)
```

ClickHouse, Databricks, Exasol, H2, Oracle, Vertica

```
(DATE '2020-02-03' - 2)
```

DB2, HSQLDB

```
(DATE '2020-02-03' - (2) day)
```

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, -2, DATE('2020-02-03')) } AS DATE)
```

DuckDB

```
CAST(date_add(DATE '2020-02-03', INTERVAL 1 DAY * -2) AS DATE)
```

Firebird, Snowflake

```
dateadd(DAY, -2, DATE '2020-02-03')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Hana

```
add_days(DATE '2020-02-03', -2)
```

Informix

```
(DATETIME(2020-02-03) YEAR TO DAY - 2 UNITS DAY)
```

MemSQL

```
date_add({d '2020-02-03'}, INTERVAL -2 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, -2, CAST('2020-02-03' AS date))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03', (CAST(-2 AS varchar) || ' day'))
```

Teradata

```
DATE '2020-02-03' - CAST(2 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', -2, DATE '2020-02-03')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.24. LOCALDATETIME

Convert an ISO 8601 TIMESTAMP string literal into a SQL TIMESTAMP type (represented by java.time.LocalDateTime).

This does the same as TIMESTAMP except that the client type representation uses JSR-310 types.

```
SELECT CAST('2020-02-03 15:30:45' AS TIMESTAMP);
```

```
create.select(localDateTime("2020-02-03 15:30:45")).fetch();
```

The result being

```
+--------------------+
| timestamp          |
+--------------------+
| 2020-02-03 15:30:45 |
+--------------------+
```

Dialect support

This example using jOOQ:

```
localDateTime("2020-02-03 15:30:45")
```

Translates to the following dialect specific expressions:

Access

```
#2020/02/03 15:30:45#
```

ASE, SQLite, Sybase

```
'2020-02-03 15:30:45.0'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Teradata, Vertica

```
TIMESTAMP '2020-02-03 15:30:45.0'
```

Derby

```
TIMESTAMP('2020-02-03 15:30:45.0')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION
```

MemSQL

```
{ts '2020-02-03 15:30:45.0'}
```

SQLDataWarehouse, SQLServer

```
CAST('2020-02-03 15:30:45.0' AS DATETIME2)
```

BigQuery, ClickHouse, Databricks, DuckDB, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.25. LOCALDATETIMEADD

Add an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) to a timestamp (represented by java.time.LocalDateTime).

This does the same as TIMESTAMPADD except that the client type representation uses JSR-310 types.

```
SELECT DATE '2020-02-03 15:30:45' + INTERVAL 3 DAYS;
```

```
create.select(localDateTimeAdd(LocalDateTime.parse("2020-02-03T15:30:45"),
 3)).fetch();
```

The result being

```
+---------------------+
| timestamp_add       |
+---------------------+
| 2020-02-06 15:30:45 |
+---------------------+
```

Dialect support

This example using jOOQ:

```
localDateTimeAdd(LocalDateTime.parse("2020-02-03T15:30:45"), 3)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', 3, #2020/02/03 15:30:45#)
```

ASE, Sybase

```
dateadd(DAY, 3, '2020-02-03 15:30:45.0')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 3 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + 3 * INTERVAL '1 day')
```

BigQuery

```
timestamp_add(DATETIME '2020-02-03 15:30:45.0', INTERVAL 3 DAY)
```

ClickHouse

```
(TIMESTAMP '2020-02-03 15:30:45' + 3)
```

Databricks, Exasol, H2, Oracle, Vertica

```
(TIMESTAMP '2020-02-03 15:30:45.0' + 3)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, HSQLDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + (3) day)
```

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, 3, TIMESTAMP('2020-02-03 15:30:45.0')) } AS TIMESTAMP)
```

DuckDB

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 1 DAY * 3)
```

Firebird, Snowflake

```
dateadd(DAY, 3, TIMESTAMP '2020-02-03 15:30:45.0')
```

Hana

```
add_days(TIMESTAMP '2020-02-03 15:30:45.0', 3)
```

Informix

```
(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION + 3 UNITS DAY)
```

MemSQL

```
date_add({ts '2020-02-03 15:30:45.0'}, INTERVAL 3 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, 3, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03 15:30:45.0', (CAST(3 AS varchar) || ' day'))
```

Teradata

```
TIMESTAMP '2020-02-03 15:30:45.0' + CAST(3 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', 3, TIMESTAMP '2020-02-03 15:30:45.0')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.26. LOCALDATETIMESUB

Subtract an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) from a timestamp (represented by java.time.LocalDateTime).

This does the same as TIMESTAMPSUB except that the client type representation uses JSR-310 types.

```
SELECT DATE '2020-02-03 15:30:45' - INTERVAL 2 DAYS;
```

```
create.select(localDateTimeSub(LocalDateTime.parse("2020-02-03T15:30:45"),
 2)).fetch();
```

The result being

```
+---------------------+
| timestamp_sub       |
+---------------------+
| 2020-02-01 15:30:45 |
+---------------------+
```

Dialect support

This example using jOOQ:

```
localDateTimeSub(LocalDateTime.parse("2020-02-03T15:30:45"), 2)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', -2, #2020/02/03 15:30:45#)
```

ASE, Sybase

```
dateadd(DAY, -2, '2020-02-03 15:30:45.0')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL -2 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + -2 * INTERVAL '1 day')
```

BigQuery

```
timestamp_sub(DATETIME '2020-02-03 15:30:45.0', INTERVAL 2 DAY)
```

ClickHouse

```
(TIMESTAMP '2020-02-03 15:30:45' - 2)
```

Databricks, Exasol, H2, Oracle, Vertica

```
(TIMESTAMP '2020-02-03 15:30:45.0' - 2)
```

DB2, HSQLDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' - (2) day)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
CAST({fn timestampadd(SQL_TSI_DAY, -2, TIMESTAMP('2020-02-03 15:30:45.0')) } AS TIMESTAMP)
```

DuckDB

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 1 DAY * -2)
```

Firebird, Snowflake

```
dateadd(DAY, -2, TIMESTAMP '2020-02-03 15:30:45.0')
```

Hana

```
add_days(TIMESTAMP '2020-02-03 15:30:45.0', -2)
```

Informix

```
(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION - 2 UNITS DAY)
```

MemSQL

```
date_add({ts '2020-02-03 15:30:45.0'}, INTERVAL -2 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, -2, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03 15:30:45.0', (CAST(-2 AS varchar) || ' day'))
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
TIMESTAMP '2020-02-03 15:30:45.0' - CAST(2 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', -2, TIMESTAMP '2020-02-03 15:30:45.0')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.27. LOCALTIME

Convert an ISO 8601 TIME string literal into a SQL TIME type (represented by java.time.LocalTime). This does the same as TIME except that the client type representation uses JSR-310 types.

```
SELECT CAST('15:30:45' AS TIME);
```

```
create.select(localTime("15:30:45")).fetch();
```

The result being

```
+----------+
| time     |
+----------+
| 15:30:45 |
+----------+
```

Dialect support

This example using jOOQ:

```
localTime("15:30:45")
```

Translates to the following dialect specific expressions:

Access, MemSQL

```
{t '15:30:45'}
```

ASE, SQLite, Sybase

```
'15:30:45'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Postgres, Teradata, Vertica

```
TIME '15:30:45'
```

Derby

```
TIME('15:30:45')
```

Informix

```
DATETIME(15:30:45) HOUR TO SECOND
```

Oracle

```
TIMESTAMP '1970-01-01 15:30:45'
```

SQLDataWarehouse, SQLServer

```
CAST(CAST('15:30:45' AS time) AS time)
```

BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.28. MILLENNIUM

Extract the MILLENNIUM value from a datetime value.

The MILLENNIUM function is a short version of the EXTRACT, passing a DatePart.MILLENNIUM value as an argument.

```
SELECT millennium(DATE '2020-02-03');
```

```
create.select(millennium(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+------------+
| millennium |
+------------+
|          3 |
+------------+
```

Dialect support

This example using jOOQ:

```
millennium(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
(cdec(((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 999)) / 1000))
 - (((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 999)) / 1000) <
 cdec(((sgn(datepart('yyyy', #2020/02/03 00:00:00#)) * (abs(datepart('yyyy', #2020/02/03 00:00:00#)) + 999)) / 1000))))
```

ASE, Sybase

```
floor(((sign(datepart(yy, '2020-02-03 00:00:00.0')) * (abs(datepart(yy, '2020-02-03 00:00:00.0')) + 999)) / 1000))
```

Aurora MySQL, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Redshift, Snowflake, Teradata, Trino, Vertica

```
floor(((sign(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')) * (abs(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')) +
 999)) / 1000))
```

Aurora Postgres, Postgres, YugabyteDB

```
extract(MILLENNIUM FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
floor(((sign(extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0')) * (abs(extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0')) +
 999)) / 1000))
```

ClickHouse

```
floor(((sign(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00')) * (abs(extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00')) + 999)) /
 1000))
```

DB2

```
floor(((sign(YEAR(TIMESTAMP '2020-02-03 00:00:00.0')) * (abs(YEAR(TIMESTAMP '2020-02-03 00:00:00.0')) + 999)) / 1000))
```

Derby

```
floor(((sign(YEAR(TIMESTAMP('2020-02-03 00:00:00.0'))) * (abs(YEAR(TIMESTAMP('2020-02-03 00:00:00.0'))) + 999)) / 1000))
```

Informix

```
floor(((sign(YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)) * (abs(YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)) +
 999)) / 1000))
```

MemSQL

```
floor(((sign(extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})) * (abs(extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})) + 999)) / 1000))
```

SQLDataWarehouse, SQLServer

```
floor(((sign(datepart(yy, CAST('2020-02-03 00:00:00.0' AS DATETIME2))) * (abs(datepart(yy, CAST('2020-02-03 00:00:00.0' AS
 DATETIME2))) + 999)) / 1000))
```

SQLite

```
floor(((CASE
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) > 0 THEN 1
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) < 0 THEN -1
  WHEN CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int) = 0 THEN 0
END * (abs(CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int)) + 999)) / 1000))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.29. MINUTE

Extract the MINUTE value from a datetime value.

The MINUTE function is a short version of the EXTRACT, passing a DatePart.MINUTE value as an argument.

```
SELECT minute(TIMESTAMP '2020-02-03 15:30:45');
```

```
create.select(minute(Timestamp.valueOf("2020-02-03
 15:30:45"))).fetch();
```

The result being

```
+--------+
| minute |
+--------+
|     30 |
+--------+
```

Dialect support

This example using jOOQ:

```
minute(Timestamp.valueOf("2020-02-03 15:30:45"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('n', #2020/02/03 15:30:45#)
```

ASE, Sybase

```
datepart(mi, '2020-02-03 15:30:45.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(MINUTE FROM TIMESTAMP '2020-02-03 15:30:45.0')
```

BigQuery

```
extract(MINUTE FROM DATETIME '2020-02-03 15:30:45.0')
```

ClickHouse

```
extract(MINUTE FROM TIMESTAMP '2020-02-03 15:30:45')
```

DB2

```
MINUTE(TIMESTAMP '2020-02-03 15:30:45.0')
```

Derby

```
MINUTE(TIMESTAMP('2020-02-03 15:30:45.0'))
```

Informix

```
CAST(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION AS CAST(DATETIME MINUTE TO MINUTE AS CAST(CHAR(2) AS INT)))
```

MemSQL

```
extract(MINUTE FROM {ts '2020-02-03 15:30:45.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(mi, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLite

```
CAST(strftime('%M', '2020-02-03 15:30:45.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.30. MONTH

Extract the MONTH value from a datetime value.

The MONTH function is a short version of the EXTRACT, passing a DatePart.MONTH value as an argument.

```
SELECT month(DATE '2020-02-03');
```

```
create.select(month(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+-------+
| month |
+-------+
|     2 |
+-------+
```

Dialect support

This example using jOOQ:

```
month(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('m', #2020/02/03 00:00:00#)
```

ASE, Sybase

```
datepart(mm, '2020-02-03 00:00:00.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
extract(MONTH FROM DATETIME '2020-02-03 00:00:00.0')
```

ClickHouse

```
extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00')
```

DB2

```
MONTH(TIMESTAMP '2020-02-03 00:00:00.0')
```

Derby

```
MONTH(TIMESTAMP('2020-02-03 00:00:00.0'))
```

Informix

```
MONTH(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)
```

MemSQL

```
extract(MONTH FROM {ts '2020-02-03 00:00:00.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(mm, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLite

```
CAST(strftime('%m', '2020-02-03 00:00:00.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.31. QUARTER

Extract the QUARTER value from a datetime value.

The QUARTER function is a short version of the EXTRACT, passing a DatePart.QUARTER value as an argument.

```
SELECT quarter(DATE '2020-02-03');
```

```
create.select(quarter(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+---------+
| quarter |
+---------+
|       1 |
+---------+
```

Dialect support

This example using jOOQ:

```
quarter(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
(cdec(((datepart('m', #2020/02/03 00:00:00#) + 2) / 3)) - (((datepart('m', #2020/02/03 00:00:00#) + 2) / 3) < cdec(((datepart('m',
 #2020/02/03 00:00:00#) + 2) / 3))))
```

ASE, Sybase

```
datepart(qq, '2020-02-03 00:00:00.0')
```

Aurora MySQL, DB2, H2, HSQLDB, MariaDB, MySQL

```
quarter(TIMESTAMP '2020-02-03 00:00:00.0')
```

Aurora Postgres, CockroachDB, Postgres, Snowflake, YugabyteDB

```
extract(QUARTER FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
floor(((extract(MONTH FROM DATETIME '2020-02-03 00:00:00.0') + 2) / 3))
```

ClickHouse

```
floor(((extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00') + 2) / 3))
```

Databricks, DuckDB, Exasol, Firebird, Hana, Redshift, Teradata, Trino, Vertica

```
floor(((extract(MONTH FROM TIMESTAMP '2020-02-03 00:00:00.0') + 2) / 3))
```

Derby

```
floor(((MONTH(TIMESTAMP('2020-02-03 00:00:00.0')) + 2) / 3))
```

Informix

```
floor(((MONTH(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION) + 2) / 3))
```

MemSQL

```
quarter({ts '2020-02-03 00:00:00.0'})
```

© 2009 - 2025 by Data Geekery™ GmbH.

Oracle

```
to_number(to_char(TIMESTAMP '2020-02-03 00:00:00.0', 'Q'))
```

SQLDataWarehouse, SQLServer

```
datepart(qq, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

SQLite

```
floor(((CAST(strftime('%m', '2020-02-03 00:00:00.0') AS int) + 2) / 3))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.32. SECOND

Extract the SECOND value from a datetime value.

The SECOND function is a short version of the EXTRACT, passing a DatePart.SECOND value as an argument.

```
SELECT second(TIMESTAMP '2020-02-03 15:30:45');
```

```
create.select(second(Timestamp.valueOf("2020-02-03
 15:30:45"))).fetch();
```

The result being

```
+--------+
| second |
+--------+
|     45 |
+--------+
```

Dialect support

This example using jOOQ:

```
second(Timestamp.valueOf("2020-02-03 15:30:45"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('s', #2020/02/03 15:30:45#)
```

ASE, Sybase

```
datepart(ss, '2020-02-03 15:30:45.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(SECOND FROM TIMESTAMP '2020-02-03 15:30:45.0')
```

BigQuery

```
extract(SECOND FROM DATETIME '2020-02-03 15:30:45.0')
```

ClickHouse

```
extract(SECOND FROM TIMESTAMP '2020-02-03 15:30:45')
```

DB2

```
SECOND(TIMESTAMP '2020-02-03 15:30:45.0')
```

Derby

```
SECOND(TIMESTAMP('2020-02-03 15:30:45.0'))
```

Informix

```
CAST(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION AS CAST(DATETIME SECOND TO SECOND AS CAST(CHAR(2) AS INT)))
```

© 2009 - 2025 by Data Geekery™ GmbH.

MemSQL

```
extract(SECOND FROM {ts '2020-02-03 15:30:45.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(ss, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%S', '2020-02-03 15:30:45.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.33. TIME

Convert an ISO 8601 TIME string literal into a SQL TIME type (represented by java.sql.Time).

```
SELECT CAST('15:30:45' AS TIME);
```

```
create.select(time("15:30:45")).fetch();
```

The result being

```
+----------+
| time     |
+----------+
| 15:30:45 |
+----------+
```

Dialect support

This example using jOOQ:

```
time("15:30:45")
```

Translates to the following dialect specific expressions:

Access, MemSQL

```
{t '15:30:45'}
```

ASE, SQLite, Sybase

```
'15:30:45'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Postgres, Teradata, Vertica

```
TIME '15:30:45'
```

Derby

```
TIME('15:30:45')
```

Informix

```
DATETIME(15:30:45) HOUR TO SECOND
```

Oracle

```
TIMESTAMP '1970-01-01 15:30:45'
```

SQLDataWarehouse, SQLServer

```
CAST(CAST('15:30:45' AS time) AS time)
```

BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.34. TIMESTAMP

Convert an ISO 8601 TIMESTAMP string literal into a SQL TIMESTAMP type (represented by java.sql.Timestamp).

```
SELECT CAST('2020-02-03 15:30:45' AS TIMESTAMP);
```

```
create.select(timestamp("2020-02-03 15:30:45")).fetch();
```

The result being

```
+--------------------+
| timestamp          |
+--------------------+
| 2020-02-03 15:30:45 |
+--------------------+
```

Dialect support

This example using jOOQ:

```
timestamp("2020-02-03 15:30:45")
```

Translates to the following dialect specific expressions:

Access

```
#2020/02/03 15:30:45#
```

ASE, SQLite, Sybase

```
'2020-02-03 15:30:45.0'
```

Aurora MySQL, Aurora Postgres, CockroachDB, DB2, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Teradata, Vertica

```
TIMESTAMP '2020-02-03 15:30:45.0'
```

Derby

```
TIMESTAMP('2020-02-03 15:30:45.0')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION
```

MemSQL

```
{ts '2020-02-03 15:30:45.0'}
```

SQLDataWarehouse, SQLServer

```
CAST('2020-02-03 15:30:45.0' AS DATETIME2)
```

BigQuery, ClickHouse, Databricks, DuckDB, Redshift, Snowflake, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.35. TIMESTAMPADD

Add an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) to a timestamp (represented by java.sql.Timestamp).

```
SELECT DATE '2020-02-03 15:30:45' + INTERVAL 3 DAYS;
```

```
create.select(timestampAdd(Timestamp.valueOf("2020-02-03
 15:30:45"), 3)).fetch();
```

The result being

```
+--------------------+
| timestamp_add      |
+--------------------+
| 2020-02-06 15:30:45 |
+--------------------+
```

Dialect support

This example using jOOQ:

```
timestampAdd(Timestamp.valueOf("2020-02-03 15:30:45"), 3)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', 3, #2020/02/03 15:30:45#)
```

ASE, Sybase

```
dateadd(DAY, 3, '2020-02-03 15:30:45.0')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 3 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + 3 * INTERVAL '1 day')
```

BigQuery

```
timestamp_add(DATETIME '2020-02-03 15:30:45.0', INTERVAL 3 DAY)
```

ClickHouse

```
(TIMESTAMP '2020-02-03 15:30:45' + 3)
```

Databricks, Exasol, H2, Oracle, Vertica

```
(TIMESTAMP '2020-02-03 15:30:45.0' + 3)
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, HSQLDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + (3) day)
```

Derby

```
{fn timestampadd(SQL_TSI_DAY, 3, TIMESTAMP('2020-02-03 15:30:45.0')) }
```

DuckDB

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 1 DAY * 3)
```

Firebird, Snowflake

```
dateadd(DAY, 3, TIMESTAMP '2020-02-03 15:30:45.0')
```

Hana

```
add_days(TIMESTAMP '2020-02-03 15:30:45.0', 3)
```

Informix

```
(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION + 3 UNITS DAY)
```

MemSQL

```
date_add({ts '2020-02-03 15:30:45.0'}, INTERVAL 3 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, 3, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03 15:30:45.0', (CAST(3 AS varchar) || ' day'))
```

Teradata

```
TIMESTAMP '2020-02-03 15:30:45.0' + CAST(3 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', 3, TIMESTAMP '2020-02-03 15:30:45.0')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.36. TIMESTAMPSUB

Subtract an interval of type java.lang.Number (number of days) or org.jooq.types.Interval (SQL interval type) from a timestamp (represented by java.sql.Timestamp).

```
SELECT DATE '2020-02-03 15:30:45' - INTERVAL 2 DAYS;
```

```
create.select(timestampSub(Timestamp.valueOf("2020-02-03
 15:30:45"), 2)).fetch();
```

The result being

```
+--------------------+
| timestamp_sub      |
+--------------------+
| 2020-02-01 15:30:45 |
+--------------------+
```

Dialect support

This example using jOOQ:

```
timestampSub(Timestamp.valueOf("2020-02-03 15:30:45"), 2)
```

Translates to the following dialect specific expressions:

Access

```
dateadd('d', -2, #2020/02/03 15:30:45#)
```

ASE, Sybase

```
dateadd(DAY, -2, '2020-02-03 15:30:45.0')
```

Aurora MySQL, MariaDB, MySQL

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL -2 DAY)
```

Aurora Postgres, CockroachDB, Postgres, Redshift, YugabyteDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' + -2 * INTERVAL '1 day')
```

BigQuery

```
timestamp_sub(DATETIME '2020-02-03 15:30:45.0', INTERVAL 2 DAY)
```

ClickHouse

```
(TIMESTAMP '2020-02-03 15:30:45' - 2)
```

Databricks, Exasol, H2, Oracle, Vertica

```
(TIMESTAMP '2020-02-03 15:30:45.0' - 2)
```

DB2, HSQLDB

```
(TIMESTAMP '2020-02-03 15:30:45.0' - (2) day)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
{fn timestampadd(SQL_TSI_DAY, -2, TIMESTAMP('2020-02-03 15:30:45.0')) }
```

DuckDB

```
date_add(TIMESTAMP '2020-02-03 15:30:45.0', INTERVAL 1 DAY * -2)
```

Firebird, Snowflake

```
dateadd(DAY, -2, TIMESTAMP '2020-02-03 15:30:45.0')
```

Hana

```
add_days(TIMESTAMP '2020-02-03 15:30:45.0', -2)
```

Informix

```
(DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION - 2 UNITS DAY)
```

MemSQL

```
date_add({ts '2020-02-03 15:30:45.0'}, INTERVAL -2 DAY)
```

SQLDataWarehouse, SQLServer

```
dateadd(DAY, -2, CAST('2020-02-03 15:30:45.0' AS DATETIME2))
```

SQLite

```
strftime('%Y-%m-%d %H:%M:%f', '2020-02-03 15:30:45.0', (CAST(-2 AS varchar) || ' day'))
```

Teradata

```
TIMESTAMP '2020-02-03 15:30:45.0' - CAST(2 || ' 00:00:00' AS INTERVAL DAY TO SECOND)
```

Trino

```
date_add('day', -2, TIMESTAMP '2020-02-03 15:30:45.0')
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.37. TO_DATE

Parse a string value to a SQL DATE type (represented by java.sql.Date) using a vendor specific formatting pattern.

The pattern is not translated by jOOQ for vendor agnosticity and may need to be adapted depending on the SQL dialect you're using.

```
SELECT TO_DATE('20200203', 'YYYYMMDD');
```

```
create.select(toDate("20200203", "YYYYMMDD")).fetch();
```

The result being

```
+------------+
| to_date    |
+------------+
| 2020-02-03 |
+------------+
```

Dialect support

This example using jOOQ:

```
toDate("20200203", "YYYYMMDD")
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, DB2, Databricks, Exasol, HSQLDB, Oracle, Postgres, Vertica, YugabyteDB

```
to_date('20200203', 'YYYYMMDD')
```

SQLDataWarehouse, SQLServer

```
convert(
  date,
  '20200203',
  112
)
```

ASE, Access, Aurora MySQL, ClickHouse, CockroachDB, Derby, DuckDB, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.38. TO_LOCALDATE

Parse a string value to a SQL DATE type (represented by java.time.LocalDate) using a vendor specific formatting pattern.

The pattern is not translated by jOOQ for vendor agnosticity and may need to be adapted depending on the SQL dialect you're using.

This does the same as TO_DATE except that the client type representation uses JSR-310 types.

```
SELECT TO_DATE('20200203', 'YYYYMMDD');
```

```
create.select(toLocalDate("20200203", "YYYYMMDD")).fetch();
```

The result being

```
+------------+
| to_date    |
+------------+
| 2020-02-03 |
+------------+
```

Dialect support

This example using jOOQ:

```
toLocalDate("20200203", "YYYYMMDD")
```

Translates to the following dialect specific expressions:

Aurora Postgres, DB2, HSQLDB, Oracle, Postgres, Vertica

```
to_date('20200203', 'YYYYMMDD')
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.39. TO_LOCALDATETIME

Parse a string value to a SQL TIMESTAMP type (represented by java.time.LocalDateTime) using a vendor specific formatting pattern.

The pattern is not translated by jOOQ for vendor agnosticity and may need to be adapted depending on the SQL dialect you're using.

This does the same as TO_TIMESTAMP except that the client type representation uses JSR-310 types.

```
SELECT TO_TIMESTAMP('20200203153045', 'YYYYMMDDHH24MISS');
```

```
create.select(toLocalDateTime("20200203153045",
 "YYYYMMDDHH24MISS")).fetch();
```

The result being

```
+---------------------+
| to_timestamp        |
+---------------------+
| 2020-02-03 15:30:45 |
+---------------------+
```

Dialect support

This example using jOOQ:

```
toLocalDateTime("20200203153045", "YYYYMMDDHH24MISS")
```

Translates to the following dialect specific expressions:

Aurora Postgres, DB2, HSQLDB, Oracle, Postgres, Vertica

```
to_timestamp('20200203153045', 'YYYYMMDDHH24MISS')
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.40. TO_TIMESTAMP

Parse a string value to a SQL TIMESTAMP type (represented by java.sql.Timestamp) using a vendor specific formatting pattern.

The pattern is not translated by jOOQ for vendor agnosticity and may need to be adapted depending on the SQL dialect you're using.

```
SELECT TO_TIMESTAMP('20200203153045', 'YYYYMMDDHH24MISS');
```

```
create.select(toTimestamp("20200203153045",
 "YYYYMMDDHH24MISS")).fetch();
```

The result being

```
+--------------------+
| to_timestamp       |
+--------------------+
| 2020-02-03 15:30:45 |
+--------------------+
```

Dialect support

This example using jOOQ:

```
toTimestamp("20200203153045", "YYYYMMDDHH24MISS")
```

Translates to the following dialect specific expressions:

Aurora Postgres, DB2, Databricks, Exasol, HSQLDB, Oracle, Postgres, SQLDataWarehouse, SQLServer, Vertica, YugabyteDB

```
to_timestamp('20200203153045', 'YYYYMMDDHH24MISS')
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, Derby, DuckDB, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLite, Snowflake, Sybase, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.41. TRUNC

Truncate a datetime value to the precision of a certain org.jooq.DatePart, or DatePart.DAY by default.

```
SELECT TRUNC(DATE '2020-02-03', 'YYYY');
```

```
create.select(trunc(Date.valueOf("2020-02-03",
 DatePart.YEAR))).fetch();
```

The result being

```
+------------+
| trunc      |
+------------+
| 2020-01-01 |
+------------+
```

Dialect support

This example using jOOQ:

```
trunc(Date.valueOf("2020-02-03"), DatePart.YEAR)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, Vertica

```
date_trunc('year', DATE '2020-02-03')
```

BigQuery

```
date_trunc(
  DATE '2020-02-03',
  YEAR
)
```

DB2, Oracle

```
trunc(DATE '2020-02-03', 'YYYY')
```

H2

```
PARSEDATETIME(FORMATDATETIME(DATE '2020-02-03', 'yyyy'), 'yyyy')
```

HSQLDB

```
trunc(DATE '2020-02-03', 'YY')
```

Informix

```
trunc(DATETIME(2020-02-03) YEAR TO DAY, 'YEAR')
```

ASE, Access, Aurora MySQL, ClickHouse, Databricks, Derby, DuckDB, Exasol, Firebird, Hana, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.13.42. YEAR

Extract the YEAR value from a datetime value.

The YEAR function is a short version of the EXTRACT, passing a DatePart.YEAR value as an argument.

```
SELECT year(DATE '2020-02-03');
```

```
create.select(year(Date.valueOf("2020-02-03"))).fetch();
```

The result being

```
+------+
| year |
+------+
| 2020 |
+------+
```

Dialect support

This example using jOOQ:

```
year(Date.valueOf("2020-02-03"))
```

Translates to the following dialect specific expressions:

Access

```
datepart('yyyy', #2020/02/03 00:00:00#)
```

ASE, Sybase

```
datepart(yy, '2020-02-03 00:00:00.0')
```

Aurora MySQL, Aurora Postgres, CockroachDB, Databricks, DuckDB, Exasol, Firebird, H2, HSQLDB, Hana, MariaDB, MySQL, Oracle, Postgres, Redshift, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0')
```

BigQuery

```
extract(YEAR FROM DATETIME '2020-02-03 00:00:00.0')
```

ClickHouse

```
extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00')
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2

```
YEAR(TIMESTAMP '2020-02-03 00:00:00.0')
```

Derby

```
YEAR(TIMESTAMP('2020-02-03 00:00:00.0'))
```

Informix

```
YEAR(DATETIME(2020-02-03 00:00:00.0) YEAR TO FRACTION)
```

MemSQL

```
extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})
```

SQLDataWarehouse, SQLServer

```
datepart(yy, CAST('2020-02-03 00:00:00.0' AS DATETIME2))
```

SQLite

```
CAST(strftime('%Y', '2020-02-03 00:00:00.0') AS int)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.14. ARRAY functions

The SQL standard specifies an ARRAY data type, which allows for nesting collections of scalar values and even ROW expressions.

In order to operate on ARRAY data types, a few functions are made available by jOOQ.

3.8.14.1. ARRAY constructor

In order to construct an ad-hoc ARRAY type from within a SQL query, the ARRAY constructor can be used.

```
SELECT ARRAY[1, 2]
```

```
create.select(array(1, 2)).fetch();
```

The result would look like this:

```
+----------+
| array    |
+----------+
| [ 1, 2 ] |
+----------+
```

Dialect support

This example using jOOQ:

```
array(1, 2)
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DuckDB, H2, HSQLDB, Postgres, Trino, YugabyteDB

```
ARRAY[1, 2]
```

ClickHouse, Databricks

```
ARRAY(1, 2)
```

ASE, Access, Aurora MySQL, DB2, Derby, Exasol, Firebird, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.15. JSON functions

jOOQ 3.12 introduced support for the org.jooq.JSON and org.jooq.JSONB types, which are used to wrap string based JSON documents in a type safe way. With these types, there are also a few standard JSON functions that have been added to the API. This section describes scalar functions. Conditions, aggregate functions, table valued functions are supported in jOOQ 3.14+.

Most functions are overloaded with a JSON and JSONB variant to make the distinction explicit for dialects where this matters. For simplicity, and because most dialects do not make a distinction, this manual will only document the JSON version of each function.

3.8.15.1. JSON_ARRAY

The JSON_ARRAY function is used to produce simple JSON arrays from scalar values, without aggregation.

```
SELECT json_array(author.first_name, author.last_name)
FROM author
```

```
create.select(jsonArray(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+----------------------+
| json_array           |
+----------------------+
| ["Paulo", "Coelho"]  |
| ["George", "Orwell"] |
+----------------------+
```

Dialect support

This example using jOOQ:

```
jsonArray(val(1), val(2))
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, YugabyteDB

```
json_build_array(CAST(1 AS int), CAST(2 AS int))
```

BigQuery, DB2, DuckDB, H2, MariaDB, MySQL, Oracle, SQLite

```
json_array(1, 2)
```

ClickHouse

```
toJSONString(tuple(1, 2))
```

CockroachDB

```
json_build_array(CAST(1 AS int4), CAST(2 AS int4))
```

Snowflake

```
array_construct(coalesce(
  to_variant(1),
  parse_json('null')
), coalesce(
  to_variant(2),
  parse_json('null')
))
```

SQLServer

```
json_modify(
  json_modify('[]', 'append $', 1),
  'append $',
  2
)
```

Trino

```
CAST(ARRAY[
  CAST(1 AS json),
  CAST(2 AS json)
] AS json)
```

ASE, Access, Aurora MySQL, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MemSQL, Redshift, SQLDataWarehouse, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.15.2. JSON_OBJECT

The JSON_OBJECT function is used to produce simple JSON objects from scalar values, without aggregation

```
SELECT json_object(
  KEY 'firstName' VALUE author.first_name,
  KEY 'lastName'  VALUE author.last_name
)
FROM author
```

```
create.select(jsonObject(
          jsonEntry("firstName", AUTHOR.FIRST_NAME),
          jsonEntry("lastName" , AUTHOR.LAST_NAME))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+-------------------------------------------+
| json_array                                |
+-------------------------------------------+
| {"firstName":"Paulo","lastName":"Coelho"}  |
| {"firstName":"George","lastName":"Orwell"} |
+-------------------------------------------+
```

Dialect support

This example using jOOQ:

```
jsonObject("firstName", AUTHOR.FIRST_NAME)
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, Postgres, YugabyteDB

```
json_build_object('firstName', AUTHOR.FIRST_NAME)
```

BigQuery, DuckDB, MariaDB, MySQL, SQLite

```
json_object('firstName', AUTHOR.FIRST_NAME)
```

ClickHouse

```
toJSONString(map('firstName', AUTHOR.FIRST_NAME))
```

DB2, H2, Oracle

```
json_object(KEY 'firstName' VALUE AUTHOR.FIRST_NAME)
```

Snowflake

```
object_construct_keep_null('firstName', AUTHOR.FIRST_NAME)
```

SQLServer

```
(
  SELECT (
    SELECT *
    FROM (
      VALUES (AUTHOR.FIRST_NAME)
    ) t (firstName)
    FOR JSON PATH, INCLUDE_NULL_VALUES, WITHOUT_ARRAY_WRAPPER
  )
)
```

Trino

```
CAST(map_from_entries(ARRAY[row(
  'firstName',
  CAST(AUTHOR.FIRST_NAME AS json)
)]) AS json)
```

ASE, Access, Aurora MySQL, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MemSQL, Redshift, SQLDataWarehouse, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16. CONNECT BY functions

The CONNECT BY clause is a vendor specific clause used for hierarchical SQL, in the context of which a few CONNECT BY specific functions, operators, and pseudo fields are available. These functions allow for retrieving recursion state and collect paths within the recursion.

3.8.16.1. CONNECT_BY_ISCYCLE

The CONNECT_BY_ISCYCLE pseudo column allows for detecting if the current row has a child which is also the row's ancestor.

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/'),
  connect_by_iscycle
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, 2),
    (2, 3)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

create.select(
            child,
            parent,
            sysConnectByPath(child, "/"),
            connectByIsCycle())
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(3, 2),
            row(2, 3)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+--------+--------------------+
| child | parent | path   | connect_by_iscycle |
+-------+--------+--------+--------------------+
| 1     |        | /1     | 0                  |
| 2     | 1      | /1/2   | 0                  |
| 3     | 2      | /1/2/3 | 1                  |
+-------+--------+--------+--------------------+
```

Dialect support

This example using jOOQ:

```
connectByIsCycle()
```

Translates to the following dialect specific expressions:

Exasol, Informix, Oracle

```
connect_by_iscycle
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16.2. CONNECT_BY_ISLEAF

The CONNECT_BY_ISLEAF pseudo column allows for detecting if the current row has no children in the hierachy (a "leaf" in a tree).

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/'),
  connect_by_isleaf
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, 2),
    (4, 1)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

ctx.select(
            child,
            parent,
            sysConnectByPath(child, "/"),
            connectByIsLeaf())
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(3, 2),
            row(4, 1)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+---------------------+-------------------+
| child | parent | sys_connect_by_path | connect_by_isleaf |
+-------+--------+---------------------+-------------------+
|     1 | {null} | /1                  | false             |
|     2 |      1 | /1/2                | false             |
|     3 |      2 | /1/2/3              | true              |
|     4 |      1 | /1/4                | true              |
+-------+--------+---------------------+-------------------+
```

Dialect support

This example using jOOQ:

```
connectByIsLeaf()
```

Translates to the following dialect specific expressions:

Exasol, Informix, Oracle

```
connect_by_isleaf
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16.3. CONNECT_BY_ROOT

The CONNECT_BY_ROOT operator allows for accessing a row's hierarchy root row and evaluate an expression on that root row.

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/'),
  connectByRoot(child)
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, null),
    (4, 3)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

create.select(
            child,
            parent,
            sysConnectByPath(child, "/"),
            connectByRoot(child))
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(val(3), val(null, INTEGER)),
            row(4, 3)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+---------------------+-----------------+
| child | parent | sys_connect_by_path | connect_by_root |
+-------+--------+---------------------+-----------------+
|     1 | {null} | /1                  |               1 |
|     2 |      1 | /1/2                |               1 |
|     3 | {null} | /3                  |               3 |
|     4 |      3 | /3/4                |               3 |
+-------+--------+---------------------+-----------------+
```

Dialect support

This example using jOOQ:

```
connectByRoot(AUTHOR.ID)
```

Translates to the following dialect specific expressions:

Exasol, Informix, Oracle, Snowflake

```
connect_by_root AUTHOR.ID
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL,

MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16.4. LEVEL

The LEVEL pseudo column allows for accessing the hierarchy level of the current row, or in other words, the depth of the current row in the hierarchy's tree.

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/'),
  level
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, 2),
    (4, 1)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

ctx.select(
            child,
            parent,
            sysConnectByPath(child, "/"),
            level())
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(3, 2),
            row(4, 1)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+---------------------+-------+
| child | parent | sys_connect_by_path | level |
+-------+--------+---------------------+-------+
|     1 | {null} | /1                  |     1 |
|     2 |      1 | /1/2                |     2 |
|     3 |      2 | /1/2/3              |     3 |
|     4 |      1 | /1/4                |     2 |
+-------+--------+---------------------+-------+
```

Dialect support

This example using jOOQ:

```
level()
```

Translates to the following dialect specific expressions:

Aurora Postgres, CockroachDB, DB2, Exasol, Firebird, H2, HSQLDB, Informix, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, YugabyteDB

```
level
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, Databricks, Derby, DuckDB, Hana, MemSQL, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16.5. PRIOR

The PRIOR operator allows for specifying an expression from the parent (the "prior") row in the hierarchy. This can be used mainly in the CONNECT BY clause itself.

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/')
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, 2)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

ctx.select(
            child,
            parent,
            sysConnectByPath(child, "/"))
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(3, 2)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+---------------------+
| child | parent | sys_connect_by_path |
+-------+--------+---------------------+
|     1 | {null} | /1                  |
|     2 |      1 | /1/2                |
|     3 |      2 | /1/2/3              |
+-------+--------+---------------------+
```

Dialect support

This example using jOOQ:

```
prior(AUTHOR.ID)
```

Translates to the following dialect specific expressions:

Exasol, Informix, Oracle, Redshift, Snowflake

```
PRIOR AUTHOR.ID
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.16.6. SYS_CONNECT_BY_PATH

The SYS_CONNECT_BY_PATH function allows for recursively generating a string for each element in the path leading to the current row in the hierarchy.

```
SELECT
  child,
  parent,
  sys_connect_by_path(child, '/')
FROM (
  VALUES
    (1, null),
    (2, 1),
    (3, 2)
) AS t (child, parent)
START WITH
  parent IS NULL
CONNECT BY NOCYCLE
  PRIOR child = parent;
```

```
Field<Integer> child = field("child", INTEGER);
Field<Integer> parent = field("parent", INTEGER);

ctx.select(
            child,
            parent,
            sysConnectByPath(child, "/"))
      .from(values(
            row(val(1), val(null, INTEGER)),
            row(2, 1),
            row(3, 2)).as(table("t"), child, parent))
      .startWith(parent.isNull())
      .connectByNoCycle(prior(child).eq(parent))
      .fetch();
```

The result being, for example

```
+-------+--------+---------------------+
| child | parent | sys_connect_by_path |
+-------+--------+---------------------+
|     1 | {null} | /1                  |
|     2 |      1 | /1/2                |
|     3 |      2 | /1/2/3              |
+-------+--------+---------------------+
```

Dialect support

This example using jOOQ:

```
sysConnectByPath(AUTHOR.ID, "/")
```

Translates to the following dialect specific expressions:

Exasol, Informix, Oracle, Snowflake

```
sys_connect_by_path(AUTHOR.ID, '/')
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.17. System functions

Some system functions are supported by jOOQ.

3.8.17.1. CURRENT_SCHEMA

The CURRENT_SCHEMA() function produces the dialect dependent expression to produce the current default schema for the JDBC connection.

```
SELECT current_schema;
```

```
create.select(currentSchema()).fetch();
```

The result being, for example

```
+----------------+
| current_schema |
+----------------+
| public         |
+----------------+
```

Dialect support

This example using jOOQ:

```
currentSchema()
```

Translates to the following dialect specific expressions:

Aurora MySQL, MariaDB, MemSQL, MySQL

```
database()
```

Aurora Postgres, CockroachDB, DB2, HSQLDB, Postgres, YugabyteDB

```
current_schema
```

ClickHouse

```
currentDatabase()
```

Databricks, Snowflake, Vertica

```
current_schema()
```

Derby

```
CURRENT SCHEMA
```

Firebird, SQLite

```
''
```

H2

```
schema()
```

Oracle

```
user
```

SQLDataWarehouse, SQLServer

```
schema_name()
```

Teradata

```
database
```

ASE, Access, BigQuery, DuckDB, Exasol, Hana, Informix, Redshift, Sybase, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.17.2. CURRENT_USER

The CURRENT_USER() function produces the dialect dependent expression to produce the currently connected user for the JDBC connection.

```
SELECT current_user;
```

```
create.select(currentUser()).fetch();
```

The result being, for example

```
+--------------+
| current_user |
+--------------+
| sa           |
+--------------+
```

Dialect support

This example using jOOQ:

```
currentUser()
```

Translates to the following dialect specific expressions:

ASE, Informix, Oracle

```
user
```

Aurora MySQL, Databricks, H2, MariaDB, MemSQL, MySQL, Snowflake

```
current_user()
```

Aurora Postgres, CockroachDB, DB2, Derby, Firebird, HSQLDB, Hana, Postgres, SQLDataWarehouse, SQLServer, Sybase, Teradata, YugabyteDB

```
current_user
```

ClickHouse

```
currentUser()
```

SQLite

```
''
```

Access, BigQuery, DuckDB, Exasol, Redshift, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18. Aggregate functions

Aggregate functions work like Java java.util.stream.Collector, as they aggregate data from a group of data into a new data structure.

This section will first explain concepts common to many aggregate functions, and then proceed to explaining individual aggregate functions supported by jOOQ.

3.8.18.1. Grouping

Aggregate functions aggregate data from groups of data into individual values. There are three main ways of forming such groups:

- A GROUP BY clause is used to define the groups for which data is aggregated
- No GROUP BY clause is defined, which means that all data from a SELECT statement (or subquery) is aggregated into a single row
- All aggregate functions can be used as window functions, in case of which they will aggregate the data of the specified window

Aggregation with GROUP BY

In the presence of GROUP BY, a SELECT statement transforms the output of its FROM clause into a new "virtual" set of tuples containing:

- The column expressions of the GROUP BY clause. In the overall data set, the values of these column expressions is unique.
- A set of data corresponding to each row produced by the GROUP BY clause. This data set can be aggregated per group using aggregate functions.

Using GROUP BY means that a new set of rules need to be observed in the rest of the query:

- Clauses that logically precede GROUP BY are not affected. These include, for example, FROM and WHERE
- All other clauses (e.g. HAVING, WINDOW, SELECT, or ORDER BY) may now only reference expressions built from the expressions in the GROUP BY clause, or aggregations on any other expression

An example:

```
SELECT AUTHOR_ID, count(*)
FROM BOOK
GROUP BY AUTHOR_ID;
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID).fetch();
```

Producing:

```
+-----------+-------+
| AUTHOR_ID | count |
+-----------+-------+
|         1 |     2 |
|         2 |     2 |
+-----------+-------+
```

Per the rules imposed by GROUP BY, it would not be possible, for example, to project the BOOK.TITLE column, because it is not defined per author. An author has written many books, so we don't know what a BOOK.TITLE is supposed to mean. Only an aggregation, such as LISTAGG or ARRAY_AGG can reference BOOK.TITLE as an argument.

© 2009 - 2025 by Data Geekery™ GmbH.

Aggregation without GROUP BY

In the absence of GROUP BY, a SELECT statement that contains at least one aggregate function in any of its clauses (e.g. HAVING, WINDOW, SELECT, or ORDER BY) will proceed to aggregating the entire data into a single row. There is an implied "empty grouping", i.e. a grouping that has no GROUP BY columns. These two are the same things:

```
SELECT count(*) FROM BOOK;
SELECT count(*) FROM BOOK GROUP BY ();
```

See also GROUPING SETS for more details about this empty GROUP BY syntax.

For example, using our sample database, which has 4 books with IDs 1-4, we can write:

```
SELECT count(*), sum(ID)
FROM BOOK
```

```
create.select(count(), sum(BOOK.ID))
      .from(BOOK).fetch();
```

Producing:

```
+----------+---------+
| count(*) | sum(ID) |
+----------+---------+
|        4 |      10 |
+----------+---------+
```

No other columns from the tables in the FROM clause may be projected by the SELECT clause, because they would not be defined for this single group. For example, no specific BOOK.TITLE is defined for the aggregated value of all books. Only an aggregation, such as LISTAGG or ARRAY_AGG can reference BOOK.TITLE as an argument.

However, any expression whose components do not depend on content of the group is allowed. For example, it is possible to combine aggregate functions and constant expressions like this:

```
SELECT count(*) + sum(ID) + 1
FROM BOOK
```

```
create.select(count().plus(sum(BOOK.ID)).plus(1))
      .from(BOOK).fetch();
```

Producing:

```
+------+
| plus |
+------+
|   15 |
+------+
```

3.8.18.2. Distinctness

A useful thing to do when aggregating data is to remove duplicate input first, prior to aggregation. A few aggregate functions support a DISTINCT keyword for that purpose. For example, we can query

```
SELECT
  count(AUTHOR_ID),
  count(DISTINCT AUTHOR_ID),
  group_concat(AUTHOR_ID),
  group_concat(DISTINCT AUTHOR_ID)
FROM BOOK
```

```
create.select(
         count(BOOK.AUTHOR_ID),
         countDistinct(BOOK.AUTHOR_ID),
         groupConcat(BOOK.AUTHOR_ID),
         groupConcatDistinct(BOOK.AUTHOR_ID))
      .from(BOOK).fetch();
```

Producing:

```
+-------+----------------+--------------+-----------------------+
| count | count_distinct | group_concat | group_concat_distinct |
+-------+----------------+--------------+-----------------------+
|     4 |              2 | 1, 1, 2, 2   | 1, 2                  |
+-------+----------------+--------------+-----------------------+
```

If DISTINCT is available through the jOOQ API, it is always appended to the aggregate function name, such as count() and countDistinct(). sum() and sumDistinct(), etc.

3.8.18.3. Filtering

The SQL standard specifies an optional FILTER clause, that can be appended to all aggregate functions including aggregated window functions. This is very useful, for example, to implement "pivot" tables, such as the following:

```
SELECT
  count(*),
  count(*) FILTER (WHERE TITLE LIKE 'A%'),
  count(*) FILTER (WHERE TITLE LIKE '%A%')
FROM BOOK
```

```
create.select(
         count(),
         count().filterWhere(BOOK.TITLE.like("A%")),
         count().filterWhere(BOOK.TITLE.like("%A%")))
      .from(BOOK)
```

Producing:

```
+-------+-------+-------+
| count | count | count |
+-------+-------+-------+
|     4 |     1 |     2 |
+-------+-------+-------+
```

Or, with GROUP BY:

```
SELECT
  AUTHOR_ID,
  count(*),
  count(*) FILTER (WHERE TITLE LIKE 'A%'),
  count(*) FILTER (WHERE TITLE LIKE '%A%')
FROM BOOK
GROUP BY AUTHOR_ID
```

```
create.select(
         BOOK.AUTHOR_ID,
         count(),
         count().filterWhere(BOOK.TITLE.like("A%")),
         count().filterWhere(BOOK.TITLE.like("%A%")))
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
```

Producing:

```
+-----------+-------+-------+-------+
| AUTHOR_ID | count | count | count |
+-----------+-------+-------+-------+
|         1 |     2 |     1 |     1 |
|         2 |     2 |     0 |     1 |
+-----------+-------+-------+-------+
```

It is usually a good idea to calculate multiple aggregate functions in a single query, if this is possible, and FILTER helps here.

Only a few dialects implement native support for the FILTER clause. In all other databases, jOOQ emulates the clause using a CASE expression. Aggregate functions exclude NULL values from aggregation.

Dialect support

This example using jOOQ:

```
count().filterWhere(BOOK.TITLE.like("A%"))
```

Translates to the following dialect specific expressions:

Access

```
count(SWITCH(BOOK.TITLE LIKE 'A%', 1))
```

ASE, Aurora MySQL, DB2, Derby, Exasol, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, Sybase, Teradata, Vertica

```
count(CASE
  WHEN BOOK.TITLE LIKE 'A%' THEN 1
END)
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Firebird, H2, HSQLDB, Postgres, SQLite, Trino, YugabyteDB

```
count(*) FILTER (WHERE BOOK.TITLE LIKE 'A%')
```

BigQuery

```
countif((BOOK.TITLE LIKE 'A%'))
```

ClickHouse

```
count() FILTER (WHERE BOOK.TITLE LIKE 'A%')
```

Snowflake

```
count_if((BOOK.TITLE LIKE 'A%'))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.4. Ordering

Some aggregate functions allow for ordering their inputs to produce an ordered output. These aggregate functions allow for specifying an optional ORDER BY clause in their argument list. This is not to be confused with the WITHIN GROUP (ORDER BY ..) clause, which is required for ordering inputs to produce a single, unordered output.

This makes a lot of sense with aggregations that produce the aggregated values in a nested or formatted data structure, such as, for example:

- ARRAY_AGG, which aggregates data into an array.
- COLLECT, which aggregates data into a nested table (Oracle).
- LISTAGG, which aggregates data into a string. The standard LISTAGG function, unfortunately, inconsistently uses the WITHIN GROUP syntax. MySQL's GROUP_CONCAT is more consistent with the rest.

An example using ARRAY_AGG could look like this:

```
SELECT
  array_agg(ID),
  array_agg(ID ORDER BY ID DESC)
FROM BOOK
```

```
create.select(
         arrayAgg(BOOK.ID),
         arrayAgg(BOOK.ID).orderBy(BOOK.ID.desc()))
      .from(BOOK)
```

Producing:

```
+--------------+--------------+
| array_agg    | array_agg    |
+--------------+--------------+
| [1, 3, 4, 2] | [4, 3, 2, 1] |
+--------------+--------------+
```

(!) Notice that in the absence of an explicit ORDER BY clause, as always, the ordering is non deterministic.

3.8.18.5. Ordering WITHIN GROUP

Some aggregate functions allow for ordering their inputs to produce an ordered output. These aggregate functions allow for specifying a mandatory WITHIN GROUP (ORDER BY ..) clause after the function. This is not to be confused with the aggregate ORDER BY clause, which allows for optionally ordering inputs to produce ordered output

Standard SQL talks about "ordered set aggregate functions" which come in three flavours

- Hypothetical set functions: Functions that check for the position of a hypothetical value inside of an ordered set. These include RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST.
- Inverse distribution functions: Functions calculating a percentile over an ordered set, including PERCENTILE_CONT, PERCENTILE_DISC, or MODE.
- LISTAGG, which is inconsistently using the WITHIN GROUP syntax, as it is used to order the output of the function, and isn't mandatory in all dialects.

An example for the PERCENTILE_CONT inverse distribution function is this:

```
SELECT
  percentile_cont(0.5) WITHIN GROUP (ORDER BY ID)
FROM BOOK
```

```
create.select(
         percentileCont(0.5).withinGroupOrderBy(BOOK.ID))
      .from(BOOK)
```

Producing the median BOOK.ID value:

```
+-----------------+
| percentile_cont |
+-----------------+
|             2.5 |
+-----------------+
```

3.8.18.6. Keeping

Oracle allows for restricting other aggregate functions using the KEEP() clause, which is supported by jOOQ. In Oracle, some aggregate functions (e.g. MIN, MAX, SUM, AVG, COUNT, VARIANCE, or STDDEV) can be restricted by this clause, hence org.jooq.AggregateFunction also allows for specifying it. Here is an example using this clause:

```
SUM(BOOK.AMOUNT_SOLD)
  KEEP(DENSE_RANK FIRST ORDER BY BOOK.AUTHOR_ID)
```

```
sum(BOOK.AMOUNT_SOLD)
  .keepDenseRankFirstOrderBy(BOOK.AUTHOR_ID)
```

3.8.18.7. ARRAY_AGG

The ARRAY_AGG aggregate function aggregates grouped values into an array. It supports being used with an ORDER BY clause.

```
SELECT
  array_agg(ID)
  array_agg(ID ORDER BY ID DESC)
FROM BOOK
```

```
create.select(
         arrayAgg(BOOK.ID),
         arrayAgg(BOOK.ID).orderBy(BOOK.ID.desc()))
      .from(BOOK)
```

Producing:

```
+--------------+--------------+
| array_agg    | array_agg    |
+--------------+--------------+
| [1, 3, 4, 2] | [4, 3, 2, 1] |
+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
arrayAgg(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, Databricks, DuckDB, H2, HSQLDB, Postgres, Trino, YugabyteDB

```
array_agg(BOOK.ID)
```

ClickHouse

```
groupArray(BOOK.ID)
```

ASE, Access, Aurora MySQL, DB2, Derby, Exasol, Firebird, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.8. AVG

The AVG() aggregate function calculates the average value of all input values

```
SELECT avg(ID)
FROM BOOK
```

```
create.select(avg(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+-----+
| avg |
+-----+
| 2.5 |
+-----+
```

Dialect support

This example using jOOQ:

```
avg(BOOK.ID)
```

Translates to the following dialect specific expressions:

All dialects

```
avg(BOOK.ID)
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.18.9. BOOL_AND

The BOOL_AND() aggregate function calculates the boolean conjunction of all the boolean values in the aggregated group. In other words, this is:

- TRUE if the argument is TRUE for every row in the group.
- FALSE if at the argument is FALSE for at least one row in the group.

As with most aggregate functions, NULL values are not aggregated, so three valued logic does not apply here.

```
SELECT
  bool_and(ID < 4),
  bool_and(ID < 5)
FROM BOOK
```

```
create.select(
         boolAnd(BOOK.ID.lt(4)),
         boolAnd(BOOK.ID.lt(5)))
      .from(BOOK)
```

Producing:

```
+----------+----------+
| bool_and | bool_and |
+----------+----------+
| false    | true     |
+----------+----------+
```

Dialect support

This example using jOOQ:

```
boolAnd(BOOK.ID.lt(4))
```

Translates to the following dialect specific expressions:

Access

```
(min(
  SWITCH(BOOK.ID < 4, 1, TRUE, 0)
) = 1)
```

ASE, DB2, Firebird, SQLDataWarehouse, SQLServer, Sybase, Teradata

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN 1
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN 0
END
```

Aurora MySQL, ClickHouse, Derby, H2, HSQLDB, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite

```
(min(
  CASE
    WHEN BOOK.ID < 4 THEN 1
    ELSE 0
  END
) = 1)
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Postgres, Trino, Vertica, YugabyteDB

```
bool_and((BOOK.ID < 4))
```

BigQuery

```
logical_and((BOOK.ID < 4))
```

Exasol

```
every((BOOK.ID < 4))
```

Hana

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN TRUE
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN FALSE
END
```

Informix

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN CAST('t' AS boolean)
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN CAST('f' AS boolean)
END
```

Snowflake

```
booland_agg((BOOK.ID < 4))
```

*(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. **Translate your own SQL on our website***

3.8.18.10. BOOL_OR

The BOOL_OR() aggregate function calculates the boolean disjunction of all the boolean values in the aggregated group. In other words, this is:

- FALSE if the argument is FALSE for every row in the group.
- TRUE if at the argument is TRUE for at least one row in the group.

As with most aggregate functions, NULL values are not aggregated, so three valued logic does not apply here.

```
SELECT
  bool_or(ID >= 4),
  bool_or(ID >= 5)
FROM BOOK
```

```
create.select(
         boolOr(BOOK.ID.ge(4)),
         boolOr(BOOK.ID.ge(5)))
      .from(BOOK)
```

Producing:

```
+---------+---------+
| bool_or | bool_or |
+---------+---------+
| true    | false   |
+---------+---------+
```

Dialect support

This example using jOOQ:

```
boolOr(BOOK.ID.ge(4))
```

Translates to the following dialect specific expressions:

Access

```
(max(
  SWITCH(BOOK.ID >= 4, 1, TRUE, 0)
) = 1)
```

ASE, DB2, Firebird, SQLDataWarehouse, SQLServer, Sybase, Teradata

```
CASE
  WHEN max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN 1
  WHEN NOT (max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN 0
END
```

Aurora MySQL, ClickHouse, Derby, H2, HSQLDB, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite

```
(max(
  CASE
    WHEN BOOK.ID >= 4 THEN 1
    ELSE 0
  END
) = 1)
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Postgres, Trino, Vertica, YugabyteDB

```
bool_or((BOOK.ID >= 4))
```

BigQuery

```
logical_or((BOOK.ID >= 4))
```

Exasol

```
any((BOOK.ID >= 4))
```

Hana

```
CASE
  WHEN max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN TRUE
  WHEN NOT (max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN FALSE
END
```

Informix

```
CASE
  WHEN max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN CAST('t' AS boolean)
  WHEN NOT (max(
    CASE
      WHEN BOOK.ID >= 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN CAST('f' AS boolean)
END
```

Snowflake

```
boolor_agg((BOOK.ID >= 4))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.11. COLLECT

The COLLECT() aggregate function is Oracle's vendor specific version of the standard SQL ARRAY_AGG function. It produces a structurally typed array, which is implemented behind the scenes as a nominally typed, system-generated array. It supports being used with an ORDER BY clause.

The following example is using an auxiliary data type and casting the COLLECT() result to that type.

```
CREATE TYPE NUMBERS AS TABLE OF NUMBER(10);
SELECT CAST(collect(ID ORDER BY ID) AS NUMBERS);
FROM BOOK;
```

```
create.select(
         collect(BOOK.ID, NumbersRecord.class).orderBy(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+--------------+
| collect      |
+--------------+
| [1, 2, 3, 4] |
+--------------+
```

3.8.18.12. COUNT

The COUNT() aggregate function comes in two flavours:

- COUNT(*): This version counts the number of tuples in a group, regardless of any contents, including NULL values.
- COUNT(expression): This version counts the number of non-NULL expression evaluations per group.

The second version can be used to emulate the FILTER clause as the argument expression effectively filters out NULL values. Alternatively, in the case of a LEFT JOIN, the outer joined rows can be counted using an expression on the primary key, because COUNT(*) always produces at least one row.

```
SELECT
  AUTHOR.ID,
  count(*),
  count(BOOK.ID)
FROM AUTHOR
LEFT JOIN BOOK
ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select(
         AUTHOR.ID,
         count(),
         count(BOOK.ID))
      .from(AUTHOR)
      .leftJoin(BOOK)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
```

Producing (assuming the presence of an author with ID = 3, but without books):

```
+----+----------+----------------+
| ID | count(*) | count(BOOK.ID) |
+----+----------+----------------+
|  1 |        2 |              2 |
|  2 |        2 |              2 |
|  3 |        1 |              0 |
+----+----------+----------------+
```

Dialect support

This example using jOOQ:

```
count(BOOK.ID)
```

Translates to the following dialect specific expressions:

All dialects

```
count(BOOK.ID)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.13. CUME_DIST

The CUME_DIST() hypothetical set function calculates the cumulative distribution of the hypothetical value, i.e. the relative rank from 1/N to 1 (PERCENT_RANK produces values from 0 to 1)

```
SELECT
  cume_dist(0) WITHIN GROUP (ORDER BY ID),
  cume_dist(2) WITHIN GROUP (ORDER BY ID),
  cume_dist(4) WITHIN GROUP (ORDER BY ID)
FROM BOOK
```

```
create.select(
         cumeDist(val(0)).withinGroupOrderBy(BOOK.ID),
         cumeDist(val(2)).withinGroupOrderBy(BOOK.ID),
         cumeDist(val(4)).withinGroupOrderBy(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+--------------+--------------+--------------+
| cume_dist(0) | cume_dist(2) | cume_dist(4) |
+--------------+--------------+--------------+
|          0.2 |          0.6 |          1.0 |
+--------------+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
cumeDist(val(0)).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, H2, Oracle, Postgres, YugabyteDB

```
cume_dist(0) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.14. DENSE_RANK

The DENSE_RANK() hypothetical set function calculates the rank without gaps of the hypothetical value, i.e. dense ranks will be 1, 1, 1, 2, 3, 3, 4 (RANK produces values with gaps, e.g. 1, 1, 1, 4, 5, 5, 7)

```
SELECT
  dense_rank(0) WITHIN GROUP (ORDER BY AUTHOR_ID),
  dense_rank(1) WITHIN GROUP (ORDER BY AUTHOR_ID),
  dense_rank(2) WITHIN GROUP (ORDER BY AUTHOR_ID)
FROM BOOK
```

```
create.select(
         denseRank(val(0)).withinGroupOrderBy(BOOK.AUTHOR_ID),
         denseRank(val(2)).withinGroupOrderBy(BOOK.AUTHOR_ID),
         denseRank(val(4)).withinGroupOrderBy(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+---------------+---------------+---------------+
| dense_rank(0) | dense_rank(1) | dense_rank(2) |
+---------------+---------------+---------------+
|             1 |             1 |             2 |
+---------------+---------------+---------------+
```

Dialect support

This example using jOOQ:

```
denseRank(val(0)).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, H2, Oracle, Postgres, YugabyteDB

```
dense_rank(0) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.15. EVERY

The EVERY() aggregate function is the standard SQL version of the BOOL_AND function.

```
SELECT
  every(ID < 4),
  every(ID < 5)
FROM BOOK
```

```
create.select(
         every(BOOK.ID.lt(4)),
         every(BOOK.ID.lt(5)))
      .from(BOOK)
```

Producing:

```
+---------------+---------------+
| every(ID < 4) | every(ID < 5) |
+---------------+---------------+
| false         | true          |
+---------------+---------------+
```

Dialect support

This example using jOOQ:

```
every(BOOK.ID.lt(4))
```

Translates to the following dialect specific expressions:

Access

```
(min(
  SWITCH(BOOK.ID < 4, 1, TRUE, 0)
) = 1)
```

ASE, DB2, Firebird, SQLDataWarehouse, SQLServer, Sybase, Teradata

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN 1
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN 0
END
```

Aurora MySQL, ClickHouse, Derby, H2, HSQLDB, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLite

```
(min(
  CASE
    WHEN BOOK.ID < 4 THEN 1
    ELSE 0
  END
) = 1)
```

Aurora Postgres, CockroachDB, Databricks, DuckDB, Postgres, Trino, Vertica, YugabyteDB

```
bool_and((BOOK.ID < 4))
```

BigQuery

```
logical_and((BOOK.ID < 4))
```

Exasol

```
every((BOOK.ID < 4))
```

Hana

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN TRUE
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN FALSE
END
```

Informix

```
CASE
  WHEN min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1 THEN CAST('t' AS boolean)
  WHEN NOT (min(
    CASE
      WHEN BOOK.ID < 4 THEN 1
      ELSE 0
    END
  ) = 1) THEN CAST('f' AS boolean)
END
```

Snowflake

```
booland_agg((BOOK.ID < 4))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.16. GROUP_CONCAT

The GROUP_CONCAT() aggregate function is the MySQL version of the standard SQL LISTAGG function, to concatenate aggregate data into a string. It supports being used with an ORDER BY clause, which uses the expected syntax, unlike LISTAGG(), which uses the WITHIN GROUP syntax.

```
SELECT
  group_concat(ID),
  group_concat(ID ORDER BY ID),
  group_concat(ID SEPARATOR '; '),
  group_concat(ID ORDER BY ID SEPARATOR '; '),
FROM BOOK
```

```
create.select(
         groupConcat(BOOK.ID),
         groupConcat(BOOK.ID).orderBy(BOOK.ID),
         groupConcat(BOOK.ID).separator("; "),
         groupConcat(BOOK.ID).orderBy(BOOK.ID).separator("; "))
      .from(BOOK).fetch();
```

Producing:

```
+--------------+--------------+--------------+--------------+
| group_concat | group_concat | group_concat | group_concat |
+--------------+--------------+--------------+--------------+
| 1, 3, 4, 2   | 1, 2, 3, 4   | 1; 3; 4; 2   | 1; 2; 3; 4   |
+--------------+--------------+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
groupConcat(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora MySQL, H2, HSQLDB, MariaDB, MemSQL, MySQL

```
group_concat(BOOK.ID SEPARATOR ',')
```

Aurora Postgres, DuckDB, Hana, Postgres

```
string_agg(CAST(BOOK.ID AS varchar), ',')
```

BigQuery, CockroachDB

```
string_agg(CAST(BOOK.ID AS string), ',')
```

DB2, Exasol, Redshift

```
listagg(BOOK.ID, ',')
```

Oracle

```
listagg(BOOK.ID, ',') WITHIN GROUP (ORDER BY NULL)
```

SQLite

```
group_concat(BOOK.ID, ',')
```

© 2009 - 2025 by Data Geekery™ GmbH.

SQLServer

```
string_agg(CAST(BOOK.ID AS varchar(max)), ',')
```

Sybase

```
list(CAST(BOOK.ID AS varchar), ',')
```

Teradata

```
substring(xmlserialize(CONTENT xmlagg((',' || CAST(BOOK.ID AS varchar(32000)))) AS varchar(32000)) FROM 2)
```

Trino

```
listagg(CAST(BOOK.ID AS varchar), ',') WITHIN GROUP (ORDER BY NULL)
```

ASE, Access, ClickHouse, Databricks, Derby, Firebird, Informix, SQLDataWarehouse, Snowflake, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.17. LISTAGG

The LISTAGG() aggregate function aggregates data into a string. It uses the WITHIN GROUP syntax.

```
SELECT
  listagg(ID) WITHIN GROUP (ORDER BY ID),
  listagg(ID, '; ') WITHIN GROUP (ORDER BY ID),
FROM BOOK
```

```
create.select(
         listAgg(BOOK.ID).withinGroupOrderBy(BOOK.ID),
         listAgg(BOOK.ID, "; ").withinGroupOrderBy(BOOK.ID))
      .from(BOOK).fetch();
```

Producing:

```
+------------+--------------+
| listagg    | listagg      |
+------------+--------------+
| 1, 2, 3, 4 | 1; 2; 3; 4   |
+------------+--------------+
```

Dialect support

This example using jOOQ:

```
listAgg(BOOK.AUTHOR_ID, ",").withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora MySQL, H2, HSQLDB, MariaDB, MySQL

```
group_concat(BOOK.AUTHOR_ID ORDER BY BOOK.ID SEPARATOR ',')
```

Aurora Postgres, DuckDB, Hana, Postgres

```
string_agg(CAST(BOOK.AUTHOR_ID AS varchar), ',' ORDER BY BOOK.ID)
```

BigQuery, CockroachDB

```
string_agg(CAST(BOOK.AUTHOR_ID AS string), ',' ORDER BY BOOK.ID)
```

DB2, Exasol, Oracle, Redshift

```
listagg(BOOK.AUTHOR_ID, ',') WITHIN GROUP (ORDER BY BOOK.ID)
```

SQLServer

```
string_agg(CAST(BOOK.AUTHOR_ID AS varchar(max)), ',') WITHIN GROUP (ORDER BY BOOK.ID)
```

Sybase

```
list(CAST(BOOK.AUTHOR_ID AS varchar), ',' ORDER BY BOOK.ID)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
substring(xmlserialize(CONTENT xmlagg((',' || CAST(BOOK.AUTHOR_ID AS varchar(32000))) ORDER BY BOOK.ID) AS varchar(32000)) FROM 2)
```

Trino

```
listagg(CAST(BOOK.AUTHOR_ID AS varchar), ',') WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, ClickHouse, Databricks, Derby, Firebird, Informix, MemSQL, SQLDataWarehouse, SQLite, Snowflake, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.18. MAX

The MAX() aggregate function calculates the maximum value of all input values

```
SELECT max(ID)
FROM BOOK
```

```
create.select(max(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+-----+
| max |
+-----+
|   4 |
+-----+
```

Dialect support

This example using jOOQ:

```
max(BOOK.ID)
```

Translates to the following dialect specific expressions:

All dialects

```
max(BOOK.ID)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.19. MEDIAN

The MEDIAN() aggregate function calculates the median value of all input values. MEDIAN(x) is equivalent to standard SQL PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY x), see PERCENTILE_CONT.

```
SELECT median(ID)
FROM BOOK
```

```
create.select(median(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+--------+
| median |
+--------+
|    2.5 |
+--------+
```

Dialect support

This example using jOOQ:

```
median(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, Teradata, YugabyteDB

```
percentile_cont(0.5) WITHIN GROUP (ORDER BY BOOK.ID)
```

BigQuery

```
percentile_cont(BOOK.ID, 0.5)
```

ClickHouse, DB2, Databricks, DuckDB, Exasol, H2, HSQLDB, Hana, MariaDB, Oracle, Redshift, Snowflake, Sybase

```
median(BOOK.ID)
```

ASE, Access, Aurora MySQL, CockroachDB, Derby, Firebird, Informix, MemSQL, MySQL, SQLDataWarehouse, SQLServer, SQLite, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.20. MIN

The MIN() aggregate function calculates the minimum value of all input values

```
SELECT min(ID)
FROM BOOK
```

```
create.select(min(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+-----+
| min |
+-----+
|   1 |
+-----+
```

Dialect support

This example using jOOQ:

```
min(BOOK.ID)
```

Translates to the following dialect specific expressions:

All dialects

```
min(BOOK.ID)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.21. MODE

The MODE() aggregate function calculates the statistical mode of all input values, i.e. the value that appears most in the data set. There can be several modes, in case of which the first one, given an ordering, will be chosen.

```
SELECT mode() WITHIN GROUP (ORDER BY BOOK.AUTHOR_ID)
FROM BOOK
```

```
create.select(mode().withinGroupOrderBy(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+------+
| mode |
+------+
|    1 |
+------+
```

Dialect support

This example using jOOQ:

```
mode().withinGroupOrderBy(BOOK.AUTHOR_ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, DuckDB, H2, Postgres, YugabyteDB

```
mode() WITHIN GROUP (ORDER BY BOOK.AUTHOR_ID)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.22. PERCENT_RANK

The PERCENT_RANK() hypothetical set function calculates the percent rank of the hypothetical value, i.e. the relative rank from 0 to 1 (CUME_DIST produces values from 1/N to 1)

```
SELECT
  percent_rank(0) WITHIN GROUP (ORDER BY ID),
  percent_rank(2) WITHIN GROUP (ORDER BY ID),
  percent_rank(4) WITHIN GROUP (ORDER BY ID)
FROM BOOK
```

```
create.select(
         percentRank(val(0)).withinGroupOrderBy(BOOK.ID),
         percentRank(val(2)).withinGroupOrderBy(BOOK.ID),
         percentRank(val(4)).withinGroupOrderBy(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+----------------+----------------+----------------+
| percent_rank(0) | percent_rank(2) | percent_rank(4) |
+----------------+----------------+----------------+
|            0.0 |           0.25 |           0.75 |
+----------------+----------------+----------------+
```

Dialect support

This example using jOOQ:

```
percentRank(val(0)).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, H2, Oracle, Postgres, YugabyteDB

```
percent_rank(0) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.23. PERCENTILE_CONT

The PERCENTILE_CONT() aggregate function is an ordered set function that calculates a given continuous percentile of all input values. A special kind of percentile is the MEDIAN, corresponding to the 50% percentile.

```
SELECT
  percentile_cont(0.00) WITHIN GROUP (ORDER BY ID),
  percentile_cont(0.25) WITHIN GROUP (ORDER BY ID),
  percentile_cont(0.50) WITHIN GROUP (ORDER BY ID),
  percentile_cont(0.75) WITHIN GROUP (ORDER BY ID),
  percentile_cont(1.00) WITHIN GROUP (ORDER BY ID)
FROM BOOK
```

```
create.select(
         percentileCont(0.00).withinGroupOrderBy(BOOK.ID),
         percentileCont(0.25).withinGroupOrderBy(BOOK.ID),
         percentileCont(0.50).withinGroupOrderBy(BOOK.ID),
         percentileCont(0.75).withinGroupOrderBy(BOOK.ID),
         percentileCont(1.00).withinGroupOrderBy(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+------+------+------+------+------+
| 0.00 | 0.25 | 0.50 | 0.75 | 1.00 |
+------+------+------+------+------+
|    1 | 1.75 |  2.5 | 3.25 |    4 |
+------+------+------+------+------+
```

Dialect support

This example using jOOQ:

```
percentileCont(0.00).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, YugabyteDB

```
percentile_cont(CAST(0E0 AS double precision)) WITHIN GROUP (ORDER BY BOOK.ID)
```

BigQuery

```
percentile_cont(BOOK.ID, 0E0)
```

Databricks, DuckDB, Exasol, MariaDB, MemSQL, Oracle, Redshift, Snowflake, Teradata

```
percentile_cont(0E0) WITHIN GROUP (ORDER BY BOOK.ID)
```

DB2, H2

```
percentile_cont(CAST(0E0 AS double)) WITHIN GROUP (ORDER BY BOOK.ID)
```

SQLDataWarehouse, SQLServer

```
percentile_cont(CAST(0E0 AS float)) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, ClickHouse, CockroachDB, Derby, Firebird, HSQLDB, Hana, Informix, MySQL, SQLite, Sybase, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.24. PERCENTILE_DISC

The PERCENTILE_DISC() aggregate function is an ordered set function that calculates a given discrete percentile of all input values.

```
SELECT
  percentile_disc(0.00) WITHIN GROUP (ORDER BY ID),
  percentile_disc(0.25) WITHIN GROUP (ORDER BY ID),
  percentile_disc(0.50) WITHIN GROUP (ORDER BY ID),
  percentile_disc(0.75) WITHIN GROUP (ORDER BY ID),
  percentile_disc(1.00) WITHIN GROUP (ORDER BY ID)
FROM BOOK
```

```
create.select(
         percentileDisc(0.00).withinGroupOrderBy(BOOK.ID),
         percentileDisc(0.25).withinGroupOrderBy(BOOK.ID),
         percentileDisc(0.50).withinGroupOrderBy(BOOK.ID),
         percentileDisc(0.75).withinGroupOrderBy(BOOK.ID),
         percentileDisc(1.00).withinGroupOrderBy(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+------+------+------+------+------+
| 0.00 | 0.25 | 0.50 | 0.75 | 1.00 |
+------+------+------+------+------+
|    1 |    1 |    2 |    3 |    4 |
+------+------+------+------+------+
```

Dialect support

This example using jOOQ:

```
percentileDisc(0.00).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, Postgres, YugabyteDB

```
percentile_disc(CAST(0E0 AS double precision)) WITHIN GROUP (ORDER BY BOOK.ID)
```

BigQuery

```
percentile_disc(BOOK.ID, 0E0)
```

Databricks, DuckDB, Exasol, MariaDB, MemSQL, Oracle, Redshift, Snowflake, Teradata

```
percentile_disc(0E0) WITHIN GROUP (ORDER BY BOOK.ID)
```

DB2, H2

```
percentile_disc(CAST(0E0 AS double)) WITHIN GROUP (ORDER BY BOOK.ID)
```

SQLDataWarehouse, SQLServer

```
percentile_disc(CAST(0E0 AS float)) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, ClickHouse, CockroachDB, Derby, Firebird, HSQLDB, Hana, Informix, MySQL, SQLite, Sybase, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.25. PRODUCT

The PRODUCT() aggregate function is an aggregate function that calculates the product of all values in the group, similar to how the SUM function calculates the sum.

```
SELECT product(ID)
FROM BOOK
```

```
create.select(product(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+---------+
| product |
+---------+
|      24 |
+---------+
```

Dialect support

This example using jOOQ:

```
product(BOOK.ID)
```

Translates to the following dialect specific expressions:

Access

```
(SWITCH(sum(
  SWITCH(BOOK.ID = 0, 1)
) > 0, 0, (sum(
  SWITCH(BOOK.ID < 0, -1)
) MOD 2) < 0, -1, TRUE, 1) * exp(sum(log(abs(iif(BOOK.ID = 0, NULL, BOOK.ID))))))
```

ASE

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN (sum(
    CASE
      WHEN BOOK.ID < 0 THEN -1
    END
  ) % 2) < 0 THEN -1
  ELSE 1
END * exp(sum(log(abs(nullif(BOOK.ID, 0))))))
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Firebird, H2, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, Snowflake, Sybase, Trino, Vertica, YugabyteDB

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN mod(
    sum(
      CASE
        WHEN BOOK.ID < 0 THEN -1
      END
    ),
    2
  ) < 0 THEN -1
  ELSE 1
END * exp(sum(ln(abs(nullif(BOOK.ID, 0))))))
```

CockroachDB

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0E0 THEN 0
  WHEN mod(
    sum(
      CASE
        WHEN BOOK.ID < 0 THEN -1
      END
    ),
    2
  ) < 0E0 THEN -1
  ELSE 1
END * exp(sum(ln(CAST(abs(nullif(BOOK.ID, 0)) AS numeric)))))
```

Databricks, Redshift

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN (sum(
    CASE
      WHEN BOOK.ID < 0 THEN -1
    END
  ) % 2) < 0 THEN -1
  ELSE 1
END * exp(sum(ln(abs(nullif(BOOK.ID, 0))))))
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
(CASE
  WHEN sum(
    CASE
      WHEN BOOK.ID = 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN mod(
    sum(
      CASE
        WHEN BOOK.ID < 0 THEN -1
      END
    ),
    2
  ) < 0 THEN -1
  ELSE 1
END * exp(sum(ln(abs(nullif(BOOK.ID, 0))))))
```

DuckDB

```
product(BOOK.ID)
```

Exasol

```
mul(BOOK.ID)
```

Informix

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN mod(
    sum(
      CASE
        WHEN BOOK.ID < 0 THEN -1
      END
    ),
    2
  ) < 0 THEN -1
  ELSE 1
END * exp(sum(logn(abs(nullif(BOOK.ID, 0))))))
```

SQLDataWarehouse, SQLServer

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > CAST(0 AS decimal(1)) THEN 0
  WHEN (sum(
    CASE
      WHEN BOOK.ID < 0 THEN -1
    END
  ) % 2) < CAST(0 AS decimal(1)) THEN -1
  ELSE 1
END * exp(sum(log(abs(nullif(BOOK.ID, 0))))))
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
(CASE
  WHEN sum(
    CASE BOOK.ID
      WHEN 0 THEN 1
    END
  ) > 0 THEN 0
  WHEN (sum(
    CASE
      WHEN BOOK.ID < 0 THEN -1
    END
  ) MOD 2) < 0 THEN -1
  ELSE 1
END * exp(sum(ln(abs(nullif(BOOK.ID, 0))))))
```

SQLite

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.26. RANK

The RANK() hypothetical set function calculates the rank with gaps of the hypothetical value, i.e. ranks will be 1, 1, 1, 4, 5, 5, 7 (DENSE_RANK produces values without gaps, e.g. 1, 1, 1, 2, 3, 3, 4)

```
SELECT
  rank(0) WITHIN GROUP (ORDER BY AUTHOR_ID),
  rank(1) WITHIN GROUP (ORDER BY AUTHOR_ID),
  rank(2) WITHIN GROUP (ORDER BY AUTHOR_ID)
FROM BOOK
```

```
create.select(
         rank(val(0)).withinGroupOrderBy(BOOK.AUTHOR_ID),
         rank(val(2)).withinGroupOrderBy(BOOK.AUTHOR_ID),
         rank(val(4)).withinGroupOrderBy(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+---------+---------+---------+
| rank(0) | rank(1) | rank(2) |
+---------+---------+---------+
|       1 |       1 |       3 |
+---------+---------+---------+
```

Dialect support

This example using jOOQ:

```
rank(val(0)).withinGroupOrderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

Aurora Postgres, H2, Oracle, Postgres, YugabyteDB

```
rank(0) WITHIN GROUP (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.18.27. SUM

The SUM() aggregate function calculates the sum of all values per group.

```
SELECT sum(ID)
FROM BOOK
```

```
create.select(sum(BOOK.ID))
      .from(BOOK)
```

Producing:

```
+-----+
| sum |
+-----+
|  10 |
+-----+
```

Dialect support

This example using jOOQ:

```
sum(BOOK.ID)
```

Translates to the following dialect specific expressions:

All dialects

```
sum(BOOK.ID)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19. Window functions

A window function calculates an aggregate or ranking value over a subset of data (the window) relative to the projected row.

This has numerous powerful applications, for example, cumulative sums or sliding averages:

```
SELECT
  id,

  -- A sliding average over 3 rows, including the current row
  avg(amount) OVER (ORDER BY id ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING),
  amount,

  -- A cumulative sum over all previous rows
  sum(amount) OVER (ORDER BY id)
FROM (VALUES
  (1, 10.0),
  (2, 15.0),
  (3, 20.0),
  (4, 25.0),
  (5, 30.0),
  (6, 35.0)
) AS t (id, amount)
```

The result being

```
+----+-----------------+--------+-----------------+
| id | sliding average | amount |  cumulative sum |
+----+-----------------+--------+-----------------+
|  1 |  12.5           |   10.0 | --\        10.0 |
|  2 |  15.0           |   15.0 |   |        25.0 |
|  3 |  20.0           |   20.0 |   |        45.0 |
|  4 |  25.0       /-- |   25.0 |   |        70.0 |
|  5 |  30.0 = AVG +   |   30.0 | --+ SUM = 100.0 |
|  6 |  32.5       \-- |   35.0 |           135.0 |
+----+-----------------+--------+-----------------+
```

As this illustration shows, the aggregation happens over a window that is defined relative to the row on which it is calculated:

- In the AVG case, the window moves along with the row, always looking 1 row behind and 1 row ahead (if applicable), spanning anything between 1-3 rows, and calculating the average over those, forming a sliding average.
- In the SUM case, the window always starts at the beginning of the data set, and sums up all the rows up to the current row, forming a cumulative sum.

The details of how this powerful feature works will be illustrated over the next pages, where the various clauses, including the PARTITION BY clause, the ORDER BY clause, and the frame clause are explained.

3.8.19.1. PARTITION BY

Just like the GROUP BY clause, the PARTITION BY clause divides the data set into mutually exclusive groups / partitions.

This is best illustrated by example, where we fetch all the books and the total number of books per author of a book:

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  count(*) OVER (PARTITION BY BOOK.AUTHOR_ID)
FROM
  BOOK
```

```
create.select(
         BOOK.ID,
         BOOK.AUTHOR_ID,
         count().over(partitionBy(BOOK.AUTHOR_ID)))
      .from(BOOK)
      .fetch();
```

Producing:

```
+----+-----------+-------+
| id | author_id | count |
+----+-----------+-------+
|  1 |         1 |     2 |
|  2 |         1 |     2 |
|  3 |         2 |     2 |
|  4 |         2 |     2 |
+----+-----------+-------+
```

Just like ordinary aggregate functions that aggregate all data per GROUP produced by the GROUP BY clause, the PARTITION BY clause allows for grouping data, but without transforming the entire result set. As such, this kind of grouping can be much more convenient, especially in reporting, as the original data set isn't modified, only a new column is added.

(!) If you omit the PARTITION BY clause, then the entire data set forms a single partition.

Dialect support

This example using jOOQ:

```
count().over(partitionBy(BOOK.AUTHOR_ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
count(*) OVER (PARTITION BY BOOK.AUTHOR_ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.2. ORDER BY

The ORDER BY clause allows for ordering a window, which may be required by a window function to produce a deterministic result.

There are 2 reasons to order window functions:

- The window function requires it (e.g. ROW_NUMBER).
- The window has a explicit or implicit window frame.

While the window frame ordering will be discussed later, it is easy to see how ROW_NUMBER relies on ordering in an example:

```
SELECT
  BOOK.ID,
  row_number() OVER (ORDER BY id DESC)
FROM
  BOOK
```

```
create.select(
         BOOK.ID,
         rowNumber().over(orderBy(BOOK.ID.desc())))
      .from(BOOK)
      .fetch();
```

Producing:

```
+----+------------+
| id | row_number |
+----+------------+
|  1 |          4 |
|  2 |          3 |
|  3 |          2 |
|  4 |          1 |
+----+------------+
```

*(!) If you omit the ORDER BY clause (and if that's supported in a window function), then no **frame clause** is implied, and thus the entire **window partition** is being considered.*

Dialect support

This example using jOOQ:

```
rowNumber().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
row_number() OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.3. ROWS, RANGE, GROUPS (frame clause)

When a window specification contains a ORDER BY clause, then a window frame may be explicit or implicit, with aggregate window functions. In short, a window frame limits the size of the window in both directions, from the current row.

For example:

- A cumulative sum can be achieved by framing the window to include all PRECEDING rows.
- A sliding average can be achieved by framing the window to include a certain number of PRECEDING rows as well as FOLLOWING rows.

A frame can be limited in 3 modes:

- ROWS: This limits the frame by an exact number of rows, similar to the LIMIT clause. For example, ROWS 3 PRECEDING will include the 3 preceding rows and the current row in the window.
- RANGE: This limits the frame logically by a value range, e.g. RANGE 3 PRECEDING will include the a value range of [current value - 3, current value] and the current row in the window. This only works for types with well defined ranges, including numeric and temporal types.
- GROUPS: This limits the frame logically by a distinct value count range, e.g. GROUPS 3 PRECEDING will include the rows containing the 3 preceding distinct values and the current row in the window.

Notice how in the above examples, the current row is always included by default. It can be excluded using the window exclusion clause.

Complete syntax

The complete syntax is best illustrated using a grammar:

In the above short form where only a single frameBound is provided (e.g. ROWS 3 PRECEDING), then ROWS BETWEEN frameBound AND CURRENT ROW is implied.

Example

This is again best explained by example:

```
SELECT
  ID,
  PUBLISHED_IN,

  -- The 2 preceding rows and the current row
  COUNT(*) OVER (ORDER BY PUBLISHED_IN ROWS 2 PRECEDING),

  -- The 42 preceding years and the current row
  COUNT(*) OVER (ORDER BY PUBLISHED_IN RANGE 42 PRECEDING),

  -- The 1 preceding groups of years and the current row
  trunc(PUBLISHED_IN, -1),
  COUNT(*) OVER (ORDER BY trunc(PUBLISHED_IN, -1)
    GROUPS 1 PRECEDING)
FROM
  BOOK
ORDER BY published_in
```

```
create.select(
         BOOK.ID,
         BOOK.PUBLISHED_IN,

         // The 2 preceding rows and the current row

 count().over(orderBy(BOOK.PUBLISHED_IN).rowsPreceding(2)),

         // The 42 preceding years and the current row

 count().over(orderBy(BOOK.PUBLISHED_IN).rangePreceding(42)),

         // The 1 preceding groups of years and the current row
         trunc(BOOK.PUBLISHED_IN, -1),
         count().over(orderBy(trunc(BOOK.PUBLISHED_IN, -1))
                 .groupsPreceding(1)))
      .from(BOOK)
      .orderBy(BOOK.PUBLISHED_IN)
      .fetch();
```

Producing:

```
+----+--------------+-------+-------+--------+--------+
| id | published_in | rows | range | decade | groups |
+----+--------------+------+-------+--------+--------+
|  2 |         1945 |    1 |     1 |   1940 |      2 |
|  1 |         1948 |    2 |     2 |   1940 |      2 |
|  3 |         1988 |    3 |     2 |   1980 |      3 |
|  4 |         1990 |    3 |     3 |   1990 |      2 |
+----+--------------+------+-------+--------+--------+
```

As you can see:

- ROWS: All rows have between 0 and 2 preceding rows, plus the current row.
- RANGE: While 1990 has 2 preceding rows within 42 years (plus the rows with the same value as the current row), the other years have less rows in that time span.
- GROUPS: The decade of the 1980s have 2 rows belonging to the previous 1 group of decades (plus the rows with the same value as the current row).

This certainly requires a bit of practice. While the ROWS clause is straightforward, the RANGE and GROUPS clauses are a bit more tricky to understand, although they're much more powerful.

> *(!) If you omit the frame clause, but have an **ORDER BY clause**, then the RANGE UNBOUNDED PRECEDING frame is implicit for **aggregate window functions**. Without ORDER BY, RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING is implicit, i.e. the entire window partition.*

Dialect support

This example using jOOQ:

```
count().over(orderBy(BOOK.ID).rowsPreceding(3))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift,

SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
count(*) OVER (
  ORDER BY BOOK.ID
  ROWS 3 PRECEDING
)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Hana

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.4. EXCLUDE

While poorly supported by most dialects, this standard SQL extension to the window frame clause can be quite handy to govern the exclusion of the current ROW, GROUP, or TIES.

Options include:

- EXCLUDE CURRENT ROW: Only the current row is excluded from the window
- EXCLUDE GROUP: The rows belonging to the current row's group are excluded from the window
- EXCLUDE TIES: The rows tied with the current row are excluded from the window
- EXCLUDE NO OTHERS: The default.

Dialect support

This example using jOOQ:

```
count().over(orderBy(BOOK.ID).rowsPreceding(3).excludeCurrentRow())
```

Translates to the following dialect specific expressions:

CockroachDB, DuckDB, Exasol, H2, Oracle, Postgres, SQLite, Trino, YugabyteDB

```
count(*) OVER (
  ORDER BY BOOK.ID
  ROWS 3 PRECEDING EXCLUDE CURRENT ROW
)
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, DB2, Databricks, Derby, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Redshift, SQLDataWarehouse, SQLServer, Snowflake, Sybase, Teradata, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.5. NULL treatment

Some window functions may offer a special NULL treatment clause, which allows for excluding NULL values from being considered for the evaluation of the function.

These include:

- LEAD
- LAG
- FIRST_VALUE
- LAST_VALUE
- NTH_VALUE

Possible clause values include:

- IGNORE NULLS: NULL values are not considered for the evaluation of the function.
- RESPECT NULLS: The default.

An example illustrates the utility of this optional clause

```
SELECT
  id,
  amount,
  lead(amount) OVER (ORDER BY id) AS respect_nulls,
  lead(amount) IGNORE NULLS OVER (ORDER BY id) AS ignore_nulls
FROM (VALUES
  (1, 10.0),
  (2, 15.0),
  (3, 20.0),
  (4, null),
  (5, 30.0),
  (6, 35.0)
) AS t (id, amount)
```

The result being

```
+----+--------+---------------+--------------+
| ID | AMOUNT | RESPECT_NULLS | IGNORE_NULLS |
+----+--------+---------------+--------------+
|  1 |     10 |            15 |           15 |
|  2 |     15 |            20 |           20 |
|  3 |     20 |               |           30 | <-- difference here
|  4 |        |            30 |           30 |
|  5 |     30 |            35 |           35 |
|  6 |     35 |               |              |
+----+--------+---------------+--------------+
```

If you will, the IGNORE NULLS clause allows for skipping all rows containing NULL values until the next non-NULL value is found. Such a function may still return NULL if no next row is found in the window, e.g. the last row in the above example!

Dialect support

This example using jOOQ:

```
lead(BOOK.ID).ignoreNulls().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

BigQuery, DuckDB, Vertica

```
lead(BOOK.ID IGNORE NULLS) OVER (ORDER BY BOOK.ID)
```

Databricks, Exasol, H2, Informix, Oracle, Redshift, SQLServer, Snowflake, Teradata, Trino

```
lead(BOOK.ID) IGNORE NULLS OVER (ORDER BY BOOK.ID)
```

DB2

```
lead(BOOK.ID, 'IGNORE NULLS') OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, Derby, Firebird, HSQLDB, Hana, MariaDB, MemSQL, MySQL, Postgres, SQLDataWarehouse, SQLite, Sybase, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.6. FROM FIRST, FROM LAST

The NTH_VALUE window function has an additional clause that allows for specifying whether the Nth value should be searched from the FIRST or the LAST row in the window.

Possible clause values include:

- FROM FIRST: The default.
- FROM LAST: N is counted from the LAST row, backwards.

Dialect support

This example using jOOQ:

```
nthValue(BOOK.ID, 2).fromLast().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

DB2, Exasol, Firebird, H2, Oracle, Snowflake

```
nth_value(BOOK.ID, 2) FROM LAST OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, Databricks, Derby, DuckDB, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.7. Nested aggregate functions

Ordinary aggregate functions are evaluated before window functions, logically, in SQL's logical order of operations.

This means that when window functions are computed, the value of aggregate functions is already defined. This allows for some impressive looking expressions, like:

```
SELECT
  AUTHOR_ID,
  COUNT(*) AS BOOKS,
  SUM(COUNT(*)) OVER () AS ALL_BOOKS
FROM
  BOOK
GROUP BY AUTHOR_ID
```

```
create.select(
         BOOK.AUTHOR_ID,
         count().as("books"),
         sum(count()).over().as("all_books"))
     .from(BOOK)
     .groupBy(BOOK.AUTHOR_ID)
     .fetch();
```

Producing:

```
+-----------+-------+-----------+
| author_id | books | all_books |
+-----------+-------+-----------+
|         1 |     2 |         4 |
|         2 |     2 |         4 |
+-----------+-------+-----------+
```

In this example, the aggregate value of COUNT(*) can again be aggregated using SUM(COUNT(*)) OVER (...).

> *(!) It is, however, not possible to nest window functions in window functions, just like aggregate functions cannot be nested in aggregate functions.*

3.8.19.8. Window aggregation

Every aggregate function can be turned into a window function simply by appending the OVER() clause. For example:

- COUNT(*) OVER ()
- SUM(x) OVER ()
- MAX(x) OVER ()
- ...

This also includes user-defined aggregate functions!

3.8.19.9. Window ordered aggregate

Some aggregate functions already specify ordering for their aggregation. These include:

- ordered aggregate functions
- ordered set aggregate functions (subtle difference, see those sections)

In principle, there's nothing preventing users from ordering an ordered aggregate window function twice. While few dialects actually support this, here's how it would look:

```
SELECT
  ID,
  array_agg(ID ORDER BY ID DESC) OVER (),
  array_agg(ID ORDER BY ID DESC)
    OVER (ORDER BY ID)
FROM
  BOOK
```

```
create.select(
         BOOK.ID,
         arrayAgg(BOOK.ID.desc()).orderBy(BOOK.ID).over(),
         arrayAgg(BOOK.ID.desc()).orderBy(BOOK.ID)
           .over(orderBy(BOOK.ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+-----------+-----------+
| ID | ARRAY_AGG | ARRAY_AGG |
+----+-----------+-----------+
|  1 | [4,3,2,1] | [1]       |
|  2 | [4,3,2,1] | [2,1]     |
|  3 | [4,3,2,1] | [3,2,1]   |
|  4 | [4,3,2,1] | [4,3,2,1] |
+----+-----------+-----------+
```

While the aggregation itself lists the values in descending order, the window specification may or may not sort the values, meaning that if it does, the window frame only includes the preceding rows, whereas otherwise, all the rows are included.

3.8.19.10. ROW_NUMBER

ROW_NUMBER assigns a series of unique, consecutive numbers to each row in the partition.

```
SELECT
  LANGUAGE_ID,
  row_number() OVER (ORDER BY LANGUAGE_ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.LANGUAGE_ID,
         rowNumber().over(orderBy(BOOK.LANGUAGE_ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+-------------+------------+
| language_id | row_number |
+-------------+------------+
|           1 |          1 |
|           1 |          2 |
|           2 |          3 |
|           4 |          4 |
+-------------+------------+
```

(!) See this article for a comparison between ROW_NUMBER, RANK, and DENSE_RANK

Dialect support

This example using jOOQ:

```
rowNumber().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift,

SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
row_number() OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.11. RANK

RANK assigns a series of rankings, which are different from ROW_NUMBER.

- The same rank is assigned to tied rows
- Ranks are skipped after tied rows

This works similarly as ranks in a sports event, when two athletes are ranked first, the next one is ranked third.

```
SELECT
  LANGUAGE_ID,
  rank() OVER (ORDER BY LANGUAGE_ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.LANGUAGE_ID,
         rank().over(orderBy(BOOK.LANGUAGE_ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+-------------+------+
| language_id | rank |
+-------------+------+
|           1 |    1 |
|           1 |    1 | <-- Tied rows are both ranked first
|           2 |    3 | <-- The next rank is 3 as 2 is skipped
|           4 |    4 |
+-------------+------+
```

(!) ***See this article for a comparison between ROW_NUMBER, RANK, and DENSE_RANK***

Dialect support

This example using jOOQ:

```
rank().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
rank() OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.12. DENSE_RANK

DENSE_RANK assigns a series of "dense" rankings (where dense means, no gaps), which are different from RANK.

- The same rank is assigned to tied rows
- Ranks are not skipped after tied rows

```
SELECT
  LANGUAGE_ID,
  dense_rank() OVER (ORDER BY LANGUAGE_ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.LANGUAGE_ID,
         denseRank().over(orderBy(BOOK.LANGUAGE_ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+-------------+------------+
| language_id | dense_rank |
+-------------+------------+
|           1 |          1 |
|           1 |          1 | <-- Tied rows are both ranked first
|           2 |          2 | <-- The next rank is 2 as no ranks are skipped
|           4 |          3 |
+-------------+------------+
```

(!) ***See this article for a comparison between ROW_NUMBER, RANK, and DENSE_RANK***

Dialect support

This example using jOOQ:

```
denseRank().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
dense_rank() OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.13. PERCENT_RANK

PERCENT_RANK works like RANK, but assigns ranks between 0 and 1

```
SELECT
  LANGUAGE_ID,
  percent_rank() OVER (ORDER BY LANGUAGE_ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.LANGUAGE_ID,
         percentRank().over(orderBy(BOOK.LANGUAGE_ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+-------------+--------------+
| language_id | percent_rank |
+-------------+--------------+
|           1 |            0 |
|           1 |            0 | <-- Tied rows are both ranked first
|           2 | 0.6666666667 |
|           4 |            1 |
+-------------+--------------+
```

Dialect support

This example using jOOQ:

```
percentRank().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
percent_rank() OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, ClickHouse, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.14. CUME_DIST

CUME_DIST (or cumulative distribution) works like PERCENT_RANK, but assigns ranks between 1/N and 1

```
SELECT
  LANGUAGE_ID,
  cume_dist() OVER (ORDER BY LANGUAGE_ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.LANGUAGE_ID,
         cumeDist().over(orderBy(BOOK.LANGUAGE_ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+-------------+-----------+
| language_id | cume_dist |
+-------------+-----------+
|           1 |       0.5 |
|           1 |       0.5 | <-- Tied rows are both ranked first
|           2 |      0.75 |
|           4 |         1 |
+-------------+-----------+
```

Dialect support

This example using jOOQ:

```
cumeDist().over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
cume_dist() OVER (ORDER BY BOOK.ID)
```

MemSQL

```
(CAST(count(*) OVER (
  ORDER BY BOOK.ID
  RANGE UNBOUNDED PRECEDING
) AS decimal) / count(*) OVER (
  ORDER BY BOOK.ID
  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
))
```

ASE, Access, Aurora MySQL, ClickHouse, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.15. NTILE

The NTILE window function allows for dividing a window into N buckets of equal size (e.g. when N = 4, then you'll get quartiles), and computes the bucket number for each row.

For example:

```
SELECT
  ID,
  ntile(2) OVER (ORDER BY ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         ntile(2).over(orderBy(BOOK.ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+-------+
| id | ntile |
+----+-------+
|  1 |     1 |
|  2 |     1 |
|  3 |     2 |
|  4 |     2 |
+----+-------+
```

Dialect support

This example using jOOQ:

```
ntile(2).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
ntile(2) OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Sybase, Teradata

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.16. LEAD

The LEAD window function allows for getting the value of an expression evaluated on the next row, or the nth next row if an offset is given.

```
SELECT
  ID,
  lead(ID) OVER (ORDER BY ID),
  lead(ID, 2) OVER (ORDER BY ID),
  lead(ID, 2, -1) OVER (ORDER BY ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         lead(BOOK.ID).over(orderBy(BOOK.ID)),
         lead(BOOK.ID, 2).over(orderBy(BOOK.ID)),
         lead(BOOK.ID, 2, -1).over(orderBy(BOOK.ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+------+------+------+
| id | lead | lead | lead |
+----+------+------+------+
|  1 |    2 |    3 |    3 |
|  2 |    3 |    4 |    4 |
|  3 |    4 |      |   -1 |
|  4 |      |      |   -1 |
+----+------+------+------+
```

Notes:

- The window frame clause has no effect on LEAD.
- LEAD supports the optional NULL treatment clause.

Dialect support

This example using jOOQ:

```
lead(BOOK.ID).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
lead(BOOK.ID) OVER (ORDER BY BOOK.ID)
```

ClickHouse

```
leadInFrame(BOOK.ID) OVER (
  ORDER BY BOOK.ID
  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Sybase

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.17. LAG

The LAG window function allows for getting the value of an expression evaluated on the previous row, or the nth previous row if an offset is given.

```
SELECT
  ID,
  lag(ID) OVER (ORDER BY ID),
  lag(ID, 2) OVER (ORDER BY ID),
  lag(ID, 2, -1) OVER (ORDER BY ID)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         lag(BOOK.ID).over(orderBy(BOOK.ID)),
         lag(BOOK.ID, 2).over(orderBy(BOOK.ID)),
         lag(BOOK.ID, 2, -1).over(orderBy(BOOK.ID)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+------+------+------+
| id |  lag |  lag |  lag |
+----+------+------+------+
|  1 |      |      |   -1 |
|  2 |    1 |      |   -1 |
|  3 |    2 |    1 |    1 |
|  4 |    3 |    2 |    2 |
+----+------+------+------+
```

- The window frame clause has no effect on LAG.
- LAG supports the optional NULL treatment clause.

Dialect support

This example using jOOQ:

```
lag(BOOK.ID).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Teradata, Trino, Vertica, YugabyteDB

```
lag(BOOK.ID) OVER (ORDER BY BOOK.ID)
```

ClickHouse

```
lagInFrame(BOOK.ID) OVER (
  ORDER BY BOOK.ID
  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB, Sybase

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.18. FIRST_VALUE

The FIRST_VALUE window function allows for getting the value of an expression evaluated on the first row of the window.

```
SELECT
  ID,
  first_value(ID) OVER (ORDER BY ID),
  first_value(ID) OVER (ORDER BY ID ROWS 1 PRECEDING)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         firstValue(BOOK.ID).over(orderBy(BOOK.ID)),

 firstValue(BOOK.ID).over(orderBy(BOOK.ID).rowsPreceding(1)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+-------------+-------------+
| id | first_value | first_value |
+----+-------------+-------------+
|  1 |           1 |           1 |
|  2 |           1 |           1 |
|  3 |           1 |           2 |
|  4 |           1 |           3 |
+----+-------------+-------------+
```

- The window frame clause is applied to FIRST_VALUE.
- FIRST_VALUE supports the optional NULL treatment clause.

Dialect support

This example using jOOQ:

```
firstValue(BOOK.ID).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
first_value(BOOK.ID) OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.19. LAST_VALUE

The LAST_VALUE window function allows for getting the value of an expression evaluated on the last row of the window.

```
SELECT
  ID,
  last_value(ID) OVER (ORDER BY ID
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING),
  last_value(ID) OVER (ORDER BY ID
    ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         lastValue(BOOK.ID).over(orderBy(BOOK.ID)

 .rowsBetweenUnboundedPreceding().andUnboundedFollowing()),
         lastValue(BOOK.ID).over(orderBy(BOOK.ID)
           .rowsBetweenCurrentRow().andFollowing(1)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+------------+------------+
| id | last_value | last_value |
+----+------------+------------+
|  1 |          4 |          2 |
|  2 |          4 |          3 |
|  3 |          4 |          4 |
|  4 |          4 |          4 |
+----+------------+------------+
```

- The window frame clause is applied to LAST_VALUE. The default of CURRENT ROW being the upper bound is questionable, so an explicit upper bound is usually required.
- LAST_VALUE supports the optional NULL treatment clause.

Dialect support

This example using jOOQ:

```
lastValue(BOOK.ID).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, DuckDB, Exasol, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Postgres, Redshift,

SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
last_value(BOOK.ID) OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Derby, HSQLDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.19.20. NTH_VALUE

The NTH_VALUE window function allows for getting the value of an expression evaluated on the Nth row of the window.

```
SELECT
  ID,
  nth_value(ID, 2) OVER (ORDER BY ID),
  nth_value(ID, 2) OVER (ORDER BY ID ROWS 2 PRECEDING)
FROM
  BOOK;
```

```
create.select(
         BOOK.ID,
         nthValue(BOOK.ID, 2).over(orderBy(BOOK.ID)),
         nthValue(BOOK.ID,
 2).over(orderBy(BOOK.ID).rowsPreceding(2)))
     .from(BOOK)
     .fetch();
```

Producing:

```
+----+-----------+-----------+
| id | nth_value | nth_value |
+----+-----------+-----------+
|  1 |           |           |
|  2 |         2 |         2 |
|  3 |         2 |         2 |
|  4 |         2 |         3 |
+----+-----------+-----------+
```

- The window frame clause is applied to NTH_VALUE.
- NTH_VALUE supports the optional NULL treatment clause.
- NTH_VALUE supports the optional FIRST / LAST clause.

Dialect support

This example using jOOQ:

```
nthValue(BOOK.ID, 2).over(orderBy(BOOK.ID))
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, DuckDB, Exasol, Firebird, H2, Hana, MariaDB, MemSQL, MySQL, Oracle, Postgres, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
nth_value(BOOK.ID, 2) OVER (ORDER BY BOOK.ID)
```

ASE, Access, Aurora MySQL, Databricks, Derby, HSQLDB, Informix, Redshift, SQLDataWarehouse, SQLServer, Sybase, Teradata

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.8.20. User-defined functions

Some databases support user-defined functions, which can be embedded in any SQL statement, if you're using jOOQ's code generator. Let's say you have the following simple function in Oracle SQL:

```
CREATE OR REPLACE FUNCTION echo (INPUT NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN INPUT;
END echo;
```

The above function will be made available from a generated Routines class. You can use it like any other column expression:

```
SELECT echo(1) FROM DUAL WHERE echo(2) = 2
```

```
create.select(echo(1)).where(echo(2).eq(2)).fetch();
```

Note that user-defined functions returning CURSOR or ARRAY data types can also be used wherever table expressions can be used, if they are unnested

3.8.21. User-defined aggregate functions

Some databases support user-defined aggregate functions, which can then be used along with GROUP BY clauses or as window functions. An example for such a database is Oracle. With Oracle, you can define the following OBJECT type (the example was taken from the Oracle 11g documentation):

```
CREATE TYPE U_SECOND_MAX AS OBJECT
(
  MAX NUMBER, -- highest value seen so far
  SECMAX NUMBER, -- second highest value seen so far
  STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT U_SECOND_MAX) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateIterate(self IN OUT U_SECOND_MAX, value IN NUMBER) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateTerminate(self IN U_SECOND_MAX, returnValue OUT NUMBER, flags IN NUMBER) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateMerge(self IN OUT U_SECOND_MAX, ctx2 IN U_SECOND_MAX) RETURN NUMBER
);

CREATE OR REPLACE TYPE BODY U_SECOND_MAX IS
STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT U_SECOND_MAX)
RETURN NUMBER IS
BEGIN
  SCTX := U_SECOND_MAX(0, 0);
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateIterate(self IN OUT U_SECOND_MAX, value IN NUMBER) RETURN NUMBER IS
BEGIN
  IF VALUE > SELF.MAX THEN
    SELF.SECMAX := SELF.MAX;
    SELF.MAX := VALUE;
  ELSIF VALUE > SELF.SECMAX THEN
    SELF.SECMAX := VALUE;
  END IF;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateTerminate(self IN U_SECOND_MAX, returnValue OUT NUMBER, flags IN NUMBER) RETURN NUMBER IS
BEGIN
  RETURNVALUE := SELF.SECMAX;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateMerge(self IN OUT U_SECOND_MAX, ctx2 IN U_SECOND_MAX) RETURN NUMBER IS
BEGIN
  IF CTX2.MAX > SELF.MAX THEN
    IF CTX2.SECMAX > SELF.SECMAX THEN
      SELF.SECMAX := CTX2.SECMAX;
    ELSE
      SELF.SECMAX := SELF.MAX;
    END IF;
    SELF.MAX := CTX2.MAX;
  ELSIF CTX2.MAX > SELF.SECMAX THEN
    SELF.SECMAX := CTX2.MAX;
  END IF;
  RETURN ODCIConst.Success;
END;
END;
```

The above OBJECT type is then available to function declarations as such:

```
CREATE FUNCTION SECOND_MAX (input NUMBER) RETURN NUMBER
PARALLEL_ENABLE AGGREGATE USING U_SECOND_MAX;
```

Using the generated aggregate function

jOOQ's code generator will detect such aggregate functions and generate them differently from regular user-defined functions. They implement the org.jooq.AggregateFunction type, as mentioned in the manual's section about aggregate functions. Here's how you can use the SECOND_MAX() aggregate function with jOOQ:

```
-- Get the second-latest publishing date by author
SELECT SECOND_MAX(PUBLISHED_IN)
FROM BOOK
GROUP BY AUTHOR_ID
```

```
// Routines.secondMax() can be static-imported
create.select(secondMax(BOOK.PUBLISHED_IN))
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

3.8.22. The CASE expression

The CASE expression is part of the standard SQL syntax. While some RDBMS also offer an IF expression, or a DECODE function, you can always rely on the two types of CASE syntax:

```
SELECT

  -- Searched case
  CASE WHEN AUTHOR.FIRST_NAME = 'Paulo'  THEN 'brazilian'
       WHEN AUTHOR.FIRST_NAME = 'George' THEN 'english'
                                         ELSE 'unknown'
  END,

  -- Simple case
  CASE AUTHOR.FIRST_NAME WHEN 'Paulo'  THEN 'brazilian'
                         WHEN 'George' THEN 'english'
                                       ELSE 'unknown'
  END
FROM AUTHOR
```

```
create.select(

  // Searched case
  when(AUTHOR.FIRST_NAME.eq("Paulo"), "brazilian")
  .when(AUTHOR.FIRST_NAME.eq("George"), "english")
  .otherwise("unknown");

  // Simple case
  choose(AUTHOR.FIRST_NAME)
  .when("Paulo", "brazilian")
  .when("George", "english")
  .otherwise("unknown"))
.from(AUTHOR)
.fetch();
```

In jOOQ, both syntaxes are supported (The second one is emulated in Derby, which only knows the first one). Unfortunately, both case and else are reserved words in Java. jOOQ chose to use decode() from the Oracle DECODE function, or choose() / case_(), and otherwise() / else_().

A CASE expression can be used anywhere where you can place a column expression (or Field). For instance, you can SELECT the above expression, if you're selecting from AUTHOR:

```
SELECT AUTHOR.FIRST_NAME, [... CASE EXPR ...] AS nationality
  FROM AUTHOR
```

Short forms of the CASE expression

The SQL standard and some vendors support a variety of short forms of the CASE expression, usually in the form of functions. These include:

- COALESCE
- CHOOSE
- DECODE
- IIF or IF
- NULLIF
- NVL
- NVL2

Sort indirection is often implemented with a CASE clause of a SELECT's ORDER BY clause. See the manual's section about the ORDER BY clause for more details.

3.8.23. Sequences and serials

Sequences implement the org.jooq.Sequence interface, providing essentially this functionality:

```
// Get a field for the CURRVAL sequence property
Field<T> currval();

// Get a field for the NEXTVAL sequence property
Field<T> nextval();
```

So if you have a sequence like this in Oracle:

```
CREATE SEQUENCE s_author_id
```

You can then use your generated sequence object directly in a SQL statement as such:

```
// Reference the sequence in a SELECT statement:
Field<BigInteger> s = S_AUTHOR_ID.nextval();
BigInteger nextID = create.select(s).fetchOne(s);

// Reference the sequence in an INSERT statement:
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(S_AUTHOR_ID.nextval(), val("William"), val("Shakespeare"))
      .execute();
```

- For more information about generated sequences, refer to the manual's section about generated sequences
- For more information about executing standalone calls to sequences, refer to the manual's section about sequence execution

3.8.24. Scalar subqueries

A scalar subquery is a subquery that produces a scalar value, i.e. one row and one column. Such values can be used as ordinary column expressions. Syntactically, any Select<Record1<?>> type qualifies as a scalar subquery, irrespective of content and whether it is "correlated".

There are mostly 3 ways of creating scalar subqueries in jOOQ

- Type safe wrapping using DSL.field(Select)
- Type unsafe wrapping using Select.asField()
- Through convenience methods, such as Field.eq(Select)

For example:

```
SELECT
  AUTHOR.ID, (
    SELECT count(*) FROM AUTHOR
  ) AS authors
FROM AUTHOR
```

```
create.select(
          AUTHOR.ID,
          field(selectCount().from(AUTHOR)).as("authors"))
      .from(AUTHOR)
      .fetch();
```

Correlated subqueries

A "correlated" subquery is a subquery (scalar or not) whose execution depends on the query that it is embedded in. It acts as a function taking the current row as an input argument.

```
SELECT
  AUTHOR.ID, (
    SELECT count(*)
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) AS books
FROM AUTHOR
```

```
create.select(
          AUTHOR.ID,
          field(selectCount()
            .from(BOOK)
            .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
      .from(AUTHOR)
      .fetch();
```

In the above example, the subquery counts the number of books for each author from the outer query.

3.8.25. Tuples or row value expressions

According to the SQL standard, row value expressions can have a degree of more than one. This is commonly used in the INSERT statement, where the VALUES row value constructor allows for providing a row value expression as a source for INSERT data. Row value expressions can appear in various other places, though. They are supported by jOOQ as records / rows. jOOQ's DSL allows for the construction of type-safe records up to the degree of 22. Higher-degree Rows are supported as well, but without any type-safety. Row types are modelled as follows:

```
// The DSL provides overloaded row value expression constructor methods:
public static <T1>             Row1<T1>             row(T1 t1)                      { ... }
public static <T1, T2>         Row2<T1, T2>         row(T1 t1, T2 t2)               { ... }
public static <T1, T2, T3>     Row3<T1, T2, T3>     row(T1 t1, T2 t2, T3 t3)        { ... }
public static <T1, T2, T3, T4> Row4<T1, T2, T3, T4> row(T1 t1, T2 t2, T3 t3, T4 t4) { ... }

// [ ... idem for Row5, Row6, Row7, ..., Row22 ]

// Degrees of more than 22 are supported without type-safety
public static RowN row(Object... values) { ... }
```

Using row value expressions in predicates

Row value expressions are incompatible with most other QueryParts, but they can be used as a basis for constructing various conditional expressions, such as:

- comparison predicates
- NULL predicates
- BETWEEN predicates
- IN predicates
- OVERLAPS predicate (for degree 2 row value expressions only)

See the relevant sections for more details about how to use row value expressions in predicates.

Using row value expressions in UPDATE statements

The UPDATE statement also supports a variant where row value expressions are updated, rather than single columns. See the relevant section for more details

Higher-degree row value expressions

jOOQ chose to explicitly support degrees up to 22 to match Scala's typesafe tuple, function and product support. Unlike Scala, however, jOOQ also supports higher degrees without the additional typesafety.

3.9. Conditional expressions

Conditions or conditional expressions are widely used in SQL and in the jOOQ API. They can be used in

- The CASE expression
- The JOIN clause (or JOIN .. ON clause, to be precise) of a SELECT statement, UPDATE statement, DELETE statement
- The WHERE clause of a SELECT statement, UPDATE statement, DELETE statement
- The CONNECT BY clause of a SELECT statement
- The HAVING clause of a SELECT statement
- The MERGE statement's ON clause

Boolean types in SQL

Before SQL:1999, boolean types did not really exist in SQL. They were modelled by 0 and 1 numeric/char values. With SQL:1999, true booleans were introduced and are now supported by most databases. In short, these are possible boolean values:

- 1 or TRUE
- 0 or FALSE
- NULL or UNKNOWN

It is important to know that SQL differs from many other languages in the way it interprets the NULL boolean value. Most importantly, the following facts are to be remembered:

- [ANY] = NULL yields NULL (not FALSE)
- [ANY] != NULL yields NULL (not TRUE)
- NULL = NULL yields NULL (not TRUE)
- NULL != NULL yields NULL (not FALSE)

For simplified NULL handling, please refer to the section about the DISTINCT predicate.

Note that jOOQ does not model these values as actual column expression compatible.

3.9.1. Condition building

With jOOQ, most conditional expressions are built from column expressions, calling various methods on them.

For instance, to build a comparison predicate, you can write the following expression:

```
TITLE  = 'Animal Farm'
```

```
BOOK.TITLE.eq("Animal Farm")
```

Create conditions from the DSL

There are a few types of conditions, that can only be created statically from the DSL. These are:

- Plain SQL conditions, that allow you to phrase your own SQL string conditional expression
- The EXISTS predicate, a standalone predicate that creates a conditional expression
- The UNIQUE predicate, another standalone predicate creating a conditional expression
- Constant TRUE and FALSE conditional expressions
- Converting a BOOLEAN column to a condition

Connect conditions using boolean operators

Conditions can also be connected using boolean operators as will be discussed in a subsequent chapter.

3.9.2. TRUE and FALSE condition

When a conditional expression is mandatory, or when using dynamic SQL, it may be required to provide a "dummy" condition that always evaluates to TRUE or FALSE. For this purpose, you can use DSL.trueCondition(), DSL.falseCondition(), or DSL.noCondition(). For example:

TRUE

TRUE is the identity value of the AND boolean operator, and can be used for procedural or functional reduction of a set of values to a condition:

```
TRUE
AND
  ID = 1
AND
  TITLE = 'Animal Farm'
```

```
Condition condition = trueCondition();
if (id != null)
    condition = condition.and(BOOK.ID.eq(id));
if (title != null)
    condition = condition.and(BOOK.TITLE.eq(title));
```

If a dialect does not support boolean column types, jOOQ will simply generate 1 = 1.

FALSE

FALSE is the identity value of the OR boolean operator, and can be used for procedural or functional reduction of a set of values to a condition:

```
FALSE
OR
  ID = 1
OR
  ID = 7
```

```
List<Integer> list = List.of(1, 7);
Condition condition = list
    .stream()
    .map(BOOK.ID::eq)
    .reduce(falseCondition(), Condition::or)
```

If a dialect does not support boolean column types, jOOQ will simply generate 1 = 0.

© 2009 - 2025 by Data Geekery™ GmbH.

NO condition

If you think that the "left-over" identity that is generated from the above reductions is ugly, you can just use the auxiliary DSL.noCondition(), which acts as a pseudo-identity for both AND and OR, not generating any SQL, except if the reduction produces nothing (from an empty set), in case of which it will behave like TRUE, which is what you usually want when placing a dynamic condition in a WHERE clause.

If used with actual data:

```
ID = 1
OR
  ID = 7
```

```
List<Integer> list = List.of(1, 7);
Condition condition = list.stream().map(BOOK.ID::eq)
    .reduce(noCondition(), Condition::or)
```

If used with empty sets:

```
TRUE
```

```
List<Integer> list = List.of(1, 7);
Condition condition = list.stream().map(BOOK.ID::eq)
    .reduce(noCondition(), Condition::or)
```

Some additional information about the noCondition() and related topics can be found in the section about dynamic SQL.

3.9.3. BOOLEAN columns

Some databases support the standard SQL BOOLEAN data type, which produces Field<Boolean> column types in jOOQ's code generator. But even if your dialect doesn't support the BOOLEAN type out of the box, you may have applied a data type rewrite to force a TINYINT(1) or CHAR(1) or NUMBER(1) column to act as a BOOLEAN.

When you have such a column, you will want to use it as a condition, and vice-versa. A org.jooq.Field<Boolean> can be turned into a org.jooq.Condition using DSL.condition(Field). The inverse operation can be achieved using DSL.field(Condition):

```
Condition condition = BOOK.TITLE.eq("Animal Farm");
Field<Boolean> field = field(condition);

// Fetch boolean values from a table
create.select(field).from(BOOK).fetch();

// Use a boolean field as a condition
create.selectFrom(BOOK).where(field).fetch();
```

3.9.4. AND, OR, NOT boolean operators

In SQL, as in most other languages, conditional expressions can be connected using the AND and OR binary operators, as well as the NOT unary operator, to form new conditional expressions. In jOOQ, this is modelled as such:

```
(TITLE = 'Animal Farm' OR TITLE = '1984')
  AND NOT (AUTHOR.LAST_NAME = 'Orwell')
```

```
BOOK.TITLE.eq("Animal Farm").or(BOOK.TITLE.eq("1984"))
  .andNot(AUTHOR.LAST_NAME.eq("Orwell"))
```

© 2009 - 2025 by Data Geekery™ GmbH.

The above example shows that the number of parentheses in Java can quickly explode. Proper indentation may become crucial in making such code readable. In order to understand how jOOQ composes combined conditional expressions, let's assign component expressions first:

```
Condition a = BOOK.TITLE.eq("Animal Farm");
Condition b = BOOK.TITLE.eq("1984");
Condition c = AUTHOR.LAST_NAME.eq("Orwell");

Condition combined1 = a.or(b);             // These OR-connected conditions form a new condition, wrapped in parentheses
Condition combined2 = combined1.andNot(c); // The left-hand side of the AND NOT () operator is already wrapped in parentheses
```

The Condition API

Here are all boolean operators on the org.jooq.Condition interface:

```
and(Condition)            // Combine conditions with AND
and(String)               // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
and(String, Object...)    // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
and(String, QueryPart...) // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
andExists(Select<?>)      // Combine conditions with AND. Convenience for adding an exists predicate to the rhs
andNot(Condition)         // Combine conditions with AND. Convenience for adding an inverted condition to the rhs
andNotExists(Select<?>)   // Combine conditions with AND. Convenience for adding an inverted exists predicate to the rhs

or(Condition)             // Combine conditions with OR
or(String)                // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
or(String, Object...)     // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
or(String, QueryPart...)  // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
orExists(Select<?>)       // Combine conditions with OR. Convenience for adding an exists predicate to the rhs
orNot(Condition)          // Combine conditions with OR. Convenience for adding an inverted condition to the rhs
orNotExists(Select<?>)    // Combine conditions with OR. Convenience for adding an inverted exists predicate to the rhs

not()                     // Invert a condition (synonym for DSL.not(Condition)
```

3.9.5. Boolean operator precedence

As previously mentioned in the manual's section about arithmetic expressions, jOOQ does not implement operator precedence. All operators are evaluated from left to right, as expected in an object-oriented API. This is important to understand when combining boolean operators, such as AND, OR, and NOT. The following expressions are equivalent:

```
   A.and(B) .or(C) .and(D) .or(E)
(((A.and(B)).or(C)).and(D)).or(E)
```

In SQL, the two expressions wouldn't be the same, as SQL natively knows operator precedence.

```
   A AND B  OR C  AND D  OR E -- Precedence is applied
(((A AND B) OR C) AND D) OR E -- Precedence is overridden
```

3.9.6. Comparison predicate

In SQL, comparison predicates are formed using common comparison operators:

- = to test for equality
- <> or != to test for non-equality
- > to test for being strictly greater
- >= to test for being greater or equal
- < to test for being strictly less
- <= to test for being less or equal

Unfortunately, Java does not support operator overloading, hence these operators are also implemented as methods in jOOQ, like any other SQL syntax elements. The relevant parts of the org.jooq.Field interface are these:

```
eq or equal(T);                                     // =  (some bind value)
eq or equal(Field<T>);                              // =  (some column expression)
eq or equal(Select<? extends Record1<T>>);          // =  (some scalar SELECT statement)
ne or notEqual(T);                                  // <> (some bind value)
ne or notEqual(Field<T>);                           // <> (some column expression)
ne or notEqual(Select<? extends Record1<T>>);       // <> (some scalar SELECT statement)
lt or lessThan(T);                                  // <  (some bind value)
lt or lessThan(Field<T>);                           // <  (some column expression)
lt or lessThan(Select<? extends Record1<T>>);       // <  (some scalar SELECT statement)
le or lessOrEqual(T);                               // <= (some bind value)
le or lessOrEqual(Field<T>);                        // <= (some column expression)
le or lessOrEqual(Select<? extends Record1<T>>);    // <= (some scalar SELECT statement)
gt or greaterThan(T);                               // >  (some bind value)
gt or greaterThan(Field<T>);                        // >  (some column expression)
gt or greaterThan(Select<? extends Record1<T>>);    // >  (some scalar SELECT statement)
ge or greaterOrEqual(T);                            // >= (some bind value)
ge or greaterOrEqual(Field<T>);                     // >= (some column expression)
ge or greaterOrEqual(Select<? extends Record1<T>>); // >= (some scalar SELECT statement)
```

Note that every operator is represented by two methods. A verbose one (such as equal()) and a two-character one (such as eq()). Both methods are the same. You may choose either one, depending on your taste. The manual will always use the more verbose one.

jOOQ's convenience methods using comparison operators

In addition to the above, jOOQ provides a few convenience methods for common operations performed on strings using comparison predicates:

```
LOWER(TITLE) = LOWER('animal farm')
```

```
BOOK.TITLE.equalIgnoreCase("animal farm")
```

3.9.7. Comparison predicate (degree > 1)

All variants of the comparison predicate that we've seen in the previous chapter also work for row value expressions. If your database does not support row value expression comparison predicates, jOOQ emulates them the way they are defined in the SQL standard:

```
-- Row value expressions (equal)
(A, B)    =  (X, Y)
(A, B, C) =  (X, Y, Z)
-- greater than
(A, B)    >  (X, Y)

(A, B, C) >  (X, Y, Z)


-- greater or equal
(A, B)    >= (X, Y)


(A, B, C) >= (X, Y, Z)



-- Inverse comparisons

(A, B)    <> (X, Y)
(A, B)    <  (X, Y)
(A, B)    <= (X, Y)
```

```
-- Equivalent factored-out predicates (equal)
(A = X) AND (B = Y)
(A = X) AND (B = Y) AND (C = Z)
-- greater than
(A > X)
  OR ((A = X) AND (B > Y))
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y) AND (C > Z))
-- greater or equal
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y))
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y) AND (C > Z))
  OR ((A = X) AND (B = Y) AND (C = Z))
-- For simplicity, these predicates are shown in terms
-- of their negated counter parts
NOT((A, B) =  (X, Y))
NOT((A, B) >= (X, Y))
NOT((A, B) >  (X, Y))
```

jOOQ supports all of the above row value expression comparison predicates, both with column expression lists and scalar subselects at the right-hand side:

```
-- With regular column expressions
(BOOK.AUTHOR_ID, BOOK.TITLE) = (1, 'Animal Farm')

-- With scalar subselects
(BOOK.AUTHOR_ID, BOOK.TITLE) = (
  SELECT PERSON.ID, 'Animal Farm'
  FROM PERSON
  WHERE PERSON.ID = 1
)
```

```
// Column expressions
row(BOOK.AUTHOR_ID, BOOK.TITLE).eq(1, "Animal Farm");

// Subselects
row(BOOK.AUTHOR_ID, BOOK.TITLE).eq(
  select(PERSON.ID, val("Animal Farm"))
  .from(PERSON)
  .where(PERSON.ID.eq(1))
);
```

Dialect support

This example using jOOQ:

```
row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).eq("John", "Doe")
```

Translates to the following dialect specific expressions:

ASE, Access, Derby, DuckDB, Exasol, Firebird, Hana, MemSQL, SQLDataWarehouse, SQLServer, Sybase

```
(
  AUTHOR.FIRST_NAME = 'John'
  AND AUTHOR.LAST_NAME = 'Doe'
)
```

Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, H2, HSQLDB, MariaDB, MySQL, Postgres, Redshift, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) = ('John', 'Doe')
```

Databricks

```
(
  coalesce(AUTHOR.FIRST_NAME),
  coalesce(AUTHOR.LAST_NAME)
) = ('John', 'Doe')
```

Informix

```
ROW (AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) = ROW ('John', 'Doe')
```

Oracle

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) = (('John', 'Doe'))
```

Teradata

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) = (
  SELECT 'John', 'Doe'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.8. Quantified comparison predicate

If the right-hand side of a comparison predicate turns out to be a non-scalar table subquery, you can wrap that subquery in a quantifier, such as ALL, ANY, or SOME. Note that the SQL standard defines ANY and SOME to be equivalent. jOOQ settled for the more intuitive ANY and doesn't support SOME. Here are some examples, supported by jOOQ:

```
TITLE = ANY('Animal Farm', '1982')
PUBLISHED_IN > ALL(1920, 1940)
```

```
BOOK.TITLE.eq(any("Animal Farm", "1982"));
BOOK.PUBLISHED_IN.gt(all(1920, 1940));
```

For the example, the right-hand side of the quantified comparison predicates were filled with argument lists. But it is easy to imagine that the source of values results from a subselect.

ANY and the IN predicate

It is interesting to note that the SQL standard defines the IN predicate in terms of the ANY-quantified predicate. The following two expressions are equivalent:

```
[ROW VALUE EXPRESSION] IN [IN PREDICATE VALUE]
```

```
[ROW VALUE EXPRESSION] = ANY [IN PREDICATE VALUE]
```

Typically, the IN predicate is more readable than the quantified comparison predicate.

Dialect support

This example using jOOQ:

```
BOOK.AUTHOR_ID.eq(any(select(AUTHOR.ID).from(AUTHOR)))
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, DB2, Derby, DuckDB, Firebird, H2, HSQLDB, Hana, Informix, MariaDB, MySQL, Oracle, Postgres, SQLServer, Sybase, Teradata, Vertica, YugabyteDB

```
BOOK.AUTHOR_ID = ANY (
  SELECT AUTHOR.ID
  FROM AUTHOR
)
```

Access, BigQuery, Databricks, Exasol, MemSQL, Redshift, SQLDataWarehouse, SQLite, Snowflake, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.9. BETWEEN predicate

The BETWEEN predicate can be seen as syntactic sugar for a pair of comparison predicates. According to the SQL standard, the following two predicates are equivalent:

```
A BETWEEN B AND C
```

```
A >= B AND A <= C
```

Note the inclusiveness of range boundaries in the definition of the BETWEEN predicate. Intuitively, this is supported in jOOQ as such:

© 2009 - 2025 by Data Geekery™ GmbH.

```
PUBLISHED_IN     BETWEEN 1920 AND 1940
PUBLISHED_IN NOT BETWEEN 1920 AND 1940
```

```
BOOK.PUBLISHED_IN.between(1920).and(1940)
BOOK.PUBLISHED_IN.notBetween(1920).and(1940)
```

Dialect support

This example using jOOQ:

```
BOOK.TITLE.between("E").and("K")
```

Translates to the following dialect specific expressions:

All dialects

```
BOOK.TITLE BETWEEN 'E' AND 'K'
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

BETWEEN SYMMETRIC

The SQL standard defines the SYMMETRIC keyword to be used along with BETWEEN to indicate that you do not care which bound of the range is larger than the other. A database system should simply swap range bounds, in case the first bound is greater than the second one. jOOQ supports this keyword as well, emulating it if necessary.

```
PUBLISHED_IN     BETWEEN SYMMETRIC 1940 AND 1920
PUBLISHED_IN NOT BETWEEN SYMMETRIC 1940 AND 1920
```

```
BOOK.PUBLISHED_IN.betweenSymmetric(1940).and(1920)
BOOK.PUBLISHED_IN.notBetweenSymmetric(1940).and(1920)
```

The emulation is done trivially:

```
A BETWEEN SYMMETRIC B AND C
```

```
(A BETWEEN B AND C) OR (A BETWEEN C AND B)
```

Dialect support

This example using jOOQ:

```
BOOK.TITLE.betweenSymmetric("K").and("E")
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, DB2, Databricks, Derby, DuckDB, Firebird, H2, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
(
  BOOK.TITLE BETWEEN 'K' AND 'E'
  OR BOOK.TITLE BETWEEN 'E' AND 'K'
)
```

Aurora Postgres, CockroachDB, Exasol, HSQLDB, Postgres, YugabyteDB

```
BOOK.TITLE BETWEEN SYMMETRIC 'K' AND 'E'
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.10. BETWEEN predicate (degree > 1)

The SQL BETWEEN predicate also works well for row value expressions. Much like the BETWEEN predicate for degree 1, it is defined in terms of a pair of regular comparison predicates:

```
A BETWEEN           B AND C
A BETWEEN SYMMETRIC B AND C
```

```
 A >= B AND A <= C
(A >= B AND A <= C) OR (A >= C AND A <= B)
```

The above can be factored out according to the rules listed in the manual's section about row value expression comparison predicates.

jOOQ supports the BETWEEN [SYMMETRIC] predicate and emulates it in all SQL dialects where necessary. An example is given here:

```
row(BOOK.ID, BOOK.TITLE).between(1, "A").and(10, "Z");
```

Dialect support

This example using jOOQ:

```
row(BOOK.ID, BOOK.TITLE).between(1, "A").and(10, "Z")
```

Translates to the following dialect specific expressions:

ASE, Access, Derby, DuckDB, Exasol, Firebird, Hana, Informix, MemSQL, Oracle, SQLDataWarehouse, SQLServer, Sybase

```
(
  (
    BOOK.ID >= 1
    AND (
      BOOK.ID > 1
      OR (
        BOOK.ID = 1
        AND BOOK.TITLE >= 'A'
      )
    )
  )
  AND (
    BOOK.ID <= 10
    AND (
      BOOK.ID < 10
      OR (
        BOOK.ID = 10
        AND BOOK.TITLE <= 'Z'
      )
    )
  )
)
```

Aurora MySQL, MariaDB, MySQL

```
(
  (BOOK.ID, BOOK.TITLE) >= (1, 'A')
  AND (BOOK.ID, BOOK.TITLE) <= (10, 'Z')
)
```

Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, H2, HSQLDB, Postgres, Redshift, SQLite, Snowflake, Trino, Vertica, YugabyteDB

```
(BOOK.ID, BOOK.TITLE) BETWEEN (1, 'A') AND (10, 'Z')
```

Databricks

```
(
  coalesce(BOOK.ID),
  coalesce(BOOK.TITLE)
) BETWEEN (1, 'A') AND (10, 'Z')
```

© 2009 - 2025 by Data Geekery™ GmbH.

Teradata

```
(
  (BOOK.ID, BOOK.TITLE) >= (
    SELECT 1, 'A'
    FROM (
      SELECT 1 AS "dual"
    ) AS "dual"
  )
  AND (BOOK.ID, BOOK.TITLE) <= (
    SELECT 10, 'Z'
    FROM (
      SELECT 1 AS "dual"
    ) AS "dual"
  )
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.11. DISTINCT predicate

Some databases support the DISTINCT predicate, which serves as a convenient, NULL-safe comparison predicate. With the DISTINCT predicate, the following truth table can be assumed:

- [ANY] IS DISTINCT FROM NULL yields TRUE
- [ANY] IS NOT DISTINCT FROM NULL yields FALSE
- NULL IS DISTINCT FROM NULL yields FALSE
- NULL IS NOT DISTINCT FROM NULL yields TRUE

For instance, you can compare two fields for distinctness, ignoring the fact that any of the two could be NULL, which would lead to funny results. This is supported by jOOQ as such:

```
TITLE IS DISTINCT FROM SUB_TITLE
TITLE IS NOT DISTINCT FROM SUB_TITLE
```

```
BOOK.TITLE.isDistinctFrom(BOOK.SUB_TITLE)
BOOK.TITLE.isNotDistinctFrom(BOOK.SUB_TITLE)
```

Dialect support

This example using jOOQ:

```
AUTHOR.FIRST_NAME.isDistinctFrom(AUTHOR.LAST_NAME)
```

Translates to the following dialect specific expressions:

ASE, Exasol, SQLDataWarehouse, Vertica

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME x
  INTERSECT
  SELECT AUTHOR.LAST_NAME x
)
```

Aurora MySQL, MariaDB, MemSQL, MySQL

```
(NOT(AUTHOR.FIRST_NAME <=> AUTHOR.LAST_NAME))
```

Aurora Postgres, BigQuery, CockroachDB, DB2, Databricks, DuckDB, Firebird, H2, HSQLDB, Postgres, Redshift, SQLServer, Snowflake, Trino, YugabyteDB

```
AUTHOR.FIRST_NAME IS DISTINCT FROM AUTHOR.LAST_NAME
```

ClickHouse

```
arrayUniq(ARRAY(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)) = 2
```

Derby

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME x
  FROM SYSIBM.SYSDUMMY1
  INTERSECT
  SELECT AUTHOR.LAST_NAME x
  FROM SYSIBM.SYSDUMMY1
)
```

Hana, Sybase

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME x
  FROM SYS.DUMMY
  INTERSECT
  SELECT AUTHOR.LAST_NAME x
  FROM SYS.DUMMY
)
```

Informix

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME x
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
  INTERSECT
  SELECT AUTHOR.LAST_NAME x
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
)
```

Oracle

```
decode(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, 1, 0) = 0
```

SQLite

```
(AUTHOR.FIRST_NAME IS NOT AUTHOR.LAST_NAME)
```

Teradata

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME x
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
  INTERSECT
  SELECT AUTHOR.LAST_NAME x
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
)
```

Access

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.12. DISTINCT predicate (degree > 1)

The DISTINCT predicate is also supported for row value expressions of degree higher than 1. If your database does not support row value expression comparison predicates, jOOQ emulates them the way they are defined in the SQL standard:

```
(FIRST_NAME, LAST_NAME) IS DISTINCT FROM ('John', 'Doe')
(FIRST_NAME, LAST_NAME) IS NOT DISTINCT FROM ('John', 'Doe')
```

```
row(AUTHOR.FIRST_NAME,
 AUTHOR.LAST_NAME).isNotDistinctFrom("John", "Doe")
row(AUTHOR.FIRST_NAME,
 AUTHOR.LAST_NAME).isNotDistinctFrom("John", "Doe")
```

Dialect support

This example using jOOQ:

```
row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).isNotDistinctFrom("John", "Doe")
```

Translates to the following dialect specific expressions:

ASE, Exasol, Oracle, SQLDataWarehouse, SQLServer, Vertica

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  INTERSECT
  SELECT 'John', 'Doe'
)
```

Aurora MySQL, MariaDB, MySQL

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) <=> ('John', 'Doe')
```

Aurora Postgres, BigQuery, CockroachDB, DuckDB, H2, HSQLDB, Postgres, Snowflake, Trino, YugabyteDB

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) is not distinct from ('John', 'Doe')
```

ClickHouse

```
arrayUniq(ARRAY(
  TUPLE (AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME),
  TUPLE ('John', 'Doe')
)) = 1
```

Databricks

```
(
  coalesce(AUTHOR.FIRST_NAME),
  coalesce(AUTHOR.LAST_NAME)
) is not distinct from ('John', 'Doe')
```

DB2

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM SYSIBM.DUAL
  INTERSECT
  SELECT 'John', 'Doe'
  FROM SYSIBM.DUAL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Derby

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM SYSIBM.SYSDUMMY1
  INTERSECT
  SELECT 'John', 'Doe'
  FROM SYSIBM.SYSDUMMY1
)
```

Firebird

```
NOT EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM RDB$DATABASE
  UNION
  SELECT 'John', 'Doe'
  FROM RDB$DATABASE
  OFFSET 1 ROWS
)
```

Hana, Sybase

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM SYS.DUMMY
  INTERSECT
  SELECT 'John', 'Doe'
  FROM SYS.DUMMY
)
```

Informix

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
  INTERSECT
  SELECT 'John', 'Doe'
  FROM (
    SELECT 1 AS dual
    FROM systables
    WHERE (tabid = 1)
  ) AS dual
)
```

MemSQL

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM DUAL
  INTERSECT
  SELECT 'John', 'Doe'
  FROM DUAL
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Redshift

```
NOT ((AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) is distinct from ('John', 'Doe'))
```

SQLite

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) IS ('John', 'Doe')
```

Teradata

```
EXISTS (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
  INTERSECT
  SELECT 'John', 'Doe'
  FROM (
    SELECT 1 AS "dual"
  ) AS "dual"
)
```

Access

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.13. EXISTS predicate

Slightly less intuitive, yet more powerful than the IN predicate is the EXISTS predicate, that can be used to form semi-joins or anti-joins. With jOOQ, the EXISTS predicate can be formed in various ways:

- From the DSL, using static methods. This is probably the most used case
- From a conditional expression using convenience methods attached to boolean operators
- From a SELECT statement using convenience methods attached to the where clause, and from other clauses

An example of an EXISTS predicate can be seen here:

```
    EXISTS (SELECT 1 FROM BOOK
            WHERE AUTHOR_ID = 3)

NOT EXISTS (SELECT 1 FROM BOOK
            WHERE AUTHOR_ID = 3)
```

```
   exists(create.selectOne().from(BOOK)
                 .where(BOOK.AUTHOR_ID.eq(3)));

notExists(create.selectOne().from(BOOK)
                 .where(BOOK.AUTHOR_ID.eq(3)));
```

Note that in SQL, the projection of a subselect in an EXISTS predicate is irrelevant. To help you write queries like the above, you can use jOOQ's selectZero() or selectOne() DSL methods

Performance of IN vs. EXISTS

In theory, the two types of predicates can perform equally well. If your database system ships with a sophisticated cost-based optimiser, it will be able to transform one predicate into the other, if you have all necessary constraints set (e.g. referential constraints, not null constraints). However, in reality, performance between the two might differ substantially. An interesting blog post investigating this topic on the MySQL database can be seen here:
https://blog.jooq.org/not-in-vs-not-exists-vs-left-join-is-null-mysql/

Dialect support

This example using jOOQ:

```
exists(select(asterisk()).from(BOOK))
```

Translates to the following dialect specific expressions:

All dialects

```
EXISTS (
  SELECT *
  FROM BOOK
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.14. IN predicate

In SQL, apart from comparing a value against several values, the IN predicate can be used to create semi-joins or anti-joins. jOOQ knows the following methods on the org.jooq.Field interface, to construct such IN predicates:

```
in(Collection<?>)                   // Construct an IN predicate from a collection of bind values
in(T...)                            // Construct an IN predicate from bind values
in(Field<?>...)                     // Construct an IN predicate from column expressions
in(Select<? extends Record1<T>>)    // Construct an IN predicate from a subselect
notIn(Collection<?>)                // Construct a NOT IN predicate from a collection of bind values
notIn(T...)                         // Construct a NOT IN predicate from bind values
notIn(Field<?>...)                  // Construct a NOT IN predicate from column expressions
notIn(Select<? extends Record1<T>>) // Construct a NOT IN predicate from a subselect
```

A sample IN predicate might look like this:

```
TITLE     IN ('Animal Farm', '1984')
TITLE NOT IN ('Animal Farm', '1984')
```

```
BOOK.TITLE.in("Animal Farm", "1984")
BOOK.TITLE.notIn("Animal Farm", "1984")
```

NOT IN and NULL values

Beware that you should probably not have any NULL values in the right hand side of a NOT IN predicate, as the whole expression would evaluate to NULL, which is rarely desired. This can be shown informally using the following reasoning:

```
-- The following conditional expressions are formally or informally equivalent
A NOT IN (B, C)
A != ANY(B, C)
A != B AND A != C

-- Substitute C for NULL, you'll get
A NOT IN (B, NULL)   -- Substitute C for NULL
A != B AND A != NULL -- From the above rules
A != B AND NULL      -- [ANY] != NULL yields NULL
NULL                 -- [ANY] AND NULL yields NULL
```

A good way to prevent this from happening is to use the EXISTS predicate for anti-joins, which is NULL-value insensitive. See the manual's section about conditional expressions to see a boolean truth table.

Dialect support

This example using jOOQ:

```
val("TITLE").in(select(BOOK.TITLE).from(BOOK))
```

Translates to the following dialect specific expressions:

All dialects

```
'TITLE' IN (
  SELECT BOOK.TITLE
  FROM BOOK
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.15. IN predicate (degree > 1)

The SQL IN predicate also works well for row value expressions. Much like the IN predicate for degree 1, it is defined in terms of a quantified comparison predicate. The two expressions are equivalent:

```
R IN [IN predicate value]
```

```
R = ANY [IN predicate value]
```

jOOQ supports the IN predicate with row value expressions. An example is given here:

```
-- Using an IN list
(BOOK.ID, BOOK.TITLE) IN ((1, 'A'), (2, 'B'))

-- Using a subselect
(BOOK.ID, BOOK.TITLE) IN (
  SELECT T.ID, T.TITLE
  FROM T
)
```

```
// Using an IN list
row(BOOK.ID, BOOK.TITLE).in(row(1, "A"), row(2, "B"));

// Using a subselect
row(BOOK.ID, BOOK.TITLE).in(
  select(T.ID, T.TITLE)
  .from(T)
);
```

In both cases, i.e. when using an IN list or when using a subselect, the type of the predicate is checked. Both sides of the predicate must be of equal degree and row type.

Emulation of the IN predicate where row value expressions aren't well supported is currently only available for IN predicates that do not take a subselect as an IN predicate value.

Dialect support

This example using jOOQ:

```
row("FIRST", "LAST").in(select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).from(AUTHOR))
```

Translates to the following dialect specific expressions:

ASE, Access, Derby, DuckDB, Exasol, Firebird, Hana, Informix, MemSQL, SQLDataWarehouse, SQLServer, Sybase

```
EXISTS (
  SELECT alias_1.v0, alias_1.v1
  FROM (
    SELECT
      AUTHOR.FIRST_NAME v0,
      AUTHOR.LAST_NAME v1
    FROM AUTHOR
  ) alias_1
  WHERE (
    'FIRST' = alias_1.v0
    AND 'LAST' = alias_1.v1
  )
)
```

Aurora MySQL, Aurora Postgres, ClickHouse, CockroachDB, H2, HSQLDB, MariaDB, MySQL, Postgres, Redshift, SQLite, Teradata, Trino, YugabyteDB

```
('FIRST', 'LAST') IN (
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM AUTHOR
)
```

BigQuery, DB2, Snowflake, Vertica

```
EXISTS (
  SELECT alias_1.v0, alias_1.v1
  FROM (
    SELECT
      AUTHOR.FIRST_NAME v0,
      AUTHOR.LAST_NAME v1
    FROM AUTHOR
  ) alias_1
  WHERE ('FIRST', 'LAST') = (alias_1.v0, alias_1.v1)
)
```

Databricks

```
EXISTS (
  SELECT alias_1.v0, alias_1.v1
  FROM (
    SELECT
      AUTHOR.FIRST_NAME v0,
      AUTHOR.LAST_NAME v1
    FROM AUTHOR
  ) alias_1
  WHERE ('FIRST', 'LAST') = (
    coalesce(alias_1.v0),
    coalesce(alias_1.v1)
  )
)
```

Oracle

```
('FIRST', 'LAST') IN ((
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM AUTHOR
))
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.16. JSON predicate

The JSON predicate can be used to check if a given expression is valid JSON

```
SELECT 1
FROM dual
WHERE '{}' IS JSON
AND '{' IS NOT JSON
```

```
create.selectOne()
      .where(val("{}").isJson())
      .and(val("{").isNotJson())
      .fetch();
```

Dialect support

This example using jOOQ:

```
val("{}").isJson()
```

Translates to the following dialect specific expressions:

DuckDB, MariaDB, MySQL

```
json_valid('{}')
```

Oracle

```
'{}' IS JSON
```

Snowflake

```
(check_json('{}') IS NULL)
```

SQLDataWarehouse, SQLServer

```
(isjson('{}') = 1)
```

ASE, Access, Aurora MySQL, Aurora Postgres, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, Exasol, Firebird, H2, HSQLDB, Hana, Informix, MemSQL, Postgres, Redshift, SQLite, Sybase, Teradata, Trino, Vertica, YugabyteDB

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.17. LIKE predicate

LIKE predicates are popular for simple wildcard-enabled pattern matching.

Supported wildcards in all SQL databases are:

- _: (single-character wildcard)
- %: (multi-character wildcard)

With jOOQ, the LIKE predicate can be created from any column expression as such:

```
TITLE     LIKE '%abc%'
TITLE NOT LIKE '%abc%'
```

```
BOOK.TITLE.like("%abc%")
BOOK.TITLE.notLike("%abc%")
```

Concatenating wildcards

A common practice is to conatenate wildcards to the actual expression. While concatenation is dangerous in plain SQL, it is safe when creating dynamic bind values using the DSL API:

```
-- Generated SQL is using a bind variable
TITLE     LIKE '%abc%'
TITLE NOT LIKE '%abc%'
```

```
// abc might be user input
BOOK.TITLE.like("%" + abc "%")
BOOK.TITLE.notLike("%" + abc + "%")
```

Escaping operands with the LIKE predicate

Often, your pattern may contain any of the wildcard characters "_" and "%", in case of which you may want to escape them. jOOQ does not automatically escape patterns in like() and notLike() methods. Instead, you can explicitly define an escape character as such:

```
TITLE     LIKE '%The !%-Sign Book%' ESCAPE '!'
TITLE NOT LIKE '%The !%-Sign Book%' ESCAPE '!'
```

```
BOOK.TITLE.like("%The !%-Sign Book%", '!')
BOOK.TITLE.notLike("%The !%-Sign Book%", '!')
```

In the above predicate expressions, the exclamation mark character is passed as the escape character to escape wildcard characters "!_" and "!%", as well as to escape the escape character itself: "!!"

Please refer to your database manual for more details about escaping patterns with the LIKE predicate.

jOOQ's convenience methods using the LIKE predicate

jOOQ also provides a few convenience methods for common operations performed on strings using the LIKE predicate. Typical operations are "contains predicates", "starts with predicates", "ends with predicates", etc.

```
-- case insensitivity
LOWER(TITLE) LIKE LOWER('%abc%')
LOWER(TITLE) NOT LIKE LOWER('%abc%')

-- contains and similar methods
TITLE LIKE '%' || 'abc' || '%'
TITLE LIKE 'abc' || '%'
TITLE LIKE '%' || 'abc'
```

```
// case insensitivity
BOOK.TITLE.likeIgnoreCase("%abc%")
BOOK.TITLE.notLikeIgnoreCase("%abc%")

// contains and similar methods
BOOK.TITLE.contains("abc")
BOOK.TITLE.startsWith("abc")
BOOK.TITLE.endsWith("abc")
```

Note, that jOOQ escapes % and _ characters in values in some of the above predicate implementations. For simplicity, this has been omitted in this manual.

© 2009 - 2025 by Data Geekery™ GmbH.

Dialect support

This example using jOOQ:

```
BOOK.TITLE.like("%abc%")
```

Translates to the following dialect specific expressions:

All dialects

```
BOOK.TITLE LIKE '%abc%'
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.18. LIKE REGEX predicate

Regular expression LIKE predicates work like the LIKE predicate, but they match against a regular expression, not just simple LIKE patterns.

```
     TITLE LIKE_REGEX '.*abc.*'
NOT (TITLE LIKE_REGEX '.*abc.*')
```

```
BOOK.TITLE.likeRegex(".*abc.*")
BOOK.TITLE.notLikeRegex(".*abc.*")
```

Dialect support

This example using jOOQ:

```
BOOK.TITLE.likeRegex(".*abc.*")
```

Translates to the following dialect specific expressions:

Aurora MySQL, Databricks, H2, MariaDB, MemSQL, MySQL, Sybase

```
BOOK.TITLE REGEXP '.*abc.*'
```

Aurora Postgres, CockroachDB, Postgres, YugabyteDB

```
(BOOK.TITLE ~ '.*abc.*')
```

HSQLDB

```
REGEXP_MATCHES(BOOK.TITLE, '.*abc.*')
```

Oracle, Snowflake, Vertica

```
REGEXP_LIKE(BOOK.TITLE, '.*abc.*')
```

ASE, Access, BigQuery, ClickHouse, DB2, Derby, DuckDB, Exasol, Firebird, Hana, Informix, Redshift, SQLDataWarehouse, SQLServer, SQLite, Teradata, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.19. Quantified LIKE predicate

jOOQ also provides the synthetic [NOT] LIKE ANY and [NOT] LIKE ALL operators, which can be used to (positively resp. negatively) match a string against multiple patterns without having to manually string together multiple [NOT] LIKE predicates with AND or OR (learn about other synthetic sql syntaxes). The following examples show how these synthetic predicates translate to SQL:

```
(TITLE     LIKE '%abc%'  OR TITLE     LIKE '%def%')
(TITLE NOT LIKE '%abc%'  OR TITLE NOT LIKE '%def%')
(TITLE     LIKE '%abc%' AND TITLE     LIKE '%def%')
(TITLE NOT LIKE '%abc%' AND TITLE NOT LIKE '%def%')
```

```
BOOK.TITLE.like(any("%abc%", "%def%"))
BOOK.TITLE.notLike(any("%abc%", "%def%"))
BOOK.TITLE.like(all("%abc%", "%def%"))
BOOK.TITLE.notLike(all("%abc%", "%def%"))
```

All corresponding Java methods Field.like(QuantifiedSelect) and Field.notLike(QuantifiedSelect) return an instance of LikeEscapeStep, which can be used to specify an ESCAPE clause that will be applied to all patterns in the list. For brevity the examples above don't show this.

Note that both the LIKE ANY and LIKE ALL predicates allow matching a string against an empty list of patterns. For example, in the case of LIKE ANY this is equivalent to a 1 = 0 predicate and in the case of NOT LIKE ALL this behaves like 1 = 1.

Dialect support

This example using jOOQ:

```
BOOK.TITLE.like(any(select(concat(val("%"), LANGUAGE.CD, val("%"))).from(LANGUAGE)))
```

Translates to the following dialect specific expressions:

ASE, DB2, Firebird, Sybase, Teradata

```
1 = ANY (
  SELECT CASE
    WHEN BOOK.TITLE LIKE pattern THEN 1
    WHEN NOT (BOOK.TITLE LIKE pattern) THEN 0
  END
  FROM (
    SELECT ('%' || LANGUAGE.CD || '%') pattern
    FROM LANGUAGE
  ) t
)
```

Aurora MySQL, MariaDB, MySQL

```
TRUE = ANY (
  SELECT (BOOK.TITLE LIKE pattern)
  FROM (
    SELECT concat('%', LANGUAGE.CD, '%') pattern
    FROM LANGUAGE
  ) t
)
```

Aurora Postgres, ClickHouse, CockroachDB, Postgres, Snowflake, YugabyteDB

```
BOOK.TITLE LIKE ANY (
  SELECT ('%' || LANGUAGE.CD || '%')
  FROM LANGUAGE
)
```

Derby, DuckDB, H2, HSQLDB, Oracle, Vertica

```
TRUE = ANY (
  SELECT (BOOK.TITLE LIKE pattern)
  FROM (
    SELECT ('%' || LANGUAGE.CD || '%') pattern
    FROM LANGUAGE
  ) t
)
```

Hana

```
TRUE = ANY (
  SELECT CASE
    WHEN BOOK.TITLE LIKE pattern THEN TRUE
    WHEN NOT (BOOK.TITLE LIKE pattern) THEN FALSE
  END
  FROM (
    SELECT ('%' || LANGUAGE.CD || '%') pattern
    FROM LANGUAGE
  ) t
)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Informix

```
CAST('t' AS boolean) = ANY (
  SELECT CASE
    WHEN BOOK.TITLE LIKE pattern THEN CAST('t' AS boolean)
    WHEN NOT (BOOK.TITLE LIKE pattern) THEN CAST('f' AS boolean)
  END
  FROM (
    SELECT ('%' || LANGUAGE.CD || '%') pattern
    FROM LANGUAGE
  ) t
)
```

SQLServer

```
1 = ANY (
  SELECT CASE
    WHEN BOOK.TITLE LIKE pattern THEN 1
    WHEN NOT (BOOK.TITLE LIKE pattern) THEN 0
  END
  FROM (
    SELECT ('%' + LANGUAGE.CD + '%') pattern
    FROM LANGUAGE
  ) t
)
```

Access, BigQuery, Databricks, Exasol, MemSQL, Redshift, SQLDataWarehouse, SQLite, Trino

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.20. NULL predicate

In SQL, you cannot compare NULL with any value using comparison predicates, as the result would yield NULL again, which is neither TRUE nor FALSE (see also the manual's section about conditional expressions). In order to test a column expression for NULL, use the NULL predicate as such:

```
TITLE IS NULL
TITLE IS NOT NULL
```

```
BOOK.TITLE.isNull()
BOOK.TITLE.isNotNull()
```

Dialect support

This example using jOOQ:

```
BOOK.TITLE.isNull()
```

Translates to the following dialect specific expressions:

All dialects

```
BOOK.TITLE IS NULL
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.21. NULL predicate (degree > 1)

The SQL NULL predicate also works well for row value expressions, although it has some subtle, counter-intuitive features when it comes to inversing predicates with the NOT() operator! Here are some examples:

```
-- Row value expressions
(A, B) IS     NULL
(A, B) IS NOT NULL

-- Inverse of the above
NOT((A, B) IS     NULL)
NOT((A, B) IS NOT NULL)
```

```
-- Equivalent factored-out predicates
(A IS     NULL) AND (B IS     NULL)
(A IS NOT NULL) AND (B IS NOT NULL)

-- Inverse
(A IS NOT NULL) OR  (B IS NOT NULL)
(A IS     NULL) OR  (B IS     NULL)
```

The SQL standard contains a nice truth table for the above rules:

```
+----------------------+-----------+---------------+---------------+-------------------+
| Expression           | R IS NULL | R IS NOT NULL | NOT R IS NULL | NOT R IS NOT NULL |
+----------------------+-----------+---------------+---------------+-------------------+
| degree 1: null       | true      | false         | false         |  true             |
| degree 1: not null   | false     | true          | true          |  false            |
| degree > 1: all null | true      | false         | false         |  true             |
| degree > 1: some null | false     | false         | true          |  true             |
| degree > 1: none null | false     | true          | true          |  false            |
+----------------------+-----------+---------------+---------------+-------------------+
```

In jOOQ, you would simply use the isNull() and isNotNull() methods on row value expressions. Again, as with the row value expression comparison predicate, the row value expression NULL predicate is emulated by jOOQ, if your database does not natively support it:

```
row(BOOK.ID, BOOK.TITLE).isNull();
row(BOOK.ID, BOOK.TITLE).isNotNull();
```

Dialect support

This example using jOOQ:

```
row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).isNull()
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL,

Oracle, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
(
  AUTHOR.FIRST_NAME IS NULL
  AND AUTHOR.LAST_NAME IS NULL
)
```

Aurora Postgres, H2, Postgres, Redshift, YugabyteDB

```
(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) IS NULL
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.22. OVERLAPS predicate

When comparing dates, the SQL standard allows for using a special OVERLAPS predicate, which checks whether two date ranges overlap each other. The following can be said:

```
-- This yields true
(DATE '2010-01-01', DATE '2010-01-03') OVERLAPS (DATE '2010-01-02' DATE '2010-01-04')

-- INTERVAL data types are also supported. This is equivalent to the above
(DATE '2010-01-01', CAST('+2 00:00:00' AS INTERVAL DAY TO SECOND)) OVERLAPS
(DATE '2010-01-02', CAST('+2 00:00:00' AS INTERVAL DAY TO SECOND))
```

The OVERLAPS predicate in jOOQ

jOOQ supports the OVERLAPS predicate on row value expressions of degree 2. The following methods are contained in org.jooq.Row2:

```
Condition overlaps(T1 t1, T2 t2);
Condition overlaps(Field<T1> t1, Field<T2> t2);
Condition overlaps(Row2<T1, T2> row);
```

This allows for expressing the above predicates as such:

```
// The date range tuples version
row(Date.valueOf('2010-01-01'), Date.valueOf('2010-01-03')).overlaps(Date.valueOf('2010-01-02'), Date.valueOf('2010-01-04'))

// The INTERVAL tuples version
row(Date.valueOf('2010-01-01'), new DayToSecond(2)).overlaps(Date.valueOf('2010-01-02'), new DayToSecond(2))
```

© 2009 - 2025 by Data Geekery™ GmbH.

jOOQ's extensions to the standard

Unlike the standard (or any database implementing the standard), jOOQ also supports the OVERLAPS predicate for comparing arbitrary row vlaue expressions of degree 2. For instance, (1, 3) OVERLAPS (2, 4) will yield true in jOOQ. This is emulated as such

```
-- This predicate
(A, B) OVERLAPS (C, D)

-- can be emulated as such
(C <= B) AND (A <= D)
```

Dialect support

This example using jOOQ:

```
row(1, 3).overlaps(row(2, 4))
```

Translates to the following dialect specific expressions:

All dialects

```
(
  2 <= 3
  AND 1 <= 4
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.23. SIMILAR TO predicate

SIMILAR TO predicates are popular for more complex wildcard and regular expression enabled pattern matching.

Supported wildcards in all SQL databases are:

- _: (single-character wildcard)
- %: (multi-character wildcard)

With jOOQ, the SIMILAR TO predicate can be created from any column expression as such:

```
TITLE     SIMILAR TO '%abc%'
TITLE NOT SIMILAR TO '%abc%'
```

```
BOOK.TITLE.similarTo("%abc%")
BOOK.TITLE.notSimilarTo("%abc%")
```

Escaping operands with the SIMILAR TO predicate

Often, your pattern may contain any of the wildcard characters "_" and "%", in case of which you may want to escape them. jOOQ does not automatically escape patterns in similarTo() and notSimilarTo() methods. Instead, you can explicitly define an escape character as such:

```
TITLE     SIMILAR TO '%The !%-Sign Book%' ESCAPE '!'
TITLE NOT SIMILAR TO '%The !%-Sign Book%' ESCAPE '!'
```

```
BOOK.TITLE.similarTo("%The !%-Sign Book%", '!')
BOOK.TITLE.notSimilarTo("%The !%-Sign Book%", '!')
```

In the above predicate expressions, the exclamation mark character is passed as the escape character to escape wildcard characters "!_" and "!%", as well as to escape the escape character itself: "!!"

Please refer to your database manual for more details about escaping patterns with the SIMILAR TO predicate as well as what regular expression syntax is supported.

Dialect support

This example using jOOQ:

```
BOOK.TITLE.similarTo("%X%")
```

Translates to the following dialect specific expressions:

Aurora Postgres, BigQuery, CockroachDB, DuckDB, Exasol, Firebird, Postgres, Snowflake, YugabyteDB

```
BOOK.TITLE SIMILAR TO '%X%'
```

ASE, Access, Aurora MySQL, ClickHouse, DB2, Databricks, Derby, H2, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, Redshift, SQLDataWarehouse, SQLServer, SQLite, Sybase, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.24. UNIQUE predicate

The UNIQUE predicate is defined by the SQL standard, yet hardly any database implements this feature. It is a standalone predicate (much like the EXISTS predicate) which is used to check for uniqueness of rows returned by a given subquery. An example of an UNIQUE predicate can be seen here:

```
    UNIQUE (SELECT PUBLISHED_IN FROM BOOK
            WHERE AUTHOR_ID = 3)

NOT UNIQUE (SELECT PUBLISHED_IN FROM BOOK
            WHERE AUTHOR_ID = 3)
```

```
   unique(create.select(BOOK.PUBLISHED_IN).from(BOOK)
                .where(BOOK.AUTHOR_ID.eq(3)));

notUnique(create.select(BOOK.PUBLISHED_IN).from(BOOK)
                .where(BOOK.AUTHOR_ID.eq(3)));
```

The first example above evaluates to TRUE only if all books written by the given author were published in distinct years, whereas the second example will be TRUE if the author published at least two books within the same year.

Currently jOOQ emulates the UNIQUE predicate for all databases using an EXISTS predicate with a GROUP BY subquery wrapping the original subquery:

```
NOT EXISTS (
  SELECT 1 FROM (
    SELECT PUBLISHED_IN
    FROM BOOK
    WHERE AUTHOR_ID = 3
  ) T
  WHERE (T.PUBLISHED_IN) IS NOT NULL
  GROUP BY T.PUBLISHED_IN
  HAVING COUNT(*) > 1
)
```

NULL values

Be aware that (as mandated by the SQL standard) any rows returned by the subquery having NULL values for any of the projected columns will be ignored by the UNIQUE predicate. Also, for a subquery which doesn't return any rows (or all rows have at least one NULL value) the UNIQUE predicate evaluates to TRUE.

Dialect support

This example using jOOQ:

```
unique(select(BOOK.PUBLISHED_IN).from(BOOK))
```

Translates to the following dialect specific expressions:

ASE, Access, Aurora MySQL, BigQuery, ClickHouse, CockroachDB, DB2, Databricks, Derby, DuckDB, Exasol, Firebird, HSQLDB, Hana, Informix, MariaDB, MemSQL, MySQL, Oracle, SQLDataWarehouse, SQLServer, SQLite, Snowflake, Sybase, Teradata, Trino, Vertica

```
NOT EXISTS (
  SELECT 1
  FROM (
    SELECT BOOK.PUBLISHED_IN
    FROM BOOK
  ) t
  WHERE t.PUBLISHED_IN IS NOT NULL
  GROUP BY t.PUBLISHED_IN
  HAVING count(*) > 1
)
```

Aurora Postgres, Postgres, Redshift, YugabyteDB

```
NOT EXISTS (
  SELECT 1
  FROM (
    SELECT BOOK.PUBLISHED_IN
    FROM BOOK
  ) t
  WHERE (t.PUBLISHED_IN) IS NOT NULL
  GROUP BY t.PUBLISHED_IN
  HAVING count(*) > 1
)
```

H2

```
UNIQUE (
  SELECT BOOK.PUBLISHED_IN
  FROM BOOK
)
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.9.25. Query By Example (QBE)

A popular approach to querying database tables is called Query by Example, meaning that an "example" of a result record is provided instead of a formal query

```
-- example book record:
ID          :
AUTHOR_ID   : 1
TITLE       :
PUBLISHED_IN: 1970
LANGUAGE_ID : 1
```

```
-- Corresponding query
SELECT *
FROM book
WHERE author_id = 1
AND published_in = 1970
AND language_id = 1
```

The translation from an example record to a query is fairly straight-forward:

- If a record attribute is set to a value, then that value is used for an equality predicate
- If a record attribute is not set, then that attribute is not used for any predicates

jOOQ knows a simple API called DSL.condition(Record), which translates a org.jooq.Record to a org.jooq.Condition:

```
BookRecord book = new BookRecord();
book.setAuthorId(1);
book.setPublishedIn(1970);
book.setLanguageId(1);

// Using the explicit condition() API
Result<BookRecord> books1 =
DSL.using(configuration)
   .selectFrom(BOOK)
   .where(condition(book))
   .fetch();

// Using the convenience API on DSLContext
Result<BookRecord> books2 = DSL.using(configuration).fetchByExample(book);
```

The latter API call makes use of the convenience API DSLContext.fetchByExample(TableRecord).

3.10. Operator precedence

jOOQ supports various operators, such as:

- Set operators
- JOIN operators
- AND, OR, NOT
- =, <, >, etc.
- +, -, *, /, etc.
- Many more

While in actual SQL, operator precedence is important (and quite vendor specific, unfortunately!), in jOOQ this topic is irrelevant, as there are always explicit parentheses imposed by the host language (Java, Kotlin, Scala), which remove all ambiguity. For example:

```
A AND B OR C -- same as (A AND B) OR C
A OR B AND C -- same as A OR (B AND C)
```

```
A.and(B).or(C)
A.or(B.and(C))
```

In other words, if you're chaining methods like this:

```
A.op1(B).op2(C).op3(D)
```

... then, you're nesting operands from left to right like this:

```
(((A op1 B) op2 C) op3 D)
```

3.11. Synthetic SQL clauses

Most of the previously mentioned SQL clauses have a native representation in at least one of jOOQ's supported SQL dialects. For example, when a function like LPAD() is unavailable, jOOQ produces an equivalent expression for it:

```
-- MySQL (native support)
lpad('a', 10, ' ')

-- SQL Server (emulation)
(replicate(' ', (10 - len('a'))) + 'a')
```

```
// In Java



lpad("a", 10, " ")
```

However, since a lot of SQL is emulated for dialect compatibility, nothing prevents jOOQ from supporting synthetic SQL clauses that do not have any native representation anywhere.

An example for this is the quantified like predicate, introduced in jOOQ 3.12, which would be really useful in any database:

```
(TITLE     LIKE '%abc%'  OR TITLE     LIKE '%def%')
(TITLE NOT LIKE '%abc%'  OR TITLE NOT LIKE '%def%')
(TITLE     LIKE '%abc%' AND TITLE     LIKE '%def%')
(TITLE NOT LIKE '%abc%' AND TITLE NOT LIKE '%def%')
```

```
BOOK.TITLE.like(any("%abc%", "%def%"))
BOOK.TITLE.notLike(any("%abc%", "%def%"))
BOOK.TITLE.like(all("%abc%", "%def%"))
BOOK.TITLE.notLike(all("%abc%", "%def%"))
```

In this section, we briefly list most such synthetic SQL clauses, which are available both through the jOOQ API, and through the jOOQ parser, yet they do not have a native representation in any dialect.

- Implicit JOIN: Implicit JOINs are implicit LEFT JOINs that are derived from to-one relationship path expressions. In order to e.g. access the COUNTRY columns from a CUSTOMER record, it is possible to write CUSTOMER.address().city().country().NAME. jOOQ will produce the necessary LEFT JOIN graph, which is much more tedious to write.
- Relational Division: Relational algebra supports a divison operator, which is the inverse operator of the cross product.
- SEEK clause: The SEEK clause is a synthetic clause of the SELECT statement, which provides an alternative way of paginating other than the OFFSET clause. From a performance perspective, it is generally the preferred way to paginate.
- SEMI JOIN and ANTI JOIN: Relational algebra defines SEMI JOIN and ANTI JOIN operators, which do not have a representation in any SQL dialect supported by jOOQ (Apache Impala has it, though). In SQL, the EXISTS predicate or IN predicate is used instead.
- Sort indirection: When sorting, sometimes, we want to sort by a derived value, not the actual value of a column. Sort indirection makes this very easy with jOOQ.
- UNIQUE predicate: This SQL standard predicate has not yet been implemented in any SQL dialect (it is being considered for H2). An esoteric, yet occasionally useful predicate that is difficult to emulate manually using the EXISTS predicate.

3.12. Dynamic SQL

In most cases, table expressions, column expressions, and conditional expressions as introduced in the previous chapters will be embedded into different SQL statement clauses as if the statement were a static SQL statement (e.g. in a view or stored procedure):

```
create.select(
          AUTHOR.FIRST_NAME.concat(AUTHOR.LAST_NAME),
          count()
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .groupBy(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .orderBy(count().desc())
      .fetch();
```

It is, however, interesting to think of all of the above expressions as what they are: expressions. And as such, nothing keeps users from extracting expressions and referencing them from outside the statement. The following statement is exactly equivalent:

```
SelectField<?>[] select = {
    AUTHOR.FIRST_NAME.concat(AUTHOR.LAST_NAME),
    count()
};
Table<?> from = AUTHOR.join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID));
GroupField[] groupBy = { AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME };
SortField<?>[] orderBy = { count().desc() };

create.select(select)
      .from(from)
      .groupBy(groupBy)
      .orderBy(orderBy)
      .fetch();
```

Each individual expression, and collection of expressions can be seen as an independent entity that can be

- Constructed dynamically
- Reused across queries

Dynamic construction is particularly useful in the case of the WHERE clause, for dynamic predicate building. For instance:

```
public Condition condition(HttpServletRequest request) {
    Condition result = noCondition();

    if (request.getParameter("title") != null)
        result = result.and(BOOK.TITLE.like("%" + request.getParameter("title") + "%"));

    if (request.getParameter("author") != null)
        result = result.and(BOOK.AUTHOR_ID.in(
            select(AUTHOR.ID).from(AUTHOR).where(
                    AUTHOR.FIRST_NAME.like("%" + request.getParameter("author") + "%")
                .or(AUTHOR.LAST_NAME .like("%" + request.getParameter("author") + "%"))
            )
        ));

    return result;
}

// And then:
create.select()
      .from(BOOK)
      .where(condition(httpRequest))
      .fetch();
```

The dynamic SQL building power may be one of the biggest advantages of using a runtime query model like the one offered by jOOQ. Queries can be created dynamically, of arbitrary complexity. In the above example, we've just constructed a dynamic WHERE clause. The same can be done for any other clauses, including dynamic FROM clauses (dynamic JOINs), or adding additional WITH clauses as needed.

The above example made use of an optional condition, a mechanism that is explained in the following sections.

3.12.1. Optional conditional expressions

A key capability when creating dynamic SQL queries is to be able to provide optional conditional expressions.

```
boolean condition = ...

create.select(BOOK.ID)
      .from(BOOK)
      .where(condition ? BOOK.ID.eq(10) : noCondition())
      .fetch();
```

The above query produces:

```
-- If condition is true
SELECT book.id FROM book WHERE book.id = 10

-- If condition is false
SELECT book.id FROM book
```

The noCondition() expression will be ignored. If that means the clause is empty, then the entire clause will be omitted. This does not apply to clauses that project a Record type, including the SELECT clause, row value expressions, as well as function calls, in case of which a NULL value will be projected.

Some additional interactions of the noCondition() can be seen in the section about TRUE and FALSE conditions.

© 2009 - 2025 by Data Geekery™ GmbH.

*(!) The noCondition() expression is supported only in the **DSL API**, not in the **model API**, where the behaviour of noCondition() is undefined.*

3.13. Plain SQL

A DSL is a nice thing to have, it feels "fluent" and "natural", especially if it models a well-known language, such as SQL.

But an internal DSL comes with two main limitations:

- It is limited by what is defined explicitly as formal API, unlike external DSLs, which can be supported completely as unchecked String types.
- It has to follow the rules of the host language (Java in this case), and thus cannot mimick the equivalent external DSL to 100%.

We have seen a few functionalities where the DSL becomes a bit verbose. This can be especially true for:

- aliasing
- derived tables

The use-cases for jOOQ where verbosity or missing functionality is a problem are rather rare, and when they come up, you do have an option. Just write SQL the way you're used to using "plain SQL."

3.13.1. Plain SQL API

jOOQ allows you to embed SQL as a String into any supported statement in these contexts:

- Plain SQL as a conditional expression
- Plain SQL as a column expression
- Plain SQL as a function
- Plain SQL as a table expression
- Plain SQL as a query

Plain SQL API methods are usually overloaded in three ways. Let's look at the condition query part constructor:

```
// Construct a condition without bind values
// Example: condition("a = b")
@PlainSQL
Condition condition(String sql);

// Construct a condition with bind values
// Example: condition("a = ?", 1);
@PlainSQL
Condition condition(String sql, Object... bindings);

// Construct a condition taking other jOOQ object arguments
// Example: condition("a = {0}", val(1));
@PlainSQL
Condition condition(String sql, QueryPart... parts);
```

*(!) Note the @PlainSQL annotation that can be used with the **jOOQ checker** to forbid using plain SQL API entirely, or locally, for improved SQL injection protection.*

Both the bind value and the query part argument overloads support jOOQ's plain SQL templating language. Please refer to the org.jooq.impl.DSL Javadoc for more details.

The following is a more complete listing of plain SQL construction methods from the DSL:

```
// A condition
Condition condition(String sql);
Condition condition(String sql, Object... bindings);
Condition condition(String sql, QueryPart... parts);

// A field with an unknown data type
Field<Object> field(String sql);
Field<Object> field(String sql, Object... bindings);
Field<Object> field(String sql, QueryPart... parts);

// A field with a known data type
<T> Field<T> field(String sql, Class<T> type);
<T> Field<T> field(String sql, Class<T> type, Object... bindings);
<T> Field<T> field(String sql, Class<T> type, QueryPart... parts);
<T> Field<T> field(String sql, DataType<T> type);
<T> Field<T> field(String sql, DataType<T> type, Object... bindings);
<T> Field<T> field(String sql, DataType<T> type, QueryPart... parts);

// A field with a known name (properly escaped)
Field<Object> field(Name name);
<T> Field<T>  field(Name name, Class<T> type);
<T> Field<T>  field(Name name, DataType<T> type);

// A function
<T> Field<T> function(String name, Class<T> type, Field<?>... arguments);
<T> Field<T> function(String name, DataType<T> type, Field<?>... arguments);

// A table
Table<?> table(String sql);
Table<?> table(String sql, Object... bindings);
Table<?> table(String sql, QueryPart... parts);

// A table with a known name (properly escaped)
Table<Record> table(Name name);

// A query without results (update, insert, etc)
Query query(String sql);
Query query(String sql, Object... bindings);
Query query(String sql, QueryPart... parts);

// A query with results
ResultQuery<Record> resultQuery(String sql);
ResultQuery<Record> resultQuery(String sql, Object... bindings);
ResultQuery<Record> resultQuery(String sql, QueryPart... parts);

// A query with results. This is the same as resultQuery(...).fetch();
Result<Record> fetch(String sql);
Result<Record> fetch(String sql, Object... bindings);
Result<Record> fetch(String sql, QueryPart... parts);
```

Apart from the general factory methods, plain SQL is also available in various other contexts for convenience. For instance, when adding a .where("a = b") clause to a query. Hence, there exist several convenience methods where plain SQL can be inserted usefully. This is an example displaying all various use-cases in one single query:

```
// You can use your table aliases in plain SQL fields
// As long as that will produce syntactically correct SQL
Field<?> LAST_NAME    = field("a.LAST_NAME");

// You can alias your plain SQL fields
Field<?> COUNT1       = field("count(*) x");

// If you know a reasonable Java type for your field, you
// can also provide jOOQ with that type
Field<Integer> COUNT2 = field("count(*) y", Integer.class);

       // Use plain SQL as select fields
create.select(LAST_NAME, COUNT1, COUNT2)

       // Use plain SQL as aliased tables (be aware of syntax!)
      .from("author a")
      .join("book b")

       // Use plain SQL for conditions both in JOIN and WHERE clauses
      .on("a.id = b.author_id")

       // Bind a variable in plain SQL
      .where("b.title != ?", "Brida")

       // Use plain SQL again as fields in GROUP BY and ORDER BY clauses
      .groupBy(LAST_NAME)
      .orderBy(LAST_NAME)
      .fetch();
```

Important things to note about plain SQL!

There are some important things to keep in mind when using plain SQL:

- You have to provide something that will be syntactically correct. If it's not, then jOOQ won't know. Only your JDBC driver or your RDBMS will detect the syntax error.
- You have to provide consistency when you use variable binding. The number of ? must match the number of variables
- Your SQL is inserted into jOOQ queries without further checks. Hence, jOOQ can't prevent SQL injection or transform your SQL string in any way.

3.13.2. Plain SQL templating language

The plain SQL API, as documented in the previous chapter, supports a string templating mini-language that allows for constructing complex SQL string content from smaller parts. A simple example can be seen below, e.g. when looking for support for one of PostgreSQL's various vendor-specific operator types:

```
ARRAY[1,4,3] && ARRAY[2,1]
```

```
condition("{0} && {1}", array1, array2);
```

Such a plain SQL template always consists of two things:

- The SQL string fragment containing {number} style placeholders.
- A set of org.jooq.QueryPart arguments, which are expected to be embedded in the SQL string

The SQL string may reference the arguments by 0-based indexing. Each argument may be referenced several times. For instance, SQLite's emulation of the REPEAT(string, count) function may look like this:

```
Field<Integer> count = val(3);
Field<String> string = val("abc");
field("replace(substr(quote(zeroblob(({0} + 1) / 2)), 3, {0}), '0', {1})", String.class, count, string);
//                                    ^                  ^          ^                    ^^^^^  ^^^^^^
//                                    |                  |          |                      |      |
// argument "count" is repeated twice: \------------------+----------|---------------------/      |
// argument "string" is used only once:                             \----------------------------/
```

For convenience, there is also a DSL.list(QueryPart...) API that allows for wrapping a comma-separated list of query parts in a single template argument:

```
Field<String> a = val("a");
Field<String> b = val("b");
Field<String> c = val("c");

// These two produce the same result:
condition("my_column IN ({0}, {1}, {2})", a, b, c); // Using distinct template arguments
condition("my_column IN ({0})", list(a, b, c));    // Using a single template argument
```

Parsing rules

When processing these plain SQL templates, a mini parser is run that handles things like

- String literals like 'abc' or $$abc$$
- Quoted names like "name", `name`, or [name]
- Comments like /* block */ or -- line
- JDBC escape sequences like {d '2000-01-01'}
- Indexed (?) or named (:identifier) bind variable placeholders

The above are recognised by the templating engine and contents inside of them are ignored when replacing numbered placeholders and/or bind variables. For instance:

```
query(
  """
  SELECT /* In a comment, this is not a placeholder: {0}. And this is not a bind variable: ? */ title AS `title {1} ?`
  -- Another comment without placeholders: {2} nor bind variables: ?
  FROM book
  WHERE title = 'In a string literal, this is not a placeholder: {3}. And this is not a bind variable: ?'
  """
);
```

The above query does not contain any numbered placeholders nor bind variables, because the tokens that would otherwise be searched for are contained inside of comments, string literals, or quoted names.

3.14. Hints

A variety of RDBMS support hints, which are vendor specific instructions to the optimiser, telling it what algorithms or meta data usage should be enforced / prevented. Typically, hints are used to enforce hash joins, nested loop joins, etc. or to enforce / prevent index access.

In most cases, you should not need to hint the optimiser, as it can be reasonably expected to make the right decisions on your behalf. But every now and then, it does not, and in those cases, hints can be useful (still, do check if your statistics are up to date, if you have all the meta data you should have, etc.)

jOOQ supports the most common hints for the following database products:

3.14.1. MySQL hints

MySQL implements multiple hint syntaxes, including its own classic hints, as well as Oracle style hints, since more recent versions.

The following sections show what different types of MySQL hints are supported by jOOQ.

3.14.1.1. Index hints

Numerous syntaxes exist to make or prevent MySQL from using an index to access a table or for certain operations. Unlike the more generic Oracle-style hint comment syntax, these hints are embedded in the SQL statement directly:

```
SELECT *
FROM BOOK USE INDEX i_book_a
```

```
create.selectFrom(BOOK.useIndex("i_book_a"))
      .fetch()
```

The following methods on org.jooq.Table can be used for that purpose:

- Table.useIndex()
- Table.useIndexForJoin()
- Table.useIndexForOrderBy()
- Table.useIndexForGroupBy()
- Table.forceIndex()
- Table.forceIndexForJoin()
- Table.forceIndexForOrderBy()
- Table.forceIndexForGroupBy()
- Table.ignoreIndex()
- Table.ignoreIndexForJoin()
- Table.ignoreIndexForOrderBy()
- Table.ignoreIndexForGroupBy()

3.14.1.2. STRAIGHT_JOIN

MySQL supports a special type of JOIN operator to hint at the optimiser that the JOIN order must be from left to right: The STRAIGHT_JOIN

```
SELECT *
FROM BOOK
STRAIGHT_JOIN AUTHOR
  ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(BOOK)
      .straightJoin(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch()
```

© 2009 - 2025 by Data Geekery™ GmbH.

3.14.1.3. Oracle style hints in MySQL

Since more recent versions of MySQL, Oracle-style hints are now also supported.

For example, this has always been possible in MySQL:

```
SELECT SQL_CALC_FOUND_ROWS field1, field2
FROM table1
```

```
create.select(field1, field2)
      .hint("SQL_CALC_FOUND_ROWS")
      .from(table1)
      .fetch()
```

But now, you can also use the Oracle syntax for hints:

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ field1, field2
FROM table1
```

```
create.select(field1, field2)
      .hint("/*+ MAX_EXECUTION_TIME(1000) */")
      .from(table1)
      .fetch()
```

3.14.2. Oracle hints

Oracle implements hints using a comment style syntax, where the multi line comment contains a special + token to distinguish it from an ordinary comment, e.g. /*+HINT*/.

For example, the following hint tells the optimiser that the client is going to consume all the rows from the result set, as opposed to aborting the fetch after a few rows:

```
SELECT /*+ALL_ROWS*/ FIRST_NAME, LAST_NAME
  FROM AUTHOR
```

This can be done in jOOQ using the .hint() clause in your SELECT statement:

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .hint("/*+ALL_ROWS*/")
      .from(AUTHOR)
      .fetch();
```

Note that you can pass any string in the .hint() clause, including any non-hint comment if you wish to use this syntax to mark your queries. If you use that clause, the passed string will always be put in between the SELECT [DISTINCT] keywords and the actual projection list. This can be useful in other databases too, such as MySQL, for instance:

```
SELECT SQL_CALC_FOUND_ROWS field1, field2
FROM table1
```

```
create.select(field1, field2)
      .hint("SQL_CALC_FOUND_ROWS")
      .from(table1)
      .fetch()
```

See also Oracle-style hints in MySQL for more details.

3.14.3. SQL Server hints

SQL Server supports hints both on a table level as well as on a query level. Unlike Oracle-style hints, which use a special comment syntax, SQL Server hints are syntactic tokens that are part of your queries.

The following sections show what different types of SQL Server hints are supported by jOOQ.

3.14.3.1. WITH

The WITH hint in SQL Server is used for table level hints.

For example:

```
SELECT *
FROM BOOK WITH (READUNCOMMITTED)
```

```
create.selectFrom(BOOK.with("READUNCOMMITTED"))
      .fetch()
```

Dialect support

This example using jOOQ:

```
selectFrom(BOOK.with("ROWLOCK, UPDLOCK"))
```

Translates to the following dialect specific expressions:

ASE, Aurora MySQL, MariaDB, MemSQL

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
FOR UPDATE
```

Aurora Postgres, CockroachDB, H2, HSQLDB, MySQL, Postgres, YugabyteDB

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
FOR UPDATE OF BOOK
```

© 2009 - 2025 by Data Geekery™ GmbH.

DB2, Derby, Hana

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
FOR UPDATE OF ID, AUTHOR_ID, TITLE, PUBLISHED_IN, LANGUAGE_ID
```

Firebird

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
FOR UPDATE OF BOOK WITH LOCK
```

Informix, Oracle

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK
FOR UPDATE OF BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE, BOOK.PUBLISHED_IN, BOOK.LANGUAGE_ID
```

SQLServer, Sybase

```
SELECT
  BOOK.ID,
  BOOK.AUTHOR_ID,
  BOOK.TITLE,
  BOOK.PUBLISHED_IN,
  BOOK.LANGUAGE_ID
FROM BOOK WITH (ROWLOCK, UPDLOCK)
```

Access, BigQuery, ClickHouse, Databricks, DuckDB, Exasol, Redshift, SQLDataWarehouse, SQLite, Snowflake, Teradata, Trino, Vertica

```
/* UNSUPPORTED */
```

(!) Generated with jOOQ 3.21. Support in older jOOQ versions may differ. ***Translate your own SQL on our website***

3.14.3.2. OPTION

The OPTION hint in SQL Server is used for query level hints.

For example:

```
SELECT field1, field2
FROM table1
OPTION (OPTIMIZE FOR UNKNOWN)
```

```
create.select(field1, field2)
      .from(table1)
      .option("OPTION (OPTIMIZE FOR UNKNOWN)")
      .fetch()
```

Not all of these query level hints include the OPTION keyword, which is why it must be added manually to the OPTION string in jOOQ.

3.15. SQL Parser

A full-fledged SQL parser is available from DSLContext.parser() and from DSLContext.parsingConnection() (see the manual's section about parsing connections for the latter).

3.15.1. SQL Parser API

Goal

Historically, jOOQ implements an internal domain-specific language in Java, which generates SQL (an external domain-specific language) for use with JDBC. The jOOQ API is built from two parts: The DSL and the model API where the DSL API adds lexical convenience for programmers on top of the model API, which is really just a SQL expression tree, similar to what a SQL parser does inside of any database.

With this parser, the whole set of jOOQ functionality will now also be made available to anyone who is not using jOOQ directly, including JDBC and/or JPA users, e.g. through the parsing connection, which proxies all JDBC Connection calls to the jOOQ parser before forwarding them to the database, or through the DSLContext.parser() API, which allows for a more low-level access to the parser directly, e.g. for tool building on top of jOOQ.

The possibilities are endless, including standardised, SQL string based database migrations that work on any SQLDialect that is supported by jOOQ.

Example

This parser API allows for parsing an arbitrary SQL string fragment into a variety of jOOQ API elements:

- Parser.parse(String): This produces the org.jooq.Queries type, containing a batch of queries.
- Parser.parseQuery(String): This produces the org.jooq.Query type, containing a single query (any type of query).
- Parser.parseResultQuery(String): This produces the org.jooq.ResultQuery type, containing a single result query (SELECT or other).
- Parser.parseSelect(String): This produces the org.jooq.Select type, containing a single SELECT query.
- Parser.parseStatement(String): This produces the org.jooq.Statement type, containing a single procedural statement.
- Parser.parseTable(String): This produces the org.jooq.Table type, containing a table expression.
- Parser.parseField(String): This produces the org.jooq.Field type, containing a field expression.
- Parser.parseRow(String): This produces the org.jooq.Row type, containing a row expression.
- Parser.parseCondition(String): This produces the org.jooq.Condition type, containing a condition expression.
- Parser.parseName(String): This produces the org.jooq.Name type, containing a name expression.

See this blog post for more information about the various jOOQ types.

The parser is able to parse any unspecified dialect to produce a jOOQ representation of the SQL expression, for instance:

```
ResultQuery<?> query =
DSL.using(configuration)
   .parser()
   .parseResultQuery("SELECT * FROM (VALUES (1, 'a'), (2, 'b')) t(a, b)")
```

The above SQL query is valid standard SQL and runs out of the box on PostgreSQL and SQL Server, among others. The jOOQ ResultQuery that is generated from this SQL string, however, will also work on any other database, as jOOQ can emulate the two interesting SQL features being used here:

- The VALUES() constructor
- The derived column list syntax (aliasing table *and* columns in one go)

The query might be rendered as follows on the H2 database, which supports VALUES(), but not derived column lists:

```
select
  t.a,
  t.b
from (
  (
    select
      null a,
      null b
    where 1 = 0
  )
  union all (
    select *
    from (values
      (1, 'a'),
      (2, 'b')
    ) t
  )
) t;
```

Or like this on Oracle, which supports neither feature:

```
select
  t.a,
  t.b
from (
  (
    select
      null a,
      null b
    from dual
    where 1 = 0
  )
  union all (
    select *
    from (
      (
        select
          1,
          'a'
        from dual
      )
      union all (
        select
          2,
          'b'
        from dual
      )
    ) t
  )
) t;
```

3.15.2. SQL Parser CLI

The Parser API can be used as a translator between source and target dialects programmatically, as we've seen in the previous section about the parser API. This functionality can also usefully be accessed on the command line as shown below:

```
$ java -cp jooq-3.12.4.jar:reactive-streams-1.0.2.jar org.jooq.ParserCLI -h
Usage:
  -f / --formatted                          Format output SQL
  -h / --help                               Display this help
  -k / --keyword      <RenderKeywordStyle>  Specify the output keyword style (org.jooq.conf.RenderKeywordStyle)
  -i / --identifier   <RenderNameStyle>     Specify the output identifier style (org.jooq.conf.RenderNameStyle)
  -F / --from-dialect <SQLDialect>          Specify the input dialect (org.jooq.SQLDialect)
  -T / --to-dialect   <SQLDialect>          Specify the output dialect (org.jooq.SQLDialect)
  -s / --sql          <String>              Specify the input SQL string

  -I / --interactive                        Enter interactive mode

$ java -cp jooq-3.12.4.jar:reactive-streams-1.0.2.jar org.jooq.ParserCLI -T ORACLE -s "SELECT substring('abcde', 2, 3)"
select substr('abcde', 2, 3) from dual;
```

Windows users: Please replace : by ; in the above examples.

Another way to use this API is the https://www.jooq.org/translate website.

3.15.3. SQL translator

The SQL parser can be used as a SQL translator, as shown in action on our website: https://www.jooq.org/translate, or via the SQL parser CLI. The fact that translation happens is just an emerging feature of combining:

- The parser API to parse a SQL string into the jOOQ query object model API
- Using jOOQ to render the query object model again into a SQL string.

The simplest example is this one:

```
System.out.println(create.parser().parse("SELECT 1").toString());
```

This will parse the generic SQL string SELECT 1, and render it according to the configured Configuration, SQLDialect, Settings, etc.

3.15.4. SQL Parser Grammar

The existing implementation of the SQL parser is a hand-written, recursive descent parser. There are great advantages of this approach over formal grammar-based, generated parsers (e.g. by using ANTLR). These advantages are, among others:

- They can be tuned easily for performance
- They are very simple and easy to maintain
- It's easy to implement corner cases of the grammar, which might require context (that's a big plus with SQL)
- It's the easiest way to bind a grammar to an existing backing expression tree implementation (which is what jOOQ really is)

Nevertheless, there is a grammar available for documentation purposes and it is included in the manual here:

The diagrams have been created with the neat RRDiagram library by Christopher Deckers.

3.16. Names and identifiers

Various SQL objects columns or tables can be referenced using names (often also called identifiers). SQL dialects differ in the way they understand names, syntactically. The differences include:

- The permitted characters to be used in "unquoted" names
- The permitted characters to be used in "quoted" names
- The name quoting characters (e.g. "double quotes", `backticks`, or [brackets]) (e.g. "double quotes", `backticks`, or [brackets])
- The standard case for case-insensitive ("unquoted") names

For the above reasons, and also to prevent an additional SQL injection risk where names might contain SQL code, jOOQ by default quotes all names in generated SQL to be sure they match what is really contained in your database. This means that the following names will be rendered

```
-- Unquoted name
AUTHOR.TITLE

-- MariaDB, MySQL
`AUTHOR`.`TITLE`

-- MS Access, SQL Server, Sybase ASE, Sybase SQL Anywhere
[AUTHOR].[TITLE]

-- All the others, including the SQL standard
"AUTHOR"."TITLE"
```

Note that you can influence jOOQ's name rendering behaviour through custom settings, if you prefer another name style to be applied.

Creating custom names

Custom, qualified or unqualified names can be created very easily using the DSL.name() constructor:

```
// Unqualified name
Name name = name("TITLE");

// Qualified name
Name name = name("AUTHOR", "TITLE");
```

Such names can be used as standalone QueryParts, or as DSL entry point for SQL expressions, like

- Common table expressions to be used with the WITH clause
- Window specifications to be used with the WINDOW clause

More details about how to use names / identifiers to construct such expressions can be found in the relevant sections of the manual.

3.17. Bind values and parameters

Bind values are used in SQL / JDBC for various reasons. Among the most obvious ones are:

- Protection against SQL injection. Instead of inlining values possibly originating from user input, you bind those values to your prepared statement and let the JDBC driver / database take care of handling security aspects.
- Increased speed. Advanced databases such as Oracle can keep execution plans of similar queries in a dedicated cache to prevent hard-parsing your query again and again. In many cases, the actual value of a bind variable does not influence the execution plan, hence it can be reused. Preparing a statement will thus be faster
- On a JDBC level, you can also reuse the SQL string and prepared statement object instead of constructing it again, as you can bind new values to the prepared statement. jOOQ currently does not cache prepared statements, internally.

The following sections explain how you can introduce bind values in jOOQ, and how you can control the way they are rendered and bound to SQL.

3.17.1. Indexed parameters

JDBC only knows indexed bind values. A typical example for using bind values with JDBC is this:

```
try (PreparedStatement stmt = connection.prepareStatement("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?")) {

    // bind values to the above statement for appropriate indexes
    stmt.setInt(1, 5);
    stmt.setString(2, "Animal Farm");
    stmt.executeQuery();
}
```

With dynamic SQL, keeping track of the number of question marks and their corresponding index may turn out to be hard. jOOQ abstracts this and lets you provide the bind value right where it is needed. A trivial example is this:

```
create.select().from(BOOK).where(BOOK.ID.eq(5)).and(BOOK.TITLE.eq("Animal Farm")).fetch();

// This notation is in fact a short form for the equivalent:
create.select().from(BOOK).where(BOOK.ID.eq(val(5))).and(BOOK.TITLE.eq(val("Animal Farm"))).fetch();
```

Note the using of DSL.val() to explicitly create an indexed bind value. You don't have to worry about that index. When the query is rendered, each bind value will render a question mark. When the query binds its variables, each bind value will generate the appropriate bind value index.

Extract bind values from a query

Should you decide to run the above query outside of jOOQ, using your own java.sql.PreparedStatement, you can do so as follows:

```
Select<?> select = create.select().from(BOOK).where(BOOK.ID.eq(5)).and(BOOK.TITLE.eq("Animal Farm"));

// Render the SQL statement:
String sql = select.getSQL();
assertEquals("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?", sql);

// Get the bind values:
List<Object> values = select.getBindValues();
assertEquals(2, values.size());
assertEquals(5, values.get(0));
assertEquals("Animal Farm", values.get(1));
```

For more details about jOOQ's internals, see the manual's section about QueryParts.

3.17.2. Named parameters

Some SQL access abstractions that are built on top of JDBC, or some that bypass JDBC may support named parameters. jOOQ allows you to give names to your parameters as well, although those names are not rendered to SQL strings by default. Here is an example of how to create named parameters using the org.jooq.Param type:

```
// Create a query with a named parameter. You can then use that name for accessing the parameter again
Query query1 = create.select().from(AUTHOR).where(LAST_NAME.eq(param("lastName", "Poe")));
Param<?> param1 = query.getParam("lastName");

// Or, keep a reference to the typed parameter in order not to lose the <T> type information:
Param<String> param2 = param("lastName", "Poe");
Query query2 = create.select().from(AUTHOR).where(LAST_NAME.eq(param2));
```

The org.jooq.Query interface also allows for setting new bind values directly, without accessing the Param type:

```
Query query1 = create.select().from(AUTHOR).where(LAST_NAME.eq("Poe"));
query1.bind(1, "Orwell");

// Or, with named parameters
Query query2 = create.select().from(AUTHOR).where(LAST_NAME.eq(param("lastName", "Poe")));
query2.bind("lastName", "Orwell");
```

In order to actually render named parameter names in generated SQL, use the DSLContext.renderNamedParams() method:

```
-- The named bind variable can be rendered

SELECT *
FROM AUTHOR
WHERE LAST_NAME = :lastName
```

```
create.renderNamedParams(
    create.select()
          .from(AUTHOR)
          .where(LAST_NAME.eq(
                 param("lastName", "Poe"))));
```

3.17.3. Inlined parameters

Sometimes, you may wish to avoid rendering bind variables while still using custom values in SQL. Some example reasons can be seen in this blog post. jOOQ refers to that as "inlined" bind values. When bind values are inlined, they render the actual value in SQL rather than a JDBC question mark. Bind value inlining can be achieved in several ways:

- Globally, by using the Settings and setting the org.jooq.conf.StatementType to STATIC_STATEMENT. This will inline all bind values for SQL statements rendered from such a Configuration.
- Per query locally, by using the Query.getSQL(ParamType) method.
- Per QueryPart locally, by using any of the DSL.inlined(Condition), DSL.inlined(Field), DSL.inlined(QueryPart), or DSL.inlined(Statement) wrapper methods.
- Per value locally, by using DSL.inline() methods.

In all cases, your inlined bind values will be properly escaped to avoid SQL syntax errors and SQL injection. Some examples:

```
// Use dedicated calls to inline() in order to specify
// single bind values to be rendered as inline values
// --------------------------------------------------
create.select()
      .from(AUTHOR)
      .where(LAST_NAME.eq(inline("Poe")))
      .fetch();

// Or render the whole query with inlined values
// ---------------------------------------------------
Settings settings = new Settings()
    .withStatementType(StatementType.STATIC_STATEMENT);

// Add the settings to the Configuration
DSLContext create = DSL.using(connection, SQLDialect.ORACLE, settings);

// Run queries that omit rendering schema names
create.select()
      .from(AUTHOR)
      .where(LAST_NAME.eq("Poe"))
      .fetch();
```

3.17.4. SQL injection

SQL injection is serious

SQL injection is a serious problem that needs to be taken care of thoroughly. A single vulnerability can be enough for an attacker to dump your whole database, and potentially seize your database server. We've blogged about the severity of this threat on the jOOQ blog.

SQL injection happens because a programming language (SQL) is used to dynamically create arbitrary server-side statements based on user input. Programmers must take lots of care not to mix the language parts (SQL) with the user input (bind variables)

SQL injection in jOOQ

With jOOQ, SQL is usually created via a type safe, non-dynamic Java abstract syntax tree, where bind variables are a part of that abstract syntax tree. It is not possible to expose SQL injection vulnerabilities this way.

However, jOOQ offers convenient ways of introducing plain SQL strings in various places of the jOOQ API (which are annotated using org.jooq.PlainSQL since jOOQ 3.6). While jOOQ's API allows you to specify bind values for use with plain SQL, you're not forced to do that. For instance, both of the following queries will lead to the same, valid result:

```
// This query will use bind values, internally.
create.fetch("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?", 5, "Animal Farm");

// This query will not use bind values, internally.
create.fetch("SELECT * FROM BOOK WHERE ID = 5 AND TITLE = 'Animal Farm'");
```

All methods in the jOOQ API that allow for plain (unescaped, untreated) SQL contain a warning message in their relevant Javadoc, to remind you of the risk of SQL injection in what is otherwise a SQL-injection-safe API.

3.18. QueryParts

A org.jooq.Query and all its contained objects is a org.jooq.QueryPart. QueryParts essentially provide this functionality:

- they can render SQL using the accept(Context) method
- they can bind variables using the accept(Context) method

Both of these methods are contained in jOOQ's internal API's org.jooq.QueryPartInternal, which is internally implemented by every QueryPart.

The following sections explain some more details about SQL rendering and variable binding, as well as other implementation details about QueryParts in general.

3.18.1. SQL rendering

Every org.jooq.QueryPart must implement the accept(Context) method to render its SQL string to a org.jooq.RenderContext. This RenderContext has two purposes:

- It provides some information about the "state" of SQL rendering.
- It provides a common API for constructing SQL strings on the context's internal java.lang.StringBuilder

An overview of the org.jooq.RenderContext API is given here:

```
// These methods are useful for generating unique aliases within a RenderContext (and thus within a Query)
String peekAlias();
String nextAlias();

// These methods return rendered SQL
String render();
String render(QueryPart part);

// These methods allow for fluent appending of SQL to the RenderContext's internal StringBuilder
RenderContext keyword(String keyword);
RenderContext literal(String literal);
RenderContext sql(String sql);
RenderContext sql(char sql);
RenderContext sql(int sql);
RenderContext sql(QueryPart part);

// These methods allow for controlling formatting of SQL, if the relevant Setting is active
RenderContext formatNewLine();
RenderContext formatSeparator();
RenderContext formatIndentStart();
RenderContext formatIndentStart(int indent);
RenderContext formatIndentLockStart();
RenderContext formatIndentEnd();
RenderContext formatIndentEnd(int indent);
RenderContext formatIndentLockEnd();

// These methods control the RenderContext's internal state
boolean       inline();
RenderContext inline(boolean inline);
boolean       qualify();
RenderContext qualify(boolean qualify);
boolean       namedParams();
RenderContext namedParams(boolean renderNamedParams);
CastMode      castMode();
RenderContext castMode(CastMode mode);
Boolean       cast();
RenderContext castModeSome(SQLDialect... dialects);
```

The following additional methods are inherited from a common org.jooq.Context, which is shared among org.jooq.RenderContext and org.jooq.BindContext:

```
// These methods indicate whether fields or tables are being declared (MY_TABLE AS MY_ALIAS) or referenced (MY_ALIAS)
boolean declareFields();
Context declareFields(boolean declareFields);
boolean declareTables();
Context declareTables(boolean declareTables);

// These methods indicate whether a top-level query is being rendered, or a subquery
boolean subquery();
Context subquery(boolean subquery);

// These methods provide the bind value indices within the scope of the whole Context (and thus of the whole Query)
int nextIndex();
int peekIndex();
```

An example of rendering SQL

A simple example can be provided by checking out jOOQ's internal representation of a (simplified) CompareCondition. It is used for any org.jooq.Condition comparing two fields as for example the AUTHOR.ID = BOOK.AUTHOR_ID condition here:

```
-- [...]
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
-- [...]
```

This is how jOOQ renders such a condition (simplified example):

```
@Override
public final void accept(Context<?> context) {
    // The CompareCondition delegates rendering of the Fields to the Fields
    // themselves and connects them using the Condition's comparator operator:
    context.visit(field1)
           .sql(" ")
           .keyword(comparator.toSQL())
           .sql(" ")
           .visit(field2);
}
```

See the manual's sections about custom QueryParts and plain SQL QueryParts to learn about how to write your own query parts in order to extend jOOQ.

3.18.2. Declaration vs reference

A few types of org.jooq.QueryPart render different SQL depending on the context in which they are being rendered. The context is whether:

- The object is being declared
- The object is being referenced

A simple example is the aliasing of tables and columns. For example:

```
// Alias the table BOOK
Book b = BOOK.as("B");

// Use the alias as a reference
create.select(b.ID)

// Use the alias as a declaration
      .from(b)
      .fetch();
```

In the above example, when referencing the table alias, jOOQ renders the alias only, i.e. B. When declaring the table alias, jOOQ renders both the original table and its alias, i.e. BOOK as B.

While it may appear useful to occasionally enter in a third rendering mode, which renders the original table BOOK only, ignoring the alias, this isn't what is happening. If you want to access the original table (or column) reference, just keep a reference to that in your code, instead.

Types that expose this behaviour

The following types of org.jooq.QueryPart expose this kind of context sensitive rendering behaviour:

- org.jooq.Table: Table expressions can be aliased tables which can be declared in the FROM clause, or similar clauses of other statements.
- org.jooq.Field: Field expressions can be aliased columns which can be declared in the SELECT clause, or the RETURNING clauses of various statements.
- org.jooq.CommonTableExpression: Common table expressions can be declared in the WITH clause of various statements.
- org.jooq.WindowDefinition: Window definitions can be declared in the WINDOW clause of the SELECT statement.

3.18.3. Pretty printing SQL

*(!) Did you know you can pretty print arbitrary SQL with our translator at **https://www.jooq.org/translate**?*

As mentioned in the previous chapter about SQL rendering, there are some elements in the org.jooq.RenderContext that are used for formatting / pretty-printing rendered SQL. In order to obtain pretty-printed SQL, just use the following custom settings:

```
// Create a DSLContext that will render "formatted" SQL
DSLContext pretty = DSL.using(dialect, new Settings().withRenderFormatted(true));
```

And then, use the above DSLContext to render pretty-printed SQL:

```
select
  "TEST"."AUTHOR"."LAST_NAME",
  count(*) "c"
from "TEST"."BOOK"
  join "TEST"."AUTHOR"
  on "TEST"."BOOK"."AUTHOR_ID" = "TEST"."AUTHOR"."ID"
where "TEST"."BOOK"."TITLE" <> '1984'
group by "TEST"."AUTHOR"."LAST_NAME"
having count(*) = 2
```

```
String sql = pretty.select(
                     AUTHOR.LAST_NAME, count().as("c"))
                   .from(BOOK)
                   .join(AUTHOR)
                   .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                   .where(BOOK.TITLE.ne("1984"))
                   .groupBy(AUTHOR.LAST_NAME)
                   .having(count().eq(2))
                   .getSQL();
```

The section about ExecuteListeners shows an example of how such pretty printing can be used to log readable SQL to the stdout.

3.18.4. Variable binding

Every org.jooq.QueryPart must implement the accept(Context<?>) method. This Context has two purposes (among many others):

- It provides some information about the "state" of the variable binding in process.
- It provides a common API for binding values to the context's internal java.sql.PreparedStatement

An overview of the org.jooq.BindContext API is given here:

```
// This method provides access to the PreparedStatement to which bind values are bound
PreparedStatement statement();

// These methods provide convenience to delegate variable binding
BindContext bind(QueryPart part) throws DataAccessException;
BindContext bind(Collection<? extends QueryPart> parts) throws DataAccessException;
BindContext bind(QueryPart[] parts) throws DataAccessException;

// These methods perform the actual variable binding
BindContext bindValue(Object value, Class<?> type) throws DataAccessException;
BindContext bindValues(Object... values) throws DataAccessException;
```

Some additional methods are inherited from a common org.jooq.Context, which is shared among org.jooq.RenderContext and org.jooq.BindContext. Details are documented in the previous chapter about SQL rendering

An example of binding values to SQL

A simple example can be provided by checking out jOOQ's internal representation of a (simplified) CompareCondition. It is used for any org.jooq.Condition comparing two fields as for example the AUTHOR.ID = BOOK.AUTHOR_ID condition here:

```
-- [...]
WHERE AUTHOR.ID = ?
-- [...]
```

This is how jOOQ binds values on such a condition:

```
@Override
public final void bind(BindContext context) throws DataAccessException {
    // The CompareCondition itself does not bind any variables.
    // But the two fields involved in the condition might do so...
    context.bind(field1).bind(field2);
}
```

See the manual's sections about custom QueryParts and plain SQL QueryParts to learn about how to write your own query parts in order to extend jOOQ.

3.18.5. Custom data type bindings

jOOQ supports all the standard SQL data types out of the box, i.e. the types contained in java.sql.Types. But your domain model might be more specific, or you might be using a vendor-specific data type, such as JSON, HSTORE, or some other data structure. If this is the case, this section will be right for you, we'll see how you can create org.jooq.Converter types and org.jooq.Binding types.

Converters

The simplest use-case of injecting custom data types is by using org.jooq.Converter. A Converter can convert from a database type <T> to a user-defined type <U> and vice versa. You'll be implementing this SPI:

```
public interface Converter<T, U> {

    // Your conversion logic goes into these two methods, that can convert
    // between the database type T and the user type U:
    U from(T databaseObject);
    T to(U userObject);

 // You need to provide Class instances for each type, too:
    Class<T> fromType();
    Class<U> toType();
}
```

If, for instance, you want to use Java 8's java.time.LocalDate for SQL DATE and java.time.LocalDateTime for SQL TIMESTAMP, you write a converter like this:

```
import java.sql.Date;
import java.time.LocalDate;

import org.jooq.Converter;

public class LocalDateConverter implements Converter<Date, LocalDate> {

    @Override
    public LocalDate from(Date t) {
        return t == null ? null : LocalDate.parse(t.toString());
    }

    @Override
    public Date to(LocalDate u) {
        return u == null ? null : Date.valueOf(u.toString());
    }

    @Override
    public Class<Date> fromType() {
        return Date.class;
    }

    @Override
    public Class<LocalDate> toType() {
        return LocalDate.class;
    }
}
```

This converter can now be used in a variety of jOOQ API, most importanly to create a new data type:

```
DataType<LocalDate> type = DATE.asConvertedDataType(new LocalDateConverter());
```

And data types, in turn, can be used with any org.jooq.Field, i.e. with any column expression, including plain SQL or name based ones:

```
DataType<LocalDate> type = DATE.asConvertedDataType(new LocalDateConverter());

// Plain SQL based
Field<LocalDate> date1 = DSL.field("my_table.my_column", type);

// Name based
Field<LocalDate> date2 = DSL.field(name("my_table", "my_column"), type);
```

Bindings

While converters are very useful for simple use-cases, org.jooq.Binding is useful when you need to customise data type interactions at a JDBC level, e.g. when you want to bind a PostgreSQL JSON data type. Custom bindings implement the following SPI:

```
public interface Binding<T, U> extends Serializable {

    // A converter that does the conversion between the database type T
    // and the user type U (see previous examples)
    Converter<T, U> converter();

    // A callback that generates the SQL string for bind values of this
    // binding type. Typically, just ?, but also ?::json, etc.
    void sql(BindingSQLContext<U> ctx) throws SQLException;

    // Callbacks that implement all interaction with JDBC types, such as
    // PreparedStatement, CallableStatement, SQLOutput, SQLinput, ResultSet
    void register(BindingRegisterContext<U> ctx) throws SQLException;
    void set(BindingSetStatementContext<U> ctx) throws SQLException;
    void set(BindingSetSQLOutputContext<U> ctx) throws SQLException;
    void get(BindingGetResultSetContext<U> ctx) throws SQLException;
    void get(BindingGetStatementContext<U> ctx) throws SQLException;
    void get(BindingGetSQLInputContext<U> ctx) throws SQLException;
}
```

Below is full fledged example implementation that uses Google Gson to model JSON documents in Java

```
import java.sql.*;
import java.util.*;

import org.jooq.*;
import org.jooq.conf.*;
import org.jooq.impl.DSL;
import com.google.gson.*;

// We're binding <T> = JSON (or JSONB), and <U> = JsonElement (user type)
// Alternatively, extend org.jooq.impl.AbstractBinding to implement fewer methods.
public class PostgresJSONGsonBinding implements Binding<JSON, JsonElement> {

    private final Gson gson = new Gson();

    // The converter does all the work
    @Override
    public Converter<JSON, JsonElement> converter() {
        return new Converter<JSON, JsonElement>() {
            @Override
            public JsonElement from(JSON t) {
                return t == null ? JsonNull.INSTANCE : gson.fromJson(t.data(), JsonElement.class);
            }

            @Override
            public JSON to(JsonElement u) {
                return u == null || u == JsonNull.INSTANCE ? null : JSON.json(gson.toJson(u));
            }

            @Override
            public Class<JSON> fromType() {
                return JSON.class;
            }

            @Override
            public Class<JsonElement> toType() {
                return JsonElement.class;
            }
        };
    }

    // Rending a bind variable for the binding context's value and casting it to the json type
    @Override
    public void sql(BindingSQLContext<JsonElement> ctx) throws SQLException {
        // Depending on how you generate your SQL, you may need to explicitly distinguish
        // between jOOQ generating bind variables or inlined literals.
        if (ctx.render().paramType() == ParamType.INLINED)
            ctx.render().visit(DSL.inline(ctx.convert(converter()).value())).sql("::json");
        else
            ctx.render().sql(ctx.variable()).sql("::json");
    }

    // Registering VARCHAR types for JDBC CallableStatement OUT parameters
    @Override
    public void register(BindingRegisterContext<JsonElement> ctx) throws SQLException {
        ctx.statement().registerOutParameter(ctx.index(), Types.VARCHAR);
    }

    // Converting the JsonElement to a String value and setting that on a JDBC PreparedStatement
    @Override
    public void set(BindingSetStatementContext<JsonElement> ctx) throws SQLException {
        JSON json = ctx.convert(converter()).value();
        ctx.statement().setString(ctx.index(), json == null ? null : json.data());
    }

    // Getting a String value from a JDBC ResultSet and converting that to a JsonElement
    @Override
    public void get(BindingGetResultSetContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(JSON.json(ctx.resultSet().getString(ctx.index())));
    }

    // Getting a String value from a JDBC CallableStatement and converting that to a JsonElement
    @Override
    public void get(BindingGetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(JSON.json(ctx.statement().getString(ctx.index())));
    }

    // Setting a value on a JDBC SQLOutput (useful for Oracle OBJECT types)
    @Override
    public void set(BindingSetSQLOutputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }

    // Getting a value from a JDBC SQLInput (useful for Oracle OBJECT types)
    @Override
    public void get(BindingGetSQLInputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }
}
```

Code generation

There is a special section in the manual explaining how to automatically tie your Converters and Bindings to your generated code. The relevant sections are:

- Custom data types for org.jooq.Converter
- Custom data type binding for org.jooq.Binding

3.18.6. Custom syntax elements

To support simple vendor specific SQL syntax extensions, jOOQ offers the plain SQL templating API. If a SQL clause is too complex to express with jOOQ or with this templating API, or you have a requirement to support different dialects, you can extend either one of the following types for use directly in a jOOQ query:

```
// Simplified API description:
public abstract class CustomField<T> implements Field<T> {}
public abstract class CustomCondition implements Condition {}
public abstract class CustomTable<R extends TableRecord<R>> implements Table<R> {}
public abstract class CustomRecord<R extends TableRecord<R>> implements TableRecord<R> {}
```

An example for implementing a custom table and its record.

Here's an example org.jooq.impl.CustomTable showing how to create a custom table with its field definitions, similar to what the code generator is doing.

```
public class BookTable extends CustomTable<BookRecord> {
    public static final BookTable BOOK = new BookTable();

    public final TableField<BookRecord, String> FIRST_NAME = createField(name("FIRST_NAME"), VARCHAR);
    public final TableField<BookRecord, String> UNMATCHED  = createField(name("UNMATCHED"), VARCHAR);
    public final TableField<BookRecord, String> LAST_NAME  = createField(name("LAST_NAME"), VARCHAR);
    public final TableField<BookRecord, Short>  ID         = createField(name("ID"), SMALLINT);
    public final TableField<BookRecord, String> TITLE      = createField(name("TITLE"), VARCHAR);

    protected BookTable() {
        super(name("BOOK"));
    }

    @Override
    public Class<? extends BookRecord> getRecordType() {
        return BookRecord.class;
    }
}

public class BookRecord extends CustomRecord<BookRecord> {
    protected BookRecord() {
        super(BookTable.BOOK);
    }
}
```

An example for implementing custom multiplication.

Here's an example org.jooq.impl.CustomField showing how to create a field multiplying another field by 2

```
// Create an anonymous CustomField, initialised with BOOK.ID arguments
final Field<Integer> IDx2 = new CustomField<Integer>(BOOK.ID.getName(), BOOK.ID.getDataType()) {
    @Override
    public void accept(Context<?> context) {
        context.visit(BOOK.ID).sql(" * ").visit(DSL.val(2));
    }
};

// Use the above field in a SQL statement:
create.select(IDx2).from(BOOK);
```

An example for implementing vendor-specific functions.

Many vendor-specific functions are not officially supported by jOOQ, but you can implement such support yourself using CustomField, for instance. Here's an example showing how to implement Oracle's TO_CHAR() function, emulating it in SQL Server using CONVERT():

```
// Create a CustomField implementation taking two arguments in its constructor
class ToChar extends CustomField<String> {

    final Field<?> arg0;
    final Field<?> arg1;

    ToChar(Field<?> arg0, Field<?> arg1) {
        super("to_char", VARCHAR);

        this.arg0 = arg0;
        this.arg1 = arg1;
    }

    @Override
    public void accept(RenderContext context) {
        context.visit(delegate(context.configuration()));
    }

    private QueryPart delegate(Configuration configuration) {
        switch (configuration.family()) {
            case ORACLE:
                return DSL.field("TO_CHAR({0}, {1})", String.class, arg0, arg1);

            case SQLSERVER:
                return DSL.field("CONVERT(VARCHAR(8), {0}, {1})", String.class, arg0, arg1);

            default:
                throw new UnsupportedOperationException("Dialect not supported");
        }
    }
}
```

The above CustomField implementation can be exposed from your own custom DSL class:

```
public class MyDSL {
    public static Field<String> toChar(Field<?> field, String format) {
        return new ToChar(field, DSL.inline(format));
    }
}
```

3.18.7. Plain SQL QueryParts

If you don't need the integration of rather complex QueryParts into jOOQ, then you might be safer using simple Plain SQL functionality, where you can provide jOOQ with a simple String representation of your embedded SQL. Plain SQL methods in jOOQ's API come in two flavours.

- method(String, Object...): This is a method that accepts a SQL string and a list of bind values that are to be bound to the variables contained in the SQL string
- method(String, QueryPart...): This is a method that accepts a SQL string and a list of QueryParts that are "injected" at the position of their respective placeholders in the SQL string

The above distinction is best explained using an example:

```
// Plain SQL using bind values. The value 5 is bound to the first variable, "Animal Farm" to the second variable:
create.selectFrom(BOOK).where(
    "BOOK.ID = ? AND TITLE = ?",     // The SQL string containing bind value placeholders ("?")
    5,                               // The bind value at index 1
    "Animal Farm"                    // The bind value at index 2
).fetch();

// Plain SQL using embeddable QueryPart placeholders (counting from zero).
// The QueryPart "index" is substituted for the placeholder {0}, the QueryPart "title" for {1}
Field<Integer> id   = val(5);
Field<String> title = val("Animal Farm");
create.selectFrom(BOOK).where(
    "BOOK.ID = {0} AND TITLE = {1}", // The SQL string containing QueryPart placeholders ("{N}")
    id,                              // The QueryPart at index 0
    title                            // The QueryPart at index 1
).fetch();
```

Note that for historic reasons the two API usages can also be mixed, although this is not recommended and the exact behaviour is unspecified.

Plain SQL templating specification

Templating with QueryPart placeholders (or bind value placeholders) requires a simple parsing logic to be applied to SQL strings. The jOOQ template parser behaves according to the following rules:

- Single-line comments (starting with -- in all databases (or #) in MySQL) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Multi-line comments (starting with /* and ending with */ in all databases) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- String literals (starting and ending with ' in all databases, where all databases support escaping of the quote character by duplication as such: '', or in MySQL by escaping as such: \' (if Settings.backslashEscaping is turned on)) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Quoted names (starting and ending with " in most databases, with ` in MySQL, or with [and] in T-SQL databases) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- JDBC escape syntax ({fn ...}, {d ...}, {t ...}, {ts ...}) is rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Bind variable placeholders (? or :name for named bind variables) are replaced by the matching bind value in case inlining is activated, e.g. through Settings.statementType == STATIC_STATEMENT.
- QueryPart placeholders ({number}) are replaced by the matching QueryPart.
- Keywords ({identifier}) are treated like keywords and rendered in the correct case according to Settings.renderKeywordStyle.

Tools for templating

A variety of API is provided to create template elements that are intended for use with the above templating mechanism. These tools can be found in org.jooq.impl.DSL

```
// Keywords (which are rendered according to Settings.renderKeywordStyle) can be specified as such:
public static Keyword keyword(String keyword) { ... }

// Identifiers / names (which are rendered according to Settings.renderNameStyle) can be specified as such:
public static Name name(String... qualifiedName) { ... }

// QueryPart lists (e.g. IN-lists for the IN predicate) can be generated via these methods:
public static QueryPart list(QueryPart... parts) { ... }
public static QueryPart list(Collection<? extends QueryPart> parts) { ... }
```

3.18.8. Serializability

A lot of jOOQ types extend and implement the java.io.Serializable interface for your convenience. Beware, however, that jOOQ will make no guarantees related to the serialisation format, and its backwards compatible evolution. This means that while it is generally safe to rely on jOOQ types being serialisable when two processes using the exact same jOOQ version transfer jOOQ state over some network, it is not safe to rely on persisting serialised jOOQ state to be deserialised again at a later time - even after a patch release upgrade!

As always with Java's serialisation, if you want reliable serialisation of Java objects, please use your own serialisation protocol, or use one of the official export formats.

What types are serializable?

The only transient, non-serializable element in any jOOQ object is the Configuration's underlying java.sql.Connection. When you want to execute queries after de-serialisation, or when you want to store/refresh/delete Updatable Records, you may have to "re-attach" them to a Configuration

```
// Deserialise a SELECT statement
ObjectInputStream in = new ObjectInputStream(...);
Select<?> select = (Select<?>) in.readObject();

// This will throw a DetachedException:
select.execute();

// In order to execute the above select, attach it first
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);
create.attach(select);
```

Automatically attaching QueryParts

Another way of attaching QueryParts automatically, or rather providing them with a new java.sql.Connection at will, is to hook into the Execute Listener support. More details about this can be found in the manual's chapter about ExecuteListeners

3.18.9. Custom SQL transformation with VisitListener

With the org.jooq.VisitListener SPI, it is possible to perform custom SQL transformation to implement things like shared-schema multi-tenancy, or a security layer centrally preventing access to certain data. This SPI is extremely powerful, as you can make ad-hoc decisions at runtime regarding local or global

transformation of your SQL statement. The following sections show a couple of simple, yet real-world use-cases.

3.18.9.1. Example: Logging abbreviated bind values

When implementing a logger, one needs to carefully assess how much information should really be disclosed on what logger level. In log4j and similar frameworks, we distinguish between FATAL, ERROR, WARN, INFO, DEBUG, and TRACE. In DEBUG level, jOOQ's internal default logger logs all executed statements including inlined bind values as such:

```
Executing query          : select * from "BOOK" where "BOOK"."TITLE" like ?
-> with bind values      : select * from "BOOK" where "BOOK"."TITLE" like 'How I stopped worrying%'
```

But textual or binary bind values can get quite long, quickly filling your log files with irrelevant information. It would be good to be able to abbreviate such long values (and possibly add a remark to the logged statement). Instead of patching jOOQ's internals, we can just transform the SQL statements in the logger implementation, cleanly separating concerns. This can be done with the following VisitListener:

```
// This listener is inserted into a Configuration through a VisitListenerProvider that creates a
// new listener instance for every rendering lifecycle
public class BindValueAbbreviator extends DefaultVisitListener {

    private boolean anyAbbreviations = false;

    @Override
    public void visitStart(VisitContext context) {

        // Transform only when rendering values
        if (context.renderContext() != null) {
            QueryPart part = context.queryPart();

            // Consider only bind variables, leave other QueryParts untouched
            if (part instanceof Param<?>) {
                Param<?> param = (Param<?>) part;
                Object value = param.getValue();

                // If the bind value is a String (or Clob) of a given length, abbreviate it
                // e.g. using commons-lang's StringUtils.abbreviate()
                if (value instanceof String && ((String) value).length() > maxLength) {
                    anyAbbreviations = true;

                    // ... and replace it in the current rendering context (not in the Query)
                    context.queryPart(val(abbreviate((String) value, maxLength)));
                }

                // If the bind value is a byte[] (or Blob) of a given length, abbreviate it
                // e.g. by removing bytes from the array
                else if (value instanceof byte[] && ((byte[]) value).length > maxLength) {
                    anyAbbreviations = true;

                    // ... and replace it in the current rendering context (not in the Query)
                    context.queryPart(val(Arrays.copyOf((byte[]) value, maxLength)));
                }
            }
        }
    }

    @Override
    public void visitEnd(VisitContext context) {

        // If any abbreviations were performed before...
        if (anyAbbreviations) {

            // ... and if this is the top-level QueryPart, then append a SQL comment to indicate the abbreviation
            if (context.queryPartsLength() == 1) {
                context.renderContext().sql(" -- Bind values may have been abbreviated");
            }
        }
    }
}
```

If maxLength were set to 5, the above listener would produce the following log output:

```
Executing query          : select * from "BOOK" where "BOOK"."TITLE" like ?
-> with bind values      : select * from "BOOK" where "BOOK"."TITLE" like 'Ho...' -- Bind values may have been abbreviated
```

The above VisitListener is in place since jOOQ 3.3 in the org.jooq.tools.LoggerListener.

3.19. Zero-based vs one-based APIs

Any API that bridges two languages / mind sets, such as Java / SQL will inevitably face the difficulty of finding a consistent strategy to solving the "based-ness" problem. Should arrays be one-based or zero-based?

Clearly, Java is zero-based and SQL is one-based, and the best strategy for jOOQ is to keep things this way. The following are a set of rules that you should remember if this ever confuses you:

All SQL API is one-based

When using SQL API, such as the index-based ORDER BY clause, or window functions such as in the example below, jOOQ will not interpret indexes but send them directly as-is to the SQL engine. For instance:

```
SELECT nth_value(title, 3) OVER (ORDER BY id)
FROM book
ORDER BY 1
```

```
create.select(nthValue(BOOK.TITLE, 3).over(orderBy(BOOK.ID)))
      .from(BOOK)
      .orderBy(1).fetch();
```

In the above example, we're looking for the 3rd value of X in T ordered by Y. Clearly, this window function uses one-based indexing. The same is true for the ORDER BY clause, which orders the result by the 1st column - again one-based counting. There is no column zero in SQL.

All jOOQ API is zero-based

jOOQ is a Java API and as such, one-basedness would be quite surprising despite the fact that JDBC is one-based (see below). For instance, when you access a record by index in a jOOQ org.jooq.Result, given that the result extends java.util.List, you will use zero-based index access:

```
Result<?> result = create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(1)
      .fetch();

for (int i = 0; i < result.size(); i++)
    System.out.println(result.get(i));
```

Unlike in JDBC, where java.sql.ResultSet#absolute(int) positions the underlying cursor at the one-based index, we Java developers really don't like that way of thinking. As can be seen in the above loop, we iterate over this result as we do over any other Java collection.

All JDBC API is one-based

An exception to the above rule is, obviously, all jOOQ API that is JDBC-interfacing. E.g. when you implement a custom data type binding, you will work with JDBC API directly from within jOOQ, which is one-based.

3.20. SQL building in Scala

jOOQ-scala is a maven module used for leveraging some advanced Scala features for those users that wish to use jOOQ with Scala.

You can integrate it using the following dependency:

Maven

```
<dependency>
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>

  <!-- Scala 2.12 and 2.13 are also supported -->
  <artifactId>jooq-scala_3.5</artifactId>
  <version>3.12.4</version>
</dependency>
```

Gradle (Kotlin)

```
dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    // Scala 2.12 and 2.13 are also supported
    implementation("org.jooq:jooq-scala_3.5:3.12.4")
}
```

Gradle (Groovy)

Using Scala's implicit defs to allow for operator overloading

The most obvious Scala feature to use in jOOQ are implicit defs for implicit conversions in order to enhance the org.jooq.Field type with SQL-esque operators.

The following depicts a trait which wraps all fields:

```
/**
 * A Scala-esque representation of {@link org.jooq.Field}, adding overloaded
 * operators for common jOOQ operations to arbitrary fields
 */
trait SAnyField[T] extends Field[T] {

    // String operations
    // -----------------

    def ||(value : String)            : Field[String]
    def ||(value : Field[_])          : Field[String]

    // Comparison predicates
    // ---------------------

    def ===(value : T)                : Condition
    def ===(value : Field[T])         : Condition

    def !==(value : T)                : Condition
    def !==(value : Field[T])         : Condition

    def <>(value : T)                 : Condition
    def <>(value : Field[T])          : Condition

    def >(value : T)                  : Condition
    def >(value : Field[T])           : Condition

    def >=(value : T)                 : Condition
    def >=(value : Field[T])          : Condition

    def <(value : T)                  : Condition
    def <(value : Field[T])           : Condition

    def <=(value : T)                 : Condition
    def <=(value : Field[T])          : Condition

    def <=>(value : T)                : Condition
    def <=>(value : Field[T])         : Condition
}
```

The following depicts a trait which wraps numeric fields:

```
/**
 * A Scala-esque representation of {@link org.jooq.Field}, adding overloaded
 * operators for common jOOQ operations to numeric fields
 */
trait SNumberField[T <: Number] extends SAnyField[T] {

    // Arithmetic operations
    // ---------------------

    def unary_-                       : Field[T]

    def +(value : Number)             : Field[T]
    def +(value : Field[_ <: Number]) : Field[T]

    def -(value : Number)             : Field[T]
    def -(value : Field[_ <: Number]) : Field[T]

    def *(value : Number)             : Field[T]
    def *(value : Field[_ <: Number]) : Field[T]

    def /(value : Number)             : Field[T]
    def /(value : Field[_ <: Number]) : Field[T]

    def %(value : Number)             : Field[T]
    def %(value : Field[_ <: Number]) : Field[T]

    // Bitwise operations
    // ------------------

    def unary_~                       : Field[T]

    def &(value : T)                  : Field[T]
    def &(value : Field[T])           : Field[T]

    def |(value : T)                  : Field[T]
    def |(value : Field[T])           : Field[T]

    def ^(value : T)                  : Field[T]
    def ^(value : Field[T])           : Field[T]

    def <<(value : T)                 : Field[T]
    def <<(value : Field[T])          : Field[T]

    def >>(value : T)                 : Field[T]
    def >>(value : Field[T])          : Field[T]
}
```

An example query using such overloaded operators would then look like this:

```
select (
  BOOK.ID * BOOK.AUTHOR_ID,
  BOOK.ID + BOOK.AUTHOR_ID * 3 + 4,
  BOOK.TITLE || " abc" || " xy")
from BOOK
leftOuterJoin (
  select (x.ID, x.YEAR_OF_BIRTH)
  from x
  limit 1
  asTable x.getName()
)
on BOOK.AUTHOR_ID === x.ID
where (BOOK.ID <> 2)
or (BOOK.TITLE in ("O Alquimista", "Brida"))
fetch
```

Scala 2.10 Macros

This feature is still being experimented with. With Scala Macros, it might be possible to inline a true SQL dialect into the Scala syntax, backed by the jOOQ API. Stay tuned!

3.21. Compile time validation

Java 8 introduced JSR 308 (type annotations) and with it, the Checker Framework was born. The Checker Framework allows for implementing compiler plugins that run sophisticated checks on your Java AST to introduce rich annotation based type semantics, e.g.

```
// This still compiles
@Positive int value1 = 1;

// This no longer compiles:
@Positive int value2 = -1;
```

jOOQ has two annotations that are very interesting for the Checker Framework to type check, namely:

- org.jooq.Support: This annotation documents jOOQ DSL API with valuable information about which database supports a given SQL clause or function, etc. For instance, only Informix and Oracle currently support the CONNECT BY clause.
- org.jooq.PlainSQL: This annotation documents jOOQ DSL API which operates on plain SQL. Plain SQL being string-based SQL that is injected into a jOOQ expression tree, these API elements introduce a certain SQL injection risk (just like JDBC in general), if users are not careful.

Using the optional jooq-checker module (available only from Maven Central), users can now type-check their code to work only with a given set of dialects, or to forbid access to plain SQL.

Example:

A detailed blog post shows how this works in depth. By adding a simple dependency to your Maven build:

Maven

```
<dependency>
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-checker</artifactId>
  <version>3.12.4</version>
</dependency>
```

Gradle (Kotlin)

```
dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    implementation("org.jooq:jooq-checker:3.12.4")
}
```

Gradle (Groovy)
... you can now include one of the two checkers:

SQLDialectChecker

The SQLDialect checker reads all of the org.jooq.Allow and org.jooq.Require annotations in your source code and checks if the jOOQ API you're using is allowed and/or required in a given context, where that context can be any scope, including:

- A package
- A class
- A method

Configure this compiler plugin:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <fork>true</fork>
    <annotationProcessors>
      <annotationProcessor>org.jooq.checker.SQLDialectChecker</annotationProcessor>
    </annotationProcessors>
    <compilerArgs>
      <arg>-Xbootclasspath/p:1.8</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

... annotate your packages, e.g.

```
// Scope: entire package (put in package-info.java)
@Allow(ORACLE)
package org.jooq.example.checker;
```

And now, you'll no longer be able to use any SQL Server specific functionality that is not available in Oracle, for instance. Perfect!

There are quite some delicate rules that play into this when you nest these annotations. Please refer to this blog post for details.

PlainSQLChecker

This checker is much simpler. Just add the following compiler plugin to deactivate plain SQL usage by default:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <fork>true</fork>
    <annotationProcessors>
      <annotationProcessor>org.jooq.checker.PlainSQLChecker</annotationProcessor>
    </annotationProcessors>
    <compilerArgs>
      <arg>-Xbootclasspath/p:1.8</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

From now on, you won't risk any SQL injection in your jOOQ code anymore, because your compiler will reject all such API usage. If, however, you need to place an exception on a given package / class / method, simply add the org.jooq.Allow.PlainSQL annotation, as such:

```
// Scope: Single method.
@Allow.PlainSQL
public List<Integer> iKnowWhatImDoing() {
    return DSL.using(configuration)
              .select(level())
              .connectBy("level < ?", bindValue)
              .fetch(0, int.class);
}
```

The Checker Framework does add some significant overhead in terms of compilation speed, and its IDE tooling is not yet at a level where such checks can be fed into IDEs for real user feedback, but the framework does work pretty well if you integrate it in your CI, nightly builds, etc.

© 2009 - 2025 by Data Geekery™ GmbH.

4. SQL execution

In a previous section of the manual, we've seen how jOOQ can be used to build SQL that can be executed with any API including JDBC or ... jOOQ. This section of the manual deals with various means of actually executing SQL with jOOQ.

SQL execution with JDBC

JDBC calls executable objects "java.sql.Statement". It distinguishes between three types of statements:

- java.sql.Statement, or "static statement": This statement type is used for any arbitrary type of SQL statement. It is particularly useful with inlined parameters
- java.sql.PreparedStatement: This statement type is used for any arbitrary type of SQL statement. It is particularly useful with indexed parameters (note that JDBC does not support named parameters)
- java.sql.CallableStatement: This statement type is used for SQL statements that are "called" rather than "executed". In particular, this includes calls to stored procedures. Callable statements can register OUT parameters

Today, the JDBC API may look weird to users being used to object-oriented design. While statements hide a lot of SQL dialect-specific implementation details quite well, they assume a lot of knowledge about the internal state of a statement. For instance, you can use the PreparedStatement.addBatch() method, to add a the prepared statement being created to an "internal list" of batch statements. Instead of returning a new type, this method forces user to reflect on the prepared statement's internal state or "mode".

jOOQ is wrapping JDBC

These things are abstracted away by jOOQ, which exposes such concepts in a more object-oriented way. For more details about jOOQ's batch query execution, see the manual's section about batch execution.

The following sections of this manual will show how jOOQ is wrapping JDBC for SQL execution

Alternative execution modes

Just because you can, doesn't mean you must. At the end of this chapter, we'll show how you can use jOOQ to generate SQL statements that are then executed with other APIs, such as Spring's JdbcTemplate, or Hibernate. For more information see the section about alternative execution models.

© 2009 - 2025 by Data Geekery™ GmbH.

4.1. Comparison between jOOQ and JDBC

Similarities with JDBC

Even if there are two general types of Query, there are a lot of similarities between JDBC and jOOQ. Just to name a few:

- Both APIs return the number of affected records in non-result queries. JDBC: Statement.executeUpdate(), jOOQ: Query.execute()
- Both APIs return a scrollable result set type from result queries. JDBC: java.sql.ResultSet, jOOQ: org.jooq.Result

Differences to JDBC

Some of the most important differences between JDBC and jOOQ are listed here:

- Query vs. ResultQuery: JDBC does not formally distinguish between queries that can return results, and queries that cannot. The same API is used for both. This greatly reduces the possibility for fetching convenience methods
- Exception handling: While SQL uses the checked java.sql.SQLException, jOOQ wraps all exceptions in an unchecked org.jooq.exception.DataAccessException
- org.jooq.Result: Unlike its JDBC counter-part, this type implements java.util.List and is fully loaded into Java memory, freeing resources as early as possible. Just like statements, this means that users don't have to deal with a "weird" internal result set state.
- org.jooq.Cursor: If you want more fine-grained control over how many records are fetched into memory at once, you can still do that using jOOQ's lazy fetching feature
- Statement type: jOOQ does not formally distinguish between static statements and prepared statements. By default, all statements are prepared statements in jOOQ, internally. Executing a statement as a static statement can be done simply using a custom settings flag
- Closing Statements: JDBC keeps open resources even if they are already consumed. With JDBC, there is a lot of verbosity around safely closing resources. In jOOQ, resources are closed after consumption, by default. If you want to keep them open after consumption, you have to explicitly say so.
- JDBC flags: JDBC execution flags and modes are not modified. They can be set fluently on a Query
- Zero-based vs one-based APIs: JDBC is a one-based API, jOOQ is a zero-based API. While this makes sense intuitively (JDBC is the less intuitive API from a Java perspective), it can lead to confusion in certain cases.

4.2. Query vs. ResultQuery

Unlike JDBC, jOOQ has a lot of knowledge about a SQL query's structure and internals (see the manual's section about SQL building). Hence, jOOQ distinguishes between these two fundamental types of queries. While every org.jooq.Query can be executed, only org.jooq.ResultQuery can return results (see

the manual's section about fetching to learn more about fetching results). With plain SQL, the distinction can be made clear most easily:

```
// Create a Query object and execute it:
Query query = create.query("DELETE FROM BOOK");
query.execute();

// Create a ResultQuery object and execute it, fetching results:
ResultQuery<Record> resultQuery = create.resultQuery("SELECT * FROM BOOK");
Result<Record> result = resultQuery.fetch();
```

4.3. Fetching

Fetching is something that has been completely neglected by JDBC and also by various other database abstraction libraries. Fetching is much more than just looping or listing records or mapped objects. There are so many ways you may want to fetch data from a database, it should be considered a first-class feature of any database abstraction API. Just to name a few, here are some of jOOQ's fetching modes:

- Untyped vs. typed fetching: Sometimes you care about the returned type of your records, sometimes (with arbitrary projections) you don't.
- Fetching arrays, maps, or lists: Instead of letting you transform your result sets into any more suitable data type, a library should do that work for you.
- Fetching through mapper callbacks: This is an entirely different fetching paradigm. With Java 8's lambda expressions, this will become even more powerful.
- Fetching custom POJOs: This is what made Hibernate and JPA so strong. Automatic mapping of tables to custom POJOs.
- Lazy vs. eager fetching: It should be easy to distinguish these two fetch modes.
- Fetching many results: Some databases allow for returning many result sets from a single query. JDBC can handle this but it's very verbose. A list of results should be returned instead.
- Fetching data asynchronously: Some queries take too long to execute to wait for their results. You should be able to spawn query execution in a separate process.
- Fetching data reactively: In a reactive programming model, you will want to fetch results from a publisher into a subscription. jOOQ implements different Publisher APIs.

Convenience and how ResultQuery, Result, and Record share API

The term "fetch" is always reused in jOOQ when you can fetch data from the database. An org.jooq.ResultQuery provides many overloaded means of fetching data:

Various modes of fetching

These modes of fetching are also documented in subsequent sections of the manual

```
// The "standard" fetch
Result<R> fetch();

// The "standard" fetch when you know your query returns at most one record. This may return null.
R fetchOne();

// The "standard" fetch when you know your query returns exactly one record. This never returns null.
R fetchSingle();

// The "standard" fetch when you know your query returns at most one record.
Optional<R> fetchOptional();

// The "standard" fetch when you only want to fetch the first record
R fetchAny();

// Create a "lazy" Cursor, that keeps an open underlying JDBC ResultSet
Cursor<R> fetchLazy();
Cursor<R> fetchLazy(int fetchSize);
Stream<R> stream();

// Fetch several results at once
List<Result<Record>> fetchMany();

// Fetch records into a custom callback
<H extends RecordHandler<R>> H fetchInto(H handler);

// Map records using a custom callback
<E> List<E> fetch(RecordMapper<? super R, E> mapper);

// Execute a ResultQuery with jOOQ, but return a JDBC ResultSet, not a jOOQ object
ResultSet fetchResultSet();
```

Fetch convenience

These means of fetching are also available from org.jooq.Result and org.jooq.Record APIs

```
// These methods are convenience for fetching only a single field,
// possibly converting results to another type
<T>    List<T> fetch(Field<T> field);
<T>    List<T> fetch(Field<?> field, Class<? extends T> type);
<T, U> List<U> fetch(Field<T> field, Converter<? super T, U> converter);
       List<?> fetch(int fieldIndex);
<T>    List<T> fetch(int fieldIndex, Class<? extends T> type);
<U>    List<U> fetch(int fieldIndex, Converter<?, U> converter);
       List<?> fetch(String fieldName);
<T>    List<T> fetch(String fieldName, Class<? extends T> type);
<U>    List<U> fetch(String fieldName, Converter<?, U> converter);

// These methods are convenience for fetching only a single field, possibly converting results to another type
// Instead of returning lists, these return arrays
<T>    T[]      fetchArray(Field<T> field);
<T>    T[]      fetchArray(Field<?> field, Class<? extends T> type);
<T, U> U[]      fetchArray(Field<T> field, Converter<? super T, U> converter);
       Object[] fetchArray(int fieldIndex);
<T>    T[]      fetchArray(int fieldIndex, Class<? extends T> type);
<U>    U[]      fetchArray(int fieldIndex, Converter<?, U> converter);
       Object[] fetchArray(String fieldName);
<T>    T[]      fetchArray(String fieldName, Class<? extends T> type);
<U>    U[]      fetchArray(String fieldName, Converter<?, U> converter);

// These methods are convenience for fetching only a single field from a single record,
// possibly converting results to another type
<T>    T      fetchOne(Field<T> field);
<T>    T      fetchOne(Field<?> field, Class<? extends T> type);
<T, U> U      fetchOne(Field<T> field, Converter<? super T, U> converter);
       Object fetchOne(int fieldIndex);
<T>    T      fetchOne(int fieldIndex, Class<? extends T> type);
<U>    U      fetchOne(int fieldIndex, Converter<?, U> converter);
       Object fetchOne(String fieldName);
<T>    T      fetchOne(String fieldName, Class<? extends T> type);
<U>    U      fetchOne(String fieldName, Converter<?, U> converter);
```

Fetch transformations

These means of fetching are also available from org.jooq.Result and org.jooq.Record APIs

```
// Transform your Records into arrays, Results into matrices
       Object[][] fetchArrays();
       Object[]   fetchOneArray();

// Reduce your Result object into maps
<K>    Map<K, R>      fetchMap(Field<K> key);
<K, V> Map<K, V>      fetchMap(Field<K> key, Field<V> value);
<K, E> Map<K, E>      fetchMap(Field<K> key, Class<E> value);
       Map<Record, R> fetchMap(Field<?>[] key);
<E>    Map<Record, E> fetchMap(Field<?>[] key, Class<E> value);

// Transform your Result object into maps
       List<Map<String, Object>> fetchMaps();
       Map<String, Object>       fetchOneMap();

// Transform your Result object into groups
<K>    Map<K, Result<R>>      fetchGroups(Field<K> key);
<K, V> Map<K, List<V>>        fetchGroups(Field<K> key, Field<V> value);
<K, E> Map<K, List<E>>        fetchGroups(Field<K> key, Class<E> value);
       Map<Record, Result<R>> fetchGroups(Field<?>[] key);
<E>    Map<Record, List<E>>   fetchGroups(Field<?>[] key, Class<E> value);

// Transform your Records into custom POJOs
<E>    List<E> fetchInto(Class<? extends E> type);

// Transform your records into another table type
<Z extends Record> Result<Z> fetchInto(Table<Z> table);
```

Note, that apart from the fetchLazy() methods, all fetch() methods will immediately close underlying JDBC result sets.

4.3.1. Record vs. TableRecord

jOOQ understands that SQL is much more expressive than Java, when it comes to the declarative typing of table expressions. As a declarative language, SQL allows for creating ad-hoc row value expressions (records with indexed columns, or tuples) and records (records with named columns). In Java, this is not possible to the same extent.

Yet, still, sometimes you wish to use strongly typed records, when you know that you're selecting only from a single table:

Fetching strongly or weakly typed records

When fetching data only from a single table, the table expression's type is known to jOOQ if you use jOOQ's code generator to generate TableRecords for your database tables. In order to fetch such strongly typed records, you will have to use the simple select API:

```
// Use the selectFrom() method:
BookRecord book = create.selectFrom(BOOK).where(BOOK.ID.eq(1)).fetchOne();

// Typesafe field access is now possible:
System.out.println("Title       : " + book.getTitle());
System.out.println("Published in: " + book.getPublishedIn());
```

When you use the DSLContext.selectFrom() method, jOOQ will return the record type supplied with the argument table. Beware though, that you will no longer be able to use any clause that modifies the type of your table expression. This includes:

- The SELECT clause
- The JOIN clause

Mapping custom row types to strongly typed records

Sometimes, you may want to explicitly select only a subset of your columns, but still use strongly typed records. Alternatively, you may want to join a one-to-one relationship and receive the two individual strongly typed records after the join.

In both of the above cases, you can map your org.jooq.Record "into" a org.jooq.TableRecord type by using Record.into(Table).

```
// Join two tables
Record record = create.select()
                      .from(BOOK)
                      .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                      .where(BOOK.ID.eq(1))
                      .fetchOne();

// "extract" the two individual strongly typed TableRecord types from the denormalised Record:
BookRecord book = record.into(BOOK);
AuthorRecord author = record.into(AUTHOR);

// Typesafe field access is now possible:
System.out.println("Title       : " + book.getTitle());
System.out.println("Published in: " + book.getPublishedIn());
System.out.println("Author      : " + author.getFirstName() + " " + author.getLastName();
```

4.3.2. Record1 to Record22

jOOQ's row value expression (or tuple) support has been explained earlier in this manual. It is useful for constructing row value expressions where they can be used in SQL. The same typesafety is also applied to records for degrees up to 22. To express this fact, org.jooq.Record is extended by org.jooq.Record1 to org.jooq.Record22. Apart from the fact that these extensions of the R type can be used throughout the jOOQ DSL, they also provide a useful API. Here is org.jooq.Record2, for instance:

```
public interface Record2<T1, T2> extends Record {

    // Access fields and values as row value expressions
    Row2<T1, T2> fieldsRow();
    Row2<T1, T2> valuesRow();

    // Access fields by index
    Field<T1> field1();
    Field<T2> field2();

    // Access values by index
    T1 value1();
    T2 value2();
}
```

*(!) **Generated records** can be made to implement Record[N] types using the <recordsImplementingRecordN/>. Because this may have significant effects on compilation speeds, the flag's default value has been changed to false in jOOQ 3.19.*

Higher-degree records

jOOQ chose to explicitly support degrees up to 22 to match Scala's typesafe tuple, function and product support. Unlike Scala, however, jOOQ also supports higher degrees without the additional typesafety.

4.3.3. Arrays, Maps and Lists

By default, jOOQ returns an org.jooq.Result object, which is essentially a java.util.List of org.jooq.Record. Often, you will find yourself wanting to transform this result object into a type that corresponds more to your specific needs. Or you just want to list all values of one specific column. Here are some examples to illustrate those use cases:

```
// Fetching only book titles (the two calls are equivalent):
List<String> titles1 = create.select().from(BOOK).fetch().getValues(BOOK.TITLE);
List<String> titles2 = create.select().from(BOOK).fetch(BOOK.TITLE);
String[]     titles3 = create.select().from(BOOK).fetchArray(BOOK.TITLE);

// Fetching only book IDs, converted to Long
List<Long> ids1 = create.select().from(BOOK).fetch().getValues(BOOK.ID, Long.class);
List<Long> ids2 = create.select().from(BOOK).fetch(BOOK.ID, Long.class);
Long[]     ids3 = create.select().from(BOOK).fetchArray(BOOK.ID, Long.class);

// Fetching book IDs and mapping each ID to their records or titles
Map<Integer, BookRecord> map1 = create.selectFrom(BOOK).fetch().intoMap(BOOK.ID);
Map<Integer, BookRecord> map2 = create.selectFrom(BOOK).fetchMap(BOOK.ID);
Map<Integer, String>     map3 = create.selectFrom(BOOK).fetch().intoMap(BOOK.ID, BOOK.TITLE);
Map<Integer, String>     map4 = create.selectFrom(BOOK).fetchMap(BOOK.ID, BOOK.TITLE);

// Group by AUTHOR_ID and list all books written by any author:
Map<Integer, Result<BookRecord>> group1 = create.selectFrom(BOOK).fetch().intoGroups(BOOK.AUTHOR_ID);
Map<Integer, Result<BookRecord>> group2 = create.selectFrom(BOOK).fetchGroups(BOOK.AUTHOR_ID);
Map<Integer, List<String>>       group3 = create.selectFrom(BOOK).fetch().intoGroups(BOOK.AUTHOR_ID, BOOK.TITLE);
Map<Integer, List<String>>       group4 = create.selectFrom(BOOK).fetchGroups(BOOK.AUTHOR_ID, BOOK.TITLE);
```

Note that most of these convenience methods are available both through org.jooq.ResultQuery and org.jooq.Result, some are even available through org.jooq.Record as well.

4.3.4. ResultQuery as Iterable

The org.jooq.ResultQuery type extends java.lang.Iterable, and as such, inherits the JDK's Iterable.forEach() method, as well as it being a possible foreach loop source:

```
// External iteration, fetching records eagerly
for (var r : create.select(BOOK.TITLE).from(BOOK))
    System.out.println(r.get(BOOK.TITLE));

// Internal iteration, fetching records lazily
create.select(BOOK.TITLE).from(BOOK).forEach(r -> {
    System.out.println(r.get(BOOK.TITLE));
});
```

*(!) Note that unlike when **lazy fetching** using the **org.jooq.Cursor** API, the external iteration style will eager-fetch the entire intermediate **org.jooq.Result** into memory as jOOQ has no control over the lifecycle of the **java.util.Iterator**, and the JDK doesn't know an **java.lang.AutoCloseable** iterator.*

4.3.5. RecordMapper

In a more functional operating mode, you might want to write callbacks that map records from your select statement results in order to do some processing. This is a common data access pattern in Spring's JdbcTemplate, and it is also available in jOOQ. With jOOQ, you can implement your own org.jooq.RecordMapper classes and plug them into jOOQ's org.jooq.ResultQuery:

```
// Write callbacks to receive records from select statements
List<Integer> ids =
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch()
      .map(BookRecord::getId);

// Or more concisely, as fetch().map(mapper) can be written as fetch(mapper):
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(BookRecord::getId);

// Or using a lambda expression:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(book -> book.getId());

// Of course, the lambda could be expanded into the following anonymous RecordMapper:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(new RecordMapper<BookRecord, Integer>() {
          @Override
          public Integer map(BookRecord book) {
              return book.getId();
          }
      });
```

Your custom RecordMapper types can be used automatically through jOOQ's POJO mapping APIs, by injecting a RecordMapperProvider into your Configuration.

4.3.6. POJOs

Fetching data in records is fine as long as your application is not really layered, or as long as you're still writing code in the DAO layer. But if you have a more advanced application architecture, you may not want to allow for jOOQ artefacts to leak into other layers. You may choose to write POJOs (Plain Old Java Objects) as your primary DTOs (Data Transfer Objects), without any dependencies on jOOQ's org.jooq.Record types, which may even potentially hold a reference to a Configuration, and thus a JDBC java.sql.Connection. Like Hibernate/JPA, jOOQ allows you to operate with POJOs. Unlike Hibernate/JPA, jOOQ does not "attach" those POJOs or create proxies with any magic in them.

If you're using jOOQ's code generator, you can configure it to generate POJOs for you, but you're not required to use those generated POJOs. You can use your own. See the manual's section about POJOs with custom RecordMappers to see how to modify jOOQ's standard POJO mapping behaviour.

> *(!) While POJOs don't have a connection to jOOQ API, they also don't have any notion of dirty flags and other useful record-based features. They're dumb POJOs. While some people like "clean" separation of concerns, this really isn't always necessary. It's perfectly fine to be pragmatic and mostly work with jOOQ's **org.jooq.UpdatableRecord**, even in the UI layer!*

equals() and hashCode()

Generated POJOs provide entirely value-based equals() and hashCode() semantics out of the box, as jOOQ operates on data in a similar way as SQL does, where rows are "equal" when they're "NOT DISTINCT." Custom built POJOs or DTOs can have different opinions about the implementation of equals() and hashCode(), including approaches where POJOs are more similar to JPA entities (if primary key present: use that only for comparison, if absent, all instances are distinct). jOOQ doesn't impose one approach over the other.

Using JPA-annotated POJOs

jOOQ tries to find JPA annotations on your POJO types. If it finds any, they are used as the primary source for mapping meta-information. Only the jakarta.persistence.Column annotation is used and understood by jOOQ. An example:

```
// A JPA-annotated POJO class
public class MyBook {
  @Column(name = "ID")
  public int myId;

  @Column(name = "TITLE")
  public String myTitle;
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook myBook        = create.select().from(BOOK).fetchAny().into(MyBook.class);
List<MyBook> myBooks = create.select().from(BOOK).fetch().into(MyBook.class);
List<MyBook> myBooks = create.select().from(BOOK).fetchInto(MyBook.class);
```

Just as with any other JPA implementation, you can put the jakarta.persistence.Column annotation on any class member, including attributes, setters and getters. Please refer to the Record.into() Javadoc for more details.

Using simple POJOs

If jOOQ does not find any JPA-annotations, columns are mapped to the "best-matching" constructor, attribute or setter. An example illustrates this:

```
// A "mutable" POJO class
public class MyBook1 {
  public int id;
  public String title;
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook1 myBook        = create.select().from(BOOK).fetchAny().into(MyBook1.class);
List<MyBook1> myBooks = create.select().from(BOOK).fetch().into(MyBook1.class);
List<MyBook1> myBooks = create.select().from(BOOK).fetchInto(MyBook1.class);
```

Please refer to the Record.into() Javadoc for more details.

Using "immutable" POJOs

If jOOQ does not find any default constructor, columns are mapped to the "best-matching" constructor. This allows for using "immutable" POJOs with jOOQ. An example illustrates this:

```
// An "immutable" POJO class
public class MyBook2 {
  public final int id;
  public final String title;

  public MyBook2(int id, String title) {
    this.id = id;
    this.title = title;
  }
}

// With "immutable" POJO classes, there must be an exact match between projected fields and available constructors:
MyBook2 myBook        = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchAny().into(MyBook2.class);
List<MyBook2> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetch().into(MyBook2.class);
List<MyBook2> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchInto(MyBook2.class);

// An "immutable" POJO class with a java.beans.ConstructorProperties annotation
public class MyBook3 {
  public final String title;
  public final int id;

  @ConstructorProperties({ "title", "id" })
  public MyBook3(String title, int id) {
    this.title = title;
    this.id = id;
  }
}

// With annotated "immutable" POJO classes, there doesn't need to be an exact match between fields and constructor arguments.
// In the below cases, only BOOK.ID is really set onto the POJO, BOOK.TITLE remains null and BOOK.AUTHOR_ID is ignored
MyBook3 myBook        = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetchAny().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetch().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetchInto(MyBook3.class);
```

Please refer to the Record.into() Javadoc for more details.

Using proxyable types

jOOQ also allows for fetching data into abstract classes or interfaces, or in other words, "proxyable" types. This means that jOOQ will return a java.util.HashMap wrapped in a java.lang.reflect.Proxy implementing your custom type. An example of this is given here:

```
// A "proxyable" type
public interface MyBook3 {
  int getId();
  void setId(int id);

  String getTitle();
  void setTitle(String title);
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook3 myBook        = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchAny().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetch().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchInto(MyBook3.class);
```

Please refer to the Record.into() Javadoc for more details.

Loading POJOs back into Records to store them

The above examples show how to fetch data into your own custom POJOs / DTOs. When you have modified the data contained in POJOs, you probably want to store those modifications back to the database. An example of this is given here:

```
// A "mutable" POJO class
public class MyBook {
  public int id;
  public String title;
}

// Create a new POJO instance
MyBook myBook = new MyBook();
myBook.id = 10;
myBook.title = "Animal Farm";

// Load a jOOQ-generated BookRecord from your POJO
BookRecord book = create.newRecord(BOOK, myBook);

// Insert it (implicitly)
book.store();

// Insert it (explicitly)
create.executeInsert(book);

// or update it (ID = 10)
create.executeUpdate(book);
```

Note: Because of your manual setting of ID = 10, jOOQ's store() method will asume that you want to insert a new record. See the manual's section about CRUD with UpdatableRecords for more details on this.

Interaction with DAOs

If you're using jOOQ's code generator, you can configure it to generate DAOs for you. Those DAOs operate on generated POJOs. An example of using such a DAO is given here:

```
// Initialise a Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(SQLDialect.ORACLE);

// Initialise the DAO with the Configuration
BookDao bookDao = new BookDao(configuration);

// Start using the DAO
Book book = bookDao.findById(5);

// Modify and update the POJO
book.setTitle("1984");
book.setPublishedIn(1948);
bookDao.update(book);

// Delete it again
bookDao.delete(book);
```

*(!) While these **org.jooq.DAO** types look useful for trivial operations, they quickly become less interesting as your SQL interactions grow more complex. Just like POJOs themselves, DAOs are very dumb and simple. It's perfectly fine to work with **org.jooq.UpdatableRecord** directly, or with SQL statements, instead!*

More complex data structures

jOOQ currently doesn't support more complex data structures, the way Hibernate/JPA attempt to map relational data onto POJOs. While future developments in this direction are not excluded, jOOQ claims that generic mapping strategies lead to an enormous additional complexity that only serves very few use cases. You are likely to find a solution using any of jOOQ's various fetching modes, with only little boiler-plate code on the client side.

4.3.7. RecordMapperProvider

In the previous sections we have seen how to create RecordMapper types to map jOOQ records onto arbitrary objects. We have also seen how jOOQ provides default algorithms to map jOOQ records onto POJOs. Your own custom domain model might be much more complex, but you want to avoid looking up the most appropriate RecordMapper every time you need one. For this, you can provide jOOQ's Configuration with your own implementation of the org.jooq.RecordMapperProvider interface. An example is given here:

```
DSL.using(new DefaultConfiguration()
   .set(connection)
   .set(SQLDialect.ORACLE)
   .set(
       new RecordMapperProvider() {
           @Override
           public <R extends Record, E> RecordMapper<R, E> provide(RecordType<R> recordType, Class<? extends E> type) {

               // UUID mappers will always try to find the ID column
               if (type == UUID.class) {
                   return new RecordMapper<R, E>() {
                       @Override
                       public E map(R record) {
                           return (E) record.getValue("ID");
                       }
                   }
               }

               // Books might be joined with their authors, create a 1:1 mapping
               if (type == Book.class) {
                   return new BookMapper();
               }

               // Fall back to jOOQ's DefaultRecordMapper, which maps records onto
               // POJOs using reflection.
               return new DefaultRecordMapper(recordType, type);
           }
       }
   ))
   .selectFrom(BOOK)
   .orderBy(BOOK.ID)
   .fetchInto(UUID.class);
```

The above is a very simple example showing that you will have complete flexibility in how to override jOOQ's record to POJO mapping mechanisms.

Using third party libraries

A couple of useful libraries exist out there, which implement custom, more generic mapping algorithms. Some of them have been specifically made to work with jOOQ. Among them are:

- ModelMapper (with an explicit jOOQ integration)
- SimpleFlatMapper (with an explicit jOOQ integration)
- Orika Mapper (without explicit jOOQ integration)

4.3.8. Lazy fetching

Unlike JDBC's java.sql.ResultSet, jOOQ's org.jooq.Result does not represent an open database cursor with various fetch modes and scroll modes, that needs to be closed after usage. jOOQ's results are simple in-memory Java java.util.List objects, containing all of the result values. If your result sets are large, or if you have a lot of network latency, you may wish to fetch records one-by-one, or in small

chunks. jOOQ supports a org.jooq.Cursor type for that purpose. In order to obtain such a reference, use the ResultQuery.fetchLazy() method. An example is given here:

```
// Obtain a Cursor reference:
try (Cursor<BookRecord> cursor = create.selectFrom(BOOK).fetchLazy()) {

    // Cursor has similar methods as Iterator<R>
    while (cursor.hasNext()) {
        BookRecord book = cursor.fetchOne();

        Util.doThingsWithBook(book);
    }
}
```

As a org.jooq.Cursor holds an internal reference to an open java.sql.ResultSet, it may need to be closed at the end of iteration. If a cursor is completely scrolled through, it will conveniently close the underlying ResultSet. However, you should not rely on that.

Fetch sizes

While using a Cursor prevents jOOQ from eager fetching all data into memory, your underlying JDBC driver may still do that. To configure a fetch size in your JDBC driver, use ResultQuery.fetchSize(int), which specifies the JDBC Statement.setFetchSize(int) when executing the query. Please refer to your JDBC driver manual to learn about fetch sizes and their possible defaults and limitations.

Cursors ship with all the other fetch features

Like org.jooq.ResultQuery or org.jooq.Result, org.jooq.Cursor gives access to all of the other fetch features that we've seen so far, i.e.

- Strongly or weakly typed records: Cursors are also typed with the <R> type, allowing to fetch custom, generated org.jooq.TableRecord or plain org.jooq.Record types.
- RecordMapper callbacks: You can use your own org.jooq.RecordMapper callbacks to map lazily fetched records.
- POJOs: You can fetch data into your own custom POJO types.

4.3.9. Lazy fetching with Streams

jOOQ 3.7+ supports Java 8, and with Java 8, it supports java.util.stream.Stream. This opens up a range of possibilities of combining the declarative aspects of SQL with the functional aspects of the new Stream API. Much like the Cursors from the previous section, such a Stream keeps an internal reference to a JDBC java.sql.ResultSet, which means that the Stream has to be treated like a resource. Here's an example of using such a stream:

```
// Obtain a Stream reference:
try (Stream<BookRecord> stream = create.selectFrom(BOOK).stream()) {
    stream.forEach(Util::doThingsWithBook);
}
```

A more sophisticated example would be using streams to transform the results and add business logic to it. For instance, to generate a DDL script with CREATE TABLE statements from the INFORMATION_SCHEMA of an H2 database:

```
create.select(
          COLUMNS.TABLE_NAME,
          COLUMNS.COLUMN_NAME,
          COLUMNS.TYPE_NAME)
      .from(COLUMNS)
      .orderBy(
          COLUMNS.TABLE_CATALOG,
          COLUMNS.TABLE_SCHEMA,
          COLUMNS.TABLE_NAME,
          COLUMNS.ORDINAL_POSITION)
      .fetch() // Eagerly load the whole ResultSet into memory first
      .stream()
      .collect(groupingBy(
          r -> r.getValue(COLUMNS.TABLE_NAME),
          LinkedHashMap::new,
          mapping(
              r -> new SimpleEntry(
                  r.getValue(COLUMNS.COLUMN_NAME),
                  r.getValue(COLUMNS.TYPE_NAME)
              ),
              toList()
          )))
      .forEach(
          (table, columns) -> {
              // Just emit a CREATE TABLE statement
              System.out.println("CREATE TABLE " + table + " (");

              // Map each "Column" type into a String containing the column specification,
              // and join them using comma and newline. Done!
              System.out.println(
                  columns.stream()
                         .map(col -> "  " + col.getKey() +
                                     " " + col.getValue())
                         .collect(Collectors.joining(",\n"))
              );

              System.out.println(");");
          });
```

The above combination of SQL and functional programming will produce the following output:

```
CREATE TABLE CATALOGS(
  CATALOG_NAME VARCHAR
);
CREATE TABLE COLLATIONS(
  NAME VARCHAR,
  KEY VARCHAR
);
CREATE TABLE COLUMNS(
  TABLE_CATALOG VARCHAR,
  TABLE_SCHEMA VARCHAR,
  TABLE_NAME VARCHAR,
  COLUMN_NAME VARCHAR,
  ORDINAL_POSITION INTEGER,
  COLUMN_DEFAULT VARCHAR,
  IS_NULLABLE VARCHAR,
  DATA_TYPE INTEGER,
  CHARACTER_MAXIMUM_LENGTH INTEGER,
  CHARACTER_OCTET_LENGTH INTEGER,
  NUMERIC_PRECISION INTEGER,
  NUMERIC_PRECISION_RADIX INTEGER,
  NUMERIC_SCALE INTEGER,
  CHARACTER_SET_NAME VARCHAR,
  COLLATION_NAME VARCHAR,
  TYPE_NAME VARCHAR,
  NULLABLE INTEGER,
  IS_COMPUTED BOOLEAN,
  SELECTIVITY INTEGER,
  CHECK_CONSTRAINT VARCHAR,
  SEQUENCE_NAME VARCHAR,
  REMARKS VARCHAR,
  SOURCE_DATA_TYPE SMALLINT
);
```

4.3.10. Many fetching

Many databases support returning several result sets, or cursors, from single queries. An example for this is Sybase ASE's sp_help command:

```
> sp_help 'author'

+--------+-----+-----------+-------------+-------------------+
|Name    |Owner|Object_type|Object_status|Create_date        |
+--------+-----+-----------+-------------+-------------------+
|  author|dbo  |user table | -- none --  |Sep 22 2011 11:20PM|
+--------+-----+-----------+-------------+-------------------+

+-------------+-------+------+----+-----+-----+
|Column_name  |Type   |Length|Prec|Scale|...  |
+-------------+-------+------+----+-----+-----+
|id           |int    |     4|NULL| NULL|    0|
|first_name   |varchar|    50|NULL| NULL|    1|
|last_name    |varchar|    50|NULL| NULL|    0|
|date_of_birth|date   |     4|NULL| NULL|    1|
|year_of_birth|int    |     4|NULL| NULL|    1|
+-------------+-------+------+----+-----+-----+
```

The correct (and verbose) way to do this with JDBC is as follows:

```
ResultSet rs = statement.executeQuery();

// Repeat until there are no more result sets
for (;;) {

  // Empty the current result set
  while (rs.next()) {
    // [ .. do something with it .. ]
  }

  // Get the next result set, if available
  if (statement.getMoreResults()) {
    rs = statement.getResultSet();
  }
  else {
    break;
  }
}

// Be sure that all result sets are closed
statement.getMoreResults(Statement.CLOSE_ALL_RESULTS);
statement.close();
```

As previously discussed in the chapter about differences between jOOQ and JDBC, jOOQ does not rely on an internal state of any JDBC object, which is "externalised" by Javadoc. Instead, it has a straightforward API allowing you to do the above in a one-liner:

```
// Get some information about the author table, its columns, keys, indexes, etc
Results results = create.fetchMany("sp_help 'author'");
```

The returned org.jooq.Results type extends the List<Result<Record>> type for backwards-compatibility reasons, but it also allows to access individual update counts that may have been returned by the database in between result sets.

4.3.11. Later fetching

Using Java 8 CompletableFutures

Java 8 has introduced the new java.util.concurrent.CompletableFuture type, which allows for functional composition of asynchronous execution units. When applying this to SQL and jOOQ, you might be writing code as follows:

```
// Initiate an asynchronous call chain
CompletableFuture

    // This lambda will supply an int value indicating the number of inserted rows
    .supplyAsync(() ->
        DSL.using(configuration)
           .insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
           .values(3, "Hitchcock")
           .execute()
    )

    // This will supply an AuthorRecord value for the newly inserted author
    .handleAsync((rows, throwable) ->
        DSL.using(configuration)
           .fetchOne(AUTHOR, AUTHOR.ID.eq(3))
    )

    // This should supply an int value indicating the number of rows,
    // but in fact it'll throw a constraint violation exception
    .handleAsync((record, throwable) -> {
        record.changed(true);
        return record.insert();
    })

    // This will supply an int value indicating the number of deleted rows
    .handleAsync((rows, throwable) ->
        DSL.using(configuration)
           .delete(AUTHOR)
           .where(AUTHOR.ID.eq(3))
           .execute()
    )
    .join();
```

The above example will execute four actions one after the other, but asynchronously in the JDK's default or common java.util.concurrent.ForkJoinPool.

For more information, please refer to the java.util.concurrent.CompletableFuture Javadoc and official documentation.

Using deprecated API

Some queries take very long to execute, yet they are not crucial for the continuation of the main program. For instance, you could be generating a complicated report in a Swing application, and while this report is being calculated in your database, you want to display a background progress bar, allowing the user to pursue some other work. This can be achived simply with jOOQ, by creating a org.jooq.FutureResult, a type that extends java.util.concurrent.Future. An example is given here:

```
// Spawn off this query in a separate process:
FutureResult<BookRecord> future = create.selectFrom(BOOK).where(... complex predicates ...).fetchLater();

// This example actively waits for the result to be done
while (!future.isDone()) {
    progressBar.increment(1);
    Thread.sleep(50);
}

// The result should be ready, now
Result<BookRecord> result = future.get();
```

Note, that instead of letting jOOQ spawn a new thread, you can also provide jOOQ with your own java.util.concurrent.ExecutorService:

```
// Spawn off this query in a separate process:
ExecutorService service = // [...]
FutureResult<BookRecord> future = create.selectFrom(BOOK).where(... complex predicates ...).fetchLater(service);
```

© 2009 - 2025 by Data Geekery™ GmbH.

4.3.12. Reactive Fetching

In a reactive programming model, a query will not be executed eagerly and blocking the current thread. Instead a query implements a Publisher API, such as JDK 9's java.util.concurrent.Flow.Publisher or the Reactive Streams Publisher API.

When using a third party reactive stream API like project reactor, jOOQ queries can easily be embedded in a Flux or Mono type, such as:

```
List<String> authors =
Flux.from(create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
                .from(AUTHOR))
                .map(r -> r.get(AUTHOR.FIRST_NAME) + " " + r.get(AUTHOR.LAST_NAME))
                .collectList()
                .block();
```

Note that the current versions of jOOQ will still bind to the blocking JDBC API behind the scenes, when executing the above publisher. Future versions of jOOQ might channel such query executions e.g. through Spring's R2DBC.

4.3.13. ResultSet fetching

When interacting with legacy applications, you may prefer to have jOOQ return a java.sql.ResultSet, rather than jOOQ's own org.jooq.Result types. This can be done simply, in two ways:

```
try (

    // jOOQ's Cursor type exposes the underlying ResultSet:
    ResultSet rs1 = create.selectFrom(BOOK).fetchLazy().resultSet();

    // But you can also directly access that ResultSet from ResultQuery:
    ResultSet rs2 = create.selectFrom(BOOK).fetchResultSet()) {

// ...
}
```

Transform jOOQ's Result into a JDBC ResultSet

Instead of operating on a JDBC ResultSet holding an open resource from your database, you can also let jOOQ's org.jooq.Result wrap itself in a java.sql.ResultSet. The advantage of this is that the so-created ResultSet has no open connection to the database. It is a completely in-memory ResultSet:

```
// Transform a jOOQ Result into a ResultSet
Result<BookRecord> result = create.selectFrom(BOOK).fetch();
ResultSet rs = result.intoResultSet();
```

The inverse: Fetch data from a legacy ResultSet using jOOQ

The inverse of the above is possible too. Maybe, a legacy part of your application produces JDBC java.sql.ResultSet, and you want to turn them into a org.jooq.Result:

```
// Transform a JDBC ResultSet into a jOOQ Result
ResultSet rs = connection.createStatement().executeQuery("SELECT * FROM BOOK");

// As a Result:
Result<Record> result = create.fetch(rs);

// As a Cursor
Cursor<Record> cursor = create.fetchLazy(rs);
```

You can also tighten the interaction with jOOQ's data type system and data type conversion features, by passing the record type to the above fetch methods:

```
// Pass an array of types:
Result<Record> result = create.fetch    (rs, Integer.class, String.class);
Cursor<Record> result = create.fetchLazy(rs, Integer.class, String.class);

// Pass an array of data types:
Result<Record> result = create.fetch    (rs, INTEGER, VARCHAR);
Cursor<Record> result = create.fetchLazy(rs, INTEGER, VARCHAR);

// Pass an array of fields:
Result<Record> result = create.fetch    (rs, BOOK.ID, BOOK.TITLE);
Cursor<Record> result = create.fetchLazy(rs, BOOK.ID, BOOK.TITLE);
```

If supplied, the additional information is used to override the information obtained from the ResultSet's java.sql.ResultSetMetaData information.

4.3.14. Auto data type conversion

Many native SQL data types can be automatically converted from one another, such as VARCHAR to INTEGER and vice versa.

The jOOQ API also supports a variety of such auto conversions through the org.jooq.tools.Convert utility API, which implements the following rules:

- null is always converted to null, or the primitive default value, or Optional.empty(), regardless of the target type.
- Identity conversion (converting a value to its own type) is always possible.
- Primitive types can be converted to their wrapper types and vice versa
- All types can be converted to String
- All types can be converted to Object
- All Number types can be converted to other Number types
- All Number or String types can be converte to Boolean. Possible (case-insensitive) values for true:

  * 1
  * 1.0
  * y
  * yes
  * true
  * on
  * enabled

  Possible (case-insensitive) values for false:

  * 0
  * 0.0
  * n
  * no
  * false
  * off
  * disabled

  All other values evaluate to null
- All java.util.Date subtypes (java.sql.Date, java.sql.Time, java.sql.Timestamp), as well as most java.time.temporal.Temporal subtypes (java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime, java.time.OffsetDateTime, as well as java.time.Instant) can be converted into each other.
- byte[] can be converted into String, using the platform's default charset
- Object[] can be converted into any other array type, if array elements can be converted, too

This auto conversion can be applied explicitly, but is also available through a variety of API, in particular anywhere a java.lang.Class reference can be provided, such as:

```
Record record = ...
int i = record.get(0, int.class);
String s = record.get(1, String.class);
```

4.3.15. Custom data type conversion

jOOQ supports a variety of built-in data types as well as user-defined types. If you prefer your own domain types to be used with your table columns or other column expressions, you can use jOOQ's org.jooq.Converter API.

A Converter allows for two-way conversion between two Java data types T and U. By convention, the T type corresponds to the built-in type in your database whereas the U type corresponds to your own user type. The Converter API is summarised here:

```
public interface Converter<T, U> extends Serializable {

    /**
     * Convert a database object to a user object
     */
    U from(T databaseObject);

    /**
     * Convert a user object to a database object
     */
    T to(U userObject);

    /**
     * The database type
     */
    Class<T> fromType();

    /**
     * The user type
     */
    Class<U> toType();
}
```

Such a Converter can be used in many parts of the jOOQ API. While they're most powerful when attached to generated code, you can also attach them to arbitrary expressions as follows:

```
record IntegerWrapper(Integer wrapped) {}
DataType<Integer> t = SQLDataType.INTEGER;

// This DataType can now be used everywhere in jOOQ. It will remember how to
// convert between Integer and the user type IntegerWrapper
DataType<IntegerWrapper> u = t.asConvertedDataType(Converter.of(
    Integer.class,
    IntegerWrapper.class,
    IntegerWrapper::new,
    IntegerWrapper::wrapped
));

// For example, you can attach it to a plain SQL template:
Field<IntegerWrapper> f1 = field("ID", u);

// Or you can coerce any expression to your type:
Field<IntegerWrapper> f2 = BOOK.ID.coerce(u);
```

4.3.16. Data type lookups

In a lot of cases, it's unnecessary to provide an explicit built-in data type or user-defined data type with a column expression in general, or a bind value in particular. For example, assuming a user-defined data type was attached to generated code using a forced type, or to a plain SQL template using an explicit converted data type, the following expressions can infer the user-defined data type from context:

```
// Comparison predicates with TableField/bind value pairs:
TABLE.COLUMN.eq(new MyType(1));

// Comparison predicates with TableField/bind value pairs in row value expressions in various forms:
row(TABLE.COLUMN1, TABLE.COLUMN2).eq(new MyType(1), new MyOtherType(2));
row(TABLE.COLUMN1, TABLE.COLUMN2).eq(row(new MyType(1), new MyOtherType(2)));
```

More complex expressions may be able to do the same thing, depending on your jOOQ version.

In some cases, however, the DataType cannot be looked up from query context. This includes, for example, plain SQL templates:

```
// Plain SQL with bind values:
DSL.condition("column = ?", new MyType(1));

// Plain SQL with expressions:
DSL.condition("column = {0}", DSL.val(new MyType(1)));

// With Class<?> literals:
DSL.field("column", MyType.class);
```

jOOQ historically can still look up the *first* statically registered Converter or Binding for such a user-defined data type, which is why historically, these expressions may still work. Starting from jOOQ 3.19, this usage is discouraged, and a warning will be logged.

```
org.jooq.impl.DefaultDataType$DiscouragedStaticTypeRegistryUsage: null
    at org.jooq.impl.DefaultDataType.check(DefaultDataType.java)
    at org.jooq.impl.Val.accept(Val.java)
    at org.jooq.impl.AbstractBindContext.bindInternal(AbstractBindContext.java)
    at org.jooq.impl.AbstractBindContext.visit0(AbstractBindContext.java)
    at org.jooq.impl.AbstractContext.visit(AbstractContext.java)
    at ...
```

When this warning is encountered, users are encouraged to avoid the practice of relying on this lookup for the following reasons:

- There may be race conditions between usages of the user-defined data type and registrations of the Converter / Binding in the static type registry
- There may be multiple Converter or Binding for the same user-defined type Class<U> (as well as race conditions between their registrations!)
- A future version of jOOQ may unsupport this static type registry for user-defined types in favour of a dynamic type registry.

If you encounter this warning and think it shouldn't appear in your query (jOOQ should be able to infer the DataType from context), or if you struggle to address the issue, please report an issue here: https://jooq.org/bug.

4.4. Static statements vs. Prepared Statements

With JDBC, you have full control over your SQL statements. You can decide yourself, if you want to execute a static java.sql.Statement without bind values, or a java.sql.PreparedStatement with (or without) bind values. But you have to decide early, which way to go. And you'll have to prevent SQL injection and syntax errors manually, when inlining your bind variables.

With jOOQ, this is easier. As a matter of fact, it is plain simple. With jOOQ, you can just set a flag in your Configuration's Settings, and all queries produced by that configuration will be executed as static statements, with all bind values inlined. An example is given here:

```
-- These statements are rendered by the two factories:
SELECT ? FROM DUAL WHERE ? = ?
SELECT 1 FROM DUAL WHERE 1 = 1
```

```
// This DSLContext executes PreparedStatements
DSLContext prepare = DSL.using(connection, SQLDialect.ORACLE);

// This DSLContext executes static Statements
DSLContext inlined = DSL.using(connection, SQLDialect.ORACLE,
  new
 Settings().withStatementType(StatementType.STATIC_STATEMENT));

prepare.select(val(1)).where(val(1).eq(1)).fetch();
inlined.select(val(1)).where(val(1).eq(1)).fetch();
```

Reasons for choosing one or the other

Not all databases are equal. Some databases show improved performance if you use java.sql.PreparedStatement, as the database will then be able to re-use execution plans for identical SQL statements, regardless of actual bind values. This heavily improves the time it takes for soft-parsing a SQL statement. In other situations, assuming that bind values are irrelevant for SQL execution plans may be a bad idea, as you might run into "bind value peeking" issues. You may be better off spending the extra cost for a new hard-parse of your SQL statement and instead having the database fine-tune the new plan to the concrete bind values.

Whichever aproach is more optimal for you cannot be decided by jOOQ. In most cases, prepared statements are probably better. But you always have the option of forcing jOOQ to render inlined bind values.

Inlining bind values on a per-bind-value basis

Note that you don't have to inline all your bind values at once. If you know that a bind value is not really a variable and should be inlined explicitly, you can do so by using DSL.inline(), as documented in the manual's section about inlined parameters

4.5. Reusing a Query's PreparedStatement

As previously discussed in the chapter about differences between jOOQ and JDBC, reusing PreparedStatements is handled a bit differently in jOOQ from how it is handled in JDBC

Keeping open PreparedStatements with JDBC

With JDBC, you can easily reuse a java.sql.PreparedStatement by not closing it between subsequent executions. An example is given here:

```
// Execute the statement
try (PreparedStatement stmt = connection.prepareStatement("SELECT 1 FROM DUAL")) {

    // Fetch a first ResultSet
    try (ResultSet rs1 = stmt.executeQuery()) { ... }

    // Without closing the statement, execute it again to fetch another ResultSet
    try (ResultSet rs2 = stmt.executeQuery()) { ... }
}
```

The above technique can be quite useful when you want to reuse expensive database resources. This can be the case when your statement is executed very frequently and your database would take non-negligible time to soft-parse the prepared statement and generate a new statement / cursor resource.

Keeping open PreparedStatements with jOOQ

This is also modeled in jOOQ. However, the difference to JDBC is that closing a statement is the default action, whereas keeping it open has to be configured explicitly. This is better than JDBC, because the default action should be the one that is used most often. Keeping open statements is rarely done in average applications. Here's an example of how to keep open PreparedStatements with jOOQ:

```
// Create a query which is configured to keep its underlying PreparedStatement open
try (ResultQuery<Record> query = create.selectOne().keepStatement(true)) {
    Result<Record> result1 = query.fetch(); // This will lazily create a new PreparedStatement
    Result<Record> result2 = query.fetch(); // This will reuse the previous PreparedStatement
}
```

The above example shows how a query can be executed twice against the same underlying java.sql.PreparedStatement. Notice how the Query must now be treated like a resource, i.e. it must be managed in a try-with-resources statement, or Query.close() must be called explicitly.

4.6. JDBC flags

JDBC knows a couple of execution flags and modes, which can be set through the jOOQ API as well. jOOQ essentially supports these flags and execution modes:

```
public interface Query extends QueryPart, Attachable {

    // [...]

    // The query execution timeout.
    // ----------------------------------------------------------
    Query queryTimeout(int timeout);

}
```

```
public interface ResultQuery<R extends Record> extends Query {

    // [...]

    // The query execution timeout.
    // ----------------------------------------------------------
    @Override
    ResultQuery<R> queryTimeout(int timeout);

    // Flags allowing to specify the resulting ResultSet modes
    // ----------------------------------------------------------
    ResultQuery<R> resultSetConcurrency(int resultSetConcurrency);
    ResultQuery<R> resultSetType(int resultSetType);
    ResultQuery<R> resultSetHoldability(int resultSetHoldability);

    // The buffer size for JDBC cursors
    // ----------------------------------------------------------
    ResultQuery<R> fetchSize(int size);

    // The maximum number of rows to be fetched by JDBC
    // ----------------------------------------------------------
    ResultQuery<R> maxRows(int rows);

}
```

Using ResultSet concurrency with ExecuteListeners

An example of why you might want to manually set a ResultSet's concurrency flag to something non-default is given here:

```
DSL.using(new DefaultConfiguration()
   .set(connection)
   .set(SQLDialect.ORACLE)
   .set(DefaultExecuteListenerProvider.providers(
        new DefaultExecuteListener() {

            @Override
            public void recordStart(ExecuteContext ctx) {
                try {

                    // Change values in the cursor before reading a record
                    ctx.resultSet().updateString(BOOK.TITLE.getName(), "New Title");
                    ctx.resultSet().updateRow();
                }
                catch (SQLException e) {
                    throw new DataAccessException("Exception", e);
                }
            }
        }
   )))
   .select(BOOK.ID, BOOK.TITLE)
   .from(BOOK)
   .orderBy(BOOK.ID)
   .resultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)
   .resultSetConcurrency(ResultSet.CONCUR_UPDATABLE)
   .fetch(BOOK.TITLE);
```

In the above example, your custom ExecuteListener callback is triggered before jOOQ loads a new Record from the java.sql.ResultSet. With the concurrency being set to ResultSet.CONCUR_UPDATABLE, you can now modify the database cursor through the standard java.sql.ResultSet API.

4.7. Using JDBC batch operations

With JDBC, you can easily execute several statements at once using the addBatch() method. Essentially, there are two modes in JDBC

- Execute several queries without bind values
- Execute one query several times with bind values

Using JDBC

In code, this looks like the following snippet:

```
// 1. several queries
// ------------------
try (Statement stmt = connection.createStatement()) {
    stmt.addBatch("INSERT INTO author(id, first_name, last_name) VALUES (1, 'Erich', 'Gamma')");
    stmt.addBatch("INSERT INTO author(id, first_name, last_name) VALUES (2, 'Richard', 'Helm')");
    stmt.addBatch("INSERT INTO author(id, first_name, last_name) VALUES (3, 'Ralph', 'Johnson')");
    stmt.addBatch("INSERT INTO author(id, first_name, last_name) VALUES (4, 'John', 'Vlissides')");
    int[] result = stmt.executeBatch();
}

// 2. a single query
// -----------------
try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO author(id, first_name, last_name) VALUES (?, ?, ?)")) {
    stmt.setInt(1, 1);
    stmt.setString(2, "Erich");
    stmt.setString(3, "Gamma");
    stmt.addBatch();

    stmt.setInt(1, 2);
    stmt.setString(2, "Richard");
    stmt.setString(3, "Helm");
    stmt.addBatch();

    stmt.setInt(1, 3);
    stmt.setString(2, "Ralph");
    stmt.setString(3, "Johnson");
    stmt.addBatch();

    stmt.setInt(1, 4);
    stmt.setString(2, "John");
    stmt.setString(3, "Vlissides");
    stmt.addBatch();

    int[] result = stmt.executeBatch();
}
```

Using jOOQ

jOOQ supports executing queries in batch mode as follows:

```
// 1. several queries
// ------------------
create.batch(
 create.insertInto(AUTHOR, ID, FIRST_NAME, LAST_NAME).values(1, "Erich"  , "Gamma"    ),
 create.insertInto(AUTHOR, ID, FIRST_NAME, LAST_NAME).values(2, "Richard", "Helm"     ),
 create.insertInto(AUTHOR, ID, FIRST_NAME, LAST_NAME).values(3, "Ralph"  , "Johnson"  ),
 create.insertInto(AUTHOR, ID, FIRST_NAME, LAST_NAME).values(4, "John"   , "Vlissides"))
.execute();

// 2. a single query
// -----------------
create.batch(create.insertInto(AUTHOR, ID, FIRST_NAME, LAST_NAME  ).values((Integer) null, null, null))
      .bind(                          1 , "Erich"   , "Gamma"    )
      .bind(                          2 , "Richard" , "Helm"     )
      .bind(                          3 , "Ralph"   , "Johnson"  )
      .bind(                          4 , "John"    , "Vlissides")
      .execute();
```

When creating a batch execution with a single query and multiple bind values, you will still have to provide jOOQ with dummy bind values for the original query. In the above example, these are set to null. For subsequent calls to bind(), there will be no type safety provided by jOOQ.

4.8. Sequence execution

Most databases support sequences of some sort, to provide you with unique values to be used for primary keys and other enumerations. If you're using jOOQ's code generator, it will generate a sequence object per sequence for you. There are two ways of using such a sequence object:

Standalone calls to sequences

Instead of actually phrasing a select statement, you can also use the DSLContext's convenience methods:

```
// Fetch the next value from a sequence
BigInteger nextID = create.nextval(S_AUTHOR_ID);

// Fetch the current value from a sequence
BigInteger currID = create.currval(S_AUTHOR_ID);
```

Inlining sequence references in SQL

You can inline sequence references in jOOQ SQL statements. The following are examples of how to do that:

```
// Reference the sequence in a SELECT statement:
Field<BigInteger> s = S_AUTHOR_ID.nextval();
BigInteger nextID = create.select(s).fetchOne(s);

// Reference the sequence in an INSERT statement:
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(S_AUTHOR_ID.nextval(), val("William"), val("Shakespeare"))
      .execute();
```

For more info about inlining sequence references in SQL statements, please refer to the manual's section about sequences and serials.

4.9. Stored procedures and functions

Many RDBMS support the concept of "routines", usually calling them procedures and/or functions. These concepts have been around in programming languages for a while, also outside of databases. Famous languages distinguishing procedures from functions are:

- Ada
- BASIC
- Pascal
- etc...

The general distinction between (stored) procedures and (stored) functions can be summarised like this:

Procedures

- Are called using JDBC CallableStatement
- Have no return value
- Usually support OUT parameters

Functions

- Can be used in SQL statements
- Have a return value
- Usually don't support OUT parameters

Exceptions to these rules

- DB2, H2, and HSQLDB don't allow for JDBC escape syntax when calling functions. Functions must be used in a SELECT statement
- H2 only knows functions (without OUT parameters)
- Oracle functions may have OUT parameters
- Oracle knows functions that must not be used in SQL statements for transactional reasons
- Postgres only knows functions (with all features combined). OUT parameters can also be interpreted as return values, which is quite elegant/surprising, depending on your taste
- The Sybase jconn3 JDBC driver doesn't handle null values correctly when using the JDBC escape syntax on functions

In general, it can be said that the field of routines (procedures / functions) is far from being standardised in modern RDBMS even if the SQL:2008 standard specifies things quite well. Every database has its ways and JDBC only provides little abstraction over the great variety of procedures / functions implementations, especially when advanced data types such as cursors / UDT's / arrays are involved.

To simplify things a little bit, jOOQ handles both procedures and functions the same way, using a more general org.jooq.Routine type.

Using jOOQ for standalone calls to stored procedures and functions

If you're using jOOQ's code generator, it will generate org.jooq.Routine objects for you. Let's consider the following example:

```
-- Check whether there is an author in AUTHOR by that name and get his ID
CREATE OR REPLACE PROCEDURE author_exists (author_name VARCHAR2, result OUT NUMBER, id OUT NUMBER);
```

The generated artefacts can then be used as follows:

```
// Make an explicit call to the generated procedure object:
AuthorExists procedure = new AuthorExists();

// All IN and IN OUT parameters generate setters
procedure.setAuthorName("Paulo");
procedure.execute(configuration);

// All OUT and IN OUT parameters generate getters
assertEquals(new BigDecimal("1"), procedure.getResult());
assertEquals(new BigDecimal("2"), procedure.getId();
```

But you can also call the procedure using a generated convenience method in a global Routines class:

```
// The generated Routines class contains static methods for every procedure.
// Results are also returned in a generated object, holding getters for every OUT or IN OUT parameter.
AuthorExists procedure = Routines.authorExists(configuration, "Paulo");

// All OUT and IN OUT parameters generate getters
assertEquals(new BigDecimal("1"), procedure.getResult());
assertEquals(new BigDecimal("2"), procedure.getId();
```

For more details about code generation for procedures, see the manual's section about procedures and code generation.

Inlining stored function references in SQL

Unlike procedures, functions can be inlined in SQL statements to generate column expressions or table expressions, if you're using unnesting operators. Assume you have a function like this:

```
-- Check whether there is an author in AUTHOR by that name and get his ID
CREATE OR REPLACE FUNCTION author_exists (author_name VARCHAR2) RETURN NUMBER;
```

The generated artefacts can then be used as follows:

```
-- This is the rendered SQL

SELECT AUTHOR_EXISTS('Paulo') FROM DUAL
```

```
// Use the static-imported method from Routines:
boolean exists =
create.select(authorExists("Paulo")).fetchOne(0, boolean.class);
```

For more info about inlining stored function references in SQL statements, please refer to the manual's section about user-defined functions.

4.9.1. Oracle Packages

Oracle uses the concept of a PACKAGE to group several procedures/functions into a sort of namespace. The SQL 92 standard talks about "modules", to represent this concept, even if this is rarely implemented as such. This is reflected in jOOQ by the use of Java sub-packages in the source code generation destination package. Every Oracle package will be reflected by

- A Java package holding classes for formal Java representations of the procedure/function in that package
- A Java class holding convenience methods to facilitate calling those procedures/functions

Apart from this, the generated source code looks exactly like the one for standalone procedures/functions.

For more details about code generation for procedures and packages see the manual's section about procedures and code generation.

4.9.2. Oracle member procedures

Oracle UDTs can have object-oriented structures including member functions and procedures. With Oracle, you can do things like this:

```
CREATE OR REPLACE TYPE u_author_type AS OBJECT (
  id NUMBER(7),
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),

  MEMBER PROCEDURE LOAD,
  MEMBER FUNCTION counBOOKs RETURN NUMBER
)

-- The type body is omitted for the example
```

These member functions and procedures can simply be mapped to Java methods:

```
// Create an empty, attached UDT record from the DSLContext
UAuthorType author = create.newRecord(U_AUTHOR_TYPE);

// Set the author ID and load the record using the LOAD procedure
author.setId(1);
author.load();

// The record is now updated with the LOAD implementation's content
assertNotNull(author.getFirstName());
assertNotNull(author.getLastName());
```

For more details about code generation for UDTs see the manual's section about user-defined types and code generation.

4.10. Exporting to XML, CSV, JSON, HTML, Text, Charts

If you are using jOOQ for scripting purposes or in a slim, unlayered application server, you might be interested in using jOOQ's exporting functionality (see also the importing functionality). You can export any Result<Record> into the formats discussed in the subsequent chapters of the manual

4.10.1. Exporting XML

```
// Fetch books and format them as XML
String xml = create.selectFrom(BOOK).fetch().formatXML();
```

The above query will result in an XML document looking like the following one:

```
<result xmlns="http://www.jooq.org/xsd/jooq-export-3.10.0.xsd">
  <fields>
    <field schema="TEST" table="BOOK" name="ID" type="INTEGER"/>
    <field schema="TEST" table="BOOK" name="AUTHOR_ID" type="INTEGER"/>
    <field schema="TEST" table="BOOK" name="TITLE" type="VARCHAR"/>
  </fields>
  <records>
    <record>
      <value field="ID">1</value>
      <value field="AUTHOR_ID">1</value>
      <value field="TITLE">1984</value>
    </record>
    <record>
      <value field="ID">2</value>
      <value field="AUTHOR_ID">1</value>
      <value field="TITLE">Animal Farm</value>
    </record>
  </records>
</result>
```

The same result as an org.w3c.dom.Document can be obtained using the Result.intoXML() method:

```
// Fetch books and format them as XML
Document xml = create.selectFrom(BOOK).fetch().intoXML();
```

See the XSD schema definition here, for a formal definition of the XML export format: https://www.jooq.org/xsd/jooq-export-3.10.0.xsd

4.10.2. Exporting CSV

```
// Fetch books and format them as CSV
String csv = create.selectFrom(BOOK).fetch().formatCSV();
```

The above query will result in a CSV document looking like the following one:

```
ID,AUTHOR_ID,TITLE
1,1,1984
2,1,Animal Farm
```

In addition to the standard behaviour, you can also specify a separator character, as well as a special string to represent NULL values (which cannot be represented in standard CSV):

```
// Use ";" as the separator character
String csv = create.selectFrom(BOOK).fetch().formatCSV(';');

// Specify "{null}" as a representation for NULL values
String csv = create.selectFrom(BOOK).fetch().formatCSV(';', "{null}");
```

4.10.3. Exporting JSON

```
// Fetch books and format them as JSON
String json = create.selectFrom(BOOK).fetch().formatJSON();
```

The above query will result in a JSON document looking like the following one:

```
{"fields":[{"schema":"schema-1","table":"table-1","name":"field-1","type":"type-1"},
           {"schema":"schema-2","table":"table-2","name":"field-2","type":"type-2"},
           ...,
           {"schema":"schema-n","table":"table-n","name":"field-n","type":"type-n"}],
 "records":[[value-1-1,value-1-2,...,value-1-n],
            [value-2-1,value-2-2,...,value-2-n]]}
```

Note: This format has been modified in jOOQ 2.6.0 and 3.7.0

4.10.4. Exporting HTML

```
// Fetch books and format them as HTML
String html = create.selectFrom(BOOK).fetch().formatHTML();
```

© 2009 - 2025 by Data Geekery™ GmbH.

The above query will result in an HTML document looking like the following one

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>AUTHOR_ID</th>
      <th>TITLE</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td>1</td>
      <td>1984</td>
    </tr>
    <tr>
      <td>2</td>
      <td>1</td>
      <td>Animal Farm</td>
    </tr>
  </tbody>
</table>
```

4.10.5. Exporting Text

```
// Fetch books and format them as text
String text = create.selectFrom(BOOK).fetch().format();
```

The above query will result in a text document looking like the following one

```
+---+---------+-----------+
| ID|AUTHOR_ID|TITLE      |
+---+---------+-----------+
|  1|        1|1984       |
|  2|        1|Animal Farm|
+---+---------+-----------+
```

A simple text representation can also be obtained by calling toString() on a Result object. See also the manual's section about DEBUG logging

4.10.6. Exporting Charts

```
// Count books per book store and format them as charts
String chart =
create.select(
           BOOK_TO_BOOK_STORE.BOOK_STORE_NAME,
           count(BOOK_TO_BOOK_STORE.BOOK_ID).as("books")
       )
       .from(BOOK_TO_BOOK_STORE)
       .groupBy(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
       .fetch()
       .formatChart();
```

When formatted, the result is this:

```
+------------------------+-----+
|BOOK_STORE_NAME         |books|
+------------------------+-----+
|Buchhandlung im Volkshaus|    1|
|Ex Libris               |    2|
|Orell Füssli            |    3|
+------------------------+-----+
```

And the chart will be looking like the following one

```
3.00|                                                  #########################
2.91|                                                  #########################
2.82|                                                  #########################
2.73|                                                  #########################
2.64|                                                  #########################
2.55|                                                  #########################
2.45|                                                  #########################
2.36|                                                  #########################
2.27|                                                  #########################
2.18|                                                  #########################
2.09|                                                  #########################
2.00|                          ##################################################
1.91|                          ##################################################
1.82|                          ##################################################
1.73|                          ##################################################
1.64|                          ##################################################
1.55|                          ##################################################
1.45|                          ##################################################
1.36|                          ##################################################
1.27|                          ##################################################
1.18|                          ##################################################
1.09|                          ##################################################
1.00|###########################################################################
----+---------------------------------------------------------------------------
    | Buchhandlung im Volkshaus         Ex Libris               Orell Füssli
```

It is possible to specify a variety of org.jooq.ChartFormat formatting specifications, such as the width, height, display type (default, stacked, 100% stacked), the column index of the category and value columns, etc.

4.11. Importing data

jOOQ's loader API can be used to import tabular data into a table from a variety of data sources. It offers a simplified API to solve common data import challenges such as:

- Mapping different data sources, like CSV, JSON, XML, records to SQL tables
- Specifying behaviour when duplicate keys are encountered
- Fine tuning batch, bulk, and commit sizes
- Error handling

4.11.1. The Loader API

The loader API is implemented like any other DSL statement in jOOQ, following a few steps:

```
create.loadInto(TARGET_TABLE)
      .[options]
      .[source and source to target mapping]
      .[listeners]
      .[execution and error handling]
```

For example:

```
create.loadInto(BOOK)

      // Options
      .onDuplicateKeyError()
      .bulkAll()
      .batchAll()
      .commitAll()

      // Source and source to target mapping
      .loadCSV(inputStream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)

      // Listeners
      .onRow(ctx -> { /* ... */ })

      // Execution and error handling
      .execute()
      .errors()
      .forEach(e -> { /* ... */ });
```

See the following sections for details about each step:

- Import options
- Import data sources
- Import listeners
- Import result and error handling

4.11.2. Import options

Prior to specifing data sources, data source independent loading options can be specified. These include throttling, duplicate handling, and error handling:

4.11.2.1. Throttling

When importing large data sets, it may be beneficial to explicitly define the optimal size for each:

- Bulk size: The number of rows that are sent to the server in one SQL statement. Defaults to 1.
- Batch size: The number of statements that are sent to the server in one JDBC statement batch. Defaults to 1.
- Commit size: The number of statement batches that are committed in one transaction. Defaults to 1.

All of the three types of throttling can be combined.

Not all RDBMS offer the same optimisation capabilities. Please refer to your database manual to learn how these tuning capabilities may affect your data import performance. Also, actual measurements may help improve these numbers. Do not optimise prematurely, or based on assumptions. Always measure if your optimisation has the desired effect!

The bulk size

Bulk data processing is important in SQL, which is a set based language. It is possible to express bulk INSERT statements as well with INSERT .. VALUES or INSERT .. SELECT. Sophisticated RDBMS may use these statements to improve the disk block allocation process, because if an INSERT statement has more than 1 row, the optimiser knows better how much space will be needed to store the incoming

data. This approach also helps "clustering" inserted data (keeping data that is inserted at the same time in local disk block "clusters"), which may be beneficial if the same data is also frequently read, later on. While the benefit may be marginal on SSDs or other random access disks, it may be significant on HDDs.

There are 3 possible, mutually exclusive configurations of specifying the bulk size:

```
create.loadInto(BOOK)

      // Put all the data in a single bulk statement.
      .bulkAll()

      // Put up to 32 rows in a single bulk statement.
      .bulkAfter(32)

      // Do not put more than 1 row in a statement.
      .bulkNone()

      .loadCSV(inputstream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

The batch size

Batch data processing allows for reducing the network traffic overhead, because it allows the JDBC driver to buffer bind values for several subsequent statement executions and send them all in one go.

There are 3 possible, mutually exclusive configurations of specifying the batch size:

```
create.loadInto(BOOK)

      // Execute all statements (bulk or not) in a single large statement batch.
      .batchAll()

      // Put up to 32 statements (bulk or not) in a single statement batch.
      .batchAfter(32)

      // Execute each statement (bulk or not) individually.
      .batchNone()

      .loadCSV(inputstream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

The commit size

Committing a transaction can be a costly operation if done too often, or not often enough. If there are too many commits, this can lead to a lot of logging overhead on the server. If a too many changes are left uncommitted for too long, there may be too much locking in 2PL transaction models, or log contention in MVCC transaction models. An empirically discovered, optimal commit size that leads to committing e.g. 1000 rows (or 10000, or 100, please measure what works best for you) may produce best results.

There are 3 possible, mutually exclusive configurations of specifying the batch size:

```
create.loadInto(BOOK)

      // Commit all statements (batch, bulk, or not) in a single large transaction.
      .commitAll()

      // Put up to 32 statements (batch, bulk, or not) in a transaction.
      .commitAfter(32)

      // Commit each statement (batch, bulk, or not) in a transaction, just like commitAfter(1)
      .commitEach()

      // Do not commit any statement, leave committing to client code
      .commitNone()

      .loadCSV(inputstream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

© 2009 - 2025 by Data Geekery™ GmbH.

4.11.2.2. Duplicate handling

When importing data, some data may already be present and needs to be updated. jOOQ supports a variety of UPSERT style statements. The ideal statement to be used for imports is MySQL's INSERT .. ON DUPLICATE KEY UPDATE statement, which can be emulated using standard SQL MERGE, or INSERT .. ON CONFLICT in PostgreSQL or SQLite.

```
create.loadInto(BOOK)

      // Insert each row using INSERT .. ON DUPLICATE KEY UPDATE
      .onDuplicateKeyUpdate()

      // Insert each row using INSERT .. ON DUPLICATE KEY IGNORE
      .onDuplicateKeyIgnore()

      // Use ordinary INSERT statements, which will produce errors on duplicate keys
      .onDuplicateKeyError()

      .loadCSV(inputstream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

4.11.2.3. Error handling

When importing large amounts of data, errors may be inevitable and may need to be processed after the import, without impacting the entire import. In these cases, it may be useful to specify error handling. In all cases, errors will be reported after the execution of the import process:

```
create.loadInto(BOOK)

      // Ignore any errors and continue inserting. Errors will be reported nonetheless.
      .onErrorIgnore()

      // Abort the import upon encountering the first error.
      .onErrorAbort()

      .loadCSV(inputstream)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

4.11.3. Import data sources

Different types of data sources are supported by jOOQ in the same formats as the export API. These include:

4.11.3.1. Importing CSV

The below CSV data represents two author records that may have been exported previously, by jOOQ's exporting functionality, and then modified in Microsoft Excel or any other spreadsheet tool:

```
ID,AUTHOR_ID,TITLE <-- Note the CSV header. By default, the first line is ignored
1,1,1984
2,1,Animal Farm
```

The following examples show how to map source and target tables.

```
// Specify fields from the target table to be matched with fields from the source CSV by position.
// Positional matching is independent of the presence of a header row in the CSV content.
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();

// Use "null" field placeholders to ignore source columns by position.
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fields(BOOK.ID, null, BOOK.TITLE)
      .execute();
```

CSV specific options

You may pass one of the following flags to specify how the CSV content should be parsed:

```
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)

      // Ignore a certain number of header rows. By default, this is 1.
      .ignoreRows(1)

      // The quote character for use with string content containing quotes or separators. By default, this is "
      .quote('"')

      // The separator character that separates columns. By default, this is ,
      .separator(',')

      // The null string allows for distinguishing between empty strings and null. By default, there is no null string.
      .nullString("{null}")
      .execute();
```

4.11.3.2. Importing JSON

The below JSON data represents two author records that may have been exported previously, by jOOQ's exporting functionality:

```
{"fields" :[{"name":"ID","type":"INTEGER"},
            {"name":"AUTHOR_ID","type":"INTEGER"},
            {"name":"TITLE","type":"VARCHAR"}],
 "records":[[1,1,"1984"],
            [2,1,"Animal Farm"]]}
```

The following examples show how to map source data and target table.

```
// Specify fields from the target table to be matched with fields from the source JSON array by position.
// Positional matching is independent of the presence of a header information in the JSON content.
create.loadInto(BOOK)
      .loadJSON(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();

// Use "null" field placeholders to ignore source columns by position.
create.loadInto(BOOK)
      .loadJSON(inputstream, encoding)
      .fields(BOOK.ID, null, BOOK.TITLE)
      .execute();
```

No other, JSON-specific options are currently available.

© 2009 - 2025 by Data Geekery™ GmbH.

4.11.3.3. Importing records

A common use-case for importing records via jOOQ's Loader API is when data needs to be transferred between databases. For instance, when fetching the following data from database 1:

```
Result<Record3<Integer, Integer, String>> result =
DSL.using(configuration1)
   .select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
   .from(BOOK)
   .fetch();
```

Now, this result should be imported back into a database 2:

```
// Specify fields from the target table to be matched with fields from the source result by position.
create.loadInto(BOOK)
      .loadRecords(result)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();

// Use "null" field placeholders to ignore source columns by position.
create.loadInto(BOOK)
      .loadRecords(result)
      .fields(BOOK.ID, null, BOOK.TITLE)
      .execute();
```

No other, Record-specific options are currently available.

4.11.3.4. Importing arrays

A common use-case for importing arrays via jOOQ's Loader API is when data is fetched into memory from some data source, or even ad-hoc data, which needs to be imported into a database.

```
// Specify fields from the target table to be matched with fields from the source result by position.
create.loadInto(BOOK)
      .loadArrays(
          new Object[] { 1, 1, "1984" },
          new Object[] { 2, 1, "Animal Farm" })
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

No other, array-specific options are currently available.

4.11.3.5. Importing XML

This is not yet supported

4.11.4. Import listeners

Import listeners allow for keeping track of import progress:

```
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .onRow(ctx -> {
          log.info(
              "Executed: {}, ignored: {}, processed: {}, stored: {}",
              ctx.executed(), ctx.ignored(), ctx.processed(), ctx.stored()
          );
      })
      .execute();
```

4.11.5. Import result and error handling

After completed execution, a number of diagnostics are available to implement error handling:

```
Loader<?> loader =
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();

// The number of processed rows
int processed = loader.processed();

// The number of stored rows (INSERT or UPDATE)
int stored = loader.stored();

// The number of ignored rows (due to errors, or duplicate rule)
int ignored = loader.ignored();

// The errors that may have occurred during loading
List<LoaderError> errors = loader.errors();
LoaderError error = errors.get(0);

// The exception that caused the error
DataAccessException exception = error.exception();

// The row that caused the error
int rowIndex = error.rowIndex();
String[] row = error.row();

// The query that caused the error
Query query = error.query();
```

4.12. CRUD with UpdatableRecords

Your database application probably consists of 50% - 80% CRUD, whereas only the remaining 20% - 50% of querying is actual querying. Most often, you will operate on records of tables without using any advanced relational concepts. This is called CRUD for

- Create (INSERT)
- Read (SELECT)
- Update (UPDATE)
- Delete (DELETE)

CRUD always uses the same patterns, regardless of the nature of underlying tables. This again, leads to a lot of boilerplate code, if you have to issue your statements yourself. Like Hibernate / JPA and other ORMs, jOOQ facilitates CRUD using a specific API involving org.jooq.UpdatableRecord types.

Primary keys and updatability

In normalised databases, every table has a primary key by which a tuple/record within that table can be uniquely identified. In simple cases, this is a (possibly auto-generated) number called ID. But in many cases, primary keys include several non-numeric columns. An important feature of such keys is the fact that in most databases, they are enforced using an index that allows for very fast random access to the table. A typical way to access / modify / delete a book is this:

```
-- Inserting uses a previously generated key value or generates it afresh
INSERT INTO BOOK (ID, TITLE) VALUES (5, 'Animal Farm');

-- Other operations can use a previously generated key value
SELECT * FROM BOOK WHERE ID = 5;
UPDATE BOOK SET TITLE = '1984' WHERE ID = 5;
DELETE FROM BOOK WHERE ID = 5;
```

Normalised databases assume that a primary key is unique "forever", i.e. that a key, once inserted into a table, will never be changed or re-inserted after deletion. In order to use jOOQ's CRUD operations correctly, you should design your database accordingly.

4.12.1. Simple CRUD

If you're using jOOQ's code generator, it will generate org.jooq.UpdatableRecord implementations for every table that has a primary key. When fetching such a record form the database, these records are "attached" to the Configuration that created them. This means that they hold an internal reference to the same database connection that was used to fetch them. This connection is used internally by any of the following methods of the UpdatableRecord:

```
// Refresh a record from the database.
void refresh() throws DataAccessException;

// Store (insert or update) a record to the database.
int store() throws DataAccessException;

// Delete a record from the database
int delete() throws DataAccessException;
```

See the manual's section about serializability for some more insight on "attached" objects.

Storing

Storing a record will perform an INSERT statement or an UPDATE statement. In general, new records are always inserted, whereas records loaded from the database are always updated. This is best visualised in code:

```
// Create a new record
BookRecord book1 = create.newRecord(BOOK);

// Insert the record: INSERT INTO BOOK (TITLE) VALUES ('1984');
book1.setTitle("1984");
book1.store();

// Update the record: UPDATE BOOK SET PUBLISHED_IN = 1984 WHERE ID = [id]
book1.setPublishedIn(1948);
book1.store();

// Get the (possibly) auto-generated ID from the record
Integer id = book1.getId();

// Get another instance of the same book
BookRecord book2 = create.fetchOne(BOOK, BOOK.ID.eq(id));

// Update the record: UPDATE BOOK SET TITLE = 'Animal Farm' WHERE ID = [id]
book2.setTitle("Animal Farm");
book2.store();
```

Some remarks about storing:

- jOOQ sets only modified values in INSERT statements or UPDATE statements. This allows for default values to be applied to inserted records, as specified in CREATE TABLE DDL statements.
- When store() performs an INSERT statement, jOOQ attempts to load any generated keys from the database back into the record. For more details, see the manual's section about IDENTITY values.
- In addition to loading identity values, store() can also be configured to refresh the entire record. See the returnAllOnUpdatableRecord setting for details
- When loading records from POJOs, jOOQ will assume the record is a new record. It will hence attempt to INSERT it.
- When you activate optimistic locking, storing a record may fail, if the underlying database record has been changed in the mean time.

Deleting

Deleting a record will remove it from the database. Here's how you delete records:

```
// Get a previously inserted book
BookRecord book = create.fetchOne(BOOK, BOOK.ID.eq(5));

// Delete the book
book.delete();
```

Refreshing

Refreshing a record from the database means that jOOQ will issue a SELECT statement to refresh all record values that are not the primary key. This is particularly useful when you use jOOQ's optimistic locking feature, in case a modified record is "stale" and cannot be stored to the database, because the underlying database record has changed in the mean time.

In order to perform a refresh, use the following Java code:

```
// Fetch an updatable record from the database
BookRecord book = create.fetchOne(BOOK, BOOK.ID.eq(5));

// Refresh the record
book.refresh();
```

CRUD and SELECT statements

CRUD operations can be combined with regular querying, if you select records from single database tables, as explained in the manual's section about SELECT statements. For this, you will need to use the selectFrom() method from the DSLContext:

```
// Loop over records returned from a SELECT statement
for (BookRecord book : create.fetch(BOOK, BOOK.PUBLISHED_IN.eq(1948))) {

  // Perform actions on BookRecords depending on some conditions
  if ("Orwell".equals(book.fetchParent(Keys.FK_BOOK_AUTHOR).getLastName())) {
    book.delete();
  }
}
```

4.12.2. Records' internal flags

All of jOOQ's Record types and subtypes maintain an internal state for every column value. This state is composed of three elements:

- The value itself
- The "original" value, i.e. the value as it was originally fetched from the database or null, if the record was never in the database
- The "changed" flag, indicating if the value was ever changed through the Record API.

The purpose of the above information is for jOOQ's CRUD operations to know, which values need to be stored to the database, and which values have been left untouched.

4.12.3. IDENTITY values

Many databases support the concept of IDENTITY values, or SEQUENCE-generated key values. This is reflected by JDBC's getGeneratedKeys() method. jOOQ abstracts using this method as many databases and JDBC drivers behave differently with respect to generated keys. Let's assume the following SQL Server BOOK table:

```
CREATE TABLE book (
  ID INTEGER IDENTITY(1,1) NOT NULL,

  -- [...]

  CONSTRAINT pk_book PRIMARY KEY (id)
)
```

If you're using jOOQ's code generator, the above table will generate a org.jooq.UpdatableRecord with an IDENTITY column. This information is used by jOOQ internally, to update IDs after calling store():

```
BookRecord book = create.newRecord(BOOK);
book.setTitle("1984");
book.store();

// The generated ID value is fetched after the above INSERT statement
System.out.println(book.getId());
```

Database compatibility

DB2, Derby, HSQLDB, Ingres

These SQL dialects implement the standard very neatly.

```
id INTEGER GENERATED BY DEFAULT AS IDENTITY
id INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1)
```

H2, MySQL, Postgres, SQL Server, Sybase ASE, Sybase SQL Anywhere

These SQL dialects implement identites, but the DDL syntax doesn't follow the standard

```
-- H2 mimicks MySQL's and SQL Server's syntax
ID INTEGER IDENTITY(1,1)
ID INTEGER AUTO_INCREMENT
-- MySQL and SQLite
ID INTEGER NOT NULL AUTO_INCREMENT

-- Postgres serials implicitly create a sequence
-- Postgres also allows for selecting from custom sequences
-- That way, sequences can be shared among tables
id SERIAL NOT NULL

-- SQL Server
ID INTEGER IDENTITY(1,1) NOT NULL
-- Sybase ASE
id INTEGER IDENTITY NOT NULL
-- Sybase SQL Anywhere
id INTEGER NOT NULL IDENTITY
```

For databases where IDENTITY columns are only emulated (e.g. Oracle prior to 12c), the jOOQ generator can also be configured to generate synthetic identities.

4.12.4. Navigation methods

org.jooq.TableRecord and org.jooq.UpdatableRecord contain foreign key navigation methods. These navigation methods allow for "navigating" inbound or outbound foreign key references by executing an appropriate query. An example is given here:

```
CREATE TABLE book (
  AUTHOR_ID NUMBER(7) NOT NULL,

  -- [...]

  FOREIGN KEY (AUTHOR_ID) REFERENCES author(ID)
)
```

```
BookRecord book = create.fetch(BOOK, BOOK.ID.eq(5));

// Find the author of a book (static imported from Keys)
AuthorRecord author = book.fetchParent(FK_BOOK_AUTHOR);

// Find other books by that author
Result<BookRecord> books = author.fetchChildren(FK_BOOK_AUTHOR);
```

Note that, unlike in Hibernate, jOOQ's navigation methods will always lazy-fetch relevant records, without caching any results. In other words, every time you run such a fetch method, a new query will be issued.

These fetch methods only work on "attached" records. See the manual's section about serializability for some more insight on "attached" objects.

4.12.5. Non-updatable records

Tables without a PRIMARY KEY are considered non-updatable by jOOQ, as jOOQ has no way of uniquely identifying such a record within the database. If you're using jOOQ's code generator, such tables will

generate org.jooq.TableRecord classes, instead of org.jooq.UpdatableRecord classes. When you fetch typed records from such a table, the returned records will not allow for calling any of the store(), refresh(), delete() methods.

Note, that some databases use internal rowid or object-id values to identify such records. jOOQ does not support these vendor-specific record meta-data.

4.12.6. Optimistic locking

jOOQ allows you to perform CRUD operations using optimistic locking. You can immediately take advantage of this feature by activating the relevant executeWithOptimisticLocking Setting. Without any further knowledge of the underlying data semantics, this will have the following impact on store() and delete() methods:

- INSERT statements are not affected by this Setting flag
- Prior to UPDATE or DELETE statements, jOOQ will run a SELECT .. FOR UPDATE statement, pessimistically locking the record for the subsequent UPDATE / DELETE
- The data fetched with the previous SELECT will be compared against the data in the record being stored or deleted
- An org.jooq.exception.DataChangedException is thrown if the record had been modified or deleted in the meantime, or if optimistic locking is performed on an unversioned record that hasn't been fetched from the database.
- The record is successfully stored / deleted, if the record had not been modified in the mean time.

The above changes to jOOQ's behaviour are transparent to the API, the only thing you need to do for it to be activated is to set the Settings flag. Here is an example illustrating optimistic locking:

```
// Properly configure the DSLContext
DSLContext optimistic = DSL.using(connection, SQLDialect.ORACLE,
  new Settings().withExecuteWithOptimisticLocking(true));

// Fetch a book two times
BookRecord book1 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));
BookRecord book2 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));

// Change the title and store this book. The underlying database record has not been modified, it can be safely updated.
book1.setTitle("Animal Farm");
book1.store();

// Book2 still references the original TITLE value, but the database holds a new value from book1.store().
// This store() will thus fail:
book2.setTitle("1984");
book2.store();
```

Optimised optimistic locking using TIMESTAMP fields

If you're using jOOQ's code generator, you can take indicate TIMESTAMP or UPDATE COUNTER fields for every generated table in the code generation configuration. Let's say we have this table:

```
CREATE TABLE book (

  -- This column indicates when each book record was modified for the last time
  MODIFIED TIMESTAMP NOT NULL,
  -- [...]
)
```

The MODIFIED column will contain a timestamp indicating the last modification timestamp for any book in the BOOK table. If you're using jOOQ and it's store() methods on UpdatableRecords, jOOQ will then generate this TIMESTAMP value for you, automatically. However, instead of running an additional

SELECT .. FOR UPDATE statement prior to an UPDATE or DELETE statement, jOOQ adds a WHERE-clause to the UPDATE or DELETE statement, checking for TIMESTAMP's integrity. This can be best illustrated with an example:

```
// Properly configure the DSLContext
DSLContext optimistic = DSL.using(connection, SQLDialect.ORACLE,
  new Settings().withExecuteWithOptimisticLocking(true));

// Fetch a book two times
BookRecord book1 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));
BookRecord book2 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));

// Change the title and store this book. The MODIFIED value has not been changed since the book was fetched.
// It can be safely updated
book1.setTitle("Animal Farm");
book1.store();

// Book2 still references the original MODIFIED value, but the database holds a new value from book1.store().
// This store() will thus fail:
book2.setTitle("1984");
book2.store();
```

As before, without the added TIMESTAMP column, optimistic locking is transparent to the API.

Optimised optimistic locking using VERSION fields

Instead of using TIMESTAMPs, you may also use numeric VERSION fields, containing version numbers that are incremented by jOOQ upon store() calls.

Note, for explicit pessimistic locking, please consider the manual's section about the FOR UPDATE clause. For more details about how to configure TIMESTAMP or VERSION fields, consider the manual's section about advanced code generator configuration.

4.12.7. Batch execution

When inserting, updating, deleting a lot of records, you may wish to profit from JDBC batch operations, which can be performed by jOOQ. These are available through jOOQ's DSLContext as shown in the following example:

```
// Fetch a bunch of books
Result<BookRecord> books = create.fetch(BOOK);

// Modify the above books, and add some new ones:
modify(books);
addMore(books);

// Batch-update and/or insert all of the above books
create.batchStore(books).execute();
```

Internally, jOOQ will render all the required SQL statements and execute them as a regular JDBC batch execution.

4.12.8. CRUD SPI: RecordListener

When performing CRUD, you may want to be able to centrally register one or several listener objects that receive notification every time CRUD is performed on an UpdatableRecord. Example use cases of such a listener are:

- Adding a central ID generation algorithm, generating UUIDs for all of your records.
- Adding a central record initialisation mechanism, preparing the database prior to inserting a new record.

An example of such a RecordListener is given here:

```
// Extending DefaultRecordListener, which provides empty implementations for all methods...
public class InsertListener extends DefaultRecordListener {

    @Override
    public void insertStart(RecordContext ctx) {

        // Generate an ID for inserted BOOKs
        if (ctx.record() instanceof BookRecord) {
            BookRecord book = (BookRecord) ctx.record();
            book.setId(IDTools.generate());
        }
    }
}
```

Now, configure jOOQ's runtime to load your listener

```
// Create a configuration with an appropriate listener provider:
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);
configuration.set(new DefaultRecordListenerProvider(new InsertListener()));

// Create a DSLContext from the above configuration
DSLContext create = DSL.using(configuration);
```

For a full documentation of what RecordListener can do, please consider the RecordListener Javadoc. Note that RecordListener instances can be registered with a Configuration independently of ExecuteListeners.

Triggers

A RecordListener does not act as a client-side trigger. As such, it does not affect any bulk DML statements (e.g. UPDATE statement), whose affected records are not available to clients. For those purposes, use a server-side trigger if records should be changed, or a org.jooq.VisitListener if the SQL query should be changed independently of data.

4.13. DAOs

If you're using jOOQ's code generator, you can configure it to generate POJOs and DAOs for you. jOOQ then generates one DAO per UpdatableRecord, i.e. per table with a single-column primary key. Generated DAOs implement a common jOOQ type called org.jooq.DAO. This type contains the following methods:

```
// <R> corresponds to the DAO's related table
// <P> corresponds to the DAO's related generated POJO type
// <T> corresponds to the DAO's related table's primary key type.
// Note that multi-column primary keys are not yet supported by DAOs
public interface DAO<R extends TableRecord<R>, P, T> {

    // These methods allow for inserting POJOs
    void insert(P object) throws DataAccessException;
    void insert(P... objects) throws DataAccessException;
    void insert(Collection<P> objects) throws DataAccessException;

    // These methods allow for updating POJOs based on their primary key
    void update(P object) throws DataAccessException;
    void update(P... objects) throws DataAccessException;
    void update(Collection<P> objects) throws DataAccessException;

    // These methods allow for deleting POJOs based on their primary key
    void delete(P... objects) throws DataAccessException;
    void delete(Collection<P> objects) throws DataAccessException;
    void deleteById(T... ids) throws DataAccessException;
    void deleteById(Collection<T> ids) throws DataAccessException;

    // These methods allow for checking record existence
    boolean exists(P object) throws DataAccessException;
    boolean existsById(T id) throws DataAccessException;
    long count() throws DataAccessException;

    // These methods allow for retrieving POJOs by primary key or by some other field
    List<P> findAll() throws DataAccessException;
    P findById(T id) throws DataAccessException;
    <Z> List<P> fetch(Field<Z> field, Z... values) throws DataAccessException;
    <Z> P fetchOne(Field<Z> field, Z value) throws DataAccessException;

    // These methods provide DAO meta-information
    Table<R> getTable();
    Class<P> getType();
}
```

Besides these base methods, generated DAO classes implement various useful fetch methods. An incomplete example is given here, for the BOOK table:

```
// An example generated BookDao class
public class BookDao extends DAOImpl<BookRecord, Book, Integer> {

    // Columns with primary / unique keys produce fetchOne() methods
    public Book fetchOneById(Integer value) { ... }

    // Other columns produce fetch() methods, returning several records
    public List<Book> fetchByAuthorId(Integer... values) { ... }
    public List<Book> fetchByTitle(String... values) { ... }
}
```

Note that you can further subtype those pre-generated DAO classes, to add more useful DAO methods to them. Using such a DAO is simple:

```
// Initialise an Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(SQLDialect.ORACLE);

// Initialise the DAO with the Configuration
BookDao bookDao = new BookDao(configuration);

// Start using the DAO
Book book = bookDao.findById(5);

// Modify and update the POJO
book.setTitle("1984");
book.setPublishedIn(1948);
bookDao.update(book);

// Delete it again
bookDao.delete(book);
```

4.14. Transaction management

jOOQ can work with any pre-existing transaction model (e.g. JDBC, Spring, Jakarta EE), or alternatively, offers its own convenience transaction API. In particular:

- You can use third-party transaction management libraries like Spring TX
- You can use a JTA-compliant Java EE transaction manager from your container.
- You can call JDBC's Connection.commit(), Connection.rollback() and other methods on your JDBC driver, or use the equivalent methods on your R2DBC driver.
- You can issue vendor-specific COMMIT, ROLLBACK and other statements directly in your database.
- You use jOOQ's transaction API.

(!) When using Spring Boot, its jOOQ starter already pre-configures the correct Spring transaction aware data source, so Spring transactions will work out of the box with jOOQ.

While jOOQ does not aim to replace any of the above, it offers a simple API (and a corresponding SPI) to provide you with jOOQ-style programmatic fluency to express your transactions. Below are some Java examples showing how to implement (nested) transactions with jOOQ. For these examples, we're using Java 8 syntax. Java 8 is not a requirement, though.

```
create.transaction(configuration -> {
    AuthorRecord author =
    DSL.using(configuration)
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    DSL.using(configuration)
       .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
       .values(author.getId(), "1984")
       .values(author.getId(), "Animal Farm")
       .execute();

    // Implicit commit executed here
});
```

Note how the lambda expression receives a new, ***derived*** configuration that should be used within the local scope:

```
create.transaction(configuration -> {

    // Wrap configuration in a new DSLContext:
    DSL.using(configuration).insertInto(...);
    DSL.using(configuration).insertInto(...);

    // Or, reuse the new DSLContext within the transaction scope:
    DSLContext ctx = DSL.using(configuration);
    ctx.insertInto(...);
    ctx.insertInto(...);

    // ... but avoid using the scope from outside the transaction:
    create.insertInto(...);
    create.insertInto(...);
});
```

While some org.jooq.TransactionProvider implementations (e.g. ones based on ThreadLocals, e.g. Spring or JTA) may allow you to reuse the globally scoped DSLContext reference, the jOOQ transaction API design allows for TransactionProvider implementations that require your transactional code to use the new, locally scoped Configuration, instead.

Transactional code is wrapped in jOOQ's org.jooq.TransactionalRunnable or org.jooq.TransactionalCallable types:

```
public interface TransactionalRunnable {
    void run(Configuration configuration) throws Exception;
}

public interface TransactionalCallable<T> {
    T run(Configuration configuration) throws Exception;
}
```

Such transactional code can be passed to transaction(TransactionRunnable) or transactionResult(TransactionCallable) methods. An example using transactionResult():

```
int updateCount = create.transactionResult(configuration -> {
    int result = 0;

    DSLContext ctx = DSL.using(configuration);
    result += ctx.insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).values("John", "Doe").execute();
    result += ctx.insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).values("Jane", "Doe").execute();

    return result;
});
```

Rollbacks

Any uncaught checked or unchecked exception thrown from your transactional code will rollback the transaction to the beginning of the block. This behaviour will allow for nesting transactions, if your configured org.jooq.TransactionProvider supports nesting of transactions. An example can be seen here:

```
create.transaction(outer -> {
    final AuthorRecord author =
    DSL.using(outer)
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    // Implicit savepoint created here
    try {
        DSL.using(outer)
           .transaction(nested -> {
               DSL.using(nested)
                  .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
                  .values(author.getId(), "1984")
                  .values(author.getId(), "Animal Farm")
                  .execute();

            // Rolls back the nested transaction
            if (oops)
                throw new RuntimeException("Oops");

            // Implicit savepoint is discarded, but no commit is issued yet.
        });
    }
    catch (RuntimeException e) {

        // We can decide whether an exception is "fatal enough" to roll back also the outer transaction
        if (isFatal(e))

            // Rolls back the outer transaction
            throw e;
    }

    // Implicit commit executed here
});
```

TransactionProvider implementations

By default, jOOQ ships with the org.jooq.impl.DefaultTransactionProvider, which implements nested transactions using JDBC java.sql.Savepoint. Spring Boot will configure an alternative TransactionProvider that should work for most use-cases. You can, however, still implement your own org.jooq.TransactionProvider and supply that to your Configuration to override jOOQ's default behaviour. A simple example implementation using Spring's DataSourceTransactionManager can be seen here:

```
import static org.springframework.transaction.TransactionDefinition.PROPAGATION_NESTED;

import org.jooq.Transaction;
import org.jooq.TransactionContext;
import org.jooq.TransactionProvider;
import org.jooq.tools.JooqLogger;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class SpringTransactionProvider implements TransactionProvider {

    private static final JooqLogger log = JooqLogger.getLogger(SpringTransactionProvider.class);

    @Autowired
    DataSourceTransactionManager txMgr;

    @Override
    public void begin(TransactionContext ctx) {
        log.info("Begin transaction");

        // This TransactionProvider behaves like jOOQ's DefaultTransactionProvider,
        // which supports nested transactions using Savepoints
        TransactionStatus tx = txMgr.getTransaction(new DefaultTransactionDefinition(PROPAGATION_NESTED));
        ctx.transaction(new SpringTransaction(tx));
    }

    @Override
    public void commit(TransactionContext ctx) {
        log.info("commit transaction");

        txMgr.commit(((SpringTransaction) ctx.transaction()).tx);
    }

    @Override
    public void rollback(TransactionContext ctx) {
        log.info("rollback transaction");

        txMgr.rollback(((SpringTransaction) ctx.transaction()).tx);
    }
}

class SpringTransaction implements Transaction {
    final TransactionStatus tx;

    SpringTransaction(TransactionStatus tx) {
        this.tx = tx;
    }
}
```

4.15. Exception handling

Checked vs. unchecked exceptions

Pros and cons have been discussed time and again, and it is mostly still a matter of taste, today. In this topic, jOOQ clearly takes a side. jOOQ's exception strategy is simple:

- All "system exceptions" are unchecked. If in the middle of a transaction involving business logic, there is no way that you can recover sensibly from a lost database connection, or a constraint violation that indicates a bug in your understanding of your database model.
- All "business exceptions" are checked. Business exceptions are true exceptions that you should handle (e.g. not enough funds to complete a transaction). Other languages may have union types or monads to implement the same behaviour, minus the control flow implications of exceptions.

With jOOQ, it's simple. All of jOOQ's exceptions are "system exceptions", hence they are all unchecked.

jOOQ's DataAccessException

jOOQ uses its own org.jooq.exception.DataAccessException to wrap any underlying java.sql.SQLException that might have occurred. Note that most methods in jOOQ that may cause such a DataAccessException document this both in the Javadoc as well as in their method signature, even if this isn't a requirement for unchecked exceptions in the Java language.

DataAccessException is subtyped several times as follows:

- ConfigurationException: The org.jooq.Configuration was set up in a way that does not allow for a particular operation.
- DataAccessException: General exception usually originating from a java.sql.SQLException
- DataChangedException: An exception indicating that the database's underlying record has been changed in the mean time (see optimistic locking)
- DataTypeException: Something went wrong during type conversion
- DetachedException: A SQL statement was executed on a "detached" UpdatableRecord or a "detached" SQL statement.
- InvalidResultException: An operation was performed expecting only one result, but several results were returned.
- LoaderConfigurationException: The Loader API was executed with an illegal loader configuration.
- MappingException: Something went wrong when loading a record from a POJO or when mapping a record into a POJO
- NoDataFoundException: No rows were returned from an org.jooq.ResultQuery when exactly one row was expected.
- TooManyRowsException: More than one rows were returned from an org.jooq.ResultQuery when at most one row was expected.

Override jOOQ's exception handling

The following section about execute listeners documents means of overriding jOOQ's exception handling, if you wish to deal separately with some types of constraint violations, or if you raise business errors from your database, etc.

4.16. ExecuteListeners

The Configuration lets you specify a list of org.jooq.ExecuteListener instances. The ExecuteListener is essentially an event listener for Query, Routine, or ResultSet render, prepare, bind, execute, fetch steps. It is a base type for loggers, debuggers, profilers, data collectors, triggers, etc. Advanced ExecuteListeners can also provide custom implementations of Connection, PreparedStatement and ResultSet to jOOQ in apropriate methods.

> *(!) For convenience and better backwards-compatibility, consider extending **org.jooq.impl.DefaultExecuteListener** instead of implementing this interface.*

Example: Query statistics ExecuteListener

Here is a sample implementation of an ExecuteListener, that is simply counting the number of queries per type that are being executed using jOOQ:

```
package com.example;

// Extending DefaultExecuteListener, which provides empty implementations for all methods...
public class StatisticsListener extends DefaultExecuteListener {

    /**
     * Generated UID
     */
    private static final long                      serialVersionUID = 7399239846062763212L;

    public static final Map<ExecuteType, Integer> STATISTICS       = new ConcurrentHashMap<>();

    @Override
    public void start(ExecuteContext ctx) {
        STATISTICS.compute(ctx.type(), (k, v) -> v == null ? 1 : v + 1);
    }
}
```

Now, configure jOOQ's runtime to load your listener

```
// Create a configuration with an appropriate listener provider:
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);
configuration.set(new DefaultExecuteListenerProvider(new StatisticsListener()));

// Create a DSLContext from the above configuration
DSLContext create = DSL.using(configuration);
```

And log results any time with a snippet like this:

```
log.info("STATISTICS");
log.info("----------");

for (ExecuteType type : ExecuteType.values()) {
    log.info(type.name(), StatisticsListener.STATISTICS.get(type) + " executions");
}
```

This may result in the following log output:

```
15:16:52,982  INFO - TEST STATISTICS
15:16:52,982  INFO - ---------------
15:16:52,983  INFO - READ                     : 919 executions
15:16:52,983  INFO - WRITE                    : 117 executions
15:16:52,983  INFO - DDL                      : 2 executions
15:16:52,983  INFO - BATCH                    : 4 executions
15:16:52,983  INFO - ROUTINE                  : 21 executions
15:16:52,983  INFO - OTHER                    : 30 executions
```

Please read the ExecuteListener Javadoc for more details

Example: Custom Logging ExecuteListener

The following depicts an example of a custom ExecuteListener, which pretty-prints all queries being executed by jOOQ to stdout:

```
import org.jooq.DSLContext;
import org.jooq.ExecuteContext;
import org.jooq.conf.Settings;
import org.jooq.impl.DefaultExecuteListener;
import org.jooq.tools.StringUtils;

public class PrettyPrinter extends DefaultExecuteListener {

    /**
     * Hook into the query execution lifecycle before executing queries
     */
    @Override
    public void executeStart(ExecuteContext ctx) {

        // Create a new DSLContext for logging rendering purposes
        // This DSLContext doesn't need a connection, only the SQLDialect...
        DSLContext create = DSL.using(ctx.dialect(),

        // ... and the flag for pretty-printing
            new Settings().withRenderFormatted(true));

        // If we're executing a query
        if (ctx.query() != null) {
            System.out.println(create.renderInlined(ctx.query()));
        }

        // If we're executing a routine
        else if (ctx.routine() != null) {
            System.out.println(create.renderInlined(ctx.routine()));
        }
    }
}
```

See also the manual's sections about logging for more sample implementations of actual ExecuteListeners.

Example: Bad query execution ExecuteListener

You can also use ExecuteListeners to interact with your SQL statements, for instance when you want to check if executed UPDATE or DELETE statements contain a WHERE clause. This can be achieved trivially with the following sample ExecuteListener:

```
public class DeleteOrUpdateWithoutWhereListener extends DefaultExecuteListener {

    @Override
    public void renderEnd(ExecuteContext ctx) {
        if (ctx.sql().matches("^(?i:(UPDATE|DELETE)(?!.* WHERE ).*)$")) {
            throw new DeleteOrUpdateWithoutWhereException();
        }
    }
}

public class DeleteOrUpdateWithoutWhereException extends RuntimeException {}
```

You might want to replace the above implementation with a more efficient and more reliable one, of course.

4.17. Database meta data

Not only the code generator, but also the runtime library offers a way to access database meta data. This API is made available as org.jooq.Meta, and it has several implementations, out of the box:

- A JDBC backed implementation querying java.sql.DatabaseMetaData
- A DDL interpreter backed implementation that parses and interprets a set of DDL scripts
- An XML backed implementation (implementing https://www.jooq.org/xsd/jooq-meta-3.12.0.xsd)
- A generated code backed implementation.

The following sections document each one of the above.

4.17.1. JDBC meta data

This is the default implementation of Database meta data, which is mainly backed by JDBC's java.sql.DatabaseMetaData, or by custom SQL queries against the database's INFORMATION_SCHEMA or vendor-specific dictionary views. If you configure your DSLContext with a JDBC Connection or DataSource, you can access its meta data like this:

```
create.meta()
      .getTables()
      .forEach(System.out::println);
```

The above may print the tables from our sample database.

```
AUTHOR
BOOK
BOOK_STORE
BOOK_TO_BOOK_STORE
LANGUAGE
```

Beware that by default, the entire catalog might be loaded to ensure referential integrity of all tables and their constraints. If you're only interested in a limited subset of your schema, you can use the various filter methods:

```
create.meta()
      .filterSchema(s -> s.getName().equals("PUBLIC"))
      .getTables()
      .forEach(System.out::println);
```

4.17.2. Interpreted meta data

jOOQ's parser is used by a DDL interpreter to create an alternative implementation of org.jooq.Meta based on your DDL scripts:

```
// Using strings:
create.meta(
         Source.of("create table a (i int);"),
         Source.of("create table b (j int);"),
         Source.of("create table c (k int);")
       )
      .getTables()
      .forEach(System.out::println);
```

The above prints all of the tables from the DDL scripts

```
a
b
c
```

All the meta data is available, including column names, types, constraints, etc.

DDL can be interpreted from org.jooq.Source, which represents any string providing source, including files, input streams, etc.

Exporting DDL

Any org.jooq.Meta implementation can be exported back to DDL statements (translated to any dialect!) using Meta.ddl()

4.17.3. XML meta data

jOOQ offers a JAXB serialisable XML representation of the standard SQL INFORMATION_SCHEMA: https://www.jooq.org/xsd/jooq-meta-3.12.0.xsd. This format can be imported and exported as org.jooq.util.xml.jaxb.InformationSchema, and converted to org.jooq.Meta:

```
// Using JAXB annotated API:
create.meta(informationSchema)
      .getTables()
      .forEach(System.out::println);
```

Assuming the following XML content:

```
<information_schema>
   <tables>
      <table>
         <table_name>a</table_name>
         <table_type>BASE TABLE</table_type>
      </table>
      <table>
         <table_name>b</table_name>
         <table_type>BASE TABLE</table_type>
      </table>
      <table>
         <table_name>c</table_name>
         <table_type>BASE TABLE</table_type>
      </table>
   </tables>
   <columns>
      <column>
         <table_name>a</table_name>
         <column_name>i</column_name>
         <data_type>int</data_type>
         <ordinal_position>1</ordinal_position>
         <is_nullable>true</is_nullable>
      </column>
      <column>
         <table_name>b</table_name>
         <column_name>j</column_name>
         <data_type>int</data_type>
         <ordinal_position>1</ordinal_position>
         <is_nullable>true</is_nullable>
      </column>
      <column>
         <table_name>c</table_name>
         <column_name>k</column_name>
         <data_type>int</data_type>
         <ordinal_position>1</ordinal_position>
         <is_nullable>true</is_nullable>
      </column>
   </columns>
</information_schema>
```

The above prints all of the tables from the DDL scripts

```
a
b
c
```

All the meta data is available, including column names, types, constraints, etc.

XML can also be read from org.jooq.Source, which represents any string providing source, including files, input streams, etc.

Exporting XML

Any org.jooq.Meta implementation can be exported back to XML using Meta.informationSchema() (the JAXB annotated API, use JAXB to get the XML content).

4.17.4. Generated meta data

Generated code already implements the usual meta data API, such as org.jooq.Table, org.jooq.Field, etc., but if you need to access the powerful org.jooq.Meta API, e.g. to export DDL or XML, you can wrap your generated code using:

```
Meta meta = create.meta(BOOK, AUTHOR);
```

4.18. JDBC Connection

Sometimes, access to the JDBC java.sql.Connection is required from code that would otherwise use jOOQ. Your DSLContext and Configuration is configured with a JDBC Connection or DataSource via a org.jooq.ConnectionProvider, but rather than going through those SPIs, you can access (and acquire) a java.sql.Connection directly from your DSLContext. This can be done easily using DSLContext.connection() or DSLContext.connectionResult(). Just write:

```
// When you don't produce any results:
create.connection((Connection c) -> {

    // Modify your JDBC connection or get information from it
    c.setClientInfo("key", "value");

    // Run statements directly with JDBC
    try (Statement s = c.createStatement()) {
        s.executeUpdate("INSERT INTO author (id, first_name, last_name) VALUES (3, 'William', 'Shakespeare')");
    }
});

// When you produce results
int rows = create.connectionResult(c -> {
    try (Statement s = c.createStatement()) {
        return s.executeUpdate("INSERT INTO author (id, first_name, last_name) VALUES (3, 'William', 'Shakespeare')");
    }
});
```

4.19. Mocking Connection

When writing unit tests for your data access layer, you have probably used some generic mocking tool offered by popular providers like Mockito, jmock, mockrunner, or even DBUnit. With jOOQ, you can take advantage of the built-in JDBC mock API that allows you to emulate a simple database on the JDBC level for precisely those SQL/JDBC use cases supported by jOOQ.

> *(!) Disclaimer: The general idea of mocking a JDBC connection with this jOOQ API is to provide quick workarounds, injection points, etc. using a very simple JDBC abstraction. It is NOT RECOMMENDED to emulate an entire database (including complex state transitions, transactions, locking, etc.) using this mock API. Once you have this requirement, please consider using an actual*

database product instead for integration testing, rather than implementing your test database inside of a MockDataProvider.

Mocking the JDBC API

JDBC is a very complex API. It takes a lot of time to write a useful and correct mock implementation, implementing at least these interfaces:

- java.sql.Connection
- java.sql.Statement
- java.sql.PreparedStatement
- java.sql.CallableStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData

Optionally, you may even want to implement interfaces, such as java.sql.Array, java.sql.Blob, java.sql.Clob, and many others. In addition to the above, you might need to find a way to simultaneously support incompatible JDBC minor versions, such as 4.0, 4.1

Using jOOQ's own mock API

This work is greatly simplified, when using jOOQ's own mock API. The org.jooq.tools.jdbc package contains all the essential implementations for both JDBC 4.0 and 4.1, which are needed to mock JDBC for jOOQ. In order to write mock tests, provide the jOOQ Configuration with a MockConnection, and implement the MockDataProvider:

```
// Initialise your data provider (implementation further down):
MockDataProvider provider = new MyProvider();
MockConnection connection = new MockConnection(provider);

// Pass the mock connection to a jOOQ DSLContext:
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);

// Execute queries transparently, with the above DSLContext:
Result<BookRecord> result = create.selectFrom(BOOK).where(BOOK.ID.eq(5)).fetch();
```

As you can see, the configuration setup is simple. Now, the MockDataProvider acts as your single point of contact with JDBC / jOOQ. It unifies any of these execution modes, transparently:

- Statements without results
- Statements without results but with generated keys
- Statements with results
- Statements with several results
- Batch statements with single queries and multiple bind value sets
- Batch statements with multiple queries and no bind values

The above are the execution modes supported by jOOQ. Whether you're using any of jOOQ's various fetching modes (e.g. pojo fetching, lazy fetching, many fetching, later fetching) is irrelevant, as those modes are all built on top of the standard JDBC API.

Implementing MockDataProvider

Now, here's how to implement MockDataProvider:

```
public class MyProvider implements MockDataProvider {

    @Override
    public MockResult[] execute(MockExecuteContext ctx) throws SQLException {

        // You might need a DSLContext to create org.jooq.Result and org.jooq.Record objects
        DSLContext create = DSL.using(SQLDialect.ORACLE);
        MockResult[] mock = new MockResult[1];

        // The execute context contains SQL string(s), bind values, and other meta-data
        String sql = ctx.sql();

        // Exceptions are propagated through the JDBC and jOOQ APIs
        if (sql.toUpperCase().startsWith("DROP")) {
            throw new SQLException("Statement not supported: " + sql);
        }

        // You decide, whether any given statement returns results, and how many
        else if (sql.toUpperCase().startsWith("SELECT")) {

            // Always return one record
            Result<Record2<Integer, String>> result = create.newResult(AUTHOR.ID, AUTHOR.LAST_NAME);
            result.add(create
                .newRecord(AUTHOR.ID, AUTHOR.LAST_NAME)
                .values(1, "Orwell"));
            mock[0] = new MockResult(1, result);
        }

        // You can detect batch statements easily
        else if (ctx.batch()) {
            // [...]
        }

        return mock;
    }
}
```

Essentially, the MockExecuteContext contains all the necessary information for you to decide, what kind of data you should return. The MockResult wraps up two pieces of information:

- Statement.getUpdateCount(): The number of affected rows
- Statement.getResultSet(): The result set

You should return as many MockResult objects as there were query executions (in batch mode) or results (in fetch-many mode). Instead of an awkward java.sql.ResultSet, however, you can construct a "friendlier" org.jooq.Result with your own record types. The jOOQ mock API will use meta data provided with this Result in order to create the necessary JDBC java.sql.ResultSetMetaData

See the MockDataProvider Javadoc for a list of rules that you should follow.

4.20. Mock File Database

An alternative to the previous programmatic implementation of a mocking connection is the more declarative approach using a org.jooq.tools.jdbc.MockFileDatabase, which reads SQL statements and their corresponding result sets directly from a file.

> *(!) Disclaimer: The general idea of mocking a JDBC connection with this jOOQ API is to provide quick workarounds, injection points, etc. using a very simple JDBC abstraction. It is NOT RECOMMENDED to emulate an entire database (including complex state transitions, transactions, locking, etc.) using this mock API. Once you have this requirement, please consider using an actual database product instead for integration testing, rather than implementing your test database inside of a MockDataProvider.*

Assuming the following file content:

```
# All lines with a leading hash are ignored. This is the MockFileDatabase comment syntax
-- SQL comments are parsed and passed to the SQL statement
/* The same is true for multi-line SQL comments */
select 'A';
> A
> -
> A
@ rows: 1

select 'A', 'B' union all 'C', 'D';
> A B
> - -
> A B
> C D
@ rows: 2

# Statements without result sets just leave that section empty
update t set x = 1;
@ rows: 3
```

The above syntax consists of the following elements to define an individual statement:

- MockFileDatabase comments are any line with a leading hash ("#") symbol. They are ignored when reading the file
- SQL comments are part of the SQL statement
- A SQL statement always starts on a new line and ends with a semi colon (;), which is the last symbol on the line (apart from whitespace)
- If the statement has a result set, it immediately succeeds the SQL statement and is prefixed by angle brackets and a whitespace ("> "). Any format that is accepted by DSLContext.fetchFromTXT(), DSLContext.fetchFromJSON(), or DSLContext.fetchFromXML() is accepted.
- The statement is always terminated by the row count, which is prefixed by an at symbol, the "rows" keyword, and a double colon ("@ rows:").

The above database supports exactly two statements in total, and is completely stateless (e.g. an INSERT statement cannot be made to affect the results of a subsequent SELECT statement on the same table). It can be loaded through the MockFileDatabase can be used as follows:

```
// Initialise your data provider:
MockFileDatabase provider = new MockFileDatabase(new File("/path/to/db.txt"));
MockConnection connection = new MockConnection(provider);

// Pass the mock connection to a jOOQ DSLContext:
DSLContext create = DSL.using(connection, SQLDialect.POSTGRES);

// Execute queries transparently, with the above DSLContext:
Result<?> result = create.select(inline("A")).fetch();
Result<?> result = create.select(inline("A"), inline("B")).fetch();

// Queries that are not listed in the MockFileDatabase will simply fail
Result<?> result = create.select(inline("C")).fetch();
```

In situations where the expected set of queries are well-defined, the MockFileDatabase can offer a very effective way of mocking parts of the database engine, without offering the complete functionality of the programmatic mocking connection.

Matching statements using regular expressions

Alternatively, regular expressions can be used to match statements in order of definition in the file.

```
# Regardless of the number of columns, if selecting 'A' as the first column,
# always return a single row containing columns A and B
select 'A', .*;
> A B
> - -
> A B
@ rows: 1

# All other select statements are assumed to return only a column A
select .*;
> A
> -
> A
@ rows: 1
```

The same rules apply as before. The first matching statement will be applied for any given input statement.

This feature is "opt-in", so it has to be configured appropriately:

```
// Initialise your data provider:
MockFileDatabase provider = new MockFileDatabase(new MockFileDatabaseConfiguration()
    .source(new File("/path/to/db.txt"))
    .patterns(true) // Turn on regular expressions here
);
MockConnection connection = new MockConnection(provider);

// Pass the mock connection to a jOOQ DSLContext:
DSLContext create = DSL.using(connection, SQLDialect.POSTGRES);

// This returns a column A only
Result<?> result = create.select(inline("X")).fetch();

// This returns columns A and B
Result<?> result = create.select(inline("A"), inline("B"), inline("C")).fetch();
```

4.21. Parsing Connection

As previously discussed in the manual's section about the SQL parser, jOOQ exposes a full-fledged SQL parser through DSLContext.parser(), and often more interestingly: Through DSLContext.parsingConnection().

A parsing connection is a JDBC java.sql.Connection, which proxies all commands through the jOOQ parser, transforming the inbound SQL statement (and bind variables) given the entirety of jOOQ's Configuration, including SQLDialect, Settings (e.g. formatting SQL or inlining bind variables) and much more. This allows for transparent SQL transformation of the SQL produced by any JDBC client (including JPA!). Here's a simple usage example:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration).parsingConnection();
     Statement s = c.createStatement();

     // This syntax is not supported in Oracle, but thanks to the parser and jOOQ,
     // it will run on Oracle and produce the expected result
     ResultSet rs = s.executeQuery("SELECT * FROM (VALUES (1, 'a'), (2, 'b')) t(x, y)")) {

    while (rs.next())
        System.out.println("x: " + rs.getInt(1) + ", y: " + rs.getString());
}
```

Running the above statement will yield:

```
x: 1, y: a
x: 2, y: b
```

4.22. Diagnostics

jOOQ includes a powerful diagnostics SPI, which can be used to detect problems and inefficiencies on different levels of your database interaction:

- On the jOOQ API level
- On the JDBC level
- On the SQL level

Just like the parsing connection, which was documented in the previous section, this functionality does not depend on using the jOOQ API in a client application, but can expose itself through a JDBC java.sql.Connection that proxies your real database connection.

```
// A custom DiagnosticsListener SPI implementation
class MyDiagnosticsListener extends DefaultDiagnosticsListener {
    // Override methods here
}
```

And then:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new MyDiagnosticsListener()))
                      .diagnosticsConnection();
     Statement s = c.createStatement()) {

    // The tooManyRowsFetched() event is triggered.
    // ---------------------------------------------
    // This logic does not consume the entire ResultSet. There is more than one row
    // ready to be fetched into the client, but the client only fetches one row.
    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book WHERE id > 1")) {
        if (rs.next())
            System.out.println("ID: " + rs.getInt(1) + ", title: " + rs.getInt(2));
    }

    // The duplicateStatements() event is triggered.
    // ---------------------------------------------
    // The statement is the same as the previous one, apart from a different "bind variable".
    // Unfortunately, no actual bind variables were used, which may
    // 1) hint at a SQL injection risk
    // 2) can cause a lot of pressure / contention on execution plan caches and SQL parsers
    //
    // The tooManyColumnsFetched() event is triggered.
    // ---------------------------------------------
    // When iterating the ResultSet, we're actually only ever reading the TITLE column, never
    // the ID column. This means we probably should not have projected it in the first place
    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book WHERE id > 2")) {
        while (rs.next())
            System.out.println("Title: " + rs.getString(2));
    }
}
```

This feature incurs a certain overhead over normal operation as it requires:

- Parsing SQL statements and re-rendering them back to normalised SQL.
- Storing a limited size list of such normalised SQL in a cache to gather statistics on that cache.

The following sections describe each individual event, how it can happen, how and why it should be remedied.

4.22.1. Too Many Rows

The SPI method handling this event is tooManyRowsFetched()

If you're using jOOQ (or an ORM) to eagerly fetch your entire result set, then this will not be a problem in your code base, but when working with jOOQ's lazy fetching API or lazy streaming support, or as well as with JDBC java.sql.ResultSet directly, then it may certainly be possible that client code is aborting the iteration over the entire result set prematurely.

Why is it bad?

While it is definitely good not to fetch too many rows from a java.sql.ResultSet, it would be even better to communicate to the database that only a limited number of rows are going to be needed in the client, by using the LIMIT clause. Not only will this prevent the pre-allocation of some resources both in the client and in the server, but it opens up the possibility of much better execution plans. For instance, the optimiser may prefer to chose nested loop joins over hash joins if it knows that the loops can be aborted early.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class TooManyRows extends DefaultDiagnosticsListener {
    @Override
    public void tooManyRowsFetched(DiagnosticsContext ctx) {
        System.out.println("Consumed rows: " + ctx.resultSetConsumedRows());

        // This incurs overhead by consuming the ResultSet! Use only if needed.
        System.out.println("Fetched rows: " + ctx.resultSetFetchedRows());
    }
}
```

And then:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new TooManyRows()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT id FROM book")) {

        // Unlike "while", "if" only consumes the first row, here.
        if (rs.next())
            System.out.println("ID: " + rs.getInt(1));
    }
}
```

4.22.2. Too Many Columns

The SPI method handling this event is tooManyColumnsFetched()

A common problem with SQL is the high probability of having too many columns in the projection. This may be due to reckless usage of SELECT * or some refactoring which removes the need to select some of the projected columns, but the query was not adapted.

If the columns are consumed by some client (e.g. an ORM), then the jOOQ diagnostics have no way of knowing whether they were actually needed or not. But if they are never consumed from the JDBC java.sql.ResultSet, then we can say with certainty that too many columns have been projected.

© 2009 - 2025 by Data Geekery™ GmbH.

Why is it bad?

The drawbacks of projecting too many columns are manifold:

- Too much data is loaded, cached, transferred between server and client. The overall resource consumption of a system is too high if too many columns are projected. This can cause orders of magnitude of overhead in extreme cases!
- Locking could occur in cases where it otherwise wouldn't happen, because two conflicting queries actually don't really need to touch the same columns.
- The probability of using "covering indexes" (or "index only scans") on some tables decreases because of the unnecessary projection. This can have drastic effects!
- The probability of applying JOIN elimination decreases, because of the unnecessary projection. This is particularly true if you're querying views.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class TooManyColumns extends DefaultDiagnosticsListener {
    @Override
    public void tooManyColumnsFetched(DiagnosticsContext ctx) {
        System.out.println("Consumed columns: " + ctx.resultSetConsumedColumnCount());
        System.out.println("Fetched columns: " + ctx.resultSetFetchedColumnCount());
    }
}
```

And then:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new TooManyColumns()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book")) {

        // On none of the rows, we retrieve the TITLE column, so selecting it would not have been necessary.
        while (rs.next())
            System.out.println("ID: " + rs.getInt(1));
    }
}
```

4.22.3. Duplicate Statements

The SPI method handling this event is duplicateStatements().

A common source of overhead in databases that have an execution plan cache (or "cursor cache", etc.) are static statements, or prepared statements that differ only by "irrelevant" things:

- They may differ by white space
- They may differ by irrelevant syntactic elements (e.g. excess parentheses, or object name qualification, table aliasing, etc.)
- They may differ by input values, which are inlined into the statement rather than parameterised as bind values
- They may differ by the length of their IN predicate's IN-list sizes

Why is it bad?

There are two main problems:

- If the duplicate SQL appears in dynamic SQL (i.e. in generated SQL), then there is an indication that the database's parser and optimiser may have to do too much work parsing the various similar (but not identical) SQL queries and finding an execution plan for them, each time. In fact, it will find the *same* execution plan most of the time, but with some significant overhead. Depending on the query complexity, this overhead can easily go from milliseconds into several seconds, blocking important resources in the database. If duplicate SQL happens at peak load times, this problem can have a significant impact in production. It never affects your (single user) development environments, where the overhead of parsing duplicate SQL is manageable.
- If the duplicate SQL appears in static SQL, this can simply indicate that the query was copy pasted, and you might be able to refactor it. There's probably not any performance issue arising from duplicate static SQL

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class DuplicateStatements extends DefaultDiagnosticsListener {
    @Override
    public void duplicateStatements(DiagnosticsContext ctx) {

        // The statement that is being executed and which has duplicates
        System.out.println("Actual statement: " + ctx.actualStatement());

        // A normalised version of the actual statement, which is shared by all duplicates
        // This statement has "normal" whitespace, bind variables, IN-lists, etc.
        System.out.println("Normalised statement: " + ctx.normalisedStatement());

        // All the duplicate actual statements that have produced the same normalised
        // statement in the recent past.
        System.out.println("Duplicate statements: " + ctx.duplicateStatements());
    }
}
```

And then:

```
// Utility to run SQL on a new JDBC Statement
void run(String sql) {

    // Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
    try (Connection c = DSL.using(configuration.derive(new DuplicateStatements()))
                          .diagnosticsConnection();
         Statement s = c.createStatement();
         ResultSet rs = s.executeQuery(sql)) {

        while (rs.next()) {
            // Consume result set
        }
    }
}
    // Everything is fine with the first execution
    run("SELECT title FROM book WHERE id = 1");

    // This query is identical to the previous one, differing only in irrelevant white space
    run("SELECT title FROM book WHERE  id = 1");

    // This query is identical to the previous one, differing only in irrelevant additional parentheses
    run("SELECT title FROM book WHERE (id = 1)");

    // This query is identical to the previous one, differing only in what should be a bind variable
    run("SELECT title FROM book WHERE id = 2");

    // Everything is fine with the first execution of a new query that has never been seen
    run("SELECT title FROM book WHERE id IN (1, 2, 3, 4, 5)");

    // This query is identical to the previous one, differing only in what should be bind variables
    run("SELECT title FROM book WHERE id IN (1, 2, 3, 4, 5, 6)");
}
```

Unlike when detecting repeated statements, duplicate statement statistics are performed globally over all JDBC connections and data sources.

4.22.4. Repeated statements

The SPI method handling this event is repeatedStatements()

Sometimes, there is no other option than to repeat an identical (or similar, see duplicate statements) statement many times in a row, but often, it is a sign that your queries can be rewritten and your repeated statements should really be joined to a larger query.

Why is it bad?

This problem is usually referred to as the N+1 problem. A parent entity is loaded (often by an ORM), and its child entities are loaded lazily. Unfortunately, there were several parent instances, so for each parent instance, we're now loading a set of child instances, resulting in many many queries. This diagnostic detects if on the same connection, there is repeated execution of the same statement, even if it is not exactly identical.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class RepeatedStatement extends DefaultDiagnosticsListener {
    @Override
    public void repeatedStatements(DiagnosticsContext ctx) {

        // The statement that is being executed and which has duplicates
        System.out.println("Actual statement: " + ctx.actualStatement());

        // A normalised version of the actual statement, which is shared by all duplicates
        // This statement has "normal" whitespace, bind variables, IN-lists, etc.
        System.out.println("Normalised statement: " + ctx.normalisedStatement());

        // All the duplicate actual statements that have produced the same normalised
        // statement in the recent past.
        System.out.println("Repeated statements: " + ctx.repeatedStatements());
    }
}
```

And then:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new RepeatedStatement()))
                      .diagnosticsConnection();
     Statement s1 = c.createStatement();
     ResultSet a = s1.executeQuery("SELECT id FROM author WHERE first_name LIKE 'A%'")) {

    while (a.next()) {
        int id = a.getInt(1);

        // This query is run once for every author, when we could have joined the author table
        try (PreparedStatement s2 = c.prepareStatement("SELECT title FROM book WHERE author_id = ?")) {
            s2.setInt(1, id);

            try (ResultSet b = s2.executeQuery()) {
                while (b.next())
                    System.out.println("ID: " + id + ", title: " + b.getString(1));
            }
        }
    }
}
```

Unlike when detecting duplicate statements, repeated statement statistics are performed locally only, for a single JDBC Connection, or, if possible, for a transaction. Repeated statements in different transactions are usually not an indication of a problem.

© 2009 - 2025 by Data Geekery™ GmbH.

4.22.5. WasNull calls

The SPI methods handling these events are unnecessaryWasNullCall() and missingWasNullCall()

This problem appears only when using the JDBC API directly, as both jOOQ and most ORMs abstract over this JDBC legacy correctly. In JDBC, when fetching a primitive types (and some types are available only in their primitive form), a subsequent call to ResultSet.wasNull() is required, to see if the previous primitive type was really a NULL value, not a 0 or false value.

Why is it bad?

There are two misuses that can arise in this area:

- The call to wasNull() wasn't made when it should have been (nullable type, fetched as a primitive type), possibly resulting in wrong results in the client.
- The call to wasNull() was made too often, or when it did not need to have been made (non-nullable type, or types fetched as reference types), possibly resulting in a very slight performance overhead, depending on the driver.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class WasNull extends DefaultDiagnosticsListener {
    @Override
    public void unnecessaryWasNullCall(DiagnosticsContext ctx) {
        System.out.println("Unnecessary wasNull() call: " + ctx.resultSetUnnecessaryWasNullCall());
    }

    @Override
    public void missingWasNullCall(DiagnosticsContext ctx) {
        System.out.println("Missing wasNull() call: " + ctx.resultSetMissingWasNullCall());
    }
}
```

And then:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new WasNull()))
                      .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT year_of_birth FROM author")) {

        // The YEAR_OF_BIRTH column is nullable, so a 0 int could really mean null
        while (rs.next())
            System.out.println("Year of birth: " + rs.getInt(1));
    }
}
```

4.23. Logging with LoggerListener

For development purposes, jOOQ logs all SQL queries and fetched result sets to its internal DEBUG logger, which is implemented as an execute listener. By default, execute logging is activated in the jOOQ Settings. In order to see any DEBUG log output, put slf4j on jOOQ's classpath or module path along with their respective configuration. A sample log4j configuration can be seen here:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{ABSOLUTE} %5p [%-50c{4}] - %m%n"/>
        </Console>
    </Appenders>

    <Loggers>
        <!-- SQL execution logging is logged to the LoggerListener logger at DEBUG level -->
        <Logger name="org.jooq.tools.LoggerListener" level="debug">
            <AppenderRef ref="Console"/>
        </Logger>

        <!-- Other jOOQ related debug log output -->
        <Logger name="org.jooq" level="debug">
            <AppenderRef ref="Console"/>
        </Logger>

        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

With the above configuration, let's fetch some data with jOOQ

```
create.select(BOOK.ID, BOOK.TITLE).from(BOOK).orderBy(BOOK.ID).limit(1, 2).fetch();
```

The above query may result in the following log output:

```
Executing query          : select "BOOK"."ID", "BOOK"."TITLE" from "BOOK" order by "BOOK"."ID" asc limit ? offset ?
-> with bind values      : select "BOOK"."ID", "BOOK"."TITLE" from "BOOK" order by "BOOK"."ID" asc limit 2 offset 1
Fetched result           : +----+------------+
                         : |  ID|TITLE       |
                         : +----+------------+
                         : |   2|Animal Farm |
                         : |   3|O Alquimista|
                         : +----+------------+
```

Essentially, jOOQ will log

- The SQL statement as rendered to the prepared statement
- The SQL statement with inlined bind values (for improved debugging)
- The first 5 records of the result. This is formatted using jOOQ's text export

If you wish to use your own logger (e.g. avoiding printing out sensitive data), you can deactivate jOOQ's logger using your custom settings and implement your own execute listener logger.

4.24. Logging Connection

In addition to logging things on a jOOQ level, there is also the option of logging things on the JDBC level using the org.jooq.tools.jdbc.LoggingConnection. This connection acts as a JDBC proxy, intercepting all relevant calls in order to DEBUG log relevant information to slfj4 or java.util.logging.

```
try (Connection logging = new LoggingConnection(datasource.getConnection())) {

    // Run statements directly with JDBC
    try (Statement s = c.createStatement()) {
        s.executeUpdate("INSERT INTO author (id, first_name, last_name) VALUES (3, 'William', 'Shakespeare')");
    }

    // ... or wrap the connection with jOOQ
    DSL.using(logging, dialect)
       .insertInto(AUTHOR)
       .columns(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values(3, "William", "Shakespeare")
       .execute();
}
```

4.25. Performance considerations

Many users may have switched from higher-level abstractions such as Hibernate to jOOQ, because of Hibernate's difficult-to-manage performance, when it comes to large database schemas and complex second-level caching strategies. However, jOOQ itself is not a lightweight database abstraction framework, and it comes with its own overhead. Please be sure to consider the following points:

- It takes some time to construct jOOQ queries. If you can reuse the same queries, you might cache them. Beware of thread-safety issues, though, as jOOQ's Configuration is not necessarily threadsafe, and queries are "attached" to their creating DSLContext
- It takes some time to render SQL strings. Internally, jOOQ reuses the same java.lang.StringBuilder for the complete query, but some rendering elements may take their time. You could, of course, cache SQL generated by jOOQ and prepare your own java.sql.PreparedStatement objects
- It takes some time to bind values to prepared statements. jOOQ does not keep any open prepared statements, internally. Use a sophisticated connection pool, that will cache prepared statements and inject them into jOOQ through the standard JDBC API
- It takes some time to fetch results. By default, jOOQ will always fetch the complete java.sql.ResultSet into memory. Use lazy fetching to prevent that, and scroll over an open underlying database cursor

Optimise wisely

Don't be put off by the above paragraphs. You should optimise wisely, i.e. only in places where you really need very high throughput to your database. jOOQ's overhead compared to plain JDBC is typically less than 1ms per query.

4.26. Alternative execution models

Just because you can, doesn't mean you must. In this chapter, we'll show how you can use jOOQ to generate SQL statements that are then executed with other APIs, such as Spring's JdbcTemplate, or Hibernate.

> *(!) Please note that we strongly recommend **executing queries directly with jOOQ** and to **use jOOQ's code generator**!*

4.26.1. Using jOOQ with Spring's JdbcTemplate

A lot of people are using Spring's useful org.springframework.jdbc.core.JdbcTemplate in their projects to simplify common JDBC interaction patterns, such as:

- Variable binding
- Result mapping
- Exception handling

When adding jOOQ to a project that is using JdbcTemplate extensively, a pragmatic first step is to use jOOQ as a SQL builder and pass the query string and bind variables to JdbcTemplate for execution. For instance, you may have the following class to store authors and their number of books in our stores:

```
public class AuthorAndBooks {
    public final String firstName;
    public final String lastName;
    public final int books;

    public AuthorAndBooks(String firstName, String lastName, int books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}
```

You can then write the following code

```
// The jOOQ part stays the same as always:
Book b = BOOK.as("b");
Author a = AUTHOR.as("a");
BookStore s = BOOK_STORE.as("s");
BookToBookStore t = BOOK_TO_BOOK_STORE.as("t");

ResultQuery<Record3<String, String, Integer>> query =
create.select(a.FIRST_NAME, a.LAST_NAME, countDistinct(s.NAME))
      .from(a)
      .join(b).on(b.AUTHOR_ID.equal(a.ID))
      .join(t).on(t.BOOK_ID.equal(b.ID))
      .join(s).on(t.BOOK_STORE_NAME.equal(s.NAME))
      .groupBy(a.FIRST_NAME, a.LAST_NAME)
      .orderBy(countDistinct(s.NAME).desc());

// But instead of executing the above query, we'll send the SQL string and the bind values to JdbcTemplate:
JdbcTemplate template = new JdbcTemplate(dataSource);
List<AuthorAndBooks> result = template.query(
    query.getSQL(),
    (r, i) -> new AuthorAndBooks(
        r.getString(1),
        r.getString(2),
        r.getInt(3)
    ),
    query.getBindValues().toArray()
);
```

This approach helps you gradually migrate from using JdbcTemplate to a jOOQ-only execution model.

4.26.2. Using jOOQ with JPA

These sections will show how to use jOOQ with JPA's native query API in order to fetch tuples or managed entities using the Java EE standards.

(!) The goal of these sections is to describe how to do this, if you have strong reasons to do so. Mostly, however, you're better off executing your queries directly with jOOQ, especially if you

*want to use jOOQ's more advanced features. **This blog post illustrates various reason why it's better to execute queries directly with jOOQ**.*

In all of the following sections, let's assume we have the following JPA entities to model our database:

```
@Entity
@Table(name = "book")
public class JPABook {

    @Id
    public int id;

    @Column(name = "title")
    public String title;

    @ManyToOne
    public JPAAuthor author;

    @Override
    public String toString() {
        return "JPABook [id=" + id + ", title=" + title + ", author=" + author + "]";
    }
}

@Entity
@Table(name = "author")
public class JPAAuthor {

    @Id
    public int id;

    @Column(name = "first_name")
    public String firstName;

    @Column(name = "last_name")
    public String lastName;

    @OneToMany(mappedBy = "author")
    public Set<JPABook> books;

    @Override
    public String toString() {
        return "JPAAuthor [id=" + id + ", firstName=" + firstName +
               ", lastName=" + lastName + ", book size=" + books.size() + "]";
    }
}
```

4.26.2.1. Using jOOQ with JPA Native Query

*(!) The goal of these sections is to describe how to do this, if you have strong reasons to do so. Mostly, however, you're better off executing your queries directly with jOOQ, especially if you want to use jOOQ's more advanced features. **This blog post illustrates various reason why it's better to execute queries directly with jOOQ**.*

If your query doesn't really map to JPA entities, you can fetch ordinary, untyped Object[] representations for your database records by using the following utility method:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL());

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL());

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

This way, you can construct complex, type safe queries using the jOOQ API and have your jakarta.persistence.EntityManager execute it with all the transaction semantics attached:

```
List<Object[]> books =
nativeQuery(em, DSL.using(configuration)
    .select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, BOOK.TITLE)
    .from(AUTHOR)
    .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
    .orderBy(BOOK.ID));

books.forEach((Object[] book) -> System.out.println(book[0] + " " + book[1] + " wrote " + book[2]));
```

4.26.2.2. Using jOOQ with JPA entities

(!) The goal of these sections is to describe how to do this, if you have strong reasons to do so. Mostly, however, you're better off executing your queries directly with jOOQ, especially if you want to use jOOQ's more advanced features. ***This blog post illustrates various reason why it's better to execute queries directly with jOOQ****.*

The simplest way to fetch entities via the native query API is by passing the entity class along to the native query method. The following example maps jOOQ query results to JPA entities (from the previous section). Just add the following utility method:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, Class<E> type) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), type);

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    // There's an unsafe cast here, but we can be sure that we'll get the right type from JPA
    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query, Class<E> type) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), type);

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

With the above simple API, we're ready to write complex jOOQ queries and map their results to JPA entities:

```
List<JPAAuthor> authors =
nativeQuery(em,
    DSL.using(configuration)
       .select()
       .from(AUTHOR)
       .orderBy(AUTHOR.ID)
, JPAAuthor.class);

authors.forEach(author -> {
    System.out.println(author.firstName + " " + author.lastName + " wrote");

    books.forEach(book -> {
        System.out.println("  " + book.title);
    });
});
```

4.26.2.3. Using jOOQ with JPA EntityResult

(!) The goal of these sections is to describe how to do this, if you have strong reasons to do so. Mostly, however, you're better off executing your queries directly with jOOQ, especially if you want to use jOOQ's more advanced features. ***This blog post illustrates various reason why it's better to execute queries directly with jOOQ****.*

While JPA specifies how the mapping should be implemented (e.g. using jakarta.persistence.SqlResultSetMapping), there are no limitations regarding how you want to generate the SQL statement. The following, simple example shows how you can produce JPABook and JPAAuthor entities (from the previous section) from a jOOQ-generated SQL statement.

In order to do so, we'll need to specify the SqlResultSetMapping. This can be done on any entity, and in this case, we're using jakarta.persistence.EntityResult:

```
@SqlResultSetMapping(
    name = "bookmapping",
    entities = {
        @EntityResult(
            entityClass = JPABook.class,
            fields = {
                @FieldResult(name = "id", column = "b_id"),
                @FieldResult(name = "title", column = "b_title"),
                @FieldResult(name = "author", column = "b_author_id")
            }
        ),
        @EntityResult(
            entityClass = JPAAuthor.class,
            fields = {
                @FieldResult(name = "id", column = "a_id"),
                @FieldResult(name = "firstName", column = "a_first_name"),
                @FieldResult(name = "lastName", column = "a_last_name")
            }
        )
    }
)
```

Note how we need to map between:

- FieldResult.name(), which corresponds to the entity's attribute name
- FieldResult.column(), which corresponds to the SQL result's column name

With the above boilerplate in place, we can now fetch entities using jOOQ and JPA:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, String resultSetMapping) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), resultSetMapping);

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, String resultSetMapping) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), resultSetMapping);

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

Using the above API

Now that we have everything setup, we can use the above API to run a jOOQ query to fetch JPA entities like this:

```
List<Object[]> result =
nativeQuery(em,
    DSL.using(configuration
       .select(
           AUTHOR.ID.as("a_id"),
           AUTHOR.FIRST_NAME.as("a_first_name"),
           AUTHOR.LAST_NAME.as("a_last_name"),
           BOOK.ID.as("b_id"),
           BOOK.AUTHOR_ID.as("b_author_id"),
           BOOK.TITLE.as("b_title")
       )
       .from(AUTHOR)
       .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
       .orderBy(BOOK.ID)),
    "bookmapping" // The name of the SqlResultSetMapping
);

result.forEach((Object[] entities) -> {
    JPAAuthor author = (JPAAuthor) entities[1];
    JPABook book = (JPABook) entities[0];

    System.out.println(author.firstName + " " + author.lastName + " wrote " + book.title);
});
```

The entities are now ready to be modified and persisted again.

Caveats:

- We have to reference the result set mapping by name (a String) - there is no type safety involved here
- We don't know the type contained in the resulting List - there is a potential for ClassCastException
- The results are in fact a list of Object[], with the individual entities listed in the array, which need explicit casting

5. Code generation

While optional, source code generation is one of jOOQ's main assets if you wish to increase developer productivity. jOOQ's code generator takes your database schema and reverse-engineers it into a set of Java classes modelling tables, records, sequences, POJOs, DAOs, stored procedures, user-defined types and many more.

The essential ideas behind source code generation are these:

- Increased IDE support: Type your Java code directly against your database schema, with all type information available
- Type-safety: When your database schema changes, your generated code will change as well. Removing columns will lead to compilation errors, which you can detect early.

The following chapters will show how to configure the code generator and how to generate various artefacts.

5.1. Configuration and setup of the generator

There are three binaries available with jOOQ, to be downloaded from https://www.jooq.org/download or from Maven central:

- jooq-3.12.4.jar
  The main library that you will include in your application to run jOOQ
- jooq-meta-3.12.4.jar
  The utility that you will include in your build to navigate your database schema for code generation. This can be used as a schema crawler as well.
- jooq-codegen-3.12.4.jar
  The utility that you will include in your build to generate your database schema

Configure jOOQ's code generator

You need to tell jOOQ some things about your database connection. Here's an example of how to do it for an Oracle database

XML (standalone and maven)

```
<configuration>
  <!-- Configure the database connection here -->
  <jdbc>
    <driver>oracle.jdbc.OracleDriver</driver>
    <url>jdbc:oracle:thin:@[your jdbc connection parameters]</url>
    <user>[your database user]</user>
    <password>[your database password]</password>

    <!-- You can also pass user/password and other JDBC properties in the optional properties tag: -->
    <properties>
      <property><key>user</key><value>[db-user]</value></property>
      <property><key>password</key><value>[db-password]</value></property>
    </properties>
  </jdbc>

  <generator>
    <database>
      <!-- The database dialect from jooq-meta. Available dialects are
           named org.jooq.meta.[database].[database]Database.

           Natively supported values are:

               org.jooq.meta.ase.ASEDatabase
               org.jooq.meta.auroramysql.AuroraMySQLDatabase
               org.jooq.meta.aurorapostgres.AuroraPostgresDatabase
               org.jooq.meta.cubrid.CUBRIDDatabase
               org.jooq.meta.db2.DB2Database
               org.jooq.meta.derby.DerbyDatabase
               org.jooq.meta.firebird.FirebirdDatabase
               org.jooq.meta.h2.H2Database
               org.jooq.meta.hana.HANADatabase
               org.jooq.meta.hsqldb.HSQLDBDatabase
               org.jooq.meta.informix.InformixDatabase
               org.jooq.meta.ingres.IngresDatabase
               org.jooq.meta.mariadb.MariaDBDatabase
               org.jooq.meta.mysql.MySQLDatabase
               org.jooq.meta.oracle.OracleDatabase
               org.jooq.meta.postgres.PostgresDatabase
               org.jooq.meta.redshift.RedshiftDatabase
               org.jooq.meta.sqldatawarehouse.SQLDataWarehouseDatabase
               org.jooq.meta.sqlite.SQLiteDatabase
               org.jooq.meta.sqlserver.SQLServerDatabase
               org.jooq.meta.sybase.SybaseDatabase
               org.jooq.meta.teradata.TeradataDatabase
               org.jooq.meta.vertica.VerticaDatabase

           This value can be used to reverse-engineer generic JDBC DatabaseMetaData (e.g. for MS Access)

               org.jooq.meta.jdbc.JDBCDatabase

           This value can be used to reverse-engineer standard jOOQ-meta XML formats

               org.jooq.meta.xml.XMLDatabase

           This value can be used to reverse-engineer schemas defined by SQL files
           (requires jooq-meta-extensions dependency)

               org.jooq.meta.extensions.ddl.DDLDatabase

           This value can be used to reverse-engineer schemas defined by JPA annotated entities
           (requires jooq-meta-extensions dependency, or jooq-meta-extensions-hibernate starting from jOOQ 3.14)

               org.jooq.meta.extensions.jpa.JPADatabase

           You can also provide your own org.jooq.meta.Database implementation
           here, if your database is currently not supported -->
      <name>org.jooq.meta.oracle.OracleDatabase</name>

      <!-- All elements that are generated from your schema (A Java regular expression.
           Use the pipe to separate several expressions) Watch out for
           case-sensitivity. Depending on your database, this might be
           important!

           You can create case-insensitive regular expressions using this syntax: (?i:expr)

           Whitespace is ignored and comments are possible.
           -->
      <includes>.*</includes>

      <!-- All elements that are excluded from your schema (A Java regular expression.
           Use the pipe to separate several expressions). Excludes match before
           includes, i.e. excludes have a higher priority -->
      <excludes>
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      </excludes>

      <!-- The schema that is used locally as a source for meta information.
           This could be your development schema or the production schema, etc
           This cannot be combined with the schemata element.

           If left empty, jOOQ will generate all available schemata. See the
           manual's next section to learn how to generate several schemata -->
      <inputSchema>[your database schema / owner / name]</inputSchema>
    </database>

    <!-- Generation flags: See advanced configuration properties -->
    <generate/>

    <target>
      <!-- The destination package of your generated classes (within the
           destination directory)

           jOOQ may append the schema name to this package if generating multiple schemas,
           e.g. org.jooq.your.packagename.schema1
                org.jooq.your.packagename.schema2 -->
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // Configure the database connection here
  .withJdbc(new Jdbc()
    .withDriver("oracle.jdbc.OracleDriver")
    .withUrl("jdbc:oracle:thin:@[your jdbc connection parameters]")
    .withUser("[your database user]")
    .withPassword("[your database password]")

    // You can also pass user/password and other JDBC properties in the optional properties tag:
    .withProperties(
      new Property()
        .withKey("user")
        .withValue("[db-user]"),
      new Property()
        .withKey("password")
        .withValue("[db-password]")
    )
  )
  .withGenerator(new Generator()
    .withDatabase(new Database()

      // The database dialect from jooq-meta. Available dialects are
      // named org.jooq.meta.[database].[database]Database.
      //
      // Natively supported values are:
      //
      // org.jooq.meta.ase.ASEDatabase
      // org.jooq.meta.auroramysql.AuroraMySQLDatabase
      // org.jooq.meta.aurorapostgres.AuroraPostgresDatabase
      // org.jooq.meta.cubrid.CUBRIDDatabase
      // org.jooq.meta.db2.DB2Database
      // org.jooq.meta.derby.DerbyDatabase
      // org.jooq.meta.firebird.FirebirdDatabase
      // org.jooq.meta.h2.H2Database
      // org.jooq.meta.hana.HANADatabase
      // org.jooq.meta.hsqldb.HSQLDBDatabase
      // org.jooq.meta.informix.InformixDatabase
      // org.jooq.meta.ingres.IngresDatabase
      // org.jooq.meta.mariadb.MariaDBDatabase
      // org.jooq.meta.mysql.MySQLDatabase
      // org.jooq.meta.oracle.OracleDatabase
      // org.jooq.meta.postgres.PostgresDatabase
      // org.jooq.meta.redshift.RedshiftDatabase
      // org.jooq.meta.sqldatawarehouse.SQLDataWarehouseDatabase
      // org.jooq.meta.sqlite.SQLiteDatabase
      // org.jooq.meta.sqlserver.SQLServerDatabase
      // org.jooq.meta.sybase.SybaseDatabase
      // org.jooq.meta.teradata.TeradataDatabase
      // org.jooq.meta.vertica.VerticaDatabase
      //
      // This value can be used to reverse-engineer generic JDBC DatabaseMetaData (e.g. for MS Access)
      //
      // org.jooq.meta.jdbc.JDBCDatabase
      //
      // This value can be used to reverse-engineer standard jOOQ-meta XML formats
      //
      // org.jooq.meta.xml.XMLDatabase
      //
      // This value can be used to reverse-engineer schemas defined by SQL files
      // (requires jooq-meta-extensions dependency)
      //
      // org.jooq.meta.extensions.ddl.DDLDatabase
      //
      // This value can be used to reverse-engineer schemas defined by JPA annotated entities
      // (requires jooq-meta-extensions dependency, or jooq-meta-extensions-hibernate starting from jOOQ 3.14)
      //
      // org.jooq.meta.extensions.jpa.JPADatabase
      //
      // You can also provide your own org.jooq.meta.Database implementation
      // here, if your database is currently not supported
      .withName("org.jooq.meta.oracle.OracleDatabase")

      // All elements that are generated from your schema (A Java regular expression.
      // Use the pipe to separate several expressions) Watch out for
      // case-sensitivity. Depending on your database, this might be
      // important!
      //
      // You can create case-insensitive regular expressions using this syntax: (?i:expr)
      //
      // Whitespace is ignored and comments are possible.
      .withIncludes(".*")

      // All elements that are excluded from your schema (A Java regular expression.
      // Use the pipe to separate several expressions). Excludes match before
      // includes, i.e. excludes have a higher priority
      .withExcludes("""
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      """)

      // The schema that is used locally as a source for meta information.
      // This could be your development schema or the production schema, etc
      // This cannot be combined with the schemata element.
      //
      // If left empty, jOOQ will generate all available schemata. See the
      // manual's next section to learn how to generate several schemata
      .withInputSchema("[your database schema / owner / name]")
    )

    // Generation flags: See advanced configuration properties
    .withGenerate()
    .withTarget(new Target()

      // The destination package of your generated classes (within the
      // destination directory)
```

See the configuration XSD and programmatic code generation for more details.

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)

Gradle (third party)

```
generationTool {

  // Configure the database connection here
  jdbc {
    driver = "oracle.jdbc.OracleDriver"
    url = "jdbc:oracle:thin:@[your jdbc connection parameters]"
    user = "[your database user]"
    password = "[your database password]"

    // You can also pass user/password and other JDBC properties in the optional properties tag:
    properties {
      property {
        key = "user"
        value = "[db-user]"
      }
      property {
        key = "password"
        value = "[db-password]"
      }
    }
  }
  generator {
    database {

      // The database dialect from jooq-meta. Available dialects are
      // named org.jooq.meta.[database].[database]Database.
      //
      // Natively supported values are:
      //
      // org.jooq.meta.ase.ASEDatabase
      // org.jooq.meta.auroramysql.AuroraMySQLDatabase
      // org.jooq.meta.aurorapostgres.AuroraPostgresDatabase
      // org.jooq.meta.cubrid.CUBRIDDatabase
      // org.jooq.meta.db2.DB2Database
      // org.jooq.meta.derby.DerbyDatabase
      // org.jooq.meta.firebird.FirebirdDatabase
      // org.jooq.meta.h2.H2Database
      // org.jooq.meta.hana.HANADatabase
      // org.jooq.meta.hsqldb.HSQLDBDatabase
      // org.jooq.meta.informix.InformixDatabase
      // org.jooq.meta.ingres.IngresDatabase
      // org.jooq.meta.mariadb.MariaDBDatabase
      // org.jooq.meta.mysql.MySQLDatabase
      // org.jooq.meta.oracle.OracleDatabase
      // org.jooq.meta.postgres.PostgresDatabase
      // org.jooq.meta.redshift.RedshiftDatabase
      // org.jooq.meta.sqldatawarehouse.SQLDataWarehouseDatabase
      // org.jooq.meta.sqlite.SQLiteDatabase
      // org.jooq.meta.sqlserver.SQLServerDatabase
      // org.jooq.meta.sybase.SybaseDatabase
      // org.jooq.meta.teradata.TeradataDatabase
      // org.jooq.meta.vertica.VerticaDatabase
      //
      // This value can be used to reverse-engineer generic JDBC DatabaseMetaData (e.g. for MS Access)
      //
      // org.jooq.meta.jdbc.JDBCDatabase
      //
      // This value can be used to reverse-engineer standard jOOQ-meta XML formats
      //
      // org.jooq.meta.xml.XMLDatabase
      //
      // This value can be used to reverse-engineer schemas defined by SQL files
      // (requires jooq-meta-extensions dependency)
      //
      // org.jooq.meta.extensions.ddl.DDLDatabase
      //
      // This value can be used to reverse-engineer schemas defined by JPA annotated entities
      // (requires jooq-meta-extensions dependency, or jooq-meta-extensions-hibernate starting from jOOQ 3.14)
      //
      // org.jooq.meta.extensions.jpa.JPADatabase
      //
      // You can also provide your own org.jooq.meta.Database implementation
      // here, if your database is currently not supported
      name = "org.jooq.meta.oracle.OracleDatabase"

      // All elements that are generated from your schema (A Java regular expression.
      // Use the pipe to separate several expressions) Watch out for
      // case-sensitivity. Depending on your database, this might be
      // important!
      //
      // You can create case-insensitive regular expressions using this syntax: (?i:expr)
      //
      // Whitespace is ignored and comments are possible.
      includes = ".*"

      // All elements that are excluded from your schema (A Java regular expression.
      // Use the pipe to separate several expressions). Excludes match before
      // includes, i.e. excludes have a higher priority
      excludes = """
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      """

      // The schema that is used locally as a source for meta information.
      // This could be your development schema or the production schema, etc
      // This cannot be combined with the schemata element.
      //
      // If left empty, jOOQ will generate all available schemata. See the
      // manual's next section to learn how to generate several schemata
      inputSchema = "[your database schema / owner / name]"
    }

    // Generation flags: See advanced configuration properties
    generate {}
    target {
```

See the configuration XSD and gradle code generation for more details.
There are also lots of advanced configuration parameters, which will be treated in the manual's section about advanced code generation features. Note, you can find the official XSD file for a formal specification at:
https://www.jooq.org/xsd/jooq-codegen-3.12.0.xsd

Run jOOQ code generation

Code generation works by calling this class with the above property file as argument.

```
org.jooq.codegen.GenerationTool /jooq-config.xml
```

Be sure that these elements are located on the classpath:

- jooq-3.12.4.jar, jooq-meta-3.12.4.jar, jooq-codegen-3.12.4.jar, reactive-streams-1.0.2.jar
- The JDBC driver you configured

A command-line example (for Windows, OSX/Linux/etc. will be similar)

- Put the XML configuration file, jooq*.jar and the JDBC driver into a directory, e.g. C:\temp\jooq
- Go to C:\temp\jooq
- Run java -cp jooq-3.12.4.jar;jooq-meta-3.12.4.jar;jooq-codegen-3.12.4.jar;reactive-streams-1.0.2.jar;[JDBC-driver].jar org.jooq.codegen.GenerationTool <XML-file>

Note that the XML configuration file can also be loaded from the classpath, in which case the path should be given as /path/to/my-configuration.xml

Run code generation from Eclipse

Of course, you can also run code generation from your IDE. In Eclipse, set up a project like this. Note that:

- this example uses jOOQ's log4j support by adding log4j.xml and log4j.jar to the project classpath.
- the actual jooq-3.12.4.jar, jooq-meta-3.12.4.jar, jooq-codegen-3.12.4.jar artefacts may contain version numbers in the file names.

Once the project is set up correctly with all required artefacts on the classpath, you can configure an Eclipse Run Configuration for org.jooq.codegen.GenerationTool.

With the XML file as an argument

And the classpath set up correctly

Finally, run the code generation and see your generated artefacts

Integrate generation with Maven

Using the official jOOQ-codegen-maven plugin, you can integrate source code generation in your Maven build process:

```
<plugin>

  <!-- Specify the maven code generator plugin -->
  <!-- Use org.jooq            for the Open Source Edition
           org.jooq.pro        for commercial editions,
           org.jooq.pro-java-8 for commercial editions with Java 8 support,
           org.jooq.pro-java-6 for commercial editions with Java 6 support,
           org.jooq.trial      for the free trial edition

       Note: Only the Open Source Edition is hosted on Maven Central.
             Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.12.4</version>

  <!-- The plugin should hook into the generate goal -->
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>

  <!-- Manage the plugin's dependency. In this example, we'll use a PostgreSQL database -->
  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.3</version>
    </dependency>
  </dependencies>

  <!-- Specify the plugin configuration.
       The configuration format is the same as for the standalone code generator -->
  <configuration>

    <!-- JDBC connection parameters -->
    <jdbc>
      <driver>org.postgresql.Driver</driver>
      <url>jdbc:postgresql:postgres</url>
      <user>postgres</user>
      <password>test</password>
    </jdbc>

    <!-- Generator parameters -->
    <generator>
      <database>
        <name>org.jooq.meta.postgres.PostgresDatabase</name>
        <includes>.*</includes>
        <excludes></excludes>
        <!-- In case your database supports catalogs, e.g. SQL Server:
        <inputCatalog>public</inputCatalog>
          -->
        <inputSchema>public</inputSchema>
      </database>
      <target>
        <packageName>org.jooq.codegen.maven.example</packageName>
        <directory>target/generated-sources/jooq</directory>
      </target>
    </generator>
  </configuration>
</plugin>
```

Use jOOQ generated classes in your application

Be sure, both jooq-3.12.4.jar and your generated package (see configuration) are located on your classpath. Once this is done, you can execute SQL statements with your generated classes.

5.2. Advanced generator configuration

In the previous section we have seen how jOOQ's source code generator is configured and run within a few steps. In this chapter we'll cover some advanced settings, individually.

5.2.1. Logging

This optional top level configuration element simply allows for overriding the log level of anything that has been specified by the runtime, e.g. in log4j or slf4j and is helpful for per-code-generation log configuration. For example, in order to mute everything that is less than WARN level, write:

XML (standalone and maven)

```
<configuration>
  <logging>WARN</logging>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withLogging(Logging.WARN)
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  logging = "WARN"
}
```

See the configuration XSD and gradle code generation for more details.
Available log levels are

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL

© 2009 - 2025 by Data Geekery™ GmbH.

5.2.2. Error handling

This optional top level configuration element allows configuring the action to be taken by the generator in case of unexpected exceptions encountered during the code generation.

XML (standalone and maven)

```
<configuration>

  <!-- Behaviour when encountering an exception. Defaults to FAIL -->
  <onError>FAIL</onError>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // Behaviour when encountering an exception. Defaults to FAIL
  .withOnError(OnError.FAIL)
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // Behaviour when encountering an exception. Defaults to FAIL
  onError = "FAIL"
}
```

See the configuration XSD and gradle code generation for more details.
The available error actions are:

- FAIL - The exception will be thrown and handled by the caller (e.g. Maven)
- LOG - The exception will be handled by the generator by logging it as a warning
- SILENT - The exception will be silently ignored by the generator

5.2.3. Jdbc

This optional top level configuration element allows for configuring a JDBC connection. By default, the jOOQ code generator requires an active JDBC connection to reverse engineer your database schema. For example, if you want to connect to a MySQL database, write this:

XML (standalone and maven)

```
<configuration>
  <jdbc>
    <driver>com.mysql.cj.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/testdb</url>

    <!-- "username" is a valid synonym for "user" -->
    <user>root</user>
    <password>secret</password>
  </jdbc>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withJdbc(new Jdbc()
    .withDriver("com.mysql.cj.jdbc.Driver")
    .withUrl("jdbc:mysql://localhost/testdb")

    // "username" is a valid synonym for "user"
    .withUser("root")
    .withPassword("secret")
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  jdbc {
    driver = "com.mysql.cj.jdbc.Driver"
    url = "jdbc:mysql://localhost/testdb"

    // "username" is a valid synonym for "user"
    user = "root"
    password = "secret"
  }
}
```

See the configuration XSD and gradle code generation for more details.
Note that when using the programmatic configuration API through the GenerationTool, you can also pass a pre-existing JDBC connection to the GenerationTool and leave this configuration element alone.

*(!) Make sure the JDBC driver is available to the code generator as a **code generator dependency***

Now, run the code generation with the JDBC URL specified as -Dcom.example.myproperty=jdbc:mysql://localhost/testdb, for example.

Optional JDBC properties

JDBC drivers allow for passing java.util.Properties to the JDBC driver when creating a connection. This is also supported in the code generator configuration with a list of key/value pairs as follows:

XML (standalone and maven)

```
<configuration>
  <jdbc>
    <driver>com.mysql.cj.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/testdb</url>
    <properties>
      <property>
        <key>user</key>
        <value>root</value>
      </property>
      <property>
        <key>password</key>
        <value>secret</value>
      </property>
    </properties>
  </jdbc>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withJdbc(new Jdbc()
    .withDriver("com.mysql.cj.jdbc.Driver")
    .withUrl("jdbc:mysql://localhost/testdb")
    .withProperties(
      new Property()
        .withKey("user")
        .withValue("root"),
      new Property()
        .withKey("password")
        .withValue("secret")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  jdbc {
    driver = "com.mysql.cj.jdbc.Driver"
    url = "jdbc:mysql://localhost/testdb"
    properties {
      property {
        key = "user"
        value = "root"
      }
      property {
        key = "password"
        value = "secret"
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Auto committing

jOOQ's code generator will use the driver's / connection's default auto commit flag. If for some reason you need to override this (e.g. in order to recover from failed transactions in PostgreSQL, by setting it to true), you can specify it here:

XML (standalone and maven)

```
<configuration>
  <jdbc>
    <autoCommit>true</autoCommit>
  </jdbc>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withJdbc(new Jdbc()
    .withAutoCommit(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  jdbc {
    autoCommit = true
  }
}
```

See the configuration XSD and gradle code generation for more details.

When the JDBC configuration is optional

There are some exceptions, where the JDBC connection does not need to be configured, for instance when using the JPADatabase (to reverse engineer JPA annotated entities) or when using the XMLDatabase (to reverse engineer an XML file). Please refer to the respective sections for more details.

5.2.4. Generator

This mandatory top level configuration element wraps all the remaining configuration elements related to code generation, including the overridable code generator class.

XML (standalone and maven)

```
<configuration>
  <generator>

    <!-- Optional: The fully qualified class name of the code generator. -->
    <name>...</name>



    <!-- Optional: The programmatic or configurative generator strategy. -->
    <strategy/>

    <!-- Optional: The jooq-meta configuration, configuring the information schema source. -->
    <database/>

    <!-- Optional: The jooq-codegen configuration, configuring the generated output content. -->
    <generate/>

    <!-- Optional: The generation output target -->
    <target/>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()

    // Optional: The fully qualified class name of the code generator.
    .withName(...)

    // Optional: The programmatic or configurative generator strategy.
    .withStrategy()

    // Optional: The jooq-meta configuration, configuring the information schema source.
    .withDatabase()

    // Optional: The jooq-codegen configuration, configuring the generated output content.
    .withGenerate()

    // Optional: The generation output target
    .withTarget()
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {

    // Optional: The fully qualified class name of the code generator.
    name = ...

    // Optional: The programmatic or configurative generator strategy.
    strategy {}

    // Optional: The jooq-meta configuration, configuring the information schema source.
    database {}

    // Optional: The jooq-codegen configuration, configuring the generated output content.
    generate {}

    // Optional: The generation output target
    target {}
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Specifying your own generator

The <name/> element allows for specifying a user-defined generator implementation. This is mostly useful when generating custom code sections, which can be added programmatically using the code generator's internal API. For more details, please refer to the relevant section of the manual.

> *(!) Make sure your custom generator's dependencies (or the generator itself, if referenced by name) is available to the code generator as a **code generator dependency***

Specifying a strategy

jOOQ by default applies standard Java naming schemes: PascalCase for classes, camelCase for members, methods, variables, parameters, UPPER_CASE_WITH_UNDERSCORES for constants and other literals. This may not be the desired default for your database, e.g. when you strongly rely on case-sensitive naming and if you wish to be able to search for names both in your Java code and in your database code (scripts, views, stored procedures) uniformly. For that purpose, you can override the <strategy/> element with your own implementation, either:

- programmatically
- configuratively

For more details, please refer to the relevant sections, above.

5.2.5. Database

This element wraps all the configuration elements that are used for the jooq-meta module, which reads the configured database meta data. In its simplest form, it can be left empty, when meaningful defaults will apply.

The two main elements in the <database/> element are <name/> and <properties>, which specify the class to implement the database meta data source, and an optional list of key/value parameters, which are described in the next chapter

Subsequent elements are:

5.2.5.1. Database name and properties

The two main elements in the <database/> element are <name/> which specifies the class to implement the database meta data source, and depending on that class, an optional list of key/value <properties/> (see later sections for details). An example for the XML Database:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.xml.XMLDatabase</name>
      <properties>
        <property>
          <key>dialect</key>
          <value>MYSQL</value>
        </property>
        <property>
          <key>xmlFile</key>
          <value>/path/to/database.xml</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.xml.XMLDatabase")
      .withProperties(
        new Property()
          .withKey("dialect")
          .withValue("MYSQL"),
        new Property()
          .withKey("xmlFile")
          .withValue("/path/to/database.xml")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.xml.XMLDatabase"
      properties {
        property {
          key = "dialect"
          value = "MYSQL"
        }
        property {
          key = "xmlFile"
          value = "/path/to/database.xml"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
The default <name/> if no name is supplied will be derived from the JDBC connection. If you want to specifically specify your SQL dialect's database name, any of these values will be supported by jOOQ, out of the box:

- org.jooq.meta.ase.ASEDatabase
- org.jooq.meta.auroramysql.AuroraMySQLDatabase
- org.jooq.meta.aurorapostgres.AuroraPostgresDatabase
- org.jooq.meta.cockroachdb.CockroachDBDatabase
- org.jooq.meta.db2.DB2Database
- org.jooq.meta.derby.DerbyDatabase
- org.jooq.meta.firebird.FirebirdDatabase
- org.jooq.meta.h2.H2Database
- org.jooq.meta.hana.HanaDatabase
- org.jooq.meta.hsqldb.HSQLDBDatabase
- org.jooq.meta.informix.InformixDatabase
- org.jooq.meta.ingres.IngresDatabase
- org.jooq.meta.mariadb.MariaDBDatabase
- org.jooq.meta.mysql.MySQLDatabase
- org.jooq.meta.oracle.OracleDatabase
- org.jooq.meta.postgres.PostgresDatabase
- org.jooq.meta.redshift.RedshiftDatabase
- org.jooq.meta.sqldatawarehouse.SQLDataWarehouseDatabase
- org.jooq.meta.sqlite.SQLiteDatabase
- org.jooq.meta.sqlserver.SQLServerDatabase
- org.jooq.meta.sybase.SybaseDatabase
- org.jooq.meta.teradata.TeradataDatabase
- org.jooq.meta.vertica.VerticaDatabase

Alternatively, you can also specify the following database if you want to reverse-engineer a generic JDBC java.sql.DatabaseMetaData source for an unsupported database version / dialect / etc:

- org.jooq.meta.jdbc.JDBCDatabase

Furthermore, there are two out-of-the-box database meta data sources, that do not rely on a JDBC connection: the JPADatabase (to reverse engineer JPA annotated entities) and the XMLDatabase (to reverse engineer an XML file). Please refer to the respective sections for more details.

Last, but not least, you can of course implement your own by implementing org.jooq.meta.Database from the jooq-meta module.

> *(!) If you're implementing your custome database, make sure it is available to the code generator as a **code generator dependency***

5.2.5.2. RegexFlags

A lot of configuration elements rely on regular expressions. The most prominent examples are the useful includes and excludes elements. All of these regular expressions use the Java java.util.regex.Pattern API, with all of its features. The Pattern API allows for specifying flags and for your configuration convenience, the applied flags are, by default:

- COMMENTS: This allows for embedding comments (and, as a side-effect: meaningless whitespace) in regular expressions, which makes them much more readable.
- CASE_INSENSITIVE: Most schemas are case insensitive, so case-sensitive regular expressions are a bit of a pain, especially in multi-vendor setups, where databases like PostgreSQL (mostly lower case) and Oracle (mostly UPPER CASE) need to be supported simultaneously.

But of course, this default setting may get in your way, for instance if you rely on case sensitive identifiers and whitespace in identifiers a lot, it might be better for you to turn off the above defaults:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <regexFlags>COMMENTS DOTALL</regexFlags>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withRegexFlags(List.COMMENTS DOTALL)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      regexFlags = "COMMENTS DOTALL"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
All the flags available from java.util.regex.Pattern are available as a whitespace-separated list in standalone XML, or a comma separated list in Maven.

5.2.5.3. Includes and Excludes

Perhaps the most important elements of the code generation configuration are used for the inclusion and exclusion of content as reported by your database meta data configuration

These expressions match any of the following object types, either by their fully qualified names (catalog.schema.object_name), or by their names only (object_name):

- Array types
- Domains
- Enums
- Links
- Packages
- Queues
- Routines
- Sequences
- Tables
- UDTs

Excludes match *before* includes, meaning that something that has been excluded cannot be included again. For example:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      </excludes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIncludes(".*")
      .withExcludes("""
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      """)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      includes = ".*"
      excludes = """
           UNUSED_TABLE                # This table (unqualified name) should not be generated
         | PREFIX_.*                   # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE              # This routine (unqualified name) ...
      """
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

© 2009 - 2025 by Data Geekery™ GmbH.

As always, when regular expressions are used, they are regular expressions with default flags.

A special, additional option allows for specifying whether the above two regular expressions should also match table columns. The following example will hide an INVISIBLE_COL in any table (and also tables called this way, of course):

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>INVISIBLE_COL</excludes>
      <includeExcludeColumns>true</includeExcludeColumns>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIncludes(".*")
      .withExcludes("INVISIBLE_COL")
      .withIncludeExcludeColumns(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      includes = ".*"
      excludes = "INVISIBLE_COL"
      includeExcludeColumns = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

If these flags are enabled, make sure to understand that they will match on ***all*** the relevant levels of identifier hierarchy. E.g. for includeExcludeColumns, a regular expression will now match ***both*** the table ***and*** the column. E.g. use this kind of expression then:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includes>
      # Table part of the expression
      (
        TABLE1
      | TABLE2
      )

      # Column part of the expression (optional for tables)
      (\..*)?
      </includes>
      <excludes>INVISIBLE_COL</excludes>
      <includeExcludeColumns>true</includeExcludeColumns>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIncludes("""
      # Table part of the expression
      (
        TABLE1
      | TABLE2
      )

      # Column part of the expression (optional for tables)
      (\..*)?
      """)
      .withExcludes("INVISIBLE_COL")
      .withIncludeExcludeColumns(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      includes = """
      # Table part of the expression
      (
        TABLE1
      | TABLE2
      )

      # Column part of the expression (optional for tables)
      (\..*)?
      """
      excludes = "INVISIBLE_COL"
      includeExcludeColumns = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

5.2.5.4. Include object types

Sometimes, you want to generate only tables. Or only routines. Or you want to exclude them from being generated. Whatever the use-case, there's a way to do this with the following, additional includes flags:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includeCheckConstraints>false</includeCheckConstraints>

      <includeEmbeddables>true</includeEmbeddables>
      <includeForeignKeys>false</includeForeignKeys>
      <includeIndexes>false</includeIndexes>
      <includeInvisibleColumns>true</includeInvisibleColumns>
      <includePackageConstants>true</includePackageConstants>
      <includePackageRoutines>true</includePackageRoutines>
      <includePackageUDTs>true</includePackageUDTs>
      <includePackages>true</includePackages>
      <includePrimaryKeys>false</includePrimaryKeys>
      <includeRoutines>true</includeRoutines>
      <includeSequences>false</includeSequences>





      <includeTables>true</includeTables>
      <includeTriggerRoutines>false</includeTriggerRoutines>
      <includeTriggers>true</includeTriggers>
      <includeUDTs>true</includeUDTs>
      <includeUniqueKeys>false</includeUniqueKeys>
      <includeXMLSchemaCollections>false</includeXMLSchemaCollections>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIncludeCheckConstraints(false)
      .withIncludeEmbeddables(true)
      .withIncludeForeignKeys(false)
      .withIncludeIndexes(false)
      .withIncludeInvisibleColumns(true)
      .withIncludePackageConstants(true)
      .withIncludePackageRoutines(true)
      .withIncludePackageUDTs(true)
      .withIncludePackages(true)
      .withIncludePrimaryKeys(false)
      .withIncludeRoutines(true)
      .withIncludeSequences(false)
      .withIncludeTables(true)
      .withIncludeTriggerRoutines(false)
      .withIncludeTriggers(true)
      .withIncludeUDTs(true)
      .withIncludeUniqueKeys(false)
      .withIncludeXMLSchemaCollections(false)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      includeCheckConstraints = false
      includeEmbeddables = true
      includeForeignKeys = false
      includeIndexes = false
      includeInvisibleColumns = true
      includePackageConstants = true
      includePackageRoutines = true
      includePackageUDTs = true
      includePackages = true
      includePrimaryKeys = false
      includeRoutines = true
      includeSequences = false
      includeTables = true
      includeTriggerRoutines = false
      includeTriggers = true
      includeUDTs = true
      includeUniqueKeys = false
      includeXMLSchemaCollections = false
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
By default, most of these flags are set to true, with the exception of:

- includeTriggerRoutines: Some databases store triggers as special ROUTINE types in the schema. These routines are not meant to be called directly, by clients, which is why their inclusion in code generation is undesirable.

5.2.5.5. Record Version and Timestamp Fields

jOOQ's org.jooq.UpdatableRecord supports an optimistic locking feature, which can be enabled in the code generator by specifying a regular expression that defines such a record's version and/or timestamp fields. These regular expressions should match at most one column per table, again either by their fully qualified names (catalog.schema.table.column_name) or by their names only (column_name):

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <recordVersionFields>REC_VERSION</recordVersionFields>
      <recordTimestampFields>REC_TIMESTAMP</recordTimestampFields>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withRecordVersionFields("REC_VERSION")
      .withRecordTimestampFields("REC_TIMESTAMP")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      recordVersionFields = "REC_VERSION"
      recordTimestampFields = "REC_TIMESTAMP"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

5.2.5.6. Synthetic identities

jOOQ's code generator recognises identity columns if they are reported as such by the database. Some databases do not support "real" identity columns, but allow for emulating them, e.g. through triggers and sequences (e.g. Oracle prior to 12c). If a column is a known "identity" without formally being one, users can specify a regular expression that matches all columns (one column per table), which will be treated as if they were formal identities. For example:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticIdentities>SCHEMA\.TABLE\.ID</syntheticIdentities>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withSyntheticIdentities("SCHEMA\\.TABLE\\.ID")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      syntheticIdentities = "SCHEMA\\.TABLE\\.ID"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

5.2.5.7. Synthetic primary keys

jOOQ's code generator recognises primary keys that are declared and reported as such by the database. But some databases don't report all keys, or some tables don't have them enabled, or sometimes, a view is a 1:1 representation of an underlying table, but it doesn't expose the key information. In these cases, this regular expression can match all columns that users wish to "pretend" are part of such a primary key. If a composite synthetic primary key is desired, the regular expression should match all columns of that table that are part of the primary key. For example, a composite synthetic primary key consists of (COLUMN1, COLUMN2) in table SCHEMA.TABLE:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticPrimaryKeys>SCHEMA\.TABLE\.COLUMN(1|2)</syntheticPrimaryKeys>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withSyntheticPrimaryKeys("SCHEMA\\.TABLE\\.COLUMN(1|2)")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      syntheticPrimaryKeys = "SCHEMA\\.TABLE\\.COLUMN(1|2)"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

If the regular expression matches column in a table that already has an existing primary key, that existing primary key will be replaced by the synthetic one. It will still be reported as a unique key, though.

5.2.5.8. Override primary keys

In some legacy contexts, there might not be any primary key, only a bunch of unique keys, defined in a table. jOOQ cannot pick one of those as the "primary key", and the table won't be generated as a

org.jooq.UpdatableRecord. Users can, however, use this regular expression to match all UNIQUE key names that should be used as a table's primary key.

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <overridePrimaryKeys>MY_UNIQUE_KEY_NAME</overridePrimaryKeys>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withOverridePrimaryKeys("MY_UNIQUE_KEY_NAME")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      overridePrimaryKeys = "MY_UNIQUE_KEY_NAME"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

If several keys match, a warning is emitted and the first one encountered will be used. This flag will also replace synthetic primary keys, if it matches.

5.2.5.9. Date as timestamp

The Oracle database doesn't know a SQL standard DATE type (YYYY-MM-DD). It's vendor-specific DATE type is really a TIMESTAMP(0), i.e. a TIMESTAMP with zero fractional seconds precision (YYYY-MM-DD HH24:MI:SS). For historic reasons, many legacy Oracle databases do not use the TIMESTAMP data type, but the DATE data type for timestamps, in case of which client applications also need to treat these columns as timestamps.

If upgrading the schema to proper TIMESTAMP usage isn't an option, and neither is using data type rewrites on a per-column basis, then this flag is the right one to activate. It will remove the DATE type from the Oracle type system (at least as far as the jOOQ code generator is concerned), and pretend all such columns are really TIMESTAMP typed. This is how to activate the flag:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <dateAsTimestamp>true</dateAsTimestamp>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withDateAsTimestamp(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      dateAsTimestamp = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
This flag will apply before any other data type related flags are applied, including forced types.

5.2.5.10. Ignore procedure return values (deprecated)

In jOOQ 3.6.0, #4106 was implemented to support Transact-SQL's optional return values from stored procedures. This turns all procedures into Routine<Integer> (instead of Routine<Void>). For backwards-compatibility reasons, users can suppress this change in jOOQ 3.x

This feature is deprecated as of jOOQ 3.6.0 and will be removed again in jOOQ 4.0.
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <ignoreProcedureReturnValues>true</ignoreProcedureReturnValues>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIgnoreProcedureReturnValues(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      ignoreProcedureReturnValues = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.5.11. Unsigned types

The JDBC and Java type system don't know any unsigned integer data types, but some databases do, most importantly MySQL. This flag allows for overriding the default mapping from unsigned to signed integers and generates jOOU types instead:

- org.jooq.types.UByte
- org.jooq.types.UShort
- org.jooq.types.UInteger
- org.jooq.types.ULong

Those types work just like ordinary java.lang.Number wrapper types, except that there is no primitive version of them. The configuration looks like follows:
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <unsignedTypes>true</unsignedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withUnsignedTypes(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      unsignedTypes = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.5.12. Catalog and schema mapping

These configuration elements combine two features in one:

- They allow for specifying one or more catalogs (default: all catalogs) as well as one or more schemas (default: all schemas) for inclusion in the code generator. This works in a similar fashion as the includes and excludes elements, but it is applied on an earlier stage.
- Once all "input" catalogs and schemas are specified, they can each be associated with a matching "output" catalog or schema, in case of which the "input" will be mapped to the "output" by the code generator. For more details about this, please refer to the manual section about schema mapping.

There are two ways to operate "input" and "output" catalogs and schemas configurations: "top level" and "nested". Note that catalogs are only supported in very few databases, so usually, users will only use the "input" and "output" schema feature.

Top level configurations

This mode is preferrable for small projects or quick tutorials, where only a single catalog and a/or a single schema need to be generated. In this case, the following "top level" configuration elements can be applied:

Read only a single schema (from all catalogs, but in most databases, there is only one "default catalog")

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <inputSchema>my_schema</inputSchema>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withInputSchema("my_schema")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      inputSchema = "my_schema"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
Read only a single catalog and all its schemas

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <inputCatalog>my_catalog</inputCatalog>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withInputCatalog("my_catalog")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      inputCatalog = "my_catalog"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
Read only a single catalog and only a single schema

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <inputCatalog>my_catalog</inputCatalog>
      <inputSchema>my_schema</inputSchema>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withInputCatalog("my_catalog")
      .withInputSchema("my_schema")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      inputCatalog = "my_catalog"
      inputSchema = "my_schema"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Nested configurations

This mode is preferrable for larger projects where several catalogs and/or schemas need to be included. The following examples show different possible configurations:

Read two or more schemas (from all catalogs, but in most databases, there is only one "default catalog")

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <schemata>
        <schema>
          <inputSchema>schema1</inputSchema>
        </schema>
        <schema>
          <inputSchema>schema2</inputSchema>
        </schema>
      </schemata>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withSchemata(
        new SchemaMappingType()
          .withInputSchema("schema1"),
        new SchemaMappingType()
          .withInputSchema("schema2")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      schemata {
        schema {
          inputSchema = "schema1"
        }
        schema {
          inputSchema = "schema2"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
Read two or more catalogs and all their schemas

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogs>
        <catalog>
          <inputCatalog>catalog1</inputCatalog>
        </catalog>
        <catalog>
          <inputCatalog>catalog2</inputCatalog>
        </catalog>
      </catalogs>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogs(
        new CatalogMappingType()
          .withInputCatalog("catalog1"),
        new CatalogMappingType()
          .withInputCatalog("catalog2")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = "catalog1"
        }
        catalog {
          inputCatalog = "catalog2"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
Read two or more schemas from one or more specific catalogs

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogs>
        <catalog>
          <inputCatalog>catalog</inputCatalog>
          <schemata>
            <schema>
              <inputSchema>schema1</inputSchema>
            </schema>
            <schema>
              <inputSchema>schema2</inputSchema>
            </schema>
          </schemata>
        </catalog>
      </catalogs>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogs(
        new CatalogMappingType()
          .withInputCatalog("catalog")
          .withSchemata(
            new SchemaMappingType()
              .withInputSchema("schema1"),
            new SchemaMappingType()
              .withInputSchema("schema2")
          )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = "catalog"
          schemata {
            schema {
              inputSchema = "schema1"
            }
            schema {
              inputSchema = "schema2"
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Catalog and schema mapping

Wherever you can place an inputCatalog or inputSchema element (top level or nested), you can also put a matching mapping instruction, if you wish to profit from the catalog and schema mapping feature. The following configurations are possible:

Map input names to a specific output name

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <inputCatalog>my_input_catalog</inputCatalog>
      <outputCatalog>my_output_catalog</outputCatalog>
      <inputSchema>my_input_schema</inputSchema>
      <outputSchema>my_output_schema</outputSchema>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withInputCatalog("my_input_catalog")
      .withOutputCatalog("my_output_catalog")
      .withInputSchema("my_input_schema")
      .withOutputSchema("my_output_schema")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      inputCatalog = "my_input_catalog"
      outputCatalog = "my_output_catalog"
      inputSchema = "my_input_schema"
      outputSchema = "my_output_schema"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
Map input names to the "default" catalog or schema (i.e. no name)

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <inputCatalog>my_input_catalog</inputCatalog>
      <outputCatalogToDefault>true</outputCatalogToDefault>
      <inputSchema>my_input_schema</inputSchema>
      <outputSchemaToDefault>true</outputSchemaToDefault>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withInputCatalog("my_input_catalog")
      .withOutputCatalogToDefault(true)
      .withInputSchema("my_input_schema")
      .withOutputSchemaToDefault(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      inputCatalog = "my_input_catalog"
      outputCatalogToDefault = true
      inputSchema = "my_input_schema"
      outputSchemaToDefault = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
For more information about the catalog and schema mapping feature, please refer to the relevant section of the manual.

5.2.5.13. Catalog and schema version providers

For continuous integration reasons, users often like to version their database schemas, e.g. with tools like Flyway. In these cases, it is usually beneficial if the catalog and/or schema version can be placed

with generated jOOQ code for documentation purposes and to prevent unnecessary re-generation of a catalog and/or schema.

For this reason, jOOQ allows for implementing a simple code generation SPI which tells jOOQ what the user-defined version of any given catalog or schema is.

There are three possible ways to implement this SPI:

- By providing a fully qualified class name that implements any of org.jooq.meta.CatalogVersionProvider or org.jooq.meta.SchemaVersionProvider respectively for programmatic version providing.
- By providing a SELECT statement returning one row with one column containing the version string. The SELECT statement may contain a named variable called :catalog_name or :schema_name respectively.
- By providing a constant, such as a Maven property, for instance.

These schema versions will be generated into the javax.annotation.Generated annotation on generated artefacts.

Example: fully qualified class name

This example is assuming that you have version provider classes on your code generation classpath available:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogVersionProvider>com.example.MyCatalogVersionProvider</catalogVersionProvider>
      <schemaVersionProvider>com.example.MySchemaVersionProvider</schemaVersionProvider>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogVersionProvider("com.example.MyCatalogVersionProvider")
      .withSchemaVersionProvider("com.example.MySchemaVersionProvider")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogVersionProvider = "com.example.MyCatalogVersionProvider"
      schemaVersionProvider = "com.example.MySchemaVersionProvider"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

> *(!) Make sure your custom version providers are available to the code generator as a **code generator dependency***

Example: SQL SELECT statement

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogVersionProvider>SELECT :catalog_name || '_' || MAX("version") FROM "schema_version"</catalogVersionProvider>
      <schemaVersionProvider>SELECT :schema_name || '_' || MAX("version") FROM "schema_version"</schemaVersionProvider>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogVersionProvider("SELECT :catalog_name || '_' || MAX(\"version\") FROM \"schema_version\"")
      .withSchemaVersionProvider("SELECT :schema_name || '_' || MAX(\"version\") FROM \"schema_version\"")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogVersionProvider = "SELECT :catalog_name || '_' || MAX(\"version\") FROM \"schema_version\""
      schemaVersionProvider = "SELECT :schema_name || '_' || MAX(\"version\") FROM \"schema_version\""
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Example: A constant

Instead of supplying the constant directly in your configuration, you can also use your build system's property expression, or some other mechanism to produce this constant.
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogVersionProvider>1</catalogVersionProvider>
      <schemaVersionProvider>2</schemaVersionProvider>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogVersionProvider(1)
      .withSchemaVersionProvider(2)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogVersionProvider = 1
      schemaVersionProvider = 2
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.5.14. Custom ordering of generated code

By default, the jOOQ code generator maintains the following ordering of objects:

- Catalogs, schemas, tables, user-defined types, packages, routines, sequences, constraints are ordered alphabetically
- Table columns, user-defined type attributes, routine parameters are ordered in their order of definition

Sometimes, it may be desireable to override this default ordering to a custom ordering. In particular, the default ordering may be case-sensitive, when case-insensitive ordering is really more desireable at times. Users may define an order provider by specifying a fully qualified class on the code generator's class path, which must implement java.util.Comparator<org.jooq.meta.Definition> as follows:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <orderProvider>com.example.CaseInsensitiveOrderProvider</orderProvider>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withOrderProvider("com.example.CaseInsensitiveOrderProvider")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      orderProvider = "com.example.CaseInsensitiveOrderProvider"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

> *(!) Make sure your custom order provider is available to the code generator as a **code generator dependency***

This order provider may then look as follows:

```
package com.example;

import java.util.Comparator;

import org.jooq.meta.Definition;

public class CaseInsensitiveOrderProvider implements Comparator<Definition> {
    @Override
    public int compare(Definition o1, Definition o2) {
        return o1.getQualifiedInputName().compareToIgnoreCase(o2.getQualifiedInputName());
    }
}
```

While changing the order of "top level types" (like tables) is irrelevant to the jOOQ runtime, there may be some side-effects to changing the order of table columns, user-defined type attributes, routine parameters, as the database might expect the exact same order as is defined in the database. In order to only change the ordering for tables, the following order provider can be implemented instead:

```
package com.example;

import java.util.Comparator;

import org.jooq.meta.Definition;
import org.jooq.meta.TableDefinition;

public class CaseInsensitiveOrderProvider implements Comparator<Definition> {
    @Override
    public int compare(Definition o1, Definition o2) {
        if (o1 instanceof TableDefinition && o2 instanceof TableDefinition)
            return o1.getQualifiedInputName().compareToIgnoreCase(o2.getQualifiedInputName());
        else
            return 0; // Retain input ordering
    }
}
```

© 2009 - 2025 by Data Geekery™ GmbH.

5.2.5.15. Forced types

The code generator allows users to override column, attribute, and parameter data types, as well as other attributes in three different ways:

- By rewriting them to some other data type using the data type rewriting feature.
- By mapping them to some user type using the data type converter feature and a custom org.jooq.Converter.
- By mapping them to some user type using the data type binding feature and a custom org.jooq.Binding.

All of this, and more, can be done using forced types.

5.2.5.15.1. Matching of forced types

The <database> configuration's <forcedTypes> element can contain many <forcedType> elements and the generator will always pick the first of these definitions (orderered by lexical appearance in the configuration), where ***all*** given match predicates match a given database attribute, column, array element, parameter, or sequence.

These are the available match predicates, all are optional:

- <objectType>: Must be one of ATTRIBUTE, COLUMN, ELEMENT, PARAMETER, SEQUENCE, or ALL and specifies what type of database objects this forced type is applicable to
- <nullability>: Must be one of NULL, NOT_NULL, or ALL and specifies if this forced type is applicable to NULL, NOT NULL, or all definitions
- <excludeExpression>: This ***exclude*** predicate is a regular expression which is matched against the definition's fully qualified, partially qualified, or unqualified name; any definition's name matching the regular expression will be considered ***not matching*** this forced type. If left empty, then nothing is excluded.
- <includeExpression>: This predicate is a regular expression which is matched against the definition's fully qualified, partially qualified, or unqualified name. If left empty, then everything is included.
- <excludeTypes>: This ***exclude*** predicate is a regular expression which is matched against the name of the definition's type (provided that it has one); any definition's type name matching the regular expression will be considered ***not matching*** this forced type. If left empty, then nothing is excluded.
- <includeTypes>: This predicate is a regular expression which is matched against the name of the definition's type (provided that it has one). If left empty, then everything is included.
- <sql>: An SQL query returning all (qualified or unqualified) object names which should be considered to match this forced type; an example is given at the end of this section.

An example using various attributes

The following example is a forced type that applies a data type rewrite of all matched objects to the type BOOLEAN:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>

      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify any data type that is supported in your database, or if unsupported,
               a type from org.jooq.impl.SQLDataType -->
          <name>BOOLEAN</name>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.IS_VALID</includeExpression>

          <!-- A Java regex matching data types to be forced to have this type.

               Data types may be reported by your database as:
               - NUMBER              regexp suggestion: NUMBER
               - NUMBER(5)           regexp suggestion: NUMBER\(5\)
               - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
               - any other form.

               It is thus recommended to use defensive regexes for types. -->
          <includeTypes>.*</includeTypes>

          <!-- Force a type depending on data type nullability. Default is ALL.

                - ALL - Force a type regardless of whether data type is nullable or not (default)
                - NULL - Force a type only when data type is nullable
                - NOT_NULL - Force a type only when data type is not null -->
          <nullability>ALL</nullability>

          <!-- Force a type on ALL or specific object types. Default is ALL. Options include:
               ATTRIBUTE, COLUMN, ELEMENT, PARAMETER, SEQUENCE -->
          <objectType>ALL</objectType>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()

      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(
        new ForcedType()

          // Specify any data type that is supported in your database, or if unsupported,
          // a type from org.jooq.impl.SQLDataType
          .withName("BOOLEAN")

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.IS_VALID")

          // A Java regex matching data types to be forced to have this type.
          //
          // Data types may be reported by your database as:
          // - NUMBER              regexp suggestion: NUMBER
          // - NUMBER(5)           regexp suggestion: NUMBER\(5\)
          // - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
          // - any other form.
          //
          // It is thus recommended to use defensive regexes for types.
          .withIncludeTypes(".*")

          // Force a type depending on data type nullability. Default is ALL.
          //
          // - ALL - Force a type regardless of whether data type is nullable or not (default)
          // - NULL - Force a type only when data type is nullable
          // - NOT_NULL - Force a type only when data type is not null
          .withNullability(Nullability.ALL)

          // Force a type on ALL or specific object types. Default is ALL. Options include:
          // ATTRIBUTE, COLUMN, ELEMENT, PARAMETER, SEQUENCE
          .withObjectType(ForcedTypeObjectType.ALL)
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {

      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {

          // Specify any data type that is supported in your database, or if unsupported,
          // a type from org.jooq.impl.SQLDataType
          name = "BOOLEAN"

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*\\.IS_VALID"

          // A Java regex matching data types to be forced to have this type.
          //
          // Data types may be reported by your database as:
          // - NUMBER              regexp suggestion: NUMBER
          // - NUMBER(5)           regexp suggestion: NUMBER\(5\)
          // - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
          // - any other form.
          //
          // It is thus recommended to use defensive regexes for types.
          includeTypes = ".*"

          // Force a type depending on data type nullability. Default is ALL.
          //
          // - ALL - Force a type regardless of whether data type is nullable or not (default)
          // - NULL - Force a type only when data type is nullable
          // - NOT_NULL - Force a type only when data type is not null
          nullability = "ALL"

          // Force a type on ALL or specific object types. Default is ALL. Options include:
          // ATTRIBUTE, COLUMN, ELEMENT, PARAMETER, SEQUENCE
          objectType = "ALL"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Using SQL to match column names

If you want to match your column names based on more complex criteria not supported by jOOQ yet, you can supply the matching column names using a SQL query that runs against your dictionary views. The following example uses a SQL query to find all columns that are "probably boolean", and applies a data type converter to them:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Rewrite types to BOOLEAN -->
          <userType>java.lang.Boolean</userType>
          <converter>com.example.YNBooleanConverter</converter>

          <!-- All Oracle columns that have a default of 'Y' or 'N' are probably boolean -->
          <sql>
            SELECT owner || '.' || table_name || '.' || column_name
            FROM all_tab_cols
            WHERE data_default IN ('Y', 'N')
          </sql>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Rewrite types to BOOLEAN
          .withUserType("java.lang.Boolean")
          .withConverter("com.example.YNBooleanConverter")

          // All Oracle columns that have a default of 'Y' or 'N' are probably boolean
          .withSql("""
            SELECT owner || '.' || table_name || '.' || column_name
            FROM all_tab_cols
            WHERE data_default IN ('Y', 'N')
          """)
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Rewrite types to BOOLEAN
          userType = "java.lang.Boolean"
          converter = "com.example.YNBooleanConverter"

          // All Oracle columns that have a default of 'Y' or 'N' are probably boolean
          sql = """
            SELECT owner || '.' || table_name || '.' || column_name
            FROM all_tab_cols
            WHERE data_default IN ('Y', 'N')
          """
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Please see the relevant sections of the manual to get more information about using converters or bindings.

5.2.5.15.2. Data type rewriting

Sometimes, the actual database data type does not match the SQL data type that you would like to use in Java. This is often the case for ill-supported SQL data types, such as BOOLEAN, UUID, or INSTANT. jOOQ's code generator allows you to apply simple data type rewriting. The following configuration will rewrite IS_VALID columns in all tables to be of type BOOLEAN.

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Specify any data type that is supported in your database, or if unsupported,
               a type from org.jooq.impl.SQLDataType -->
          <name>BOOLEAN</name>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.IS_VALID</includeExpression>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Specify any data type that is supported in your database, or if unsupported,
          // a type from org.jooq.impl.SQLDataType
          .withName("BOOLEAN")

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.IS_VALID")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify any data type that is supported in your database, or if unsupported,
          // a type from org.jooq.impl.SQLDataType
          name = "BOOLEAN"

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*\\.IS_VALID"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
After such a data type rewrite, neither jOOQ's code generator, nor the runtime have any information about the original data type that your column may have had. Hence, this approach works best when the original data type (e.g. INTEGER), and the configured data type (e.g. BIGINT) are reasonably compatible.

For more details about how to match columns, please refer to the section about matching columns for forced types.

5.2.5.15.3. Qualified converters

When using a custom type in jOOQ, you need to let jOOQ know about its associated org.jooq.Converter. Ad-hoc usages of such converters has been discussed in the chapter about data type conversion. However, when mapping a custom type onto a standard JDBC type, a more common use-case is to let jOOQ know about custom types at code generation time.

Assuming you have a converter like this:

```
public class IntegerToYearConverter extends AbstractConverter<Integer, Year> {
    public IntegerToYearConverter() {
        super(Integer.class, Year.class);
    }

    @Override
    public Year from(Integer i) {
        return i == null ? null : Year.of(i);
    }

    @Override
    public Integer to(Year y) {
        return y == null ? null : y.getValue();
    }
}
```

The following example shows how to reference a custom Converter from your <forcedType> configuration:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>java.time.Year</userType>

          <!-- Associate that custom type with your converter. -->
          <converter>com.example.IntegerToYearConverter</converter>


          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.YEAR.*</includeExpression>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          .withUserType("java.time.Year")

          // Associate that custom type with your converter.
          .withConverter("com.example.IntegerToYearConverter")

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.YEAR.*")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          userType = "java.time.Year"

          // Associate that custom type with your converter.
          converter = "com.example.IntegerToYearConverter"

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*\\.YEAR.*"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
For more details about how to match columns, please refer to the section about matching columns for forced types.

The above configuration will lead e.g. to AUTHOR.YEAR_OF_BIRTH being generated like this:

```
public class TAuthor extends TableImpl<TAuthorRecord> {

    // [...]
    public final TableField<TAuthorRecord, Year> YEAR_OF_BIRTH =    // [...]
    // [...]

}
```

This means that the bound type of <T> will be Year, wherever you reference YEAR_OF_BIRTH. jOOQ will use your custom converter when binding variables and when fetching data from java.sql.ResultSet:

```
// Get all date of births of authors born after 1980
List<Year> result =
create.selectFrom(AUTHOR)
      .where(AUTHOR.YEAR_OF_BIRTH.gt(Year.of(1980))
      .fetch(AUTHOR.YEAR_OF_BIRTH);
```

© 2009 - 2025 by Data Geekery™ GmbH.

5.2.5.15.4. Inline converters

For convenience, you can inline your converter code directly into the configuration instead of providing a class reference, as shown previously.

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>java.time.Year</userType>

          <!-- Associate that custom type with your inline converter. -->
          <converter>org.jooq.Converter.ofNullable(
            Integer.class, Year.class,
            Year::of, Year::getValue
          )</converter>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.YEAR.*</includeExpression>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          .withUserType("java.time.Year")

          // Associate that custom type with your inline converter.
          .withConverter("""org.jooq.Converter.ofNullable(
            Integer.class, Year.class,
            Year::of, Year::getValue
          )""")

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.YEAR.*")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          userType = "java.time.Year"

          // Associate that custom type with your inline converter.
          converter = """org.jooq.Converter.ofNullable(
            Integer.class, Year.class,
            Year::of, Year::getValue
          )"""

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*\\.YEAR.*"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

The effect on your query user code will be the same as with explicit converters.

For more details about how to match columns, please refer to the section about matching columns for forced types.

5.2.5.15.5. Enum converters

If your user type is a Java enum, you can use the <enumConverter/> convenience flag instead of an explicit converter per enum type. This will apply the built-in org.jooq.impl.EnumConverter.

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>com.example.MyEnum</userType>

          <!-- Apply the built in org.jooq.impl.EnumConverter. -->
          <enumConverter>true</enumConverter>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.MY_STATUS</includeExpression>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          .withUserType("com.example.MyEnum")

          // Apply the built in org.jooq.impl.EnumConverter.
          .withEnumConverter(true)

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.MY_STATUS")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          userType = "com.example.MyEnum"

          // Apply the built in org.jooq.impl.EnumConverter.
          enumConverter = true

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*\\.MY_STATUS"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

The effect on your query user code will be the same as with explicit converters.

For more details about how to match columns, please refer to the section about matching columns for forced types.

5.2.5.15.6. Data type bindings

The previous sections discussed the case where your custom data type is mapped onto a standard JDBC type as contained in org.jooq.impl.SQLDataType. In some cases, however, you want to map your own type onto a type that is not explicitly supported by JDBC, such as for instance, PostgreSQL's various advanced data types (though do check out the jooq-postgres-extensions module to see what we're already offering in this area). For this, you can register an org.jooq.Binding for relevant columns in your code generator. Consider the following trivial implementation of a binding for PostgreSQL's JSON data type, which binds the JSON string in PostgreSQL to a Google GSON object:

```
import java.sql.*;
import java.util.*;

import org.jooq.*;
import org.jooq.conf.*;
import org.jooq.impl.DSL;
import com.google.gson.*;

// We're binding <T> = JSON (or JSONB), and <U> = JsonElement (user type)
// Alternatively, extend org.jooq.impl.AbstractBinding to implement fewer methods.
public class PostgresJSONGsonBinding implements Binding<JSON, JsonElement> {

    private final Gson gson = new Gson();

    // The converter does all the work
    @Override
    public Converter<JSON, JsonElement> converter() {
        return new Converter<JSON, JsonElement>() {
            @Override
            public JsonElement from(JSON t) {
                return t == null ? JsonNull.INSTANCE : gson.fromJson(t.data(), JsonElement.class);
            }

            @Override
            public JSON to(JsonElement u) {
                return u == null || u == JsonNull.INSTANCE ? null : JSON.json(gson.toJson(u));
            }

            @Override
            public Class<JSON> fromType() {
                return JSON.class;
            }

            @Override
            public Class<JsonElement> toType() {
                return JsonElement.class;
            }
        };
    }

    // Rending a bind variable for the binding context's value and casting it to the json type
    @Override
    public void sql(BindingSQLContext<JsonElement> ctx) throws SQLException {
        // Depending on how you generate your SQL, you may need to explicitly distinguish
        // between jOOQ generating bind variables or inlined literals.
        if (ctx.render().paramType() == ParamType.INLINED)
            ctx.render().visit(DSL.inline(ctx.convert(converter()).value())).sql("::json");
        else
            ctx.render().sql(ctx.variable()).sql("::json");
    }

    // Registering VARCHAR types for JDBC CallableStatement OUT parameters
    @Override
    public void register(BindingRegisterContext<JsonElement> ctx) throws SQLException {
        ctx.statement().registerOutParameter(ctx.index(), Types.VARCHAR);
    }

    // Converting the JsonElement to a String value and setting that on a JDBC PreparedStatement
    @Override
    public void set(BindingSetStatementContext<JsonElement> ctx) throws SQLException {
        JSON json = ctx.convert(converter()).value();
        ctx.statement().setString(ctx.index(), json == null ? null : json.data());
    }

    // Getting a String value from a JDBC ResultSet and converting that to a JsonElement
    @Override
    public void get(BindingGetResultSetContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(JSON.json(ctx.resultSet().getString(ctx.index())));
    }

    // Getting a String value from a JDBC CallableStatement and converting that to a JsonElement
    @Override
    public void get(BindingGetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(JSON.json(ctx.statement().getString(ctx.index())));
    }

    // Setting a value on a JDBC SQLOutput (useful for Oracle OBJECT types)
    @Override
    public void set(BindingSetSQLOutputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }

    // Getting a value from a JDBC SQLInput (useful for Oracle OBJECT types)
    @Override
    public void get(BindingGetSQLInputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }
}
```

Registering bindings to the code generator

The above org.jooq.Binding implementation intercepts all the interaction on a JDBC level, such that jOOQ will never need to know how to correctly serialise / deserialise your custom data type. Similar to what we've seen in the previous section about how to register Converters to the code generator, we can now register such a binding to the code generator. Note that you will reuse the same types of XML elements (<forcedType/>):

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Binding's <U> type. -->
          <userType>com.google.gson.JsonElement</userType>

          <!-- Associate that custom type with your binding. -->
          <binding>com.example.PostgresJSONGsonBinding</binding>


          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*JSON.*</includeExpression>

          <!-- A Java regex matching data types to be forced to
               have this type.

               Data types may be reported by your database as:
               - NUMBER              regexp suggestion: NUMBER
               - NUMBER(5)           regexp suggestion: NUMBER\(5\)
               - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
               - any other form

               It is thus recommended to use defensive regexes for types.

               If provided, both "includeExpressions" and "includeTypes" must match. -->
          <includeTypes>(?i:JSON)</includeTypes>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForcedTypes(
        new ForcedType()

          // Specify the Java type of your custom type. This corresponds to the Binding's <U> type.
          .withUserType("com.google.gson.JsonElement")

          // Associate that custom type with your binding.
          .withBinding("com.example.PostgresJSONGsonBinding")

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*JSON.*")

          // A Java regex matching data types to be forced to
          // have this type.
          //
          // Data types may be reported by your database as:
          // - NUMBER              regexp suggestion: NUMBER
          // - NUMBER(5)           regexp suggestion: NUMBER\(5\)
          // - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
          // - any other form
          //
          // It is thus recommended to use defensive regexes for types.
          //
          // If provided, both "includeExpressions" and "includeTypes" must match.
          .withIncludeTypes("(?i:JSON)")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Binding's <U> type.
          userType = "com.google.gson.JsonElement"

          // Associate that custom type with your binding.
          binding = "com.example.PostgresJSONGsonBinding"

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = ".*JSON.*"

          // A Java regex matching data types to be forced to
          // have this type.
          //
          // Data types may be reported by your database as:
          // - NUMBER             regexp suggestion: NUMBER
          // - NUMBER(5)          regexp suggestion: NUMBER\(5\)
          // - NUMBER(5, 2)       regexp suggestion: NUMBER\(5,\s*2\)
          // - any other form
          //
          // It is thus recommended to use defensive regexes for types.
          //
          // If provided, both "includeExpressions" and "includeTypes" must match.
          includeTypes = "(?i:JSON)"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
For more details about how to match columns, please refer to the section about matching columns for forced types.

The above configuration will lead to AUTHOR.CUSTOM_DATA_JSON being generated like this:

```
public class TAuthor extends TableImpl<TAuthorRecord> {

    // [...]
    public final TableField<TAuthorRecord, JsonElement> CUSTOM_DATA_JSON =    // [...]
    // [...]

}
```

5.2.5.16. Table valued functions

jOOQ supports table valued functions in many databases, including Oracle, PostgreSQL, SQL Server. By default, table valued functions are treated as:

- ordinary tables in most databases including PostgreSQL, SQL Server - because that's what they are. They're intended for use in FROM clauses of SELECT statements, not as standalone routines.
- ordinary routines in some databases including Oracle - for historic reasons. While Oracle also allows for embedding (pipelined) table functions in FROM clauses of SELECT statements, it is not uncommon to call these as standalone routines in Oracle.

The <tableValuedFunctions/> flag is thus set to false by default on Oracle, and true otherwise. Here's how to explicitly change this behaviour:
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <tableValuedFunctions>true</tableValuedFunctions>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withTableValuedFunctions(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      tableValuedFunctions = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6. Generate

This element wraps all the configuration elements that are used for the jooq-codegen module, which generates Java or Scala code, or XML from your database.

Contained elements are:

5.2.6.1. Annotations

The code generator supports a set of annotations on generated code, which can be turned on using the following flags. These annotations include:

- Generated annotations: The JDK generated annotation can be added to all generated classes to include some useful meta information, like the jOOQ version, or the schema version, or the generation date. Depending on the configured generatedAnnotationType, the annotation is one of:

  * javax.annotation.processing.Generated (JDK 9+)
  * javax.annotation.Generated (JDK 8-)

- JPA annotations: A minimal set of JPA annotations can be generated on POJOs and other artefacts to convey type and metadata information that is available to the code generator. These annotations include:

  * javax.persistence.Column
  * javax.persistence.Entity
  * javax.persistence.GeneratedValue
  * javax.persistence.GenerationType
  * javax.persistence.Id
  * javax.persistence.Index (JPA 2.1 and later)
  * javax.persistence.Table
  * javax.persistence.UniqueConstraint

  While jOOQ generated code cannot really be used as full-fledged entities (use e.g. Hibernate or EclipseLink to generate such entities), this meta information can still be useful as documentation on your generated code. Some of the annotations (e.g. @Column) can be used by the org.jooq.impl.DefaultRecordMapper for mapping records to POJOs.
- Validation annotations: A set of Bean Validation API annotations can be added to the generated code to convey type information. They include:

  * javax.validation.constraints.NotNull
  * javax.validation.constraints.Size

  jOOQ does not implement the validation spec, nor does it validate your data, but you can use third-party tools to read the jOOQ-generated validation annotations.
- Spring annotations: Some useful Spring annotations can be generated on DAOs for better Spring integration. These include:

  * org.springframework.beans.factory.annotation.Autowired
  * org.springframework.stereotype.Repository

The flags governing the generation of these annotations are:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Possible values for generatedAnnotationType
         - DETECT_FROM_JDK
         - JAVAX_ANNOTATION_GENERATED
         - JAVAX_ANNOTATION_PROCESSING_GENERATED -->
      <generatedAnnotation>true</generatedAnnotation>
      <generatedAnnotationType>DETECT_FROM_JDK</generatedAnnotationType>

      <jpaAnnotations>true</jpaAnnotations>
      <jpaVersion>2.2</jpaVersion>

      <validationAnnotations>true</validationAnnotations>
      <springAnnotations>true</springAnnotations>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Possible values for generatedAnnotationType
      // - DETECT_FROM_JDK
      // - JAVAX_ANNOTATION_GENERATED
      // - JAVAX_ANNOTATION_PROCESSING_GENERATED
      .withGeneratedAnnotation(true)
      .withGeneratedAnnotationType(GeneratedAnnotationType.DETECT_FROM_JDK)
      .withJpaAnnotations(true)
      .withJpaVersion(2.2)
      .withValidationAnnotations(true)
      .withSpringAnnotations(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Possible values for generatedAnnotationType
      // - DETECT_FROM_JDK
      // - JAVAX_ANNOTATION_GENERATED
      // - JAVAX_ANNOTATION_PROCESSING_GENERATED
      generatedAnnotation = true
      generatedAnnotationType = "DETECT_FROM_JDK"
      jpaAnnotations = true
      jpaVersion = 2.2
      validationAnnotations = true
      springAnnotations = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.2. Covariant overrides

Generated tables can extend the DSL in a way for a DSL method return type to return again the generated type. This leads to improved type safety in client code by allowing for type safe dereferencing of table columns also from certain table expressions, not just the tables themselves.

The following sections discuss options for such covariant overrides:

5.2.6.2.1. Overriding as()

The most popular covariant override is the override of the Table.as(Name) method and its overloads, in order to offer type safe column aliasing.

```
// Overridden Book.as() returns Book, not Table<Book>:
Book b = BOOK.as("b");

// Type safe dereferencing of TableField:
Field<String> title = b.TITLE;
```

5.2.6.2.2. Overriding rename()

*(!) As of jOOQ 3.12.4, the rename() method is not declared in any super type yet, but it will be eventually, see **#13937**.*

Occasionally, users will want to rename a table for the scope of a single query, rather than globally as with the runtime schema and table mapping feature. This can be done with Table.rename(Name):

```
// Overridden Book.rename() returns Book, not Table<Book>:
Book b = BOOK.rename("b");

// Type safe dereferencing of TableField:
Field<String> title = b.TITLE;
```

Unlike as(), this does not affect generated SQL other than assuming a table b of the same structure as BOOK exists in the schema.

5.2.6.3. Default catalog and schema

Generated catalog and schema objects have purposes other than qualifying their contents, such as tables, sequences, routines, etc. Some of these purposes include:

- mapping of qualifiers
- declaring of version providers to prevent unnecessary re-generation
- declaring of qualifier based global artefacts

This is why jOOQ generates a DefaultCatalog and DefaultSchema instance even when there are no catalogs or schemas in the database, or when the catalogs or schemas have been mapped to the default.

5.2.6.4. Fluent setters

By default, jOOQ generated artefacts follow JavaBeans conventions, where setters return void. If that is not a hard requirement, fluent setters can be generated where the setter returns the record/interface/pojo itself. To activate this behaviour, use:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <fluentSetters>true</fluentSetters>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()
      .withFluentSetters(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {
      fluentSetters = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.5. Fully Qualified Types

By default, the jOOQ code generator references all types as unqualified types, generating the necessary import statement at the beginning of generated classes.

In rare cases, this can cause problems when two types conflict with each other, e.g. when there is both a TABLE and a TABLE_RECORD table (generating a TableRecord org.jooq.Record type for TABLE as well as a TableRecord org.jooq.Table type for TABLE_RECORD). In this case, users can specify a regular expression that matches all objects whose corresponding generated artefacts should never be imported, but always be fully qualified.

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <fullyQualifiedTypes>.*\.MY_TABLE</fullyQualifiedTypes>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()
      .withFullyQualifiedTypes(".*\\.MY_TABLE")
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {
      fullyQualifiedTypes = ".*\\.MY_TABLE"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

5.2.6.6. Global Artefacts

For convenience, jOOQ generates a set of global artefacts, which group static constants of the same type in a well-known class. The following set of flags allows for turning off the generation of these artefacts, individually:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- This overrides all the other individual flags -->
      <globalObjectReferences>true</globalObjectReferences>

      <!-- Individual flags for each object type -->
      <globalCatalogReferences>true</globalCatalogReferences>
      <globalSchemaReferences>true</globalSchemaReferences>
      <globalTableReferences>true</globalTableReferences>
      <globalSequenceReferences>true</globalSequenceReferences>


      <globalUDTReferences>true</globalUDTReferences>
      <globalRoutineReferences>true</globalRoutineReferences>
      <globalQueueReferences>true</globalQueueReferences>
      <globalLinkReferences>true</globalLinkReferences>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // This overrides all the other individual flags
      .withGlobalObjectReferences(true)

      // Individual flags for each object type
      .withGlobalCatalogReferences(true)
      .withGlobalSchemaReferences(true)
      .withGlobalTableReferences(true)
      .withGlobalSequenceReferences(true)
      .withGlobalUDTReferences(true)
      .withGlobalRoutineReferences(true)
      .withGlobalQueueReferences(true)
      .withGlobalLinkReferences(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // This overrides all the other individual flags
      globalObjectReferences = true

      // Individual flags for each object type
      globalCatalogReferences = true
      globalSchemaReferences = true
      globalTableReferences = true
      globalSequenceReferences = true
      globalUDTReferences = true
      globalRoutineReferences = true
      globalQueueReferences = true
      globalLinkReferences = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.7. Implicit JOIN paths

Implicit JOINs are one of jOOQ's most powerful synthetic SQL features, which depends entirely on code generation. The code generator produces links between tables that follow "to-one" relationships (i.e. from child table to parent table). This way, it is possible to greatly simplify your queries:

```
// Get all books, their authors, and their respective language
create.select(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.TITLE,
          BOOK.language().CD.as("language"))
      .from(BOOK)
      .fetch();
```

The feature has a few feature toggles in the code generator, including:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Allowing to turn off the feature entirely. The default is true. -->
      <implicitJoinPathsToOne>true</implicitJoinPathsToOne>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Allowing to turn off the feature entirely. The default is true.
      .withImplicitJoinPathsToOne(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Allowing to turn off the feature entirely. The default is true.
      implicitJoinPathsToOne = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.8. Java Time Types

With jOOQ 3.9, support for JSR-310 java.time types has been added to the jOOQ API and to the code generator. Users of Java 8 can now specify that the jOOQ code generator should prefer JSR 310 types over their equivalent JDBC types. This includes:

- java.time.LocalDate instead of java.sql.Date
- java.time.LocalTime instead of java.sql.Time
- java.time.LocalDateTime instead of java.sql.Timestamp

Semantically, the above types are exactly equivalent, although the new types do away with the many flaws of the JDBC types. If there is no JDBC type for an equivalent JSR 310 type, then the JSR 310 type is generated by default. This includes

- java.time.OffsetTime (for SQL TIME WITH TIME ZONE)
- java.time.OffsetDateTime (for SQL TIMESTAMP WITH TIME ZONE)

© 2009 - 2025 by Data Geekery™ GmbH.

To get more fine-grained control of the above, you may wish to consider applying data type rewriting.

In order to activate the generation of these types, use:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <javaTimeTypes>true</javaTimeTypes>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()
      .withJavaTimeTypes(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {
      javaTimeTypes = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.9. Whitespace (newlines and indentation)

By default, jOOQ's code generator produces unix newline characters (\n) and 4 space indentation (Java) or 2 space indentation (Scala). This can be overridden by using the below configuration flags. Depending on how you're loading the configuration, whitespace characters may get lost, which is why you may need to escape the backslash \ to \\. Supported escape sequences include:

- Indentation: \t (tab)
- Newline: \r (carriage return) and \n (newline)

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <indentation>\t</indentation>
      <newline>\r\n</newline>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.

Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()
      .withIndentation("\\t")
      .withNewline("\\r\\n")
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {
      indentation = "\\t"
      newline = "\\r\\n"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.6.10. Zero Scale Decimal Types

A zero-scale decimal, such as DECIMAL(10) or NUMBER(10, 0) is really an integer type with a decimal precision rather than a binary / bitwise precision. Some databases (e.g. Oracle) do not support actual integer types at all, only decimal types. Historically, jOOQ generates the most appropriate integer wrapper type instead of BigDecimal or BigInteger:

- NUMBER(2, 0) and less: java.lang.Byte
- NUMBER(4, 0) and less: java.lang.Short
- NUMBER(9, 0) and less: java.lang.Integer
- NUMBER(18, 0) and less: java.lang.Long

If this is not a desireable default, it can be deactivated either explicitly on a per-column basis using forced types, or globally using the following flag:
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forceIntegerTypesOnZeroScaleDecimals>true</forceIntegerTypesOnZeroScaleDecimals>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withForceIntegerTypesOnZeroScaleDecimals(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      forceIntegerTypesOnZeroScaleDecimals = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.2.7. Output target configuration

In the previous sections, we've seen the <target/> element which configures the location of your generated output. The following XML snippet illustrates some additional flags that can be specified in that section:
XML (standalone and maven)

```
<configuration>
  <generator>
    <target>
      <packageName>org.jooq.your.packagename</packageName>
      <directory>/path/to/your/dir</directory>
      <encoding>UTF-8</encoding>
      <clean>true</clean>
    </target>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withTarget(new Target()
      .withPackageName("org.jooq.your.packagename")
      .withDirectory("/path/to/your/dir")
      .withEncoding("UTF-8")
      .withClean(true)
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    target {
      packageName = "org.jooq.your.packagename"
      directory = "/path/to/your/dir"
      encoding = "UTF-8"
      clean = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

- packageName: Specifies the root package name inside of which all generated code is located. This package is located inside of the <directory/>. The package name is part of the generator strategy and can be modified by a custom implementation, if so desired.
- directory: Specifies the root directoy inside of which all generated code is located.
- encoding: The encoding that should be used for generated classes.
- clean: Whether the target package (<packageName/>) should be cleaned to contain only generated code after a generation run. Defaults to true.

5.3. Generated object types

RDBMS have a rich set of object types, each of which are represented by one or more specific jOOQ types, respectively. The following sections explain what types exist, and how each of them can be configured in jOOQ's code generator.

5.3.1. Generated tables

Every table and view in your database will generate a org.jooq.Table implementation that looks like this:

```
public class Book extends TableImpl<BookRecord> {

    // The singleton instance
    public static final Book BOOK = new Book();

    // Generated columns
    public final TableField<BookRecord, Integer> ID        = createField("ID",        INTEGER, this);
    public final TableField<BookRecord, Integer> AUTHOR_ID = createField("AUTHOR_ID", INTEGER, this);
    public final TableField<BookRecord, String>  TITLE     = createField("TITLE",     VARCHAR, this);

    // Covariant aliasing method, returning a table of the same type as BOOK
    @Override
    public Book as(java.lang.String alias) {
        return new Book(alias);
    }

    // [...]
}
```

Flags influencing generated tables

These flags from the code generation configuration influence generated tables:

- recordVersionFields: Relevant methods from super classes are overridden to return the VERSION field
- recordTimestampFields: Relevant methods from super classes are overridden to return the TIMESTAMP field
- syntheticPrimaryKeys: This overrides existing primary key information to allow for "custom" primary key column sets
- overridePrimaryKeys: This overrides existing primary key information to allow for unique key to primary key promotion
- dateAsTimestamp: This influences all relevant columns
- unsignedTypes: This influences all relevant columns
- relations: Relevant methods from super classes are overridden to provide primary key, unique key, foreign key and identity information
- instanceFields: This flag controls the "static" keyword on table columns, as well as aliasing convenience
- records: The generated record type is referenced from tables allowing for type-safe single-table record fetching

Flags controlling table generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Allows for turning on tables generation: default true -->
      <tables>true</tables>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Allows for turning on tables generation: default true
      .withTables(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Allows for turning on tables generation: default true
      tables = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.3.2. Generated records

Every table and view in your database will generate a org.jooq.TableRecord (or org.jooq.UpdatableRecord if there's a primary key) implementation that looks like this:

```
// JPA annotations can be generated, optionally
@Entity
@Table(name = "BOOK", schema = "TEST")
public class BookRecord extends UpdatableRecordImpl<BookRecord>

// An interface common to records and pojos can be generated, optionally
implements IBook {

    // Every column generates a setter and a getter
    @Override
    public void setId(Integer value) {
        setValue(BOOK.ID, value);
    }

    @Id
    @Column(name = "ID", unique = true, nullable = false, precision = 7)
    @Override
    public Integer getId() {
        return getValue(BOOK.ID);
    }

    // More setters and getters
    public void setAuthorId(Integer value) {...}
    public Integer getAuthorId() {...}

    // Convenience methods for foreign key methods
    public void setAuthorId(AuthorRecord value) {
        if (value == null) {
            setValue(BOOK.AUTHOR_ID, null);
        }
        else {
            setValue(BOOK.AUTHOR_ID, value.getValue(AUTHOR.ID));
        }
    }

    // Navigation methods
    public AuthorRecord fetchAuthor() {
        return create.selectFrom(AUTHOR).where(AUTHOR.ID.eq(getValue(BOOK.AUTHOR_ID))).fetchOne();
    }

    // [...]
}
```

TableRecord vs UpdatableRecord

If primary key information is available to the code generator, an org.jooq.UpdatableRecord will be generated. If no such information is available, a org.jooq.TableRecord will be generated. Primary key information can be absent because:

- The table is a view, which does not expose the underlying primary keys
- The table does not have a primary key
- The code generator configuration has turned off primary keys usage information usage through one of various flags (see below)
- The primary key information is not available to the code generator

Flags controlling record generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Allows for turning off records generation: default true -->
      <records>true</records>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Allows for turning off records generation: default true
      .withRecords(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Allows for turning off records generation: default true
      records = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
The recordsIncludes and recordsExcludes regular expressions work just like the global includes and excludes regular expressions

Flags influencing generated records

Additional flags from the code generation configuration influence generated records:

- syntheticPrimaryKeys: This overrides existing primary key information to allow for "custom" primary key column sets, possibly promoting a TableRecord to an UpdatableRecord
- overridePrimaryKeys: This overrides existing primary key information to allow for unique key to primary key promotion, possibly promoting a TableRecord to an UpdatableRecord
- includePrimaryKeys: This includes or excludes all primary key information in the generator's database meta data
- dateAsTimestamp: This influences all relevant getters and setters
- unsignedTypes: This influences all relevant getters and setters
- relations: This is needed as a prerequisite for navigation methods
- daos: Records are a pre-requisite for DAOs. If DAOs are generated, records are generated as well
- interfaces: If interfaces are generated, records will implement them
- jpaAnnotations: JPA annotations are used on generated records (details here)
- jpaVersion: Version of JPA specification is to be used to generate version-specific annotations. If it is omitted, the latest version is used by default. (details here)
- recordsImplementingRecordN: Whether generated records should implement Record[N] types. Please note that this may impact compilation speeds! Starting from jOOQ 3.19, the default for this flag is false, for this reason.

5.3.3. Generated POJOs

Every table and view in your database will generate a POJO implementation that looks like this:

```
// JPA annotations can be generated, optionally
@Entity
@Table(name = "BOOK", schema = "TEST")
public class Book implements java.io.Serializable

// An interface common to records and pojos can be generated, optionally
, IBook {

    // JSR-303 annotations can be generated, optionally
    @NotNull
    private Integer id;

    @NotNull
    private Integer authorId;

    @NotNull
    @Size(max = 400)
    private String title;

    // Every column generates a getter and a setter
    @Id
    @Column(name = "ID", unique = true, nullable = false, precision = 7)
    @Override
    public Integer getId() {
        return this.id;
    }

    @Override
    public void setId(Integer id) {
        this.id = id;
    }

    // [...]
}
```

Flags controlling POJO generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Allows for turning on POJOs generation: default false -->
      <pojos>true</pojos>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Allows for turning on POJOs generation: default false
      .withPojos(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Allows for turning on POJOs generation: default false
      pojos = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Flags influencing generated POJOs

Additional flags from the code generation configuration influence generated POJOs:

- daos: POJOs are a pre-requisite for DAOs. If DAOs are generated, POJOs are generated as well
- dateAsTimestamp: This influences all relevant getters and setters
- immutablePojos: Immutable POJOs have final members and no setters. All members must be passed to the constructor
- interfaces: If interfaces are generated, POJOs will implement them
- jpaAnnotations: JPA annotations are used on generated records (details here)
- jpaVersion: Version of JPA specification is to be used to generate version-specific annotations. If it is omitted, the latest version is used by default. (details here)
- pojosToString: Whether POJOs should have a generated toString() implementation.
- pojosEqualsAndHashCode: Whether POJOs should have generated equals() and hashCode() implementations. These implementations are purely value-based, just like with records, i.e. two POJOs are equal if all their attributes are equal.
- validationAnnotations: JSR-303 validation annotations are used on generated records (details here)

5.3.4. Generated Interfaces

Every table, view, udt in your database will generate an interface that looks like this:

```
public interface IBook extends java.io.Serializable {

    // Every column generates a getter and a setter
    public void setId(Integer value);
    public Integer getId();

    // [...]
}
```

The purpose of these interfaces is to be able to abstract over jOOQ generated records and POJOs.

Flags controlling interface generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Turn on the generation of interfaces -->
      <interfaces>true</interfaces>

      <!-- Generated interfaces will not expose mutable components of their implementations, such as setters -->
      <immutableInterfaces>true</immutableInterfaces>

      <!-- Whether generated interfaces are Serializable -->
      <serializableInterfaces>true</serializableInterfaces>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Turn on the generation of interfaces
      .withInterfaces(true)

      // Generated interfaces will not expose mutable components of their implementations, such as setters
      .withImmutableInterfaces(true)

      // Whether generated interfaces are Serializable
      .withSerializableInterfaces(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Turn on the generation of interfaces
      interfaces = true

      // Generated interfaces will not expose mutable components of their implementations, such as setters
      immutableInterfaces = true

      // Whether generated interfaces are Serializable
      serializableInterfaces = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

*(!) Note, there are numerous problems related to generated interfaces as can be seen in **#10509**.*

5.3.5. Generated DAOs

Every table and view in your database will generate a org.jooq.DAO implementation that looks like this:

```
public class BookDao extends DAOImpl<BookRecord, Book, Integer> {

    // Generated constructors
    public BookDao() {
        super(BOOK, Book.class);
    }

    public BookDao(Configuration configuration) {
        super(BOOK, Book.class, configuration);
    }

    // Every column generates at least one fetch method
    public List<Book> fetchById(Integer... values) {
        return fetch(BOOK.ID, values);
    }

    public Book fetchOneById(Integer value) {
        return fetchOne(BOOK.ID, value);
    }

    public List<Book> fetchByAuthorId(Integer... values) {
        return fetch(BOOK.AUTHOR_ID, values);
    }

    // [...]
}
```

Flags controlling DAO generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Generate the DAO classes -->
      <daos>true</daos>

      <!-- Annotate DAOs (and other types) with spring annotations, such as @Repository and @Autowired
           for auto-wiring the Configuration instance, e.g. from Spring Boot's jOOQ starter -->
      <springAnnotations>true</springAnnotations>


    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.

Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Generate the DAO classes
      .withDaos(true)

      // Annotate DAOs (and other types) with spring annotations, such as @Repository and @Autowired
      // for auto-wiring the Configuration instance, e.g. from Spring Boot's jOOQ starter
      .withSpringAnnotations(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Generate the DAO classes
      daos = true

      // Annotate DAOs (and other types) with spring annotations, such as @Repository and @Autowired
      // for auto-wiring the Configuration instance, e.g. from Spring Boot's jOOQ starter
      springAnnotations = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
See generation annotations for more information about the annotation specific flags, above.

5.3.6. Generated sequences

Every sequence in your database will generate a org.jooq.Sequence implementation that looks like this:

```
public final class Sequences {

    // Every sequence generates a member
    public static final Sequence<Integer> S_AUTHOR_ID = new SequenceImpl<Integer>("S_AUTHOR_ID", TEST, INTEGER);
}
```

Flags controlling sequence generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Generate the Sequences class -->
      <sequences>true</sequences>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Generate the Sequences class
      .withSequences(true)
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Generate the Sequences class
      sequences = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.3.7. Generated procedures

Every procedure or function (routine) in your database will generate a org.jooq.Routine implementation that looks like this:

```
public class AuthorExists extends AbstractRoutine<java.lang.Void> {

    // All IN, IN OUT, OUT parameters and function return values generate a static member
    public static final Parameter<String>     AUTHOR_NAME = createParameter("AUTHOR_NAME", VARCHAR);
    public static final Parameter<BigDecimal> RESULT      = createParameter("RESULT",      NUMERIC);

    // A constructor for a new "empty" procedure call
    public AuthorExists() {
        super("AUTHOR_EXISTS", TEST);

        addInParameter(AUTHOR_NAME);
        addOutParameter(RESULT);
    }

    // Every IN and IN OUT parameter generates a setter
    public void setAuthorName(String value) {
        setValue(AUTHOR_NAME, value);
    }

    // Every IN OUT, OUT and RETURN_VALUE generates a getter
    public BigDecimal getResult() {
        return getValue(RESULT);
    }

    // [...]
}
```

Package and member procedures or functions

Procedures or functions contained in packages or UDTs are generated in a sub-package that corresponds to the package or UDT name.

Flags controlling routine generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Generate the Routines class and a class for each individual routine -->
      <routines>true</routines>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.

Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Generate the Routines class and a class for each individual routine
      .withRoutines(true)
  )
```

See the configuration XSD and programmatic code generation for more details.

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)

Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Generate the Routines class and a class for each individual routine
      routines = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.3.8. Generated UDTs

Every UDT in your database will generate a org.jooq.UDT implementation that looks like this:

```
public class AddressType extends UDTImpl<AddressTypeRecord> {

    // The singleton UDT instance
    public static final UAddressType U_ADDRESS_TYPE = new UAddressType();

    // Every UDT attribute generates a static member
    public static final UDTField<AddressTypeRecord, String> ZIP     =
      createField("ZIP",     VARCHAR, U_ADDRESS_TYPE);
    public static final UDTField<AddressTypeRecord, String> CITY    =
      createField("CITY",    VARCHAR, U_ADDRESS_TYPE);
    public static final UDTField<AddressTypeRecord, String> COUNTRY =
      createField("COUNTRY", VARCHAR, U_ADDRESS_TYPE);

    // [...]
}
```

Besides the org.jooq.UDT implementation, a org.jooq.UDTRecord implementation is also generated

```
public class AddressTypeRecord extends UDTRecordImpl<AddressTypeRecord> {

    // Every attribute generates a getter and a setter

    public void setZip(String value) {...}
    public String getZip() {...}
    public void setCity(String value) {...}
    public String getCity() {...}
    public void setCountry(String value) {...}
    public String getCountry() {...}

    // [...]
}
```

Flags controlling UDT generation

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Generate the UDTs class, records and UDT literals for each UDT -->
      <udts>true</udts>
    </generate>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.

Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(
    new Generate()

      // Generate the UDTs class, records and UDT literals for each UDT
      .withUdts(true)
  )
```

See the configuration XSD and programmatic code generation for more details.

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)

Gradle (third party)

```
generationTool {
  generator {
    generate {

      // Generate the UDTs class, records and UDT literals for each UDT
      udts = true
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.3.9. Generated global artefacts

For increased convenience at the use-site, jOOQ generates "global" artefacts at the code generation root location, referencing tables, routines, sequences, etc. In detail, these global artefacts include the following:

- Keys.java: This file contains all of the required primary key, unique key, foreign key and identity references in the form of static members of type org.jooq.Key.
- Routines.java: This file contains all standalone routines (not in packages) in the form of static factory methods for org.jooq.Routine types.
- Sequences.java: This file contains all sequence objects in the form of static members of type org.jooq.Sequence.
- Tables.java: This file contains all table objects in the form of static member references to the actual singleton org.jooq.Table object
- UDTs.java: This file contains all UDT objects in the form of static member references to the actual singleton org.jooq.UDT object

Referencing global artefacts

When referencing global artefacts from your client application, you would typically static import them as such:

```
// Static imports for all global artefacts (if they exist)
import static com.example.generated.Keys.*;
import static com.example.generated.Routines.*;
import static com.example.generated.Sequences.*;
import static com.example.generated.Tables.*;

// You could then reference your artefacts as follows:
create.insertInto(MY_TABLE)
      .values(MY_SEQUENCE.nextval(), myFunction())

// as a more concise form of this:
create.insertInto(com.example.generated.Tables.MY_TABLE)
      .values(com.example.generated.Sequences.MY_SEQUENCE.nextval(), com.example.generated.Routines.myFunction())
```

Configuring these artefacts

The generation of these artefacts can be turned off. For details, see the relevant section in the manual.

5.4. Class names, method names, identifiers

By default, generated class names, method names, identifiers, etc. follow Java's typical naming conventions.

Assuming a table like this:

```
CREATE TABLE account_transactions (
  id INTEGER NOT NULL,
  created_at TIMESTAMP NOT NULL,
  created_by VARCHAR(100) NOT NULL,
  ...
);
```

As a rule of thumb, the following defaults can be assumed for named objects:

- Class names follow PascalCase naming, i.e. the org.jooq.Table class name is AccountTransactions
- Records have a Record suffix, i.e. the org.jooq.UpdatableRecord class name is AccountTransactionsRecord
- Daos have a Dao suffix, i.e. the org.jooq.DAO class name is AccountTransactionsDao
- Identifiers follow SNAKE_CASE naming, i.e. the org.jooq.Table instance name is ACCOUNT_TRANSACTIONS
- Path methods follow a camleCase naming, i.e. the org.jooq.Path method name is accountTransactions() (e.g. on a child table)

If these defaults do not convene, or if a workaround needs to be applied for a limitation of the default naming, generator strategies allow for overriding these defaults.

5.4.1. Custom generator strategies

Using custom generator strategies to override naming schemes

jOOQ allows you to override default implementations of the code generator or the generator strategy. Specifically, the latter can be very useful if you want to inject custom behaviour into jOOQ's code generator with respect to naming classes, members, methods, and other Java objects.

XML (standalone and maven)

```
<configuration>
  <generator>

    <!-- A programmatic naming strategy implementation, referened by class name. -->
    <strategy>
      <name>com.example.AsInDatabaseStrategy</name>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()

    // A programmatic naming strategy implementation, referened by class name.
    .withStrategy(new Strategy()
      .withName("com.example.AsInDatabaseStrategy")
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {

    // A programmatic naming strategy implementation, referened by class name.
    strategy {
      name = "com.example.AsInDatabaseStrategy"
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

> *(!) Make sure your custom generator strategy and its dependencies are available to the code generator as a **code generator dependency***

The following example shows how you can override the DefaultGeneratorStrategy to render table and column names the way they are defined in the database, rather than switching them to camel case:

```
/**
 * It is recommended that you extend the DefaultGeneratorStrategy. Most of the
 * GeneratorStrategy API is already declared final. You only need to override any
 * of the following methods, for whatever generation behaviour you'd like to achieve.
 *
 * Also, the DefaultGeneratorStrategy takes care of disambiguating quite a few object
 * names in case of conflict. For example, MySQL indexes do not really have a name, so
 * a synthetic, non-ambiguous name is generated based on the table name. If you
 * override the default behaviour, you must ensure that this disambiguation still takes
 * place for generated code to be compilable.
 *
 * Beware that most methods also receive a "Mode" object, to tell you whether a
 * TableDefinition is being rendered as a Table, Record, POJO, etc. Depending on
 * that information, you can add a suffix only for TableRecords, but not for Tables,
 * for example.
 *
 * This example will name objects in Java as they are named in the database, without
 * applying any camelCase, PascalCase transformations, etc.
 */
public class AsInDatabaseStrategy extends DefaultGeneratorStrategy {

    /**
     * Override this to specify what identifiers, such as Catalog, Schema, Table,
     * TableField identifiers will look like.
     *
     * This example will just generate the identifier as defined in the database,
     * e.g. MY_TABLE or MY_TABLE.MY_FIELD.
     */
    @Override
    public String getJavaIdentifier(Definition definition) {
        // The DefaultGeneratorStrategy disambiguates some synthetic object names,
        // such as the MySQL PRIMARY key names, which do not really have a name
        // Uncomment the below code if you want to reuse that logic.
        // if (definition instanceof IndexDefinition)
        //     return super.getJavaIdentifier(definition);
        return definition.getOutputName();
    }

    /**
     * Override these to specify what a setter in Java should look like. Setters
     * are used in TableRecords, UDTRecords, and POJOs, depending on Mode.
     *
     * This example will name setters "set[NAME_IN_DATABASE]"
     */
    @Override
    public String getJavaSetterName(Definition definition, Mode mode) {
        return "set" + definition.getOutputName();
    }

    /**
     * Override these to specify what a getter in Java should look like. Getters
     * are used in TableRecords, UDTRecords, and POJOs, depending on Mode.
     *
     * This example will name getters "set[NAME_IN_DATABASE]"
     */
    @Override
    public String getJavaGetterName(Definition definition, Mode mode) {
        return "get" + definition.getOutputName();
    }

    /**
     * Override this method to define what a Java method generated from a database
     * Definition should look like. This is used e.g. for convenience methods
     * when calling stored procedures and functions, or for path methods.
     *
     * This example shows how to set a prefix to a CamelCase version of your
     * procedure.
     */
    @Override
    public String getJavaMethodName(Definition definition, Mode mode) {
        if (definition instanceof RoutineDefinition)
            return "call" + org.jooq.tools.StringUtils.toCamelCase(definition.getOutputName());
        else
            return super.getJavaMethodName(definition, mode);
    }

    /**
     * Override this method to define how your Java classes and Java files should
     * be named.
     *
     * This example applies no custom setting and uses CamelCase versions instead.
     */
    @Override
    public String getJavaClassName(Definition definition, Mode mode) {
        return super.getJavaClassName(definition, mode);
    }

    /**
     * Override this method to re-define the package names of your generated
     * artefacts.
     */
    @Override
    public String getJavaPackageName(Definition definition, Mode mode) {
        return super.getJavaPackageName(definition, mode);
    }

    /**
     * Override this method to define how Java members should be named. This is
     * used for POJO members and method arguments, for example, or when using the
     * KotlinGenerator or ScalaGenerator also for record or interface properties.
     */
    @Override
    public String getJavaMemberName(Definition definition, Mode mode) {
        return definition.getOutputName();
    }

    /**
     * Override this method to define the base class for those artefacts that
```

An org.jooq.Table example:

This is an example showing which generator strategy method will be called in what place when generating tables. For improved readability, full qualification is omitted:

```
package com.example.tables;
//   1: ^^^^^^^^^^^^^^^^^^
public class Book extends TableImpl<com.example.tables.records.BookRecord> {
//        2: ^^^^                                          3: ^^^^^^^^^^
    public static final Book    BOOK = new Book();
//                   2: ^^^^ 4: ^^^^
    public final TableField<BookRecord, Integer> ID = /* ... */
//                       3: ^^^^^^^^^^        5: ^^
}

// 1: strategy.getJavaPackageName(table)
// 2: strategy.getJavaClassName(table)
// 3: strategy.getJavaClassName(table, Mode.RECORD)
// 4: strategy.getJavaIdentifier(table)
// 5: strategy.getJavaIdentifier(column)
```

An org.jooq.Record example:

This is an example showing which generator strategy method will be called in what place when generating records. For improved readability, full qualification is omitted:

```
package com.example.tables.records;
//   1: ^^^^^^^^^^^^^^^^^^^^^^^^^^
public class BookRecord extends UpdatableRecordImpl<BookRecord> {
//        2: ^^^^^^^^^^                          2: ^^^^^^^^^^
    public void setId(Integer value) { /* ... */ }
//           3: ^^^^^
    public Integer getId() { /* ... */ }
//              4: ^^^^^
}

// 1: strategy.getJavaPackageName(table, Mode.RECORD)
// 2: strategy.getJavaClassName(table, Mode.RECORD)
// 3: strategy.getJavaSetterName(column, Mode.RECORD)
// 4: strategy.getJavaGetterName(column, Mode.RECORD)
```

A POJO example:

This is an example showing which generator strategy method will be called in what place when generating pojos. For improved readability, full qualification is omitted:

```
package com.example.tables.pojos;
//   1: ^^^^^^^^^^^^^^^^^^^^^^^^
public class Book implements java.io.Serializable {
//        2: ^^^^
    private Integer id;
//                3: ^^
    public void setId(Integer value) { /* ... */ }
//           4: ^^^^^
    public Integer getId() { /* ... */ }
//              5: ^^^^^

}

// 1: strategy.getJavaPackageName(table, Mode.POJO)
// 2: strategy.getJavaClassName(table, Mode.POJO)
// 3: strategy.getJavaMemberName(column, Mode.POJO)
// 4: strategy.getJavaSetterName(column, Mode.POJO)
// 5: strategy.getJavaGetterName(column, Mode.POJO)
```

© 2009 - 2025 by Data Geekery™ GmbH.

Out-of-the-box strategies

The jooq-codegen module provides the following out-of-the-box strategies, which can be used directly, or taken as inspiration:

- org.jooq.codegen.KeepNamesGeneratorStrategy: Keeps all names and identifiers exactly as they are in the database, including case and underscores. I.e. snake_case identifiers will produce snake_case classes and members, whereas PascalCase identifiers will produce PascalCase classes and members.

More examples can be found here:

- org.jooq.codegen.example.JPrefixGeneratorStrategy
- org.jooq.codegen.example.JVMArgsGeneratorStrategy

5.4.2. Matcher strategies

Using custom matcher strategies

In the previous section, we have seen how to override generator strategies programmatically. In this chapter, we'll see how such strategies can be configured in the XML or Maven code generator configuration. Instead of specifying a strategy name, you can also specify a <matchers/> element as explained in the following subsections.

The generic structure of such matchers looks as follows:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <tables>
          <table>

            <!-- Match unqualified or qualified table names. If left empty, this matcher applies to all tables. -->
            <expression>MY_TABLE</expression>

            <!-- These elements influence the naming of a generated org.jooq.Table object. -->
            <tableClass> a MatcherRule specification </tableClass>
            <tableIdentifier> a MatcherRule specification </tableIdentifier>

            <tableImplements>com.example.MyOptionalCustomInterface</tableImplements>
          </table>
        </tables>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()
        .withTables(
          new MatchersTableType()

            // Match unqualified or qualified table names. If left empty, this matcher applies to all tables.
            .withExpression("MY_TABLE")

            // These elements influence the naming of a generated org.jooq.Table object.
            .withTableClass(MatcherRule. a MatcherRule specification )
            .withTableIdentifier(MatcherRule. a MatcherRule specification )
            .withTableImplements("com.example.MyOptionalCustomInterface")
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {
        tables {
          table {

            // Match unqualified or qualified table names. If left empty, this matcher applies to all tables.
            expression = "MY_TABLE"

            // These elements influence the naming of a generated org.jooq.Table object.
            tableClass = " a MatcherRule specification "
            tableIdentifier = " a MatcherRule specification "
            tableImplements = "com.example.MyOptionalCustomInterface"
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

A matcher configuration element can be any of these things:

- A regular expression matching an identifier
- A org.jooq.meta.jaxb.MatcherRule specification (see MatcherRule)
- A constant value

(!) All regular expressions that match object identifiers try to match identifiers first by unqualified name (org.jooq.meta.Definition.getName()), then by qualified name (org.jooq.meta.Definition.getQualifiedName()).

The following sections explain the above more in detail:

5.4.2.1. MatcherRule

Most matchers use references to MatcherRule, which is consists of two elements:

- The regex replacement experssion to replace the matched name with.
- The transformation directive, which allows for specifying how to transform the resulting name.

Transformation directives

The following transformation directives are supported:

- AS_IS: Leave the database name as it is, e.g. MY_name => MY_name
- LOWER: ransform the database name into lower case, e.g. MY_name => my_name
- LOWER_FIRST_LETTER: Transform the first letter into lower case, e.g. MY_name => mY_name
- UPPER: Transform the database name into upper case, e.g. MY_name => MY_NAME
- UPPER_FIRST_LETTER: Transform the first letter into upper case, e.g. my_NAME => My_NAME
- CAMEL: Transform the database name into camel case, e.g. MY_name => myName
- PASCAL: Transform the database name into pascal case, e.g. MY_name => MyName

Example: Adding a prefix / suffix to names

XML (standalone and maven)

```
<configuration>
  <generator>
    <strategy>
      <matchers>
        <schemas>
          <schema>

            <!-- Without an input expression, this rule applies to all schemas -->
            <schemaClass>

              <!-- Optional transform directive -->
              <transform>CAMEL</transform>

              <!-- The mandatory expression element lets you specify a replacement expression to be used when
                   replacing the matcher's regular expression. You can use indexed variables $0, $1, $2. -->
              <expression>PREFIX_$0_SUFFIX</expression>
            </schemaClass>
          </schema>
        </schemas>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()
        .withSchemas(
          new MatchersSchemaType()

            // Without an input expression, this rule applies to all schemas
            .withSchemaClass(new MatcherRule()

              // Optional transform directive
              .withTransform(MatcherTransformType.CAMEL)

              // The mandatory expression element lets you specify a replacement expression to be used when
              // replacing the matcher's regular expression. You can use indexed variables $0, $1, $2.
              .withExpression("PREFIX_$0_SUFFIX")
            )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    strategy {
      matchers {
        schemas {
          schema {

            // Without an input expression, this rule applies to all schemas
            schemaClass {

              // Optional transform directive
              transform = "CAMEL"

              // The mandatory expression element lets you specify a replacement expression to be used when
              // replacing the matcher's regular expression. You can use indexed variables $0, $1, $2.
              expression = "PREFIX_$0_SUFFIX"
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Example: Removing a prefix / suffix from names

XML (standalone and maven)

```
<configuration>
  <generator>
    <strategy>
      <matchers>
        <tables>
          <table>

            <!-- Provide an optional input expression to apply this rule only to certain tables.
                 We can match groups with the regex (group expression) -->
            <expression>^T_(.*)$</expression>

            <tableClass>

              <!-- Optional transform directive -->
              <transform>CAMEL</transform>

              <!-- This ignores the prefix and replaces the name by the first matched group. -->
              <expression>$1</expression>
            </tableClass>
          </table>
        </tables>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()
        .withTables(
          new MatchersTableType()

            // Provide an optional input expression to apply this rule only to certain tables.
            // We can match groups with the regex (group expression)
            .withExpression("^T_(.*)$")
            .withTableClass(new MatcherRule()

              // Optional transform directive
              .withTransform(MatcherTransformType.CAMEL)

              // This ignores the prefix and replaces the name by the first matched group.
              .withExpression("$1")
            )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    strategy {
      matchers {
        tables {
          table {

            // Provide an optional input expression to apply this rule only to certain tables.
            // We can match groups with the regex (group expression)
            expression = "^T_(.*)$"
            tableClass {

              // Optional transform directive
              transform = "CAMEL"

              // This ignores the prefix and replaces the name by the first matched group.
              expression = "$1"
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

In other words, a MatcherRule describes how a specific object type name (e.g. a class name representing a generated org.jooq.Schema) should be declared and referenced based on the object's input name.

5.4.2.2. Matching schemas

The following example shows how to define a MatcherStrategy for generated org.jooq.Schema types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n schema matchers to provide a strategy for naming objects created from schemas. -->
        <schemas>
          <schema>

            <!-- Match unqualified or qualified schema names. If left empty, this matcher applies to all schemas. -->
            <expression>MY_SCHEMA</expression>

            <!-- These elements influence the naming of a generated org.jooq.Schema object. -->
            <schemaClass> a MatcherRule specification </schemaClass>
            <schemaIdentifier> a MatcherRule specification </schemaIdentifier>

            <schemaImplements>com.example.MyOptionalCustomInterface</schemaImplements>
          </schema>
        </schemas>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n schema matchers to provide a strategy for naming objects created from schemas.
        .withSchemas(
          new MatchersSchemaType()

            // Match unqualified or qualified schema names. If left empty, this matcher applies to all schemas.
            .withExpression("MY_SCHEMA")

            // These elements influence the naming of a generated org.jooq.Schema object.
            .withSchemaClass(MatcherRule. a MatcherRule specification )
            .withSchemaIdentifier(MatcherRule. a MatcherRule specification )
            .withSchemaImplements("com.example.MyOptionalCustomInterface")
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n schema matchers to provide a strategy for naming objects created from schemas.
        schemas {
          schema {

            // Match unqualified or qualified schema names. If left empty, this matcher applies to all schemas.
            expression = "MY_SCHEMA"

            // These elements influence the naming of a generated org.jooq.Schema object.
            schemaClass = " a MatcherRule specification "
            schemaIdentifier = " a MatcherRule specification "
            schemaImplements = "com.example.MyOptionalCustomInterface"
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.3. Matching tables

The following example shows how to define a MatcherStrategy for generated org.jooq.Table and org.jooq.TableRecord types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n table matchers to provide a strategy for naming objects created from tables. -->
        <tables>
          <table>

            <!-- Match unqualified or qualified table names. If left empty, this matcher applies to all tables. -->
            <expression>MY_TABLE</expression>

            <!-- These elements influence the naming of a generated org.jooq.Table object. -->
            <tableClass> a MatcherRule specification </tableClass>
            <tableIdentifier> a MatcherRule specification </tableIdentifier>

            <tableImplements>com.example.MyOptionalCustomInterface</tableImplements>


            <!-- These elements influence the naming of a generated org.jooq.Record object. -->
            <recordClass> a MatcherRule specification </recordClass>

            <recordImplements>com.example.MyOptionalCustomInterface</recordImplements>

            <!-- These elements influence the naming of a generated interface, implemented by
                 generated org.jooq.Record objects and by generated POJOs. -->
            <interfaceClass> a MatcherRule specification </interfaceClass>
            <interfaceImplements>com.example.MyOptionalCustomInterface</interfaceImplements>

            <!-- These elements influence the naming of a generated org.jooq.DAO object. -->
            <daoClass> a MatcherRule specification </daoClass>

            <daoImplements>com.example.MyOptionalCustomInterface</daoImplements>

            <!-- These elements influence the naming of a generated POJO object.  -->
            <pojoClass> a MatcherRule specification </pojoClass>
            <pojoExtends>com.example.MyOptionalPojoBaseClass</pojoExtends>
            <pojoImplements>com.example.MyOptionalCustomInterface</pojoImplements>
          </table>
        </tables>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n table matchers to provide a strategy for naming objects created from tables.
        .withTables(
          new MatchersTableType()

            // Match unqualified or qualified table names. If left empty, this matcher applies to all tables.
            .withExpression("MY_TABLE")

            // These elements influence the naming of a generated org.jooq.Table object.
            .withTableClass(MatcherRule. a MatcherRule specification )
            .withTableIdentifier(MatcherRule. a MatcherRule specification )
            .withTableImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated org.jooq.Record object.
            .withRecordClass(MatcherRule. a MatcherRule specification )
            .withRecordImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated interface, implemented by
            // generated org.jooq.Record objects and by generated POJOs.
            .withInterfaceClass(MatcherRule. a MatcherRule specification )
            .withInterfaceImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated org.jooq.DAO object.
            .withDaoClass(MatcherRule. a MatcherRule specification )
            .withDaoImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated POJO object.
            .withPojoClass(MatcherRule. a MatcherRule specification )
            .withPojoExtends("com.example.MyOptionalPojoBaseClass")
            .withPojoImplements("com.example.MyOptionalCustomInterface")
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n table matchers to provide a strategy for naming objects created from tables.
        tables {
          table {

            // Match unqualified or qualified table names. If left empty, this matcher applies to all tables.
            expression = "MY_TABLE"

            // These elements influence the naming of a generated org.jooq.Table object.
            tableClass = " a MatcherRule specification "
            tableIdentifier = " a MatcherRule specification "
            tableImplements = "com.example.MyOptionalCustomInterface"

            // These elements influence the naming of a generated org.jooq.Record object.
            recordClass = " a MatcherRule specification "
            recordImplements = "com.example.MyOptionalCustomInterface"

            // These elements influence the naming of a generated interface, implemented by
            // generated org.jooq.Record objects and by generated POJOs.
            interfaceClass = " a MatcherRule specification "
            interfaceImplements = "com.example.MyOptionalCustomInterface"

            // These elements influence the naming of a generated org.jooq.DAO object.
            daoClass = " a MatcherRule specification "
            daoImplements = "com.example.MyOptionalCustomInterface"

            // These elements influence the naming of a generated POJO object.
            pojoClass = " a MatcherRule specification "
            pojoExtends = "com.example.MyOptionalPojoBaseClass"
            pojoImplements = "com.example.MyOptionalCustomInterface"
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.4. Matching fields

The following example shows how to define a MatcherStrategy for generated org.jooq.Field types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n field matchers to provide a strategy for naming objects created from fields. -->
        <fields>
          <field>

            <!-- Match unqualified or qualified field names. If left empty, this matcher applies to all fields. -->
            <expression>MY_FIELD</expression>

            <!-- These elements influence the naming of a generated org.jooq.Field object. -->
            <fieldIdentifier> a MatcherRule specification </fieldIdentifier>
            <fieldMember> a MatcherRule specification </fieldMember>
            <fieldSetter> a MatcherRule specification </fieldSetter>
            <fieldGetter> a MatcherRule specification </fieldGetter>


          </field>
        </fields>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n field matchers to provide a strategy for naming objects created from fields.
        .withFields(
          new MatchersFieldType()

            // Match unqualified or qualified field names. If left empty, this matcher applies to all fields.
            .withExpression("MY_FIELD")

            // These elements influence the naming of a generated org.jooq.Field object.
            .withFieldIdentifier(MatcherRule. a MatcherRule specification )
            .withFieldMember(MatcherRule. a MatcherRule specification )
            .withFieldSetter(MatcherRule. a MatcherRule specification )
            .withFieldGetter(MatcherRule. a MatcherRule specification )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n field matchers to provide a strategy for naming objects created from fields.
        fields {
          field {

            // Match unqualified or qualified field names. If left empty, this matcher applies to all fields.
            expression = "MY_FIELD"

            // These elements influence the naming of a generated org.jooq.Field object.
            fieldIdentifier = " a MatcherRule specification "
            fieldMember = " a MatcherRule specification "
            fieldSetter = " a MatcherRule specification "
            fieldGetter = " a MatcherRule specification "
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.5. Matching routines

The following example shows how to define a MatcherStrategy for generated org.jooq.Routine types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n routine matchers to provide a strategy for naming objects created from routines. -->
        <routines>
          <routine>

            <!-- Match unqualified or qualified routine names. If left empty, this matcher applies to all routines. -->
            <expression>MY_ROUTINE</expression>

            <!-- These elements influence the naming of a generated org.jooq.Routine object. -->
            <routineClass> a MatcherRule specification </routineClass>
            <routineMethod> a MatcherRule specification </routineMethod>

            <routineImplements>com.example.MyOptionalCustomInterface</routineImplements>
          </routine>
        </routines>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n routine matchers to provide a strategy for naming objects created from routines.
        .withRoutines(
          new MatchersRoutineType()

            // Match unqualified or qualified routine names. If left empty, this matcher applies to all routines.
            .withExpression("MY_ROUTINE")

            // These elements influence the naming of a generated org.jooq.Routine object.
            .withRoutineClass(MatcherRule. a MatcherRule specification )
            .withRoutineMethod(MatcherRule. a MatcherRule specification )
            .withRoutineImplements("com.example.MyOptionalCustomInterface")
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n routine matchers to provide a strategy for naming objects created from routines.
        routines {
          routine {

            // Match unqualified or qualified routine names. If left empty, this matcher applies to all routines.
            expression = "MY_ROUTINE"

            // These elements influence the naming of a generated org.jooq.Routine object.
            routineClass = " a MatcherRule specification "
            routineMethod = " a MatcherRule specification "
            routineImplements = "com.example.MyOptionalCustomInterface"
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.6. Matching sequences

The following example shows how to define a MatcherStrategy for generated org.jooq.Sequence types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n sequence matchers to provide a strategy for naming objects created from sequences. -->
        <sequences>
          <sequence>

            <!-- Match unqualified or qualified sequence names. If left empty, this matcher applies to all sequences. -->
            <expression>MY_SEQUENCE</expression>

            <!-- These elements influence the naming of the generated Sequences class. -->
            <sequenceIdentifier> a MatcherRule specification </sequenceIdentifier>
          </sequence>
        </sequences>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n sequence matchers to provide a strategy for naming objects created from sequences.
        .withSequences(
          new MatchersSequenceType()

            // Match unqualified or qualified sequence names. If left empty, this matcher applies to all sequences.
            .withExpression("MY_SEQUENCE")

            // These elements influence the naming of the generated Sequences class.
            .withSequenceIdentifier(MatcherRule. a MatcherRule specification )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n sequence matchers to provide a strategy for naming objects created from sequences.
        sequences {
          sequence {

            // Match unqualified or qualified sequence names. If left empty, this matcher applies to all sequences.
            expression = "MY_SEQUENCE"

            // These elements influence the naming of the generated Sequences class.
            sequenceIdentifier = " a MatcherRule specification "
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.7. Matching enums

The following example shows how to define a MatcherStrategy for generated org.jooq.EnumType types and related objects:

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n enum matchers to provide a strategy for naming objects created from enums. -->
        <enums>
          <enum>

            <!-- Match unqualified or qualified enum names. If left empty, this matcher applies to all enums. -->
            <expression>MY_ENUM</expression>

            <!-- These elements influence the naming of a generated org.jooq.EnumType object. -->
            <enumClass> a MatcherRule specification </enumClass>
            <enumImplements>com.example.MyOptionalCustomInterface</enumImplements>

          </enum>
        </enums>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()

  // These properties can be added directly to the generator element:
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()

        // Specify 0..n enum matchers to provide a strategy for naming objects created from enums.
        .withEnums(
          new MatchersEnumType()

            // Match unqualified or qualified enum names. If left empty, this matcher applies to all enums.
            .withExpression("MY_ENUM")

            // These elements influence the naming of a generated org.jooq.EnumType object.
            .withEnumClass(MatcherRule. a MatcherRule specification )
            .withEnumImplements("com.example.MyOptionalCustomInterface")
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n enum matchers to provide a strategy for naming objects created from enums.
        enums {
          enum_ {

            // Match unqualified or qualified enum names. If left empty, this matcher applies to all enums.
            expression = "MY_ENUM"

            // These elements influence the naming of a generated org.jooq.EnumType object.
            enumClass = " a MatcherRule specification "
            enumImplements = "com.example.MyOptionalCustomInterface"
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

See MatcherRule for more information about MatcherRule specifications.

5.4.2.8. Matcher examples

The following example shows a matcher strategy that adds a "T_" prefix to all table classes and to table identifiers:

XML (standalone and maven)

```
<configuration>
  <generator>
    <strategy>
      <matchers>
        <tables>
          <table>

            <!-- Expression is omitted. This will make this rule apply to all tables -->
            <tableIdentifier>
              <transform>UPPER</transform>
              <expression>T_$0</expression>
            </tableIdentifier>
            <tableClass>
              <transform>PASCAL</transform>
              <expression>T_$0</expression>
            </tableClass>
          </table>
        </tables>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()
        .withTables(
          new MatchersTableType()

            // Expression is omitted. This will make this rule apply to all tables
            .withTableIdentifier(new MatcherRule()
              .withTransform(MatcherTransformType.UPPER)
              .withExpression("T_$0")
            )
            .withTableClass(new MatcherRule()
              .withTransform(MatcherTransformType.PASCAL)
              .withExpression("T_$0")
            )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    strategy {
      matchers {
        tables {
          table {

            // Expression is omitted. This will make this rule apply to all tables
            tableIdentifier {
              transform = "UPPER"
              expression = "T_$0"
            }
            tableClass {
              transform = "PASCAL"
              expression = "T_$0"
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

The following example shows a matcher strategy that renames BOOK table identifiers (or table identifiers containing BOOK) into BROCHURE (or tables containing BROCHURE):

XML (standalone and maven)

```
<configuration>
  <generator>
    <strategy>
      <matchers>
        <tables>
          <table>
            <expression>^(.*?)_BOOK_(.*)$</expression>
            <tableIdentifier>
              <transform>UPPER</transform>
              <expression>$1_BROCHURE_$2</expression>
            </tableIdentifier>
          </table>
        </tables>
      </matchers>
    </strategy>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withStrategy(new Strategy()
      .withMatchers(new Matchers()
        .withTables(
          new MatchersTableType()
            .withExpression("^(.*?)_BOOK_(.*)$")
            .withTableIdentifier(new MatcherRule()
              .withTransform(MatcherTransformType.UPPER)
              .withExpression("$1_BROCHURE_$2")
            )
        )
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    strategy {
      matchers {
        tables {
          table {
            expression = "^(.*?)_BOOK_(.*)$"
            tableIdentifier {
              transform = "UPPER"
              expression = "$1_BROCHURE_$2"
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

For more information about each XML tag, please refer to the https://www.jooq.org/xsd/jooq-codegen-3.12.0.xsd XSD file.

5.5. Custom code sections

Power users might choose to re-implement large parts of the org.jooq.codegen.JavaGenerator class. If you only want to add some custom code sections, however, you can extend the JavaGenerator and override only parts of it.

An example for generating custom class footers

```
public class MyGenerator1 extends JavaGenerator {

    @Override
    protected void generateRecordClassFooter(TableDefinition table, JavaWriter out) {
        out.println();
        out.tab(1).println("public String toString() {");
        out.tab(2).println("return \"MyRecord[\" + valuesRow() + \"]\";");
        out.tab(1).println("}");
    }
}
```

The above example simply adds a class footer to generated records, in this case, overriding the default toString() implementation.

An example for generating custom class Javadoc

```
public class MyGenerator2 extends JavaGenerator {

    @Override
    protected void generateRecordClassJavadoc(TableDefinition table, JavaWriter out) {
        out.println("/**");
        out.println(" * This record belongs to table " + table.getOutputName() + ".");

        if (table.getComment() != null && !"".equals(table.getComment())) {
            out.println(" * <p>");
            out.println(" * Table comment: " + table.getComment());
        }

        out.println(" */");
    }
}
```

Any of the below methods can be overridden:

```
generateArray(SchemaDefinition, ArrayDefinition)           // Generates an Oracle array class
generateArrayClassFooter(ArrayDefinition, JavaWriter)      // Callback for an Oracle array class footer
generateArrayClassJavadoc(ArrayDefinition, JavaWriter)     // Callback for an Oracle array class Javadoc

generateDao(TableDefinition)                               // Generates a DAO class
generateDaoClassFooter(TableDefinition, JavaWriter)        // Callback for a DAO class footer
generateDaoClassJavadoc(TableDefinition, JavaWriter)       // Callback for a DAO class Javadoc

generateEnum(EnumDefinition)                               // Generates an enum
generateEnumClassFooter(EnumDefinition, JavaWriter)        // Callback for an enum footer
generateEnumClassJavadoc(EnumDefinition, JavaWriter)       // Callback for an enum Javadoc

generateInterface(TableDefinition)                         // Generates an interface
generateInterfaceClassFooter(TableDefinition, JavaWriter)  // Callback for an interface footer
generateInterfaceClassJavadoc(TableDefinition, JavaWriter) // Callback for an interface Javadoc

generatePackage(SchemaDefinition, PackageDefinition)       // Generates an Oracle package class
generatePackageClassFooter(PackageDefinition, JavaWriter)  // Callback for an Oracle package class footer
generatePackageClassJavadoc(PackageDefinition, JavaWriter) // Callback for an Oracle package class Javadoc

generatePojo(TableDefinition)                              // Generates a POJO class
generatePojoClassFooter(TableDefinition, JavaWriter)       // Callback for a POJO class footer
generatePojoClassJavadoc(TableDefinition, JavaWriter)      // Callback for a POJO class Javadoc

generateRecord(TableDefinition)                            // Generates a Record class
generateRecordClassFooter(TableDefinition, JavaWriter)     // Callback for a Record class footer
generateRecordClassJavadoc(TableDefinition, JavaWriter)    // Callback for a Record class Javadoc

generateRoutine(SchemaDefinition, RoutineDefinition)       // Generates a Routine class
generateRoutineClassFooter(RoutineDefinition, JavaWriter)  // Callback for a Routine class footer
generateRoutineClassJavadoc(RoutineDefinition, JavaWriter) // Callback for a Routine class Javadoc

generateCatalog(CatalogDefinition)                         // Generates a Catalog class
generateCatalogClassFooter(CatalogDefinition, JavaWriter)  // Callback for a Catalog class footer
generateCatalogClassJavadoc(CatalogDefinition, JavaWriter) // Callback for a Catalog class Javadoc

generateSchema(SchemaDefinition)                           // Generates a Schema class
generateSchemaClassFooter(SchemaDefinition, JavaWriter)    // Callback for a Schema class footer
generateSchemaClassJavadoc(SchemaDefinition, JavaWriter)   // Callback for a Schema class Javadoc

generateTable(SchemaDefinition, TableDefinition)           // Generates a Table class
generateTableClassFooter(TableDefinition, JavaWriter)      // Callback for a Table class footer
generateTableClassJavadoc(TableDefinition, JavaWriter)     // Callback for a Table class Javadoc

generateUDT(SchemaDefinition, UDTDefinition)               // Generates a UDT class
generateUDTClassFooter(UDTDefinition, JavaWriter)          // Callback for a UDT class footer
generateUDTClassJavadoc(UDTDefinition, JavaWriter)         // Callback for a UDT class Javadoc

generateUDTRecord(UDTDefinition)                           // Generates a UDT Record class
generateUDTRecordClassFooter(UDTDefinition, JavaWriter)    // Callback for a UDT Record class footer
generateUDTRecordClassJavadoc(UDTDefinition, JavaWriter)   // Callback for a UDT Record class Javadoc
```

When you override any of the above, do note that according to jOOQ's understanding of semantic versioning, incompatible changes may be introduced between minor releases, even if this should be the exception.

© 2009 - 2025 by Data Geekery™ GmbH.

*(!) Make sure your custom generator and its dependencies are available to the code generator as a **code generator dependency***

5.6. Mapping generated catalogs and schemas

We've seen previously in the chapter about runtime schema mapping, that catalogs, schemata and tables can be mapped at runtime to other names. But you can also hard-wire catalog and schema mapping in generated artefacts at code generation time, e.g. when you have 5 developers with their own dedicated developer databases, and a common integration database. In the code generation configuration, you would then write.

Schema mapping

The following configuration applies mapping only for schemata, not for catalogs. The <schemata/> element is a standalone element that can be put in the code generator's <database/> configuration element:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <schemata>
        <schema>

          <!-- Use this as the developer's schema: -->
          <inputSchema>LUKAS_DEV_SCHEMA</inputSchema>

          <!-- Use this as the integration / production database: -->
          <outputSchema>PROD</outputSchema>
        </schema>
      </schemata>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withSchemata(
        new SchemaMappingType()

          // Use this as the developer's schema:
          .withInputSchema("LUKAS_DEV_SCHEMA")

          // Use this as the integration / production database:
          .withOutputSchema("PROD")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      schemata {
        schema {

          // Use this as the developer's schema:
          inputSchema = "LUKAS_DEV_SCHEMA"

          // Use this as the integration / production database:
          outputSchema = "PROD"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
The following configuration applies mapping for catalogs and their schemata. The <catalogs/> element is a standalone element that can be put in the code generator's <database/> configuration element:
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <catalogs>
        <catalog>

          <!-- Use this as the developer's catalog: -->
          <inputCatalog>LUKAS_DEV_CATALOG</inputCatalog>

          <!-- Use this as the integration / production database: -->
          <outputCatalog>PROD</outputCatalog>

          <!-- Optionally, nest also schema mapping configurations: -->
          <schemata>
          </schemata>
        </catalog>
      </catalogs>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withCatalogs(
        new CatalogMappingType()

          // Use this as the developer's catalog:
          .withInputCatalog("LUKAS_DEV_CATALOG")

          // Use this as the integration / production database:
          .withOutputCatalog("PROD")

          // Optionally, nest also schema mapping configurations:
          .withSchemata()
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      catalogs {
        catalog {

          // Use this as the developer's catalog:
          inputCatalog = "LUKAS_DEV_CATALOG"

          // Use this as the integration / production database:
          outputCatalog = "PROD"

          // Optionally, nest also schema mapping configurations:
          schemata {}
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

5.7. Alternative meta data sources

Most code generation configurations will reference an actual RDBMS connection using a JDBC URL to connect to a real RDBMS instance for meta data reverse engineering. The main benefit is that an actual database product is used, similar to what is going to be used in production. As such, all vendor specific features are available and code generation isn't limited.

But there are valid reasons for using an alternative meta data source, or "source of truth" for meta data. For example, there may be a pre-existing migration using DDL scripts (such as used by Flyway). Or, you may work with JPA annotated entities and wish to use those as a source of truth.

The following sections illustrated your options in this area.

5.7.1. JPADatabase: Code generation from entities

Many jOOQ users use jOOQ as a complementary SQL API in applications that mostly use JPA for their database interactions, e.g. to perform reporting, batch processing, analytics, etc.

In such a setup, you might have a pre-existing schema implemented using JPA-annotated entities. Your real database schema might not be accessible while developing, or it is not a first-class citizen in your application (i.e. you follow a Java-first approach). This section explains how you can generate jOOQ classes from such a JPA model. Consider this model:

```
@Entity
@Table(name = "author")
public class Author {

    @Id
    int       id;

    @Column(name = "first_name")
    String    firstName;

    @Column(name = "last_name")
    String    lastName;

    @OneToMany(mappedBy = "author")
    Set<Book> books;

    // Getters and setters...
}

@Entity
@Table(name = "book")
public class Book {

    @Id
    public int    id;

    @Column(name = "title")
    public String title;

    @ManyToOne
    public Author author;

    // Getters and setters...
}
```

Now, instead of connecting the jOOQ code generator to a database that holds a representation of the above schema, you can use jOOQ's JPADatabase and feed that to the code generator. The JPADatabase uses Hibernate internally, to generate an in-memory H2 database from your entities, and reverse-engineers that again back to jOOQ classes.

The easiest way forward is to use Maven in order to include the jooq-meta-extensions-hibernate library (which then includes the H2 and Hibernate dependencies)

Maven

```
<dependency>
    <!-- Use org.jooq              for the Open Source Edition
             org.jooq.pro          for commercial editions with Java 11 support,
             org.jooq.pro-java-8   for commercial editions with Java 8 support,
             org.jooq.pro-java-6   for commercial editions with Java 6 support,
             org.jooq.trial        for the free trial edition with Java 11 support,
             org.jooq.trial-java-8 for the free trial edition with Java 8 support,
             org.jooq.trial-java-6 for the free trial edition with Java 6 support

     Note: Only the Open Source Edition is hosted on Maven Central.
           Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
           See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

    <groupId>org.jooq</groupId>

    <!-- The artifactId has been renamed to jooq-meta-extensions-hibernate in jOOQ 3.14 -->
    <artifactId>jooq-meta-extensions</artifactId>
    <version>3.12.4</version>
</dependency>
```

Gradle (Kotlin)

```
dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    jooqCodegen("org.jooq:jooq-meta-extensions-hibernate:3.12.4")
}
```

Gradle (Groovy)

With that dependency in place, you can now specify the JPADatabase in your code generator configuration:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.jpa.JPADatabase</name>
      <properties>

        <!-- A comma separated list of Java packages, that contain your entities -->
        <property>
          <key>packages</key>
          <value>com.example.entities</value>
        </property>

        <!-- Whether JPA 2.1 AttributeConverters should be auto-mapped to jOOQ Converters.
             Custom <forcedType/> configurations will have a higher priority than these auto-mapped converters.
             This defaults to true. -->
        <property>
          <key>useAttributeConverters</key>
          <value>true</value>
        </property>
        <!-- The default schema for unqualified objects:

             - public: all unqualified objects are located in the PUBLIC (upper case) schema
             - none: all unqualified objects are located in the default schema (default)

             This configuration can be overridden with the schema mapping feature -->
        <property>
          <key>unqualifiedSchema</key>
          <value>none</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.extensions.jpa.JPADatabase")
      .withProperties(

        // A comma separated list of Java packages, that contain your entities
        new Property()
          .withKey("packages")
          .withValue("com.example.entities"),

        // Whether JPA 2.1 AttributeConverters should be auto-mapped to jOOQ Converters.
        // Custom <forcedType/> configurations will have a higher priority than these auto-mapped converters.
        // This defaults to true.
        new Property()
          .withKey("useAttributeConverters")
          .withValue(true),

        // The default schema for unqualified objects:
        //
        // - public: all unqualified objects are located in the PUBLIC (upper case) schema
        // - none: all unqualified objects are located in the default schema (default)
        //
        // This configuration can be overridden with the schema mapping feature
        new Property()
          .withKey("unqualifiedSchema")
          .withValue("none")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.extensions.jpa.JPADatabase"
      properties {

        // A comma separated list of Java packages, that contain your entities
        property {
          key = "packages"
          value = "com.example.entities"
        }

        // Whether JPA 2.1 AttributeConverters should be auto-mapped to jOOQ Converters.
        // Custom <forcedType/> configurations will have a higher priority than these auto-mapped converters.
        // This defaults to true.
        property {
          key = "useAttributeConverters"
          value = true
        }

        // The default schema for unqualified objects:
        //
        // - public: all unqualified objects are located in the PUBLIC (upper case) schema
        // - none: all unqualified objects are located in the default schema (default)
        //
        // This configuration can be overridden with the schema mapping feature
        property {
          key = "unqualifiedSchema"
          value = "none"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
The above will generate all jOOQ artefacts for your AUTHOR and BOOK tables.

Passing settings to Hibernate

The current versions of jOOQ use Hibernate behind the scenes to generate an in-memory H2 database from which to reverse engineer jOOQ code. In order to influence Hibernate's schema generation, Hibernate specific flags can be passed to MetadataSources. All properties that are prefixed with hibernate. or jakarta.persistence. will be passed along to Hibernate. For example, this is possible:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <properties>
        <property>
          <key>hibernate.physical_naming_strategy</key>
          <value>org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withProperties(
        new Property()
          .withKey("hibernate.physical_naming_strategy")
          .withValue("org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      properties {
        property {
          key = "hibernate.physical_naming_strategy"
          value = "org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

How to organise your dependencies

The JPADatabase will use Spring to look up your annotated entities from the classpath. This means that you have to create several modules with a dependency graph that looks like this:

```
                        +-------------------+
                        | Your JPA entities |
                        +-------------------+
                             ^         ^
                   depends on |         | depends on
                              |         |
          +---------------------+   +---------------------+
          | jOOQ codegen plugin |   | Your application    |
          +---------------------+   +---------------------+
                              |         |
                    generates |         | depends on
                              v         v
                     +-------------------------+
                     | jOOQ generated classes  |
                     +-------------------------+
```

You cannot put your JPA entities in the same module as the one that runs the jOOQ code generator. See also code generator dependencies.

5.7.2. XMLDatabase: Code generation from XML files

By default, jOOQ's code generator takes live database connections as a database meta data source. In many project setups, this might not be optimal, as the live database is not always available.

One way to circumvent this issue is by providing jOOQ with a database meta definition file in XML format and by passing this XML file to jOOQ's XMLDatabase.

The XMLDatabase can read a standardised XML file that implements the https://www.jooq.org/xsd/jooq-meta-3.12.0.xsd schema. Essentially, this schema is an XML representation of the SQL standard INFORMATION_SCHEMA, as implemented by databases like H2, HSQLDB, MySQL, PostgreSQL, or SQL Server.

An example schema definition containing simple schema, table, column definitions can be seen below:

```
<?xml version="1.0"?>
<information_schema xmlns="http://www.jooq.org/xsd/jooq-meta-3.12.0.xsd">
    <schemata>
        <schema>
            <schema_name>TEST</schema_name>
        </schema>
    </schemata>

    <tables>
        <table>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
        </table>
        <table>
            <table_schema>TEST</table_schema>
            <table_name>BOOK</table_name>
        </table>
    </tables>

    <columns>
        <column>
            <table_schema>PUBLIC</table_schema>
            <table_name>AUTHOR</table_name>
            <column_name>ID</column_name>
            <data_type>NUMBER</data_type>
            <numeric_precision>7</numeric_precision>
            <ordinal_position>1</ordinal_position>
            <is_nullable>false</is_nullable>
        </column>
        <!-- ... -->
    </columns>
</information_schema>
```

Constraints can be defined with the following elements:

```
<?xml version="1.0"?>
<information_schema xmlns="http://www.jooq.org/xsd/jooq-meta-3.12.0.xsd">
    <table_constraints>
        <table_constraint>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>PK_AUTHOR</constraint_name>
            <constraint_type>PRIMARY KEY</constraint_type>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
        </table_constraint>
        <!-- ... -->
    </table_constraints>

    <key_column_usages>
        <key_column_usage>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>PK_AUTHOR</constraint_name>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
            <column_name>ID</column_name>
            <ordinal_position>1</ordinal_position>
        </key_column_usage>
        <!-- ... -->
    </key_column_usages>

    <referential_constraints>
        <referential_constraint>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>FK_BOOK_AUTHOR_ID</constraint_name>
            <unique_constraint_schema>TEST</unique_constraint_schema>
            <unique_constraint_name>PK_AUTHOR</unique_constraint_name>
        </referential_constraint>
        <!-- ... -->
    </referential_constraints>
</information_schema>
```

The above file can be made available to the code generator configuration by using the XMLDatabase as follows:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.xml.XMLDatabase</name>
      <properties>

        <!-- Use any of the SQLDialect values here -->
        <property>
          <key>dialect</key>
          <value>ORACLE</value>
        </property>

        <!-- Specify the location of your database file -->
        <property>
          <key>xmlFile</key>
          <value>src/main/resources/database.xml</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.xml.XMLDatabase")
      .withProperties(

        // Use any of the SQLDialect values here
        new Property()
          .withKey("dialect")
          .withValue("ORACLE"),

        // Specify the location of your database file
        new Property()
          .withKey("xmlFile")
          .withValue("src/main/resources/database.xml")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.xml.XMLDatabase"
      properties {

        // Use any of the SQLDialect values here
        property {
          key = "dialect"
          value = "ORACLE"
        }

        // Specify the location of your database file
        property {
          key = "xmlFile"
          value = "src/main/resources/database.xml"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
If you already have a different XML format for your database, you can either XSL transform your own format into the one above via an additional Maven plugin, or pass the location of an XSL file to the XMLDatabase by providing an additional property:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.xml.XMLDatabase</name>
      <properties>

        <!-- Specify the location of your xsl file -->
        <property>
          <key>xslFile</key>
          <value>src/main/resources/transform-to-jooq-format.xsl</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.xml.XMLDatabase")
      .withProperties(

        // Specify the location of your xsl file
        new Property()
          .withKey("xslFile")
          .withValue("src/main/resources/transform-to-jooq-format.xsl")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.xml.XMLDatabase"
      properties {

        // Specify the location of your xsl file
        property {
          key = "xslFile"
          value = "src/main/resources/transform-to-jooq-format.xsl"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
This XML configuration can now be checked in and versioned, and modified independently from your live database schema.

5.7.3. DDLDatabase: Code generation from SQL files

In many cases, the schema is defined in the form of a SQL script, which can be used with Flyway, or some other database migration tool.

If you have a complete schema definition in a single file, or perhaps a set of incremental files that can reproduce your schema in any SQL dialect, then the DDLDatabase might be the right choice for you. It uses the SQL parser internally and applies all your DDL increments to an in-memory H2 database, in order to produce a replica of your schema prior to reverse engineering it again using ordinary code generation.

For example, the following database.sql script (the sample database from this manual) could be used:

```
CREATE TABLE language (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  cd              CHAR(2)       NOT NULL,
  description     VARCHAR2(50)
);

CREATE TABLE author (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  first_name      VARCHAR2(50),
  last_name       VARCHAR2(50)  NOT NULL,
  date_of_birth   DATE,
  year_of_birth   NUMBER(7),
  distinguished   NUMBER(1)
);

CREATE TABLE book (
  id              NUMBER(7)     NOT NULL PRIMARY KEY,
  author_id       NUMBER(7)     NOT NULL,
  title           VARCHAR2(400) NOT NULL,
  published_in    NUMBER(7)     NOT NULL,
  language_id     NUMBER(7)     NOT NULL,

  CONSTRAINT fk_book_author     FOREIGN KEY (author_id)   REFERENCES author(id),
  CONSTRAINT fk_book_language   FOREIGN KEY (language_id) REFERENCES language(id)
);

CREATE TABLE book_store (
  name            VARCHAR2(400) NOT NULL UNIQUE
);

CREATE TABLE book_to_book_store (
  name            VARCHAR2(400) NOT NULL,
  book_id         INTEGER       NOT NULL,
  stock           INTEGER,

  PRIMARY KEY(name, book_id),
  CONSTRAINT fk_b2bs_book_store FOREIGN KEY (name)        REFERENCES book_store (name) ON DELETE CASCADE,
  CONSTRAINT fk_b2bs_book       FOREIGN KEY (book_id)     REFERENCES book (id)         ON DELETE CASCADE
);
```

While the script uses pretty standard SQL constructs, you may well use some vendor-specific extensions, and even DML statements in between to set up your schema - it doesn't matter. You will simply need to set up your code generation configuration as follows:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.ddl.DDLDatabase</name>
      <properties>

        <!-- Specify the location of your SQL script.
             You may use ant-style file matching, e.g. /path/**/to/*.sql

             Where:
             - ** matches any directory subtree
             - * matches any number of characters in a directory / file name
             - ? matches a single character in a directory / file name -->
        <property>
          <key>scripts</key>
          <value>src/main/resources/database.sql</value>
        </property>

        <!-- The sort order of the scripts within a directory, where:

             - semantic: sorts versions, e.g. v-3.10.0 is after v-3.9.0 (default)
             - alphanumeric: sorts strings, e.g. v-3.10.0 is before v-3.9.0
             - flyway: sorts files the same way as flyway does
             - none: doesn't sort directory contents after fetching them from the directory -->
        <property>
          <key>sort</key>
          <value>semantic</value>
        </property>
        <!-- The default schema for unqualified objects:

             - public: all unqualified objects are located in the PUBLIC (upper case) schema
             - none: all unqualified objects are located in the default schema (default)

             This configuration can be overridden with the schema mapping feature -->
        <property>
          <key>unqualifiedSchema</key>
          <value>none</value>
        </property>

        <!-- The default name case for unquoted objects:

             - as_is: unquoted object names are kept unquoted
             - upper: unquoted object names are turned into upper case (most databases)
             - lower: unquoted object names are turned into lower case (e.g. PostgreSQL) -->
        <property>
          <key>defaultNameCase</key>
          <value>as_is</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.extensions.ddl.DDLDatabase")
      .withProperties(

        // Specify the location of your SQL script.
        // You may use ant-style file matching, e.g. /path/**/to/*.sql
        //
        // Where:
        // - ** matches any directory subtree
        // - * matches any number of characters in a directory / file name
        // - ? matches a single character in a directory / file name
        new Property()
          .withKey("scripts")
          .withValue("src/main/resources/database.sql"),

        // The sort order of the scripts within a directory, where:
        //
        // - semantic: sorts versions, e.g. v-3.10.0 is after v-3.9.0 (default)
        // - alphanumeric: sorts strings, e.g. v-3.10.0 is before v-3.9.0
        // - flyway: sorts files the same way as flyway does
        // - none: doesn't sort directory contents after fetching them from the directory
        new Property()
          .withKey("sort")
          .withValue("semantic"),

        // The default schema for unqualified objects:
        //
        // - public: all unqualified objects are located in the PUBLIC (upper case) schema
        // - none: all unqualified objects are located in the default schema (default)
        //
        // This configuration can be overridden with the schema mapping feature
        new Property()
          .withKey("unqualifiedSchema")
          .withValue("none"),

        // The default name case for unquoted objects:
        //
        // - as_is: unquoted object names are kept unquoted
        // - upper: unquoted object names are turned into upper case (most databases)
        // - lower: unquoted object names are turned into lower case (e.g. PostgreSQL)
        new Property()
          .withKey("defaultNameCase")
          .withValue("as_is")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.

Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)

Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.extensions.ddl.DDLDatabase"
      properties {

        // Specify the location of your SQL script.
        // You may use ant-style file matching, e.g. /path/**/to/*.sql
        //
        // Where:
        // - ** matches any directory subtree
        // - * matches any number of characters in a directory / file name
        // - ? matches a single character in a directory / file name
        property {
          key = "scripts"
          value = "src/main/resources/database.sql"
        }

        // The sort order of the scripts within a directory, where:
        //
        // - semantic: sorts versions, e.g. v-3.10.0 is after v-3.9.0 (default)
        // - alphanumeric: sorts strings, e.g. v-3.10.0 is before v-3.9.0
        // - flyway: sorts files the same way as flyway does
        // - none: doesn't sort directory contents after fetching them from the directory
        property {
          key = "sort"
          value = "semantic"
        }

        // The default schema for unqualified objects:
        //
        // - public: all unqualified objects are located in the PUBLIC (upper case) schema
        // - none: all unqualified objects are located in the default schema (default)
        //
        // This configuration can be overridden with the schema mapping feature
        property {
          key = "unqualifiedSchema"
          value = "none"
        }

        // The default name case for unquoted objects:
        //
        // - as_is: unquoted object names are kept unquoted
        // - upper: unquoted object names are turned into upper case (most databases)
        // - lower: unquoted object names are turned into lower case (e.g. PostgreSQL)
        property {
          key = "defaultNameCase"
          value = "as_is"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Ignoring unsupported content

The jOOQ parser supports parsing everything that is representable through the jOOQ API, as well as ignores some well known vendor specific syntax. But RDBMS have a lot more features and syntax that are not known to jOOQ. In this case, you can specify two comment tokens around the SQL syntax that jOOQ should ignore. The tokens are located in ordinary single line or multi line comments, so they do not affect your DDL scripts in any other way. For example:

```
-- [jooq ignore start]
-- Anything between these two tokens is ignored by the jOOQ parser
CREATE EXTENSION postgis;
-- [jooq ignore stop]

CREATE TABLE a (i INT);
CREATE TABLE b (i INT);

/* [jooq ignore start] */
-- This table will not be generated by jOOQ:
CREATE TABLE c (i INT);
/* [jooq ignore stop] */
```

The tokens can be overridden, or the feature can be turned off entirely using the following properties:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.ddl.DDLDatabase</name>
      <properties>

        <!-- Turn on/off ignoring contents between such tokens. Defaults to true -->
        <property>
          <key>parseIgnoreComments</key>
          <value>true</value>
        </property>

        <!-- Change the starting token -->
        <property>
          <key>parseIgnoreCommentStart</key>
          <value>[jooq ignore start]</value>
        </property>

        <!-- Change the stopping token -->
        <property>
          <key>parseIgnoreCommentStop</key>
          <value>[jooq ignore stop]</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withName("org.jooq.meta.extensions.ddl.DDLDatabase")
      .withProperties(

        // Turn on/off ignoring contents between such tokens. Defaults to true
        new Property()
          .withKey("parseIgnoreComments")
          .withValue(true),

        // Change the starting token
        new Property()
          .withKey("parseIgnoreCommentStart")
          .withValue("[jooq ignore start]"),

        // Change the stopping token
        new Property()
          .withKey("parseIgnoreCommentStop")
          .withValue("[jooq ignore stop]")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {
      name = "org.jooq.meta.extensions.ddl.DDLDatabase"
      properties {

        // Turn on/off ignoring contents between such tokens. Defaults to true
        property {
          key = "parseIgnoreComments"
          value = true
        }

        // Change the starting token
        property {
          key = "parseIgnoreCommentStart"
          value = "[jooq ignore start]"
        }

        // Change the stopping token
        property {
          key = "parseIgnoreCommentStop"
          value = "[jooq ignore stop]"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.

Dependencies

Note that the org.jooq.meta.extensions.ddl.DDLDatabase class is located in an external dependency, which needs to be placed on the classpath of the jOOQ code generator. E.g. using Maven:

Maven

```
<dependency>
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta-extensions</artifactId>
  <version>3.12.4</version>
</dependency>
```

Gradle (Kotlin)

```
dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    jooqCodegen("org.jooq:jooq-meta-extensions:3.12.4")
}
```

Gradle (Groovy)

5.8. Alternative output languages

The following sections describe alternative output languages that can be generated using the code generator, including XML, Kotlin, Scala.

5.8.1. XMLGenerator: Generating XML

By default the code generator produces Java files for use with the jOOQ API as documented throughout this manual. In some cases, however, it may be desireable to generate other meta data formats, such as an XML document. This can be done with the XMLGenerator.

The format produced by the XMLGenerator is the same as the one consumed by the XMLDatabase, which can read such XML content to produce Java code. It is specified in the https://www.jooq.org/xsd/jooq-meta-3.12.0.xsd schema. Essentially, this schema is an XML representation of the SQL standard INFORMATION_SCHEMA, as implemented by databases like H2, HSQLDB, MySQL, PostgreSQL, or SQL Server.

In order to use the XMLGenerator, simply place the following class reference into your code generation configuration:

XML (standalone and maven)

```
<configuration>
  <generator>
    <name>org.jooq.codegen.XMLGenerator</name>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withName("org.jooq.codegen.XMLGenerator")
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    name = "org.jooq.codegen.XMLGenerator"
  }
}
```

See the configuration XSD and gradle code generation for more details.
This configuration does not interfere with most of the remaining code generation configuration, e.g. you can still specify the JDBC connection or the generation output target as usual.

5.9. Code generation execution

There are various ways to execute the code generator, either as a standalone application or using third party build systems like Maven, Gradle, Ant, or others.

5.9.1. Running the code generator with Maven

There is no substantial difference between running the code generator with Maven or in standalone mode. Both modes use the exact same <configuration/> element. The Maven plugin configuration adds some additional boilerplate around that:

```
<plugin>
  <!-- Specify the maven code generator plugin -->
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.12.4</version>

  <executions>
    <execution>
      <id>my-id</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>

  <configuration>
    ...
  </configuration>
</plugin>
```

(!) Note that in Maven, you can place configurations both in the plugin as well as in the execution of the plugin. The latter is useful especially if you have multiple executions, see below. If you do, however, please keep in mind that you may have to supply the execution ID explicitly when running the code generator from the command line, e.g. using mvn jooq-codegen:generate@my-id

Additional Maven-specific flags

There are, however, some additional, Maven-specific flags that can be specified with the jooq-codegen-maven plugin only:

```
<plugin>
  <configuration>

    <!-- A boolean property (or constant) can be specified here to tell the plugin not to do anything -->
    <skip>${skip.jooq.generation}</skip>

    <!-- Instead of providing an inline configuration here, you can specify an external XML configuration file here -->
    <configurationFile>${externalfile}</configurationFile>

    <!-- Alternatively, you can provide several external configuration files. These will be merged by using
         Maven's combine.children="append" policy -->
    <configurationFiles>
      <configurationFile>${file1}</configurationFile>
      <configurationFile>${file2}</configurationFile>
      <configurationFile>...</configurationFile>
    </configurationFiles>
  </configuration>
</plugin>
```

Maven plugin inheritance mechanism

As with any other Maven plugin configuration, you can profit from Maven's powerful plugin inheritance mechanism. This is particularly useful when you configure several code generation runs for jOOQ, e.g.

- Different configurations for different tenants
- Different configurations for different schemas
- Different configurations for different environments

One approach would be to use a <pluginManagement/> configuration

```
<pluginManagement>
  <plugins>
    <plugin>
      <!-- Use org.jooq              for the Open Source Edition
               org.jooq.pro          for commercial editions,
               org.jooq.pro-java-8   for commercial editions with Java 8 support,
               org.jooq.pro-java-6   for commercial editions with Java 6 support,
               org.jooq.trial        for the free trial edition,
               org.jooq.trial-java-8 for the free trial edition with Java 8 support,
               org.jooq.trial-java-6 for the free trial edition with Java 6 support

           Note: Only the Open Source Edition is hosted on Maven Central.
                 Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org -->
      <groupId>org.jooq</groupId>
      <artifactId>jooq-codegen-maven</artifactId>
      <configuration>

        <!-- Log at WARN level by default -->
        <logging>WARN</logging>
        <generator>
          <generate>

            <!-- Never generate deprecated code -->
            <deprecated>false</deprecated>
          </generate>
          <database>
            <forcedTypes>

              <!-- Use BIGINT for all ID columns -->
              <forcedType>
                <name>BIGINT</name>
                <includeExpression>ID</includeExpression>
              </forcedType>
            </forcedTypes>
          </database>
        </generator>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
```

The above now applies to all of your jooq-codegen-maven plugin configurations *by default*. Now, in some module or execution, you may want to enhance / override the above defaults:

```
<plugin>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <configuration>

    <!-- Log at INFO level by in this particular configuration -->
    <logging>INFO</logging>
    <generator>
      <database>
        <!-- Append additional forced type children to the default, rather
             than merging or replacing the default -->
        <forcedTypes combine.children="append">
          <forcedType>
            <name>BIGINT</name>
            <includeExpression>.*_ID</includeExpression>
          </forcedType>
        </forcedTypes>
      </database>
    </generator>
  </configuration>
</plugin>
```

Another approach is to use multiple executions:

Multiple executions

As with any Maven plugin, jOOQ's maven code generation plugin can be executed multiple times, inheriting the common configuration in each execution:

```
<plugin>
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>

  <!-- Common configuration inherited by all executions -->
  <configuration>

    <!-- Log at WARN level by default -->
    <logging>WARN</logging>
    <generator>
      <generate>

        <!-- Never generate deprecated code -->
        <deprecated>false</deprecated>
      </generate>
      <database>
        <forcedTypes>

          <!-- Use BIGINT for all ID columns -->
          <forcedType>
            <name>BIGINT</name>
            <includeExpression>ID</includeExpression>
          </forcedType>
        </forcedTypes>
      </database>
    </generator>
  </configuration>

  <!-- Multiple executions with their own, specific configurations -->
  <executions>
    <execution>
      <id>generate-tenant-1</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        ...
      </configuration>
    <execution>

    <execution>
      <id>generate-tenant-2</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        ...
      </configuration>
    <execution>
  </executions>
</plugin>
```

Please refer to the Maven documentation for more details.

5.9.2. Running the code generator with Ant

Run generation with Ant

When running code generation with ant's <java/> task, you may have to set fork="true":

```
<!-- Run the code generation task -->
<target name="generate-test-classes">
  <java fork="true"
        classname="org.jooq.codegen.GenerationTool">
    <arg value="/path/to/configuration.xml"/>
    <classpath>
      <pathelement location="/path/to/jooq-3.12.4.jar"/>
      <pathelement location="/path/to/jooq-meta-3.12.4.jar"/>
      <pathelement location="/path/to/jooq-codegen-3.12.4.jar"/>
      <!-- Add JDBC drivers and other required artifacts -->
    </classpath>
  </java>
</target>
```

Using the Ant Maven plugin

Sometimes, ant can be useful to work around a limitation (misunderstanding?) of the Maven build. Just as with the above standalone ant usage example, the jOOQ code generator can be called from the maven-antrun-plugin:

```
<!-- Run the code generation task -->
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>

  <configuration>
    <tasks>
      <java fork="true"
            classname="org.jooq.codegen.GenerationTool"
            classpathref="maven.compile.classpath">
          <arg value="/path/to/configuration.xml"/>
      </java>
    </tasks>
  </configuration>

  <dependencies>
    <dependency>
      <!-- JDBC driver -->
    </dependency>
    <dependency>
      <!-- Use org.jooq              for the Open Source Edition
               org.jooq.pro          for commercial editions with Java 11 support,
               org.jooq.pro-java-8   for commercial editions with Java 8 support,
               org.jooq.pro-java-6   for commercial editions with Java 6 support,
               org.jooq.trial        for the free trial edition with Java 11 support,
               org.jooq.trial-java-8 for the free trial edition with Java 8 support,
               org.jooq.trial-java-6 for the free trial edition with Java 6 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
             See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

      <groupId>org.jooq</groupId>
      <artifactId>jooq-codegen</artifactId>
      <version>3.12.4</version>
    </dependency>
    <!-- Add JDBC drivers and other required artifacts -->
  </dependencies>
</plugin>
```

5.9.3. Running the code generator with Gradle

Run generation with the Gradle plugin

Historically, we used to recommend using the Gradle plugin by Etienne Studer (from Gradle Inc.). It provides a concise DSL that allows you to tune all configuration properties supported by each jOOQ

version. Please direct any support questions or issues you may find directly to the third party plugin vendor.

Consider also the various examples provided there: https://github.com/etiennestuder/gradle-jooq-plugin/tree/master/example

> *(!) When upgrading to jOOQ 3.19, please consider using the out of the box gradle plugin support, instead.*

Alternatively, programmatic configuration can be used

If you don't want to use the above plugin approach, there's also the possibility to use jOOQ's standalone code generator for simplicity. The following working example build.gradle script should work out of the box:

```
// Configure the Java plugin and the dependencies
// ----------------------------------------------
apply plugin: 'java'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    // Use org.jooq                for the Open Source Edition
    //     org.jooq.pro            for commercial editions with Java 17 support,
    //     org.jooq.pro-java-8     for commercial editions with Java 8 support,
    //     org.jooq.pro-java-6     for commercial editions with Java 6 support,
    //     org.jooq.trial          for the free trial edition with Java 17 support,
    //     org.jooq.trial-java-8   for the free trial edition with Java 8 support,
    //     org.jooq.trial-java-6   for the free trial edition with Java 6 support
    //
    // Note: Only the Open Source Edition is hosted on Maven Central.
    //       Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
    //       See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk

    compile 'org.jooq:jooq:3.12.4'

    runtime 'com.h2database:h2:1.4.200'
    testCompile 'junit:junit:4.11'
}

buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        /* See above for the correct groupId */
        classpath 'org.jooq:jooq-codegen:3.12.4'
        classpath 'com.h2database:h2:200'
    }
}

import org.jooq.codegen.GenerationTool
import org.jooq.meta.jaxb.*

GenerationTool.generate(new Configuration()
    .withJdbc(new Jdbc()
        .withDriver('org.h2.Driver')
        .withUrl('jdbc:h2:~/test-gradle')
        .withUser('sa')
        .withPassword(''))
    .withGenerator(new Generator()
        .withDatabase(new Database())
        .withGenerate(new Generate()
            .withPojos(true)
            .withDaos(true))
        .withTarget(new Target()
            .withPackageName('org.jooq.example.gradle.db')
            .withDirectory('src/main/java'))))
```

5.9.4. Programmatic configuration and execution

Configuring your code generator with Java, Groovy, etc.

In the previous sections, we have covered how to set up jOOQ's code generator using XML, either by running a standalone Java application, or by using Maven. However, it is also possible to use jOOQ's GenerationTool programmatically. The XSD file used for the configuration (https://www.jooq.org/xsd/jooq-codegen-3.12.0.xsd) is processed using XJC to produce Java artefacts. The below example uses those artefacts to produce the equivalent configuration of the previous PostgreSQL / Maven example:

```
// Use the fluent-style API to construct the code generator configuration
import org.jooq.meta.jaxb.*;
import org.jooq.meta.jaxb.Configuration;

// [...]

Configuration configuration = new Configuration()
    .withJdbc(new Jdbc()
        .withDriver("org.postgresql.Driver")
        .withUrl("jdbc:postgresql:postgres")
        .withUser("postgres")
        .withPassword("test"))
    .withGenerator(new Generator()
        .withDatabase(new Database()
            .withName("org.jooq.meta.postgres.PostgresDatabase")
            .withIncludes(".*")
            .withExcludes("")
            .withInputSchema("public"))
        .withTarget(new Target()
            .withPackageName("org.jooq.codegen.maven.example")
            .withDirectory("target/generated-sources/jooq")));

GenerationTool.generate(configuration);
```

For the above example, you will need all of jooq-3.12.4.jar, jooq-meta-3.12.4.jar, and jooq-codegen-3.12.4.jar, on your classpath.

Manually loading the XML file

Alternatively, you can also load parts of the configuration from an XML file using JAXB and programmatically modify other parts using the code generation API:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
    <jdbc>
        <driver>org.h2.Driver</driver>
        <!-- ... -->
    </jdbc>
</configuration>
```

Load the above using standard JAXB API:

```
import java.io.File;
import javax.xml.bind.JAXB;
import org.jooq.meta.jaxb.Configuration;

// [...]
// and then

Configuration configuration = JAXB.unmarshal(new File("jooq.xml"), Configuration.class);
configuration.getJdbc()
             .withUser("username")
             .withPassword("password");

GenerationTool.generate(configuration);
```

... and then, modify parts of your configuration programmatically, for instance the JDBC user / password:

5.10. System properties governing code generation

Regardless if you're using a standalone code generation configuration, or if you're generating code with Maven, ant, or gradle, you can always provide default values for certain configuration elements through the following system properties:

- -Djooq.codegen.configurationFile (path): Specify an external configuration file, rather than using the inline configuration, e.g. in Maven
- -Djooq.codegen.jdbc.driver (class name): The JDBC driver to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.url (url): The JDBC URL to use for JDBC connection based code generation (the name of this property can be overridden with the urlProperty value)
- -Djooq.codegen.jdbc.user (string): The JDBC user name to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.username (string, same as user): The JDBC user name to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.password (string): The JDBC password to use for JDBC connection based code generation
- -Djooq.codegen.logging (TRACE, DEBUG, INFO, WARN, ERROR, FATAL): The log level to use
- -Djooq.codegen.skip (boolean): Allows for skipping the execution of jOOQ code generation. Useful for larger builds, e.g. with Maven

In case of conflict between the above default value and a more concrete, local configuration, the latter prevails and the default is overridden.

5.11. Code generation dependencies

Various code generation features allow for providing the code generator with a custom implementation of something, like:

- The generator: The logic generating your code
- The database: The source for database meta data source
- The generator strategy: The strategy defining generated object names and identifiers
- Much more

Other use-cases require a classpath dependency to look up things like JPA entities.

Many of these custom implementation hooks come in two flavours:

- Using Java in-memory compilation of the implementation.
- Using a named reference to a previously compiled implementation.

Depending on the above approach, the following section explains how to set up your code generation dependencies. In both cases, any dependencies to the code generator need to be declared in your build:

Maven

```
<plugin>
  <!-- Specify the maven code generator plugin -->
  <!-- Use org.jooq              for the Open Source Edition
           org.jooq.pro          for commercial editions with Java 11 support,
           org.jooq.pro-java-8   for commercial editions with Java 8 support,
           org.jooq.pro-java-6   for commercial editions with Java 6 support,
           org.jooq.trial        for the free trial edition with Java 11 support,
           org.jooq.trial-java-8 for the free trial edition with Java 8 support,
           org.jooq.trial-java-6 for the free trial edition with Java 6 support

   Note: Only the Open Source Edition is hosted on Maven Central.
         Install the others locally using the provided scripts, or access them from here: https://repo.jooq.org
         See the JDK version support matrix here: https://www.jooq.org/download/support-matrix-jdk -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.12.4</version>

  <executions>...</executions>
  <configuration>...</configuration>

  <dependencies>
    <dependency>...</dependency>
  </dependencies>
</plugin>
```

Gradle (Kotlin)

```
dependencies {
    jooqCodegen("...")
}
```

Gradle (Groovy)

Using in-memory compilation

When using in-memory compilation, everything that is available to jOOQ's code generator is available to your in-memory compiled implementation of the extension point. You do not have to do anything else than declare those dependencies.

Using a named reference

A named reference of a custom implementation needs to be on the classpath of the code in a pre-compiled form (unlike the previously mentioned in-memory compiled form). In both Maven and Gradle, this is typically best done by setting up a multi modular project of this form:

```
                +-----------------------------------+
                | Your custom code generation logic |
                +-----------------------------------+
                             ^         ^
                  depends on |         | may or may not depend on
                             |         |
          +---------------------+   +---------------------+
          | jOOQ codegen plugin |   | Your application    |
          +---------------------+   +---------------------+
                             |         |
                   generates |         | depends on
                             v         v
                    +-------------------------+
                    | jOOQ generated classes  |
                    +-------------------------+
```

It is important to understand that your custom code generation logic has to be compiled (and packaged, etc.) ***before*** you use it in your code generation logic. Think of your custom code generation logic like any library provided by a third party, available from Maven Central or elsewhere. It's a dependency like any other.

5.12. Code generation for large schemas

Databases can become very large in real-world applications. This is not a problem for jOOQ's code generator, but it can be for the Java compiler. jOOQ generates some classes for global access. These classes can hit two sorts of limits of the compiler / JVM:

- Methods (including static / instance initialisers) are allowed to contain only 64kb of bytecode.
- Classes are allowed to contain at most 64k of constant literals

While there exist workarounds for the above two limitations (delegating initialisations to nested classes, inheriting constant literals from implemented interfaces), the preferred approach is either one of these:

- Distribute your database objects in several schemas. That is probably a good idea anyway for such large databases
- Configure jOOQ's code generator to exclude excess database objects
- Configure jOOQ's code generator to avoid generating global objects using <globalObjectReferences/>
- Remove uncompilable classes after code generation

5.13. Code generation and version control

When using jOOQ's code generation capabilities, you will need to make a strategic decision about whether you consider your generated code as

- Part of your code base
- Derived artefacts

In this section we'll see that both approaches have their merits and that none of them is clearly better.

Part of your code base

When you consider generated code as part of your code base, you will want to:

- Check in generated sources in your version control system
- Use manual source code generation
- Possibly use even partial source code generation

This approach is particularly useful when your Java developers are not in full control of or do not have full access to your database schema, or if you have many developers that work simultaneously on the same database schema, which changes all the time. It is also useful to be able to track side-effects of database changes, as your checked-in database schema can be considered when you want to analyse the history of your schema.

With this approach, you can also keep track of the change of behaviour in the jOOQ code generator, e.g. when upgrading jOOQ, or when modifying the code generation configuration.

© 2009 - 2025 by Data Geekery™ GmbH.

The drawback of this approach is that it is more error-prone as the actual schema may go out of sync with the generated schema.

Derived artefacts

When you consider generated code to be derived artefacts, you will want to:

- Check in only the actual DDL (e.g. controlled via Flyway)
- Regenerate jOOQ code every time the schema changes
- Regenerate jOOQ code on every machine - including continuous integration

This approach is particularly useful when you have a smaller database schema that is under full control by your Java developers, who want to profit from the increased quality of being able to regenerate all derived artefacts in every step of your build.

The drawback of this approach is that the build may break in perfectly acceptable situations, when parts of your database are temporarily unavailable.

Pragmatic combination

In some situations, you may want to choose a pragmatic combination, where you put only some parts of the generated code under version control. For example, you may want to version control your POJOs, but not your tables / records. This can be achieved in multiple ways:

- You could use a generator strategy to relocate your POJOs to a different package, making excluding one or the other artifact using e.g. .gitignore much simpler
- You could run the code generator twice, once for POJOs only, and once for the rest.

5.14. Features requiring generated code

There are numerous features in jOOQ that require the use of generated code, which is why we always recommend using generated code by default. The main and only reason why you wouldn't use the code generator is because your schema is dynamic and known only at runtime.

However, even when you use the jOOQ code generator, there are occasions where you may need to bypass it by using plain SQL or Name based Field and Table types. In those cases, you will not benefit from the following features either (the features are effectively bypassed). Such features include:

- Calling stored procedures and functions
- Binding user defined types
- CRUD with UpdatableRecords
- Implicit JOIN paths
- INSERT RETURNING, UPDATE RETURNING, and DELETE RETURNING emulations need access to constraint meta data.
- Forced types are a code generation feature that is attached to a org.jooq.DataType.
- Render mapping is a runtime feature, depending on org.jooq.Catalog, org.jooq.Schema, or org.jooq.Table equality.

© 2009 - 2025 by Data Geekery™ GmbH.

*(!) Note that it is possible to use some of the above features even with plain SQL if you copy a **org.jooq.DataType** from generated code to your template, although, you will have to remember this step every time you create such a template.*

6. Coming from JPA

If you've written most of your Java persistence logic using JPA, then the following sections are for you.

In the following sections, we'll explain the main conceptual differences between JPA and jOOQ, as well as show how individual JPA features are best mapped to jOOQ or SQL features.

6.1. Set based thinking

Many conceptual differences that you may encounter between using JPA and jOOQ are not technology specific, but a matter of how you think about your database interactions. Neither approach is "the best" one, both approaches are each better suited to certain use-cases. The approaches being discussed here are:

- Working with entity state transitions (where JPA shines)
- Working with data set transformations (where jOOQ / SQL shine)

A typical application will have both of the above problems, maybe one type of problem more than the other. It is important to recognise that if you do have both problems, using both types of technology next to each other is a perfectly fine option! Using jOOQ and using JPA aren't mutually exclusive.

Having said so, jOOQ is used most efficiently when following the SQL paradigm of set based thinking. I.e. don't do this:

```
FOR rec IN (SELECT id FROM book WHERE title LIKE 'A%') LOOP
  UPDATE book
  SET last_update = CURRENT_TIMESTAMP
  WHERE book.id = rec.id;
END LOOP;
```

But do this, instead:

```
UPDATE book
SET last_update = CURRENT_TIMESTAMP
WHERE title LIKE 'A%';
```

The example deliberately didn't use any Java or even JPA code to show the idea that set based thinking isn't strictly related to writing SQL. The first example exposes the N+1 problem via PL/SQL code. It may be perfectly fine to implement entity state transitions on an individual per-row basis in some cases, but in a lot of cases, it's much better to treat your data as data sets and run bulk queries.

Getting a deep understanding of this distinction is out of scope for this reference manual. It takes a lot of practice to do both approaches. But it's important to be reminded of this distinction when you want to switch from JPA to jOOQ. jOOQ embraces the SQL paradigm of set based thinking, and so should you, when you want to use jOOQ most effectively.

© 2009 - 2025 by Data Geekery™ GmbH.

6.2. Database first

In principle, JPA is agnostic of an entity model first or DDL first / database first approach. In principle, you can:

- Write DDL (e.g. migrate it with Flyway or Liquibase), then re-generate your entity code.
- Write entities, then re-generate your database or derive DDL increments from your git history, etc.
- Manually maintain both in parallel.

jOOQ is more opinionated here. It assumes your database already exists, outside of jOOQ. It does not assume your database follows any rules imposed by jOOQ regarding the design. For example, when using jOOQ, it's completely irrelevant:

- If you're using surrogate keys or natural keys
- If your keys have single columns or multiple columns
- If your tables even have keys, or if they're not in the first normal form
- If your tables are even tables, or if they're views, table valued functions, etc

Not just that jOOQ doesn't care about your database model, it also doesn't care about your client side data representation (e.g. the DTOs). I.e. it doesn't care:

- If your DTO classes and attributes are public or private
- If your DTOs are immutable or mutable
- If your DTOs implement equals() or hashCode() in any way
- If your DTOs have a meaningful identity, or are just value based classes (or soon even actual value types!)

There are no best practices from a jOOQ perspective. jOOQ has seen everything, and jOOQ won't judge you. jOOQ knows, that if your tables are not even in the first normal form, you'll have enough problems already, so you don't also need a problem with jOOQ.

But

But, again, jOOQ expects you to have a database that already exists. Yes, you can even use jOOQ for your DDL, but the point here is that you write DDL first, *then* your jOOQ queries later, ideally using source code generation.

Client applications come and go. They used to be written in Delphi. Then PHP. Then Java. Now Kotlin. Soon TypeScript/Rust/Go/Whatever? But your database stays. It will survive your client application, and as such, jOOQ understands that your database must come first.

Obviously, there exist other ways to think about business logic and databases. But jOOQ was designed for the database first model.

© 2009 - 2025 by Data Geekery™ GmbH.

6.3. Eager or lazy loading

In jOOQ, you always load data explicitly, never automatically via eager or lazy loading, both of which seem useful at first, but come at a heavy price, or say, complexity tax.

- Eager loading prevents extra round trips when you ***need*** the data, but cannot be easily prevented when you ***do not need*** the data, so chances are, eager loading is generating too much unnecessary effort
- Lazy loading prevents unnecessary loading when you ***don't need*** the data, but generates delayed extra work (the dreaded N+1 problem) when you do need the data

In the JPA ecosystem, a lot of folks advocate using queries to avoid both of the above problems. That's why in jOOQ, the above problems don't exist, because you will ***always*** spell out the exact query that you need to produce the data you want.

6.4. First level cache and second level cache

In the jOOQ world, a first or second level cache is not needed, because it fundamentally opposes the idea that you interact with your database only using queries, not using entity graph navigation.

In the JPA world, this idea exists as well when you use ***(DTO) projections***, which cannot reasonably be cached using first and second level caches (at least not without moving the query execution partially into the client).

As these caches solve problems that jOOQ doesn't have, because jOOQ doesn't have eager or lazy loading, nor does it operate on the entity graph, there is no need for such a cache.

Should you find the need for caching other types of data (e.g. your DTOs of some master data), then it should be quite trivial to use any off-the-shelf cache product and cache that data in the service layer directly. From a jOOQ perspective, the query layer is the wrong place to cache any data.

6.5. AttributeConverter

JPA knows the jakarta.persistence.AttributeConverter, which is called org.jooq.Converter in jOOQ. They work exactly the same way. See also the manual's section on how to attach a Converter to generated code.

6.6. User types

Some JPA implementations offer user types, which are called org.jooq.Binding in jOOQ. They work exactly the same way. See also the manual's section on how to attach a Binding to generated code.

© 2009 - 2025 by Data Geekery™ GmbH.

6.7. Implicit JOIN

One of HQL's greatest features is the implicit JOIN feature, which is generated from path expressions when you follow foreign key "paths" from child entities to parent entities. The same feature is available in jOOQ as well, if you're using code generation.

In HQL, you might write:

```
from Book as book
where book.language.cd = 'en'
```

And in jOOQ, this just translates to:

```
create.selectFrom(BOOK)
      .where(BOOK.language().CD.eq("en"))
      .fetch();
```

Some of the jOOQ features used in this section are:

- implicit joins (these are optional to the task of nesting, but greatly simplify it)

7. Reference

These chapters hold some general jOOQ reference information

7.1. Supported RDBMS

A list of supported databases

- Aurora MySQL Edition
- Aurora PostgreSQL Edition
- Azure SQL Data Warehouse (Azure Synapse Analytics)
- Azure SQL Database
- DB2 LUW
- Derby
- Firebird
- H2
- HANA
- HSQLDB
- Informix
- MariaDB
- Microsoft Access
- MySQL
- Oracle
- PostgreSQL
- Redshift
- SQL Server
- SQLite
- SingleStore (MemSQL)
- Sybase Adaptive Server Enterprise
- Sybase SQL Anywhere
- Teradata
- Vertica

For an up-to-date list of currently supported RDBMS and minimal versions, please refer to https://www.jooq.org/legal/licensing/#databases and https://www.jooq.org/download/support-matrix.

7.2. Commercial only features

The commercial jOOQ editions give access to drivers for certain commercial RDBMS as well as to commercial only features, which are not available in the jOOQ Open Source Edition. These features mostly involve:

- Procedural language support (procedures, functions, triggers, anonymous blocks)

The below list of sections documents all the sections about commercial only features in jOOQ:

- SQL building
  * SQL Statements (DML)
    o The SELECT statement
      o CONNECT BY clause
  * Procedural statements
    o CONTINUE statement
    o EXECUTE statement
    o EXIT statement
    o FOR statement
    o GOTO statement
    o IF statement
    o Labels
    o LOOP statement
    o REPEAT statement
    o Variables
    o WHILE statement
  * Column expressions
    o Aggregate functions
      o COLLECT
- Reference
  * Commercial only features

See also the Feature comparison for a high-level description.

7.3. Experimental features

To improve our development process and get more early adopter feedback, we've started publishing experimental functionality with our ordinary distribution. All such functionality is clearly documented as experimental either in the manual, and/or annotated with the @Experimental annotation in the API to make it clear that usage is done at the user's own risk as the API might still change in the future.

© 2009 - 2025 by Data Geekery™ GmbH.

The below list of sections documents all the sections about experimental features in jOOQ:

- This version of jOOQ does not have any experimental features.

7.4. Built-in data types

There is always a small mismatch between SQL data types and Java data types. This is for two reasons:

- SQL data types are insufficiently covered by the JDBC API.
- Java data types are often less expressive than SQL data types

This chapter should document the most important notes about SQL, JDBC and jOOQ data types.

7.4.1. BLOBs and CLOBs

jOOQ aims for hiding all JDBC details from jOOQ client API. Specifically, java.sql.Clob and java.sql.Blob are quite "harsh" APIs with a few caveats that may even depend on JDBC driver specifics.

Clob and Blob are resources (but not java.lang.AutoCloseable!) with open connections to the database. This makes no sense in an ordinary jOOQ context, when eagerly fetching all the results through fetch() methods. fetchLazy() and fetchStream() might be candidates where Clob and Blob types could make sense as the underlying java.sql.ResultSet and java.sql.PreparedStatement are still open while consuming these resources.

ByteArrayInputStream and ByteArrayOutputStream on the other hand are two different types which cannot be represented as a single Field<T> type. If either would be chosen as the <T> type, we'd get read-only or write-only fields. So for full lazy streaming support, we'd need another 2-way wrapper type, similar to Clob and Blob.

In many cases, streaming binary data isn't really necessary as thebyte[] can be easily kept in memory (and it is done so for further processing anyway, e.g. when working with images), so the extra work might not really be needed. This is particularly true in Oracle, where BLOBs are the only binary types in the absences of a formal (VAR)BINARY type, and CLOBs start at 4000 bytes.

If you want to override jOOQ's default behaviour, use either org.jooq.impl.ClobBinding or org.jooq.impl.BlobBinding, see custom data type bindings as well as using a binding in code generation for more details.

7.4.2. BOOLEAN data type

The SQL standard and a lot of RDBMS support a BOOLEAN data type with values TRUE, FALSE, and NULL. All databases support this data type in the form of an org.jooq.Condition, a condition or predicate that can be placed in the WHERE clause, among many other places. But true SQL BOOLEAN data type support means that the data type can be used everywhere a column expression can be used, including the WHERE clause.

If the BOOLEAN data type is not natively supported, JDBC and most databases translate it to 1 or '1' (TRUE), 0 or '0' (FALSE), and NULL by convention, although other translations may be possible, including 'Y' / 'N', 'T' / 'F', 'TRUE' / 'FALSE', and many more.

jOOQ, by default, follows the most popular convention and translates to 1/0.

See the manual's section about BOOLEAN columns for details on how to use the BOOLEAN data type as a conditional expression.

7.4.3. Unsigned integer types

Some databases explicitly support unsigned integer data types. In most normal JDBC-based applications, they would just be mapped to their signed counterparts letting bit-wise shifting and tweaking to the user. jOOQ ships with a set of unsigned java.lang.Number implementations modelling the following types:

- org.jooq.types.UByte: Unsigned byte, an 8-bit unsigned integer
- org.jooq.types.UShort: Unsigned short, a 16-bit unsigned integer
- org.jooq.types.UInteger: Unsigned int, a 32-bit unsigned integer
- org.jooq.types.ULong: Unsigned long, a 64-bit unsigned integer

Each of these wrapper types extends java.lang.Number, wrapping a higher-level integer type, internally:

- UByte wraps java.lang.Short
- UShort wraps java.lang.Integer
- UInteger wraps java.lang.Long
- ULong wraps java.math.BigInteger

7.4.4. INTERVAL data types

jOOQ fills a gap opened by JDBC, which neglects an important SQL data type as defined by the SQL standards: INTERVAL types. SQL knows two different types of intervals:

- YEAR TO MONTH: This interval type models a number of months and years
- DAY TO SECOND: This interval type models a number of days, hours, minutes, seconds and milliseconds

Both interval types ship with a variant of subtypes, such as DAY TO HOUR, HOUR TO SECOND, etc. jOOQ models these types as Java objects extending java.lang.Number: org.jooq.types.YearToMonth (where Number.intValue() corresponds to the absolute number of months) and org.jooq.types.DayToSecond (where Number.intValue() corresponds to the absolute number of milliseconds)

Interval arithmetic

In addition to the arithmetic expressions documented previously, interval arithmetic is also supported by jOOQ. Essentially, the following operations are supported:

- DATETIME - DATETIME => INTERVAL
- DATETIME + or - INTERVAL => DATETIME
- INTERVAL + DATETIME => DATETIME
- INTERVAL + - INTERVAL => INTERVAL
- INTERVAL * or / NUMERIC => INTERVAL
- NUMERIC * INTERVAL => INTERVAL

7.4.5. JSON data types

jOOQ supports non-JDBC but standard SQL JSON data types, which come in two forms:

- org.jooq.JSON: A text representation of a JSON document, which typically might include formatting.
- org.jooq.JSONB: A binary representation of a JSON document, which doesn't maintain formatting or object key ordering.

7.4.6. CURSOR data types

Some databases support cursors returned from stored procedures. They are mapped to the following jOOQ data type:

```
Field<Result<Record>> cursor;
```

In fact, such a cursor will be fetched immediately by jOOQ and wrapped in an org.jooq.Result object.

7.4.7. ARRAY and TABLE data types

The SQL standard specifies ARRAY data types, that can be mapped to Java arrays as such:

```
Field<Integer[]> intArray;
```

The above array type is supported by at least these SQL dialects:

- H2
- HANA
- HSQLDB
- Postgres

Oracle typed arrays

Oracle has strongly-typed arrays and table types (as opposed to the previously seen anonymously typed arrays). These arrays are wrapped by org.jooq.ArrayRecord types.

7.4.8. Oracle DATE data type

Oracle's DATE data type does not conform to the SQL standard. It is really a TIMESTAMP(0), i.e. a TIMESTAMP with a fractional second precision of zero. The most appropriate JDBC type for Oracle DATE types is java.sql.Timestamp.

Performance implications

When binding TIMESTAMP variables to SQL statements, instead of truncating such variables to DATE, the cost based optimiser may choose to widen the database column from DATE to TIMESTAMP using an Oracle INTERNAL_FUNCTION(), which prevents index usage. Details about this behaviour can be seen in this Stack Overflow question.

Use a data type binding to work around this issue

The best way to work around this issue is to implement a custom data type binding, which generates the CAST expression for every bind variable:

```
@Override
public final void sql(BindingSQLContext<Timestamp> ctx) throws SQLException {
    ctx.render()
       .visit(keyword("cast")).sql('(')
       .visit(val(ctx.value())).sql(' ')
       .visit(keyword("as date")).sql(')');
}
```

Deprecated functionality

Historic versions of jOOQ used to support a <dateAsTimestamp/> flag, which can be used with the out-of-the-box org.jooq.impl.DateAsTimestampBinding as a custom data type binding:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <!-- Use this flag to force DATE columns to be of type TIMESTAMP. This has been deprecated. -->
      <dateAsTimestamp>true</dateAsTimestamp>

      <!-- Define a custom binding for such DATE as TIMESTAMP columns -->
      <forcedTypes>
        <forcedType>

          <userType>java.sql.Timestamp</userType>
          <binding>org.jooq.impl.DateAsTimestampBinding</binding>
          <includeTypes>DATE</includeTypes>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.

Programmatic

```
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()

      // Use this flag to force DATE columns to be of type TIMESTAMP. This has been deprecated.
      .withDateAsTimestamp(true)

      // Define a custom binding for such DATE as TIMESTAMP columns
      .withForcedTypes(
        new ForcedType()
          .withUserType("java.sql.Timestamp")
          .withBinding("org.jooq.impl.DateAsTimestampBinding")
          .withIncludeTypes("DATE")
      )
    )
  )
```

See the configuration XSD and programmatic code generation for more details.
Gradle (Kotlin)

```
// The jOOQ-codegen-gradle plugin has been introduced in version 3.19 only.
```

Gradle (Groovy)
Gradle (third party)

```
generationTool {
  generator {
    database {

      // Use this flag to force DATE columns to be of type TIMESTAMP. This has been deprecated.
      dateAsTimestamp = true

      // Define a custom binding for such DATE as TIMESTAMP columns
      forcedTypes {
        forcedType {
          userType = "java.sql.Timestamp"
          binding = "org.jooq.impl.DateAsTimestampBinding"
          includeTypes = "DATE"
        }
      }
    }
  }
}
```

See the configuration XSD and gradle code generation for more details.
For more information, please refer to the manual's section about custom data type bindings and forced types.

7.5. SQL to DSL mapping rules

jOOQ takes SQL as an external domain-specific language and maps it onto Java, creating an internal domain-specific language. Internal DSLs cannot 100% implement their external language counter parts, as they have to adhere to the syntax rules of their host or target language (i.e. Java). This section explains the various problems and workarounds encountered and implemented in jOOQ.

SQL allows for "keywordless" syntax

SQL syntax does not always need keywords to form expressions. The UPDATE .. SET clause takes various argument assignments:

```
UPDATE t SET a = 1, b = 2
```

```
update(t).set(a, 1).set(b, 2)
```

The above example also shows missing operator overloading capabilities, where "=" is replaced by "," in jOOQ. Another example are row value expressions, which can be formed with parentheses only in SQL:

```
(a, b) IN ((1, 2), (3, 4))
```

```
row(a, b).in(row(1, 2), row(3, 4))
```

In this case, ROW is an actual (optional) SQL keyword, implemented by at least PostgreSQL.

SQL contains "composed" keywords

As most languages, SQL does not attribute any meaning to whitespace. However, whitespace is important when forming "composed" keywords, i.e. SQL clauses composed of several keywords. jOOQ follows standard Java method naming conventions to map SQL keywords (case-insensitive) to Java methods (case-sensitive, camel-cased). Some examples:

```
GROUP BY
ORDER BY
WHEN MATCHED THEN UPDATE
```

```
groupBy()
orderBy()
whenMatchedThenUpdate()
```

Future versions of jOOQ may use all-uppercased method names in addition to the camel-cased ones (to prevent collisions with Java keywords):

```
GROUP BY
ORDER BY
WHEN MATCHED THEN UPDATE
```

```
GROUP_BY()
ORDER_BY()
WHEN_MATCHED_THEN_UPDATE()
```

SQL contains "superfluous" keywords

Some SQL keywords aren't really necessary. They are just part of a keyword-rich language, the way Java developers aren't used to anymore. These keywords date from times when languages such as ADA, BASIC, COBOL, FORTRAN, PASCAL were more verbose:

- BEGIN .. END
- REPEAT .. UNTIL
- IF .. THEN .. ELSE .. END IF

jOOQ omits some of those keywords when it is too tedious to write them in Java.

```
CASE WHEN .. THEN .. END
```

```
decode().when(.., ..)
```

The above example omits THEN and END keywords in Java. Future versions of jOOQ may comprise a more complete DSL, including such keywords again though, to provide a more 1:1 match for the SQL language.

SQL contains "superfluous" syntactic elements

Some SQL constructs are hard to map to Java, but they are also not really necessary. SQL often expects syntactic parentheses where they wouldn't really be needed, or where they feel slightly inconsistent with the rest of the SQL language.

© 2009 - 2025 by Data Geekery™ GmbH.

```
LISTAGG(a, b) WITHIN GROUP (ORDER BY c)
              OVER (PARTITION BY d)
```

```
listagg(a, b).withinGroupOrderBy(c)
             .over().partitionBy(d)
```

The parentheses used for the WITHIN GROUP (..) and OVER (..) clauses are required in SQL but do not seem to add any immediate value. In some cases, jOOQ omits them, although the above might be optionally re-phrased in the future to form a more SQLesque experience:

```
LISTAGG(a, b) WITHIN GROUP (ORDER BY c)
              OVER (PARTITION BY d)
```

```
listagg(a, b).withinGroup(orderBy(c))
             .over(partitionBy(d))
```

SQL uses some of Java's reserved words

Some SQL keywords map onto Java Language Keywords if they're mapped using camel-casing. These keywords currently include:

- CASE
- ELSE
- FOR

jOOQ uses a suffix on those keywords to prevent a collision:

```
CASE .. ELSE
PIVOT .. FOR .. IN ..
```

```
case_() .. else_()
pivot(..).for_(..).in(..)
```

There is more future collision potential with, each resolved with a suffix:

- BOOLEAN
- CHAR
- DEFAULT
- DOUBLE
- ENUM
- FLOAT
- IF
- INT
- LONG
- PACKAGE

SQL operators cannot be overloaded in Java

Most SQL operators have to be mapped to descriptive method names in Java, as Java does not allow operator overloading:

```
=
<>, !=
||
SET a = b
```

```
equal(), eq()
notEqual(), ne()
concat()
set(a, b)
```

For those users using jOOQ with Scala or Groovy, operator overloading and implicit conversion can be leveraged to enhance jOOQ:

```
=
<>, !=
||
```

```
===
<>, !==
||
```

SQL's reference before declaration capability

This is less of a syntactic SQL feature than a semantic one. In SQL, objects can be referenced before (i.e. "lexicographically before") they are declared. This is particularly true for aliasing

```
SELECT t.a
FROM my_table t
```

```
MyTable t = MY_TABLE.as("t");
select(t.a).from(t)
```

A more sophisticated example are common table expressions (CTE), which are currently not supported by jOOQ:

```
WITH t(a, b) AS (
  SELECT 1, 2 FROM DUAL
)
SELECT t.a, t.b
FROM t
```

Common table expressions define a "derived column list", just like table aliases can do. The formal record type thus created cannot be typesafely verified by the Java compiler, i.e. it is not possible to formally dereference t.a from t.

7.6. Quality Assurance

jOOQ is running some of your most mission-critical logic: the interface layer between your Java / Scala application and the database. You have probably chosen jOOQ for any of the following reasons:

- To evade JDBC's verbosity and error-proneness due to string concatenation and index-based variable binding
- To add lots of type-safety to your inline SQL
- To increase productivity when writing inline SQL using your favourite IDE's autocompletion capabilities

With jOOQ being in the core of your application, you want to be sure that you can trust jOOQ. That is why jOOQ is heavily unit and integration tested with a strong focus on integration tests:

Unit tests

Unit tests are performed against dummy JDBC interfaces using mocks, including our own JDBC mocking abstraction. These tests verify that various org.jooq.QueryPart implementations render correct SQL and bind variables correctly.

In addition to the above, numerous types of unit tests include

- SQL formatting tests
- JSON and XML data import / export tests
- Parser tests
- Mapper and converter tests
- SPI tests
- SQL transformation tests
- And much more

Integration tests

This is the most important part of the jOOQ test suites. 10k+ queries are currently run against a standard integration test database, for which we use testcontainers and other virtualisation utilities. Both the test database and the queries are translated into every one of the 30+ supported SQL dialects to ensure that regressions are unlikely to be introduced into the code base.

For libraries like jOOQ, integration tests are much more expressive than unit tests, as there are so many subtle differences in SQL dialects. Simple mocks just don't give as much feedback as an actual database instance.

jOOQ integration tests run the weirdest and most unrealistic queries. As a side-effect of these extensive integration test suites, many corner-case bugs for JDBC drivers and/or open source databases have been discovered, feature requests submitted through jOOQ and reported mainly to Derby, H2, HSQLDB.

Code generation tests

For every one of the 30+ supported integration test databases, source code is generated and the tiniest differences in generated source code can be discovered. In case of compilation errors in generated source code, new test tables/views/columns are added to avoid regressions in this field.

API Usability tests and proofs of concept

jOOQ is used in jOOQ-meta as a proof of concept. This includes complex queries such as the following Postgres query

```
Routines r1 = ROUTINES.as("r1");
Routines r2 = ROUTINES.as("r2");

for (Record record : create.select(
        r1.ROUTINE_SCHEMA,
        r1.ROUTINE_NAME,
        r1.SPECIFIC_NAME,

        // Ignore the data type when there is at least one out parameter
        DSL.when(exists(
                selectOne()
                .from(PARAMETERS)
                .where(PARAMETERS.SPECIFIC_SCHEMA.eq(r1.SPECIFIC_SCHEMA))
                .and(PARAMETERS.SPECIFIC_NAME.eq(r1.SPECIFIC_NAME))
                .and(upper(PARAMETERS.PARAMETER_MODE).ne("IN"))),
                    val("void"))
           .else_(r1.DATA_TYPE).as("data_type"),
        r1.CHARACTER_MAXIMUM_LENGTH,
        r1.NUMERIC_PRECISION,
        r1.NUMERIC_SCALE,
        r1.TYPE_UDT_NAME,

        // Calculate overload index if applicable
        DSL.when(
            exists(
                selectOne()
                .from(r2)
                .where(r2.ROUTINE_SCHEMA.in(getInputSchemata()))
                .and(r2.ROUTINE_SCHEMA.eq(r1.ROUTINE_SCHEMA))
                .and(r2.ROUTINE_NAME.eq(r1.ROUTINE_NAME))
                .and(r2.SPECIFIC_NAME.ne(r1.SPECIFIC_NAME))),
            select(count())
                .from(r2)
                .where(r2.ROUTINE_SCHEMA.in(getInputSchemata()))
                .and(r2.ROUTINE_SCHEMA.eq(r1.ROUTINE_SCHEMA))
                .and(r2.ROUTINE_NAME.eq(r1.ROUTINE_NAME))
                .and(r2.SPECIFIC_NAME.le(r1.SPECIFIC_NAME)).asField())
        .as("overload"))
    .from(r1)
    .where(r1.ROUTINE_SCHEMA.in(getInputSchemata()))
    .orderBy(
        r1.ROUTINE_SCHEMA.asc(),
        r1.ROUTINE_NAME.asc())
    .fetch()) {

    result.add(new PostgresRoutineDefinition(this, record));
}
```

These rather complex queries show that the jOOQ API is fit for advanced SQL use-cases, compared to the rather simple, often unrealistic queries in the integration test suite.

Clean API and implementation. Code is kept DRY

As a general rule of thumb throughout the jOOQ code, everything is kept DRY. Some examples:

- There is only one place in the entire code base, which consumes values from a JDBC ResultSet
- There is only one place in the entire code base, which transforms jOOQ Records into custom POJOs

Keeping things DRY leads to longer stack traces, but in turn, also increases the relevance of highly reusable code-blocks. Chances that some parts of the jOOQ code base slips by integration test coverage decrease significantly.

7.7. Security

This section talks about a few common security related issues in jOOQ, which developers should be aware of.

7.7.1. SQL Injection

For most standard use-cases jOOQ is SQL injection safe because ordinary jOOQ usage does not involve concatenation of SQL strings. At the same time, every bit of user input is generated as a bind value in a java.sql.PreparedStatement, or escaped properly, if inlined explicitly (For more information, please refer to the section about SQL injection).

In order to completely forbid usage of API that could lead to SQL injection vulnerabilities in jOOQ (i.e. the plain SQL templating API), you can use a compiler plugin that prevents using such API.

7.7.2. Debug logging

For the convenience of the out-of-the-box jOOQ experience, jOOQ includes a lot of debug log information in its execute logging. Due to the sheer amount of logged data, users will unlikely keep this log around in production environments.

Another important reason to turn off this feature in production is the fact that Personally identifiable information (PII) can be contained in logged queries and logged result sets, including e.g. usernames, credit card numbers, etc. Logs may be distributed to other servers for technical analysis in case of production incidents, and thus give away such PII to third parties which shouldn't have access. This is also important from a GDPR compliance perspective.

7.7.3. Exception message

The org.jooq.exception.DataAccessException may contain the SQL string that has produced the exception for an improved debugging experience, including debugging in production. The assumption here is that the exception and its stack trace will never be disclosed to clients, including web browsers.

If the exception and SQL string is disclosed, then third parties may be able to deduce schema meta data information from the error (e.g. what tables and columns are available). While this may not be a significant problem by itself, if combined with another vulnerability (e.g. SQL Injection), this could help facilitate an attack.

7.7.4. Contact

In case you think you found a security issue, please look at https://github.com/jOOQ/jOOQ/security for information about how to get in touch.

7.8. Migrating to jOOQ 3.0

This section is for all users of jOOQ 2.x who wish to upgrade to the next major release. In the next sub-sections, the most important changes are explained. Some code hints are also added to help you fix compilation errors.

© 2009 - 2025 by Data Geekery™ GmbH.

Type-safe row value expressions

Support for row value expressions has been added in jOOQ 2.6. In jOOQ 3.0, many API parts were thoroughly (but often incompatibly) changed, in order to provide you with even more type-safety.

Here are some affected API parts:

- [N] in Row[N] has been raised from 8 to 22. This means that existing row value expressions with degree >= 9 are now type-safe
- Subqueries returned from DSL.select(...) now implement Select<Record[N]>, not Select<Record>
- IN predicates and comparison predicates taking subselects changed incompatibly
- INSERT and MERGE statements now take typesafe VALUES() clauses

Some hints related to row value expressions:

```
// SELECT statements are now more typesafe:
Record2<String, Integer> record         = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).where(ID.eq(1)).fetchOne();
Result<Record2<String, Integer>> result = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).fetch();

// But Record2 extends Record. You don't have to use the additional typesafety:
Record record    = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).where(ID.eq(1)).fetchOne();
Result<?> result = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).fetch();
```

SelectQuery and SelectXXXStep are now generic

In order to support type-safe row value expressions and type-safe Record[N] types, SelectQuery is now generic: SelectQuery<R>

SimpleSelectQuery and SimpleSelectXXXStep API were removed

The duplication of the SELECT API is no longer useful, now that SelectQuery and SelectXXXStep are generic.

Factory was split into DSL (query building) and DSLContext (query execution)

The pre-existing Factory class has been split into two parts:

- The DSL: This class contains only static factory methods. All QueryParts constructed from this class are "unattached", i.e. queries that are constructed through DSL cannot be executed immediately. This is useful for subqueries.
  The DSL class corresponds to the static part of the jOOQ 2.x Factory type
- The DSLContext: This type holds a reference to a Configuration and can construct executable ("attached") QueryParts.
  The DSLContext type corresponds to the non-static part of the jOOQ 2.x Factory / FactoryOperations type.

The FactoryOperations interface has been renamed to DSLContext. An example:

```
// jOOQ 2.6, check if there are any books
Factory create = new Factory(connection, dialect);
create.selectOne()
      .whereExists(
        create.selectFrom(BOOK) // Reuse the factory to create subselects
      ).fetch();                // Execute the "attached" query

// jOOQ 3.0
DSLContext create = DSL.using(connection, dialect);
create.selectOne()
      .whereExists(
        selectFrom(BOOK)        // Create a static subselect from the DSL
      ).fetch();                // Execute the "attached" query
```

Quantified comparison predicates

Field.equalAny(...) and similar methods have been removed in favour of Field.eq(any(...)). This greatly simplified the Field API. An example:

```
// jOOQ 2.6
Condition condition = BOOK.ID.equalAny(create.select(BOOK.ID).from(BOOK));

// jOOQ 3.0 adds some typesafety to comparison predicates involving quantified selects
QuantifiedSelect<Record1<Integer>> subselect = any(select(BOOK.ID).from(BOOK));
Condition condition = BOOK.ID.eq(subselect);
```

FieldProvider

The FieldProvider marker interface was removed. Its methods still exist on FieldProvider subtypes. Note, they have changed names from getField() to field() and from getIndex() to indexOf()

GroupField

GroupField has been introduced as a DSL marker interface to denote fields that can be passed to GROUP BY clauses. This includes all org.jooq.Field types. However, fields obtained from ROLLUP(), CUBE(), and GROUPING SETS() functions no longer implement Field. Instead, they only implement GroupField. An example:

```
// jOOQ 2.6
Field<?>   field1a = Factory.rollup(...); // OK
Field<?>   field2a = Factory.one();       // OK

// jOOQ 3.0
GroupField field1b = DSL.rollup(...); // OK
Field<?>   field1c = DSL.rollup(...); // Compilation error
GroupField field2b = DSL.one();       // OK
Field<?>   field2c = DSL.one();       // OK
```

NULL predicate

Beware! Previously, Field.eq(null) was translated internally to an IS NULL predicate. This is no longer the case. Binding Java "null" to a comparison predicate will result in a regular comparison predicate (which never returns true). This was changed for several reasons:

- To most users, this was a surprising "feature".
- Other predicates didn't behave in such a way, e.g. the IN predicate, the BETWEEN predicate, or the LIKE predicate.
- Variable binding behaved unpredictably, as IS NULL predicates don't bind any variables.
- The generated SQL depended on the possible combinations of bind values, which creates unnecessary hard-parses every time a new unique SQL statement is rendered.

Here is an example how to check if a field has a given value, without applying SQL's ternary NULL logic:

```
String possiblyNull = null; // Or else...

// jOOQ 2.6
Condition condition1 = BOOK.TITLE.eq(possiblyNull);

// jOOQ 3.0
Condition condition2 = BOOK.TITLE.eq(possiblyNull).or(BOOK.TITLE.isNull().and(val(possiblyNull).isNull()));
Condition condition3 = BOOK.TITLE.isNotDistinctFrom(possiblyNull);
```

Configuration

DSLContext, ExecuteContext, RenderContext, BindContext no longer extend Configuration for "convenience". From jOOQ 3.0 onwards, composition is chosen over inheritance as these objects are not really configurations. Most importantly

- DSLContext is only a DSL entry point for constructing "attached" QueryParts
- ExecuteContext has a well-defined lifecycle, tied to that of a single query execution
- RenderContext has a well-defined lifecycle, tied to that of a single rendering operation
- BindContext has a well-defined lifecycle, tied to that of a single variable binding operation

In order to resolve confusion that used to arise because of different lifecycle durations, these types are now no longer formally connected through inheritance.

ConnectionProvider

In order to allow for simpler connection / data source management, jOOQ externalised connection handling in a new ConnectionProvider type. The previous two connection modes are maintained backwards-compatibly (JDBC standalone connection mode, pooled DataSource mode). Other connection modes can be injected using:

```
public interface ConnectionProvider {

    // Provide jOOQ with a connection
    Connection acquire() throws DataAccessException;

    // Get a connection back from jOOQ
    void release(Connection connection) throws DataAccessException;
}
```

These are some side-effects of the above change

- Connection-related JDBC wrapper utility methods (commit, rollback, etc) have been moved to the new DefaultConnectionProvider. They're no longer available from the DSLContext. This had been confusing to some users who called upon these methods while operating in pool DataSource mode.

ExecuteListeners

ExecuteListeners can no longer be configured via Settings. Instead they have to be injected into the Configuration. This resolves many class loader issues that were encountered before. It also helps listener implementations control their lifecycles themselves.

Data type API

The data type API has been changed drastically in order to enable some new DataType-related features. These changes include:

- [SQLDialect]DataType and SQLDataType no longer implement DataType. They're mere constant containers
- Various minor API changes have been done.

Object renames

These objects have been moved / renamed:

- jOOU: a library used to represent unsigned integer types was moved from org.jooq.util.unsigned to org.jooq.util.types (which already contained INTERVAL data types)

Feature removals

Here are some minor features that have been removed in jOOQ 3.0

- The ant task for code generation was removed, as it was not up to date at all. Code generation through ant can be performed easily by calling jOOQ's GenerationTool through a <java> target.
- The navigation methods and "foreign key setters" are no longer generated in Record classes, as they are useful only to few users and the generated code is very collision-prone.
- The code generation configuration no longer accepts comma-separated regular expressions. Use the regex pipe | instead.
- The code generation configuration can no longer be loaded from .properties files. Only XML configurations are supported.
- The master data type feature is no longer supported. This feature was unlikely to behave exactly as users expected. It is better if users write their own code generators to generate master enum data types from their database tables. jOOQ's enum mapping and converter features sufficiently cover interacting with such user-defined types.
- The DSL subtypes are no longer instanciable. As DSL now only contains static methods, subclassing is no longer useful. There are still dialect-specific DSL types providing static methods for dialect-specific functions. But the code-generator no longer generates a schema-specific DSL
- The concept of a "main key" is no longer supported. The code generator produces UpdatableRecords only if the underlying table has a PRIMARY KEY. The reason for this removal is the fact that "main keys" are not reliable enough. They were chosen arbitrarily among UNIQUE KEYs.
- The UpdatableTable type has been removed. While adding significant complexity to the type hierarchy, this type adds not much value over a simple Table.getPrimaryKey() != null check.
- The USE statement support has been removed from jOOQ. Its behaviour was ill-defined, while it didn't work the same way (or didn't work at all) in some databases.

7.9. Don't do this

Like any software, jOOQ has a few pitfalls, known issues, historic design flaws, etc., which the seasoned jOOQ developer should know to avoid. This section summarises both of jOOQ's and SQL's own pitfalls.

7.9.1. jOOQ: Implementing the DSL types

Almost the entire jOOQ DSL API as well as the model API should be considered as sealed in the sense of Java 17. In fact, starting from jOOQ 3.16, we have started sealing types and will continue to do so in the near future.

This means, users should ***never*** attempt to extend jOOQ via inheritance, unless explicitly stated otherwise. Exceptions to this rule include:

- CustomQueryPart and related types, which are designed to be subclassed.
- Configuration SPIs which are designed to be implemented by users, following a strategy pattern style design.
- Public classes starting with AbstractXYZ (must be extended to be used) or DefaultXYZ (may be extended, but can be used out of the box, too).

7.9.2. jOOQ: Referencing the Step types

By convention, all of jOOQ's DSL API consists of interfaces whose name ends in Step, such as org.jooq.SelectFromStep, which is ***the step in the DSL API where users can append the FROM clause*** .

When writing dynamic SQL queries, users may be tempted to reference these types explicitly, such as

```
SelectConditionStep<?> c =
create.select(T.A, T.B)
      .from(T)
      .where(T.C.eq(1));

if (something)
    c = c.and(T.D.eq(2));

Result<?> result = c.fetch();
```

With this user-code, it is possible to add predicates dynamically to a query. But there are caveats:

- In jOOQ 3.x, the DSL API is mostly mutable, but this may change in the future. Code that assumes mutability might break.
- The DSL API guarantees source compatibility ***of methods*** between minor versions, but not ***of types***, as set out in this section of the manual. In order to evolve the DSL, it is sometimes necessary to replace the type hierarchy that implements the DSL. If you chain your method calls to look like actual SQL, you will not notice these changes (as the method calls remain compatible). But if you assign the Step types to local variables, or pass them around in your API, then that code might break.

It is much better to use one of the approaches mentioned in the section about dynamic SQL. Also, this blog post elaborates more in detail on the topic of creating optional SQL clauses.

7.9.3. Schema: NULL columns

In most RDBMS, the default nullability on any column is NULL, even if NOT NULL is mostly a more reasonable default. Whenever you know your data is not supposed to contain NULL values, then add an explicit NOT NULL constraint. This has the following benefits:

- Data integrity: One case of incorrect data less to worry about.
- Documentation: Even if your client application might make sure you'll never get NULL values in a column, it's still better to formally communicate this fact through a constraint.
- Performance: With NULL being an impossible value, quite a few optimisations can be applied that couldn't be, otherwise.

For example

```
CREATE TABLE customer (
  -- [...]
  phone   TEXT,          -- Here, the default of being nullable applies (in most RDBMS), but should it?
  address TEXT NULL,     -- The address might optional, you can mark it as such, explicitly, in many RDBMS
  email   TEXT NOT NULL  -- Every customer needs an email, this isn't an optional field
);
```

This rule obviously doesn't apply when a value is optional, in case of which NULL might be a desirable value.

© 2009 - 2025 by Data Geekery™ GmbH.

7.9.4. Schema: Unnamed constraints

Most RDBMS are able to generate a constraint name if you're not specifying one explicitly:

```
CREATE TABLE actor (
  actor_id BIGINT PRIMARY KEY
);

CREATE TABLE film (
  film_id BIGINT PRIMARY KEY
);

CREATE TABLE film_actor (
  actor_id BIGINT NOT NULL REFERENCES actor,
  film_id BIGINT NOT NULL REFERENCES film,

  PRIMARY KEY (actor_id, film_id)
);
```

While this is correct, it makes evolving such a schema much harder. It is usually better to give an explicit name to each constraint, such that constraints can easily be dropped, deactivated temporarily, etc.:

7.9.5. Schema: Unnecessary surrogate keys

The surrogate key vs natural key discussion is almost as old as SQL itself. Both approaches have their pros and cons, depending on the nature of the table you're designing. In short:

- A surrogate key (e.g. an IDENTITY or UUID) has no business value, and can thus be generated on any data, including not well normalised data.
- A natural key is a true placeholder for the data it represents (e.g. an ISO country code), which can help prevent joins to look up the useful information, as the useful information is already referenced in the foreign keys.

While there is a lot of debate whether a schema should be uniformly designed for consistency reasons (usually all pro surrogate key), or whether a few exceptions on a few tables are possible, there is one exception where a surrogate key is often the wrong choice: Relationship tables! In a relationship table like this:

```
CREATE TABLE actor (
  actor_id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL
);

CREATE TABLE film (
  film_id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  title TEXT NOT NULL
);

CREATE TABLE film_actor (
  actor_id BIGINT NOT NULL REFERENCES actor,
  film_id BIGINT NOT NULL REFERENCES film,

  PRIMARY KEY (actor_id, film_id)
);
```

Schema designers might be tempted to add a FILM_ACTOR_ID to the FILM_ACTOR table, but why? There is never any need to reference a many-to-many relationship entry by its surrogate key alone. We always use the foreign keys, which, if the schema is properly normalised, should have at least a UNIQUE constraint on them. Why not just make that the PRIMARY KEY?

© 2009 - 2025 by Data Geekery™ GmbH.

This blog post discusses the topic of the cost of useless surrogate keys in relationship tables, from a performance perspective.

7.9.6. Schema: Wrong data types

Most RDBMS don't have too many data types, with PostgreSQL being an exception via its powerful EXTENSIONS system. Even so, using the correct data type has at least these benefits:

- Data integrity: One case of incorrect data less to worry about.
- Documentation: Even if your client application might make sure you'll never get e.g. non-numeric values in a VARCHAR column, it's still better to formally communicate this fact through types.
- Performance: Without proper statistics, the optimiser might make the wrong estimates simply because of wrong data types being used.

For example

```
CREATE TABLE transaction (
  -- [...]
  amount     TEXT NOT NULL, -- Amount is probably a DECIMAL or NUMERIC value, so why not use that?
  value_date TEXT NOT NULL  -- But it's a date, so why not use DATE or TIMESTAMP?
);
```

7.9.7. SQL: COUNT(*) instead of EXISTS()

Do you go to the supermarket to count all their apples just to see if they have any apples? You don't. Likewise, you shouldn't run a COUNT(*) query to check if the value is bigger than 0. Use the EXISTS predicate, instead.

We've blogged about this and benchmarked the difference. It really does make a difference!

So, don't do this:

```
if (create.fetchValue(selectCount().from(AUTHOR)) > 0) {
    // ...
}
```

But do this, instead:

```
if (create.fetchValue(exists(selectOne().from(AUTHOR)))) {
    // ...
}
```

Of course, it's totally possible to embed this EXISTS predicate in a more complex query and possibly avoid the unnecessary secondary roundtrip...

7.9.8. SQL: N+1

"Hooray, we're using jOOQ (i.e. SQL), so we got rid of our N+1 problems."

Well, there's a class of N+1 problems that jOOQ won't ever run into. It's those "accidental" N+1 queres that happen because of lazy loading. With jOOQ everything is always loaded "eagerly", exactly as you specify it in your queries. Unlike in ORMs, eager loading isn't automatic either. jOOQ doesn't just materialise large parts of your object graph on its own. Everything is done explicitly. But that means you can still run into *explicit* N+1 problems

> *(!) Our diagnostics module can auto detect repeated queries for you, which are usually caused by N +1 problems. See* ***repeated statements***

An example of an N+1 problem caused by jOOQ queries:

```
// 1 query
for (Integer id : create
    .select(AUTHOR.ID)
    .from(AUTHOR)
    .fetch(AUTHOR.ID)
) {

    // N queries
    List<Integer> books =
    create.select(BOOK.ID)
          .from(BOOK)
          .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          .fetch(BOOK.ID);
}
```

The N+1 problem (technically, it should have been named 1+N problem) can be seen easily above:

- 1 query is executed to fetch all AUTHOR records.
- N queries are executed to fetch all BOOK records *per* AUTHOR.

This particular query would much better be implemented with a simple SQL JOIN:

```
// 1 query
for (Record2<Integer, Integer> record : create
    .select(AUTHOR.ID)
    .from(AUTHOR)
    .join(BOOK)
    .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
) {

    // No additional queries needed...
}
```

You can use one of jOOQ's many mapping capabilities to nest your collections directly in your Java logic as long as you just don't run several roundtrips to the database.

N+1 isn't a problem that is strictly related to SQL. It just got popularised by ORMs that emphasise automatic population of object graphs via lazy loading, rather than emphasising querying. The underlying problem, however, is latency between a client (your Java code) and a server (your RDBMS), which is caused by too many round trips. See this blog post for an example that compares calling a stored procedure 1 time to fetch N items, vs. calling another stored procedure N times to fetch 1 item, each time: https://blog.jooq.org/the-cost-of-jdbc-server-roundtrips/. The result is the same.

But with SQL, things get even worse. If you're pushing the entire declarative query into the database, the database has freedom to choose between various eligible algorithms to produce the result (e.g. hash join vs nested loop join, etc.). If you loop over your N parent rows yourself, you're enforcing a nested loop join, which can be worse *in addition to* the extra latency, in case a hash join or merge join would have been better.

7.9.9. SQL: NATURAL JOIN or JOIN USING

NATURAL JOIN or JOIN .. USING seem to be useful utilities at first when a normalised schema is very well designed in a way for PRIMARY KEY and FOREIGN KEY column names to always match (and in the case of NATURAL JOIN, all other column names to never match). The Sakila database is an example for the latter, with tables like these:

```
CREATE TABLE actor (
  actor_id BIGINT PRIMARY KEY,
  first_name TEXT,
  last_name TEXT,
  last_update TIMESTAMP
);

CREATE TABLE film (
  film_id BIGINT PRIMARY KEY,
  title TEXT,
  last_update TIMESTAMP
);

CREATE TABLE film_actor (
  actor_id BIGINT REFERENCES film,
  film_id BIGINT REFERENCES film,
  last_update TIMESTAMP
)
```

And indeed, the fact that e.g. ACTOR_ID is present in both ACTOR and FILM_ACTOR tables, and that it means the same thing in both tables is very convenient for queries like these:

```
SELECT *
FROM actor
JOIN film_actor USING (actor_id)
JOIN film USING (film_id);
```

But already this simple schema shows that we cannot use NATURAL JOIN due to the LAST_UPDATE columns, which shouldn't be part of the JOIN predicate.

```
SELECT *
FROM actor
NATURAL JOIN film_actor -- Wrong, because it joins on actor.last_update = film_actor.last_update
NATURAL JOIN film       -- Wrong, because it joins on film_actor.last_update = film.last_update
```

Even USING can suffer from such ambiguity, e.g. when we happen to add an ACTOR_ID column to the FILM table:

```
-- Where actor_id means the "main" actor ID
ALTER TABLE film ADD actor_id BIGINT REFERENCES actor;
```

This addition is debatable as:

- The naming seems to be insufficiently describing the fact that it is the "main" actor
- From a normalisation perspective, it seems to be redundant to repeat this information on the FILM table, as the FILM_ACTOR table could just have an additional flag about an actor being the "main" actor in a film.

As can be seen, queries containing NATURAL JOIN or JOIN .. USING are easy to get (often subtly) wrong, and even easier to break when the schema evolves. As such, it is best to just avoid the language feature entirely, except for the occasional ad-hoc query.

© 2009 - 2025 by Data Geekery™ GmbH.

7.9.10. SQL: NOT IN predicate

The NOT IN predicate seems to be just the inverse of the useful IN predicate, but in SQL, this isn't entirely true, thanks to SQL's three valued logic.

Look at the following transformations of equivalent predicates:

```
-- IN predicate is equivalent to:
A IN (B, C)
A = ANY (B, C)
A = B OR A = C

-- NOT IN predicate is equivalent to:
A NOT IN (B, C)
A <> ANY (B, C)
A <> B AND A <> C
```

Now, imagine if one of the values is NULL, then, informally:

```
-- IN predicate is equivalent to:
A IN (B, NULL)
A = ANY (B, NULL)
A = B OR A = NULL
A = B OR NULL
A = B

-- NOT IN predicate is equivalent to:
A NOT IN (B, NULL)
A <> ANY (B, NULL)
A <> B AND A <> NULL
A <> B AND NULL
NULL
```

Think of NULL as UNKNOWN:

- If one value of a disjunction (OR) is UNKNOWN, the result is either TRUE or UNKNOWN, the latter behaving like FALSE in a query. We're fine.
- If one value of a conjunction (AND) is UNKNOWN, the result is either FALSE or UNKNOWN, so the predicate always behaves as if it were FALSE. This is never what we want!

To make things worse, if you're using NOT IN (subquery), this problem can happen occasionally only, when the subquery returns a single NULL value. It's logical, but never useful. So better just use the NOT EXISTS predicate instead.

See also this blog post, which talks about compatibility across dialects.

7.9.11. SQL: ORDER BY [column index]

The SQL-92 standard and most implementations have always supported ordering by column index (though the feature has been removed already from SQL-99!). It can be useful for quick and dirty queries like these:

```
SELECT ID, FIRST_NAME || ' ' || LAST_NAME
FROM AUTHOR
ORDER BY 2;
```

The above is equivalent to this:

```
SELECT ID, FIRST_NAME || ' ' || LAST_NAME
FROM AUTHOR
ORDER BY FIRST_NAME || ' ' || LAST_NAME;
```

But it is easy to see that this query may quickly break by introducing another column in the projection, anywhere before the sorted column with index 2

```
-- Adding a column has broken the query
SELECT ID, LAST_UPDATE, FIRST_NAME || ' ' || LAST_NAME
FROM AUTHOR
ORDER BY 2;
```

Alternatively, these variants would not have broken:

```
-- The query is not affected by this change
SELECT ID, LAST_UPDATE, FIRST_NAME || ' ' || LAST_NAME
FROM AUTHOR
ORDER BY FIRST_NAME || ' ' || LAST_NAME;

-- Also, aliasing the expression (to avoid repetition) can work in many dialects
SELECT ID, LAST_UPDATE, FIRST_NAME || ' ' || LAST_NAME AS NAME
FROM AUTHOR
ORDER BY NAME;
```

While for quick ad-hoc queries, the ORDER BY [column index] feature can be occasionally useful, it's generally good to simply avoid the feature, especially when using jOOQ, which makes reusing query parts very simple.

7.9.12. SQL: Rely on implicit ordering

A SQL query produces reliable ordering *only* if:

- An ORDER BY clause is provided, explicitly.
- That ORDER BY clause is deterministic.

For example:

```
SELECT *
FROM BOOK
ORDER BY title, id
```

In many of the above examples, it would be weird if the RDBMS didn't produce any implicit ordering. But since SQL is a 4GL, and optimisers are free to produce *any* execution plan that implements the query's specifications (which don't specify any ordering explicitly), your assumptions may break at any time.

Implicit row insertion ordering

```
SELECT TITLE FROM BOOK;
```

You might think that rows are simple read out of a table, but what if this table is a view? What if it is partitioned? What if it is distributed? What if a covering index suddenly applies?

Implicit derived table ordering

```
SELECT TITLE FROM (
  SELECT TITLE FROM BOOK ORDER BY TITLE
) AS B;
```

The derived table specifies an explicit ordering, but the outer query does not. Nothing prevents an optimiser from deciding that in this case, the explicit ordering is unnecessary, even if it might be unlikely to do so for arbitrary reasons.

Implicit union subquery ordering

```
(SELECT TITLE FROM BOOK ORDER BY TITLE) UNION (SELECT 'Not really a book')
```

The UNION subquery specifies an explicit ordering, but there's no requirement at all for the UNION operator to maintain this ordering. In fact, UNION might be implemented using hashing, so the ordering wouldn't be stable.

Implicit window ordering

```
SELECT TITLE, ROW_NUMBER () OVER (ORDER BY TITLE)
FROM BOOK;
```

A window function may be ordered, and that ordering may be stable within a query, such that *by accident*, the results are as one would expect. But there's no guarantee for this. The RDBMS may produce results in any other order than the one from the window function.

Implicit GROUP BY ordering

```
SELECT AUTHOR_ID, COUNT(*) FROM BOOK GROUP BY AUTHOR_ID;
```

The GROUP BY clause may be implemented using a sort operation, but it may as well be implemented using hashing. There's no guarantee of one algorithm being used rather than the other. With sorting, there might be an accidental order that remains stable, but there's no guarantee for that.

Implicit non-deterministic ordering

```
SELECT ID, TITLE FROM BOOK ORDER BY TITLE;
```

In our schema, TITLE is not a UNIQUE column, so multiple books could share the same title. There's no guarantee of any stable ordering among the resulting ID values, unless ID is included in the ORDER BY clause.

© 2009 - 2025 by Data Geekery™ GmbH.

7.9.13. SQL: SELECT *

The SELECT * syntax has been introduced mostly for convenience of ad-hoc SQL. It's very useful to be able to quickly check out data on a production system to see what's available. For those cases, we often don't care about supplying a meaningful SELECT clause. We just want to project everything, e.g.

```
SELECT * FROM book
```

```
create.select(asterisk()).from(BOOK).fetch();
```

In real world applications, however, we shouldn't do this practice, neither in SQL, nor with jOOQ. We should limit ourselves to project only those columns that we really need. The key here is to project only what we need, so this isn't about the * (the asterisk as a syntactic token), but it could equally be about listing all columns explicitly, e.g.

```
SELECT
  id,
  author_id,
  title,
  published_in,
  language_id,
  ...
FROM book
```

```
create.select(
          BOOK.ID,
          BOOK.AUTHOR_ID,
          BOOK.TITLE,
          BOOK.PUBLISHED_IN,
          BOOK.LANGUAGE_ID,
          ...)
      .from(BOOK).fetch();
```

This blog post explains in depth why SELECT * is bad practice (not the asterisk is at fault, but the blind projection of everything, including when you use jOOQ's selectFrom(Table)). The main problem is that you're creating unnecessary, mandatory work on the server:

- Unnecessary, because you're throwing away the data right after fetching it
- Mandatory, because the SQL optimiser doesn't know that, so it must provide the data

This has at least the following effects:

- Disk I/O: The server has to read all the data from disk, which may be offloaded to lob storages or whatever. If your tables are wide, then this can be significant!
- Memory consumption: Both on the server and on the client, you're wasting memory and the associated CPU cycles of transferring data from/to memory, just to discard it again. The server might even cache all this data in a "buffer cache", completely unnecessarily.
- Index usage: So called "covering indexes" cannot be used this way, e.g. when your relationship tables have additional columns other than the foreign keys, the projection of those columns will likely make the query ***much*** slower than if you could just have used a covering index.
- Join elimination: Some more advanced SQL transformations are impossible to do, such as the JOIN elimination transformation, where whole joins are removed from your query, as they're provably unnecessary. But they're only unnecessary if you're not projecting anything from a table. If you do, then the JOIN must be executed.

Seems obvious, no? Best not be lazy, design your queries carefully. Again, this blog post explains the topic in depth.

7.9.14. SQL: SELECT DISTINCT

The SELECT DISTINCT syntax can occasionally be useful to remove duplicates in case your schema isn't normalised. For example, in a log table, you may want to query for distinct IP addresses:

```
SELECT DISTINCT ip FROM access_log
```

```
create.selectDistinct(ACCESS_LOG.IP).from(ACCESS_LOG).fetch();
```

This is especially useful for ad-hoc querying.

A lot of times, when removing such duplicates in application queries, you'll want to add aggregate functions (e.g. number of log entries per IP, number of distinct logins per IP, etc.), so you'll use GROUP BY, instead of the DISTINCT clause.

However, be wary of using DISTINCT for removing duplicates that you're not entirely sure why they happen, i.e. typically in very large and complex queries, but also a lot of times when there are (INNER) JOIN expressions, when in fact there should be SEMI JOIN expressions. For example, the following query looks for all authors of books available in book stores starting with the letter A:

```
SELECT DISTINCT
  author.first_name,
  author.last_name
FROM author
JOIN book ON author.id = book.author_id
JOIN book_to_book_store
  ON book.id = book_to_book_store.book_id
WHERE book_to_book_store.name LIKE 'A%'
```

```
create.selectDistinct(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .join(BOOK_TO_BOOK_STORE)
        .on(BOOK.ID.eq(BOOK_TO_BOOK_STORE.BOOK_ID))
      .where(BOOK_TO_BOOK_STORE.NAME.like("A%")).fetch();
```

Now, due to the nature of our schema:

- Authors have written many books
- Books are available from multiple book stores

Hence, it is very likely that the same author appears in the result multiple times, thus DISTINCT. From a database performance perspective, this may be very bad, because the authors are duplicated into some internal data structure before they're removed again from the result. Besides, depending on the optimiser, we might have to scan all the books and all the book stores even if we already found an author. Optimisers are very different, so this may or may not be a problem of equal severity in each RDBMS, but certainly something to keep an eye out for.

A much better solution here is to use a SEMI JOIN, e.g. using the IN predicate or EXISTS predicate (they're semantically equivalent):

```
SELECT
  author.first_name,
  author.last_name
FROM author
WHERE author.id IN (
  SELECT book.author_id
  FROM book
  JOIN book_to_book_store
    ON book.id = book_to_book_store.book_id
  WHERE book_to_book_store.name LIKE 'A%'
)
```

```
create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .where(AUTHOR.ID).in(
          select(BOOK.AUTHOR_ID)
          .from(BOOK)
          .join(BOOK_TO_BOOK_STORE)
            .on(BOOK.ID.eq(BOOK_TO_BOOK_STORE.BOOK_ID))
          .where(BOOK_TO_BOOK_STORE.NAME.like("A%")
      ).fetch();
```

Now, we'll never get any duplicate authors.

If in doubt, check your execution plan and execution metrics about what is the best query here, but in a lot of cases, DISTINCT is a hint that something unnecessary is being done, and then duplicates are removed again.

© 2009 - 2025 by Data Geekery™ GmbH.

7.9.15. SQL: Unnecessary UNION instead of UNION ALL

The UNION operator removes duplicate rows, whereas UNION ALL retains them. It isn't always possible for an optimiser to prove that there are no duplicates possible. If you, as a developer, *know* that there can't be any duplicates, or if you don't care about the duplicates, or even want them, then it's always better to use UNION ALL instead of UNION, as that avoids a potentially costly sort or hash operation to remove the duplicates.

For example:

```
SELECT 'Book' AS OBJECT_TYPE, ID FROM BOOK
UNION ALL -- No removal of duplicates necessary in this case
SELECT 'Author' AS OBJECT_TYPE, ID FROM AUTHOR;
```

7.10. The most important jOOQ types

For new users working with jOOQ for the first time, the number of types in the jOOQ API can be overwhelming. The SQL language doesn't have many such "visible" types, although if you think about SQL the way jOOQ does, then they're there just the same, but hidden from users via an English style syntax.

This overview will list the most important jOOQ types in a cheat sheet form.

7.11. Credits

jOOQ lives in a very challenging ecosystem. The Java to SQL interface is still one of the most important system interfaces. Yet there are still a lot of open questions, best practices and no "true" standard has been established. This situation gave way to a lot of tools, APIs, utilities which essentially tackle the same problem domain as jOOQ. jOOQ has gotten great inspiration from pre-existing tools and this section should give them some credit. Here is a list of inspirational tools in alphabetical order:

- Hibernate: The de-facto standard (JPA) with its useful table-to-POJO mapping features have influenced jOOQ's org.jooq.ResultQuery facilities
- JPA: The de-facto standard in the javax.persistence packages, supplied by Oracle. Its annotations are useful to jOOQ as well.
- QueryDSL: A "LINQ-port" to Java. It has a similar fluent API, a similar code-generation facility, yet quite a different purpose. While jOOQ is all about SQL, QueryDSL (like LINQ) is mostly about querying.
- SLICK: A "LINQ-like" database abstraction layer for Scala. Unlike LINQ, its API doesn't really remind of SQL. Instead, it makes SQL look like Scala.
- Spring Data: Spring's JdbcTemplate knows RowMappers, which are reflected by jOOQ's RecordMapper

© 2009 - 2025 by Data Geekery™ GmbH.