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
- Tools
  This section is dedicated to tools that ship with jOOQ.
- Reference
  This section is a reference for elements in this manual

Table of contents

1. Preface

jOOQ's reason for being - compared to JPA

Java and SQL have come a long way. SQL is an "old", yet established and well-understood technology. Java is a legacy too, although its platform JVM allows for many new and contemporary languages built on top of it. Yet, after all these years, libraries dealing with the interface between SQL and Java have come and gone, leaving JPA to be a standard that is accepted only with doubts, short of any surviving options.

So far, there had been only few database abstraction frameworks or libraries, that truly respected SQL as a first class citizen among languages. Most frameworks, including the industry standards JPA, EJB, Hibernate, JDO, Criteria Query, and many others try to hide SQL itself, minimising its scope to things called JPQL, HQL, JDOQL and various other inferior query languages

jOOQ has come to fill this gap.

jOOQ's reason for being - compared to LINQ

Other platforms incorporate ideas such as LINQ (with LINQ-to-SQL), or Scala's SLICK, or also Java's QueryDSL to better integrate querying as a concept into their respective language. By querying, they understand querying of arbitrary targets, such as SQL, XML, Collections and other heterogeneous data stores. jOOQ claims that this is going the wrong way too.

In more advanced querying use-cases (more than simple CRUD and the occasional JOIN), people will want to profit from the expressivity of SQL. Due to the relational nature of SQL, this is quite different from what object-oriented and partially functional languages such as C#, Scala, or Java can offer.

It is very hard to formally express and validate joins and the ad-hoc table expression types they create. It gets even harder when you want support for more advanced table expressions, such as pivot tables, unnested cursors, or just arbitrary projections from derived tables. With a very strong object-oriented typing model, these features will probably stay out of scope.

In essence, the decision of creating an API that looks like SQL or one that looks like C#, Scala, Java is a definite decision in favour of one or the other platform. While it will be easier to evolve SLICK in similar ways as LINQ (or QueryDSL in the Java world), SQL feature scope that clearly communicates its underlying intent will be very hard to add, later on (e.g. how would you model Oracle's partitioned outer join syntax? How would you model ANSI/ISO SQL:1999 grouping sets? How can you support scalar subquery caching? etc...).

jOOQ has come to fill this gap.

jOOQ's reason for being - compared to SQL / JDBC

So why not just use SQL?

SQL can be written as plain text and passed through the JDBC API. Over the years, people have become wary of this approach for many reasons:

- No typesafety
- No syntax safety
- No bind value index safety
- Verbose SQL String concatenation
- Boring bind value indexing techniques
- Verbose resource and exception handling in JDBC
- A very "stateful", not very object-oriented JDBC API, which is hard to use

For these many reasons, other frameworks have tried to abstract JDBC away in the past in one way or another. Unfortunately, many have completely abstracted SQL away as well

jOOQ has come to fill this gap.

jOOQ is different

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!

2. 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.
- 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
- 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
- Luc Marchaud
- Lukas Eder
- Matti Tahvonen
- Michael Doberenz
- Michael Simons
- Michał Kołodziejski
- Miguel Gonzalez Sanchez
- Mustafa Yücel
- Nathaniel Fischer
- Oliver Flege
- Peter Ertl
- Richard Bradley
- Robin Stocker
- Samy Deghou
- Sander Plas
- Sean Wellington
- Sergey Epik
- Sergey Zhuravlev
- Stanislas Nanchen
- Stephan Schroevers
- Sugiharto Lim
- Sven Jacobs
- Szymon Jachim
- Terence Zhang
- Timothy Wilson
- Timur Shaidullin
- Thomas Darimont
- Tsukasa Kitachi
- Victor Bronstein
- Victor Z. Peng
- Vladimir Kulev
- Vladimir Vinogradov

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

3. 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.

3.1. 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:

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

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

```xml
<!-- 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:

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

```java
// 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
-- SQL assumptions
------------------

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

```
// 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.

3.2. 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 );
```

3.3. 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.

3.3.1. 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.

3.3.2. 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.

3.3.3. 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

3.3.4. 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:

```
// 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

3.3.5. 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.

3.4. 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.

3.4.1. jOOQ in 7 easy steps

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

3.4.1.1. 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>3.17.8</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>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

Commercial Editions (Java 17+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq</artifactId>
  <version>3.17.8</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>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

Commercial Editions (Java 11+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro-java-11</groupId>
  <artifactId>jooq</artifactId>
  <version>3.17.8</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.pro-java-11</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-11</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

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>3.17.8</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>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-8</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

Commercial Editions (Free Trial, Java 17+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq</artifactId>
  <version>3.17.8</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>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

Commercial Editions (Free Trial, Java 11+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.trial-java-11</groupId>
  <artifactId>jooq</artifactId>
  <version>3.17.8</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.trial-java-11</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial-java-11</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</version>
</dependency>
```

Commercial Editions (Free Trial, Java 8+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.trial-java-8</groupId>
  <artifactId>jooq</artifactId>
  <version>3.17.8</version>
</dependency>

<!-- These may not be required, unless you use the GenerationTool manually for code generation -->
<dependency>
  <groupId>org.jooq.trial-java-8</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial-java-8</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.17.8</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!

3.4.1.2. 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`)
);
```

3.4.1.3. 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.KotlinGenerator
         - 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-3.17.8.jar;^
jooq-meta-3.17.8.jar;^
jooq-codegen-3.17.8.jar;^
reactive-streams-1.0.3.jar;^
r2dbc-spi-0.9.0.RELEASE.jar;^
jakarta.xml.bind-api-3.0.0.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-3.17.8.jar:\
jooq-meta-3.17.8.jar:\
jooq-codegen-3.17.8.jar:\
reactive-streams-1.0.3.jar:\
r2dbc-spi-0.9.0.RELEASE.jar:\
jakarta.xml.bind-api-3.0.0.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 3.17.8 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-3.17.8.jar;jooq-meta-3.17.8.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
```

3.4.1.4. 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.

3.4.1.5. 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.

3.4.1.6. 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();
        }
    }
}
```

3.4.1.7. 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/

3.4.2. 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. You will also find the source code of this tutorial on GitHub at https://github.com/jOOQ/jOOQ/tree/main/jOOQ-examples/jOOQ-flyway-example, and the full pom.xml file here.

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 with Java 17 support,
             org.jooq.pro-java-11    for commercial editions with Java 11 support,
             org.jooq.pro-java-8     for commercial editions with Java 8 support,
             org.jooq.trial          for the free trial edition with Java 17 support,
             org.jooq.trial-java-11  for the free trial edition with Java 11 support,
             org.jooq.trial-java-8   for the free trial edition with Java 8 support

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq</artifactId>
    <version>3.17.8</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 suggests 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 with Java 17 support,
             org.jooq.pro-java-11    for commercial editions with Java 11 support,
             org.jooq.pro-java-8     for commercial editions with Java 8 support,
             org.jooq.trial          for the free trial edition with Java 17 support,
             org.jooq.trial-java-11  for the free trial edition with Java 11 support,
             org.jooq.trial-java-8   for the free trial edition with Java 8 support

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <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:3.17.8:generate (default) @ jooq-flyway-example ---
[INFO] --- jooq-codegen-maven:3.17.8: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.

3.4.3. 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
```

3.5. 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(");");
       }
   );
```

The above example is explained more in depth in this blog post: https://blog.jooq.org/java-8-friday-no-more-need-for-orms/. For more information about Java 8, consider these resources:

- Our Java 8 Friday blog series
- A great Java 8 resources collection by the folks at Baeldung.com

3.6. jOOQ and JavaFX

One of the major improvements of Java 8 is the introduction of JavaFX into the JavaSE. With jOOQ and Java 8 Streams and lambdas, it is now very easy and idiomatic to transform SQL results into JavaFX XYChart.Series or other, related objects:

Creating a bar chart from a jOOQ Result

As we've seen in the previous section about jOOQ and Java 8, jOOQ integrates seamlessly with Java 8's Streams API. The fluent style can be maintained throughout the data transformation chain.

In this example, we're going to use Open Data from the world bank to show a comparison of countries GDP and debts:

```
DROP SCHEMA IF EXISTS world;

CREATE SCHEMA world;

CREATE TABLE world.countries (
  code CHAR(2) NOT NULL,
  year INT NOT NULL,
  gdp_per_capita DECIMAL(10, 2) NOT NULL,
  govt_debt DECIMAL(10, 2) NOT NULL
);

INSERT INTO world.countries
VALUES ('CA', 2009, 40764, 51.3),
       ('CA', 2010, 47465, 51.4),
       ('CA', 2011, 51791, 52.5),
       ('CA', 2012, 52409, 53.5),
       ('DE', 2009, 40270, 47.6),
       ('DE', 2010, 40408, 55.5),
       ('DE', 2011, 44355, 55.1),
       ('DE', 2012, 42598, 56.9),
       ('FR', 2009, 40488, 85.0),
       ('FR', 2010, 39448, 89.2),
       ('FR', 2011, 42578, 93.2),
       ('FR', 2012, 39759,103.8),
       ('GB', 2009, 35455,121.3),
       ('GB', 2010, 36573, 85.2),
       ('GB', 2011, 38927, 99.6),
       ('GB', 2012, 38649,103.2),
       ('IT', 2009, 35724,121.3),
       ('IT', 2010, 34673,119.9),
       ('IT', 2011, 36988,113.0),
       ('IT', 2012, 33814,131.1),
       ('JP', 2009, 39473,166.8),
       ('JP', 2010, 43118,174.8),
       ('JP', 2011, 46204,189.5),
       ('JP', 2012, 46548,196.5),
       ('RU', 2009,  8616,  8.7),
       ('RU', 2010, 10710,  9.1),
       ('RU', 2011, 13324,  9.3),
       ('RU', 2012, 14091,  9.4),
       ('US', 2009, 46999, 76.3),
       ('US', 2010, 48358, 85.6),
       ('US', 2011, 49855, 90.1),
       ('US', 2012, 51755, 93.8);
```

Once this data is set up (e.g. in an H2 or PostgreSQL database), we'll run jOOQ's code generator and implement the following code to display our chart:

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis();
xAxis.setLabel("Country");
yAxis.setLabel("% of GDP");

BarChart<String, Number> bc = new BarChart<String, Number>(xAxis, yAxis);
bc.setTitle("Government Debt");
bc.getData().addAll(

    // SQL data transformation, executed in the database
    // -------------------------------------------------
    DSL.using(connection)
       .select(
           COUNTRIES.YEAR,
           COUNTRIES.CODE,
           COUNTRIES.GOVT_DEBT)
       .from(COUNTRIES)
       .join(
           table(
               select(COUNTRIES.CODE, avg(COUNTRIES.GOVT_DEBT).as("avg"))
               .from(COUNTRIES)
               .groupBy(COUNTRIES.CODE)
           ).as("c1")
       )
       .on(COUNTRIES.CODE.eq(field(name("c1", COUNTRIES.CODE.getName()), String.class)))

       // order countries by their average projected value
       .orderBy(
           field(name("avg")),
           COUNTRIES.CODE,
           COUNTRIES.YEAR)

       // The result produced by the above statement looks like this:
       // +----+----+---------+
       // |year|code|govt_debt|
       // +----+----+---------+
       // |2009|RU  |     8.70|
       // |2010|RU  |     9.10|
       // |2011|RU  |     9.30|
       // |2012|RU  |     9.40|
       // |2009|CA  |    51.30|
       // +----+----+---------+

    // Java data transformation, executed in application memory
    // --------------------------------------------------------

       // Group results by year, keeping sort order in place
       .fetchGroups(COUNTRIES.YEAR)

       // Stream<Entry<Integer, Result<Record3<BigDecimal, String, Integer>>>>
       .entrySet()
       .stream()

       // Map each entry into a { Year -> Projected value } series
       .map(entry -> new XYChart.Series<>(
           entry.getKey().toString(),
           observableArrayList(

               // Map each country record into a chart Data object
               entry.getValue()
                    .map(country -> new XYChart.Data<String, Number>(
                         country.getValue(COUNTRIES.CODE),
                         country.getValue(COUNTRIES.GOVT_DEBT)
                    ))
           )
       ))
       .collect(toList())
);
```

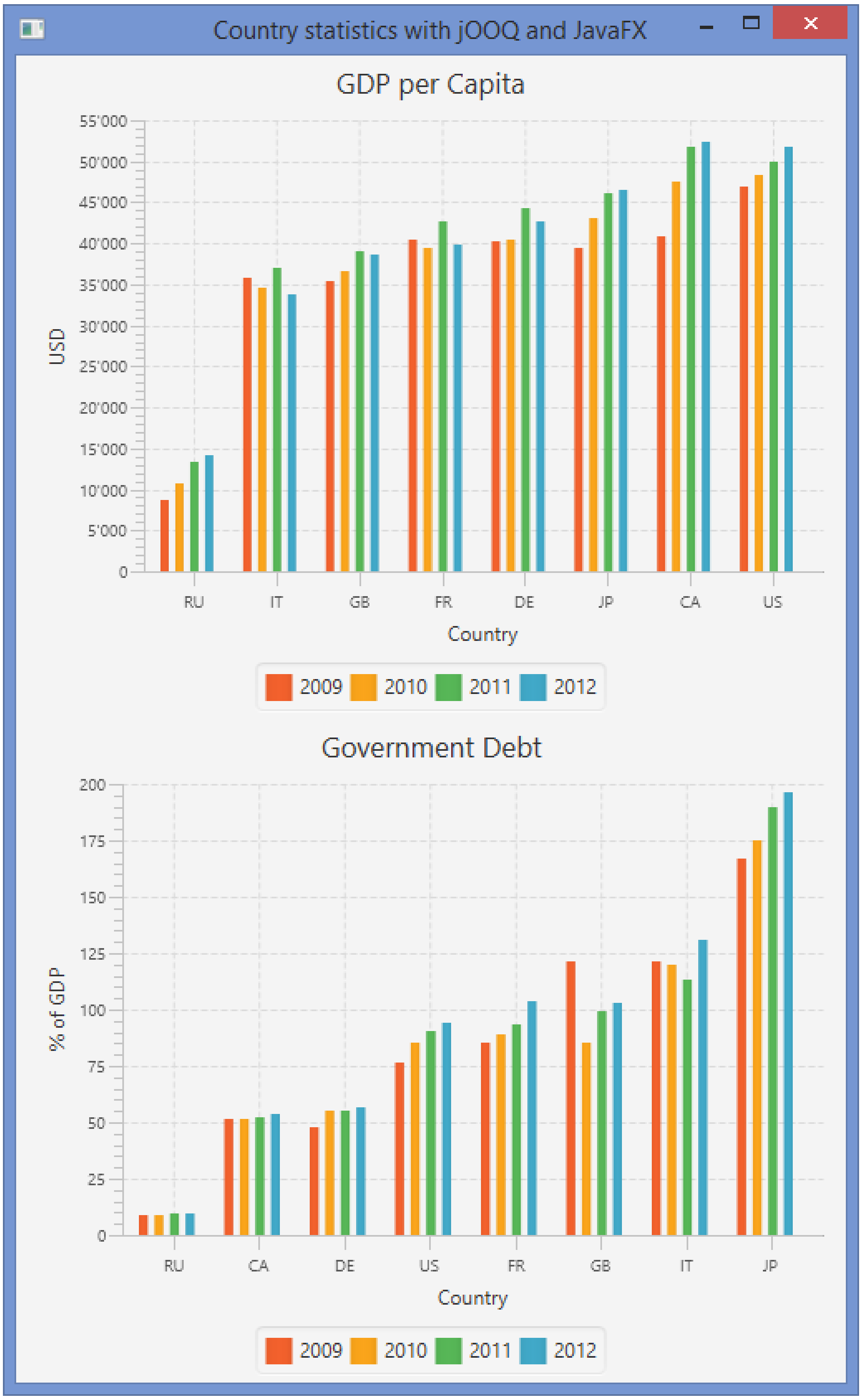
The above example uses basic SQL-92 syntax where the countries are ordered using aggregate information from a derived table, which is supported in all databases. If you're using a database that supports window functions, e.g. PostgreSQL or any commercial database, you could have also written a simpler query like this:00

```
DSL.using(connection)
   .select(
       COUNTRIES.YEAR,
       COUNTRIES.CODE,
       COUNTRIES.GOVT_DEBT)
   .from(COUNTRIES)

   // order countries by their average projected value
   .orderBy(
       DSL.avg(COUNTRIES.GOVT_DEBT).over(partitionBy(COUNTRIES.CODE)),
       COUNTRIES.CODE,
       COUNTRIES.YEAR)
   .fetch()
   ;

return bc;
```

When executed, we'll get nice-looking bar charts like these:



The complete example can be downloaded and run from GitHub:
https://github.com/jOOQ/jOOQ/tree/main/jOOQ-examples/jOOQ-javafx-example

3.7. jOOQ and Nashorn

With Java 8 and the new built-in JavaScript engine Nashorn, a whole new ecosystem of software can finally make easy use of jOOQ in server-side JavaScript. A very simple example can be seen here:

```
// Let's assume these objects were generated
// by the jOOQ source code generator
var Tables = Java.type("org.jooq.db.h2.information_schema.Tables");
var t = Tables.TABLES;
var c = Tables.COLUMNS;

// This is the equivalent of Java's static imports
var count = DSL.count;
var row = DSL.row;

// We can now execute the following query:
print(
    DSL.using(conn)
       .select(
           t.TABLE_SCHEMA,
           t.TABLE_NAME,
           c.COLUMN_NAME)
       .from(t)
       .join(c)
       .on(row(t.TABLE_SCHEMA, t.TABLE_NAME)
           .eq(c.TABLE_SCHEMA, c.TABLE_NAME))
       .orderBy(
           t.TABLE_SCHEMA.asc(),
           t.TABLE_NAME.asc(),
           c.ORDINAL_POSITION.asc())
       .fetch()
);
```

More details about how to use jOOQ, JDBC, and SQL with Nashorn can be seen here.

3.8. 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.

3.9. 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.

3.10. 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.

3.11. 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.

jOOQ is about SQL, and about SQL alone. Read more about our visions in the manual's preface.

3.12. 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 ***persisting an object graph***, JPA is a good fit. Though note that starting with jOOQ 3.15 you can also load trees with the MULTISET_AGG function and the MULTISET value constructor very easily.

And sometimes, it's best to combine both

3.13. 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

3.14. 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.

4. 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.

4.1. 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));
```

4.1.1. DSL subclasses

There are a couple of subclasses for the general query DSL. Each SQL dialect has its own dialect-specific DSL. For instance, if you're only using the MySQL dialect, you can choose to reference the MySQLDSL instead of the standard DSL:

The advantage of referencing a dialect-specific DSL lies in the fact that you have access to more proprietary RDMBS functionality. This may include:

- MySQL's encryption functions
- PL/SQL constructs, pgplsql, or any other dialect's ROUTINE-language (maybe in the future)

4.2. 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 : To provide execution lifecycle listeners (see ExecuteListeners for more details)
- org.jooq.ParseListenerProvider : To provide custom parser extensions (see SQL Parser Listener for more details)
- org.jooq.RecordListenerProvider : To provide record listeners for your CRUD operations (see CRUD SPI: RecordListener for more details)
- org.jooq.RecordMapperProvider : To provide an alternative default record mapper implementation (see POJOs with RecordMappers for more details)
- org.jooq.FormattingProvider : To provide custom default data export formats (see FormattingProvider for more details)
- JDBC access:

    * 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)

- R2DBC access:

    * io.r2dbc.spi.Connection : An optional R2DBC 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.
    * io.r2dbc.spi.ConnectionFactory : An optional R2DBC ConnectionFactory that will be re-used for the whole lifecycle of your Configuration. If you prefer using ConnectionFactories over Connections, jOOQ will internally fetch new Connections from your ConnectionFactory, conveniently closing them again after query execution. This is particularly useful in Spring contexts (see Connection vs. DataSource for more details)

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();
```

4.2.1. 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 Informix 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.INFORMIX, 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

4.2.2. 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).

4.2.3. 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;
}
```

Reactive querying

If you wish to use an R2DBC driver, you do not have to supply a org.jooq.ConnectionProvider to your Configuration. Instead, jOOQ can work with a io.r2dbc.spi.Connection (jOOQ will never close it) or io.r2dbc.spi.ConnectionFactory (jOOQ will close all R2DBC Connections that it creates).

4.2.4. 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:

```
// Implement an ExecuteListener
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.

4.2.5. 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.

4.2.6. 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("...");
```

4.2.7. 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-3.17.0.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-3.17.0.xsd

4.2.7.1. 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
```

4.2.7.2. Auto-inline bind values

Bind values are an important concept in SQL, for performance reasons, as they simplify caching of prepared statements in some RDBMS. jOOQ always creates bind values by default, when you write this:

```
-- Normally, a bind parameter marker is generated
AUTHOR.ID = ?
```

```
// This is the same as AUTHOR.ID.eq(val(1, AUTHOR.ID))
AUTHOR.ID.eq(1);
```

In some cases, however, it is better not to use a bind variable, but to create inline values, instead, so the optimiser can better apply its statistics. This is useful mainly when:

- The column is a constant discriminator column in a view, for example
- The column has very skewed statistics and only few possible values (e.g. a BOOLEAN, an ENUM type or a CHECK COL IN (1, 2, 3)) constraint.

In those cases, it can be useful to enable Settings.transformInlineBindValuesForFieldComparisons and implement a org.jooq.TransformProvider as follows:

```
Configuration configuration = ...
configuration.settings().setTransformInlineBindValuesForFieldComparisons(true);
configuration.set(new TransformProvider() {
    @Override
    public boolean inlineBindValuesForFieldComparisons(Field<?> field) {
        return field.getType() == Boolean.class
            || field.getDataType().isEnum(); // Or, perhaps, limit this only to certain enums
    }
});
```

Now, all queries whose predicates match the above TransformProvider content will have their relevant bind values inlined. For example:

```
-- Inlining applies to some columns now, not all
AUTHOR.ID = ? AND AUTHOR.STATUS = 'ACTIVE'
```

```
AUTHOR.ID.eq(1).and(
    AUTHOR.STATUS.eq(Status.ACTIVE));
```

Related settings include:

- Inline Threshold
- Statement Type

4.2.7.3. 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
```

4.2.7.4. Batch size

jOOQ offers a transparent batching API, which can buffer all statements generated by jOOQ and other JDBC backed APIs transparently in order to batch them:

```
// Everything in the below lambda will be buffered and batched
DSL.using(configuration).batched(c -> {
    module1.insertSomething(c);
    module2.insertSomethingElse(c);
});
```

Use the Settings.batchSize flag to govern the maximum batch statement size of this API:

```
Settings settings = new Settings()
    .withBatchSize(100); // Default Integer.MAX_VALUE
```

4.2.7.5. 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
```

4.2.7.6. 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
```

4.2.7.7. GROUP_CONCAT Configuration

The MySQL GROUP_CONCAT function suffers from a controversial design decision where results are truncated after a certain length, the @@group_concat_max_len.

Whenever jOOQ generates a GROUP_CONCAT function, by default, that MySQL system variable is increased to the maximum value for the scope of a single statement, e.g.

```
SET @T = @@GROUP_CONCAT_MAX_LEN;
SET @@GROUP_CONCAT_MAX_LEN = 4294967295;
SELECT GROUP_CONCAT(TITLE SEPARATOR ', ')
FROM BOOK;
SET @@GROUP_CONCAT_MAX_LEN = @T;
```

More details here. While this is a reasonable default behaviour (as opposed to the random truncation), it may occasionally be undesired, e.g. if statement batches (; separated statements) aren't possible in a single JDBC statement. The feature can be turned off with

Example configuration

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

4.2.7.8. 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. 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.

The two flags are independent of one another. If your database supports quoted, case sensitive identifiers, then using LOWER or UPPER on quoted identifiers may not work.

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.

4.2.7.9. Implicit join type

jOOQ's very useful implicit JOIN feature can be used to use a path notation to join tables on their actual, or synthetic foreign keys. For example:

```
// 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();
```

By default, this produces:

- An INNER JOIN if all columns of the foreign key are NOT NULL
- A LEFT JOIN if the foreign key is nullable / optional

This behaviour means that implicit joins do not filter results when placed in clauses that are not meant to filter, such as the SELECT clause or the ORDER BY clause.

Users may prefer to enforce a different behaviour, including:

- Always produce a LEFT JOIN, e.g. because this was the behaviour before jOOQ 3.14
- Always produce an INNER JOIN, e.g. because they're migrating off Hibernate / JPA, and depend on Hibernate's implicit joins producing inner joins

This change of behaviour can be achieved with the following setting:

Example configuration

```
Settings settings = new Settings()
    .withRenderImplicitJoinType(RenderImplicitJoinType.INNER_JOIN);
```

4.2.7.10. 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

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
```

4.2.7.11. 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
```

4.2.7.12. Interpreter Configuration

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

- interpreterDialect: The interpreter input dialect. This dialect is used to decide whether DDL interpretation should be done on an actual in-memory database of a specific type, or using jOOQ's built in DDL interpretation.
- interpreterDelayForeignKeyDeclarations: Whether the interpreter should delay the application of foreign key declarations (in case of which forward references are possible).
- interpreterLocale: The locale to use for things like case insensitive comparisons.
- interpreterNameLookupCaseSensitivity: The identifier case sensitivity that should be applied when interpreting SQL, depending on whether identifiers are quoted or not.
- interpreterSearchPath: The search path for unqualified schema objects used by the interpreter.

Example configuration

```
Settings settings = new Settings()
    .withInterpreterDialect(H2)                            // Defaults to DEFAULT
    .withInterpreterDelayForeignKeyDeclarations(true)      // Defaults to false
    .withInterpreterLocale(Locale.forLanguageTag("de"))    // Defaults to Locale.getDefault()
    .withInterpreterNameLookupCaseSensitivity(NEVER)       // Defaults to WHEN_QUOTED
    .withInterpreterSearchPath(...);                       // Defaults to an empty list
```

4.2.7.13. 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);
```

4.2.7.14. 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
```

4.2.7.15. 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
```

4.2.7.16. 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
```

4.2.7.17. 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
```

4.2.7.18. 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.

4.2.7.19. 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.

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.

4.2.7.20. 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 ":"
```

4.2.7.21. 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.

4.2.7.22. 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.
- parseDateFormat: The date format that is applied automatically when parsing date formatting functions without an explicit format.
- 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.
- parseRetainCommentsBetweenQueries: Whether comments in between statements from Parser.parse() are retained and parsed as ignored queries. Comments inside of statements (including procedural statements) currently aren't supported by jOOQ.
- 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.
- parseTimestampFormat: The timestamp format that is applied automatically when parsing timestamp formatting functions without an explicit format.
- 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]"
```

In addition to the above settings, there is also a powerful parser listener SPI called the org.jooq.ParseListener.

4.2.7.23. 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
```

4.2.7.24. Return all columns on store

When using the updatable records feature, jOOQ always fetches the generated identity value, if such a value is available and 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
```

4.2.7.25. 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
```

4.2.7.26. Runtime catalog, schema and table mapping

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")));
```

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"))));
```

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.

Mapping of catalogs

For databases like SQL Server, it is also possible to map catalogs in addition to schemata. The mechanism is exactly the same. So let's assume that we generated code for a table [dev].[dbo].[author] and want to map it to [my_book_world].[dbo].[author] at runtime. This can be achieved as follows:

Example configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withCatalogs(
        new MappedCatalog().withInput("DEV")
                           .withOutput("MY_BOOK_WORLD")));
```

To give you full control of how each and every table gets mapped, a MappedCatalog object can contain MappedSchema (and thus also MappedTable) definitions.

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"))));
```

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.

4.2.7.27. 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);
```

4.2.7.28. 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
```

4.2.7.29. 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
```

4.2.8. 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.

4.3. The DSL API

The DSL API is the primary way to construct queries or query parts in jOOQ. See the model API for an alternative way to interact with the jOOQ query object model.

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

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();
```

4.3.1. Mutability (historic)

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.

4.4. The model API

The model API is the secondary way to interact with queries or query parts in jOOQ. See the DSL API for the main way to interact with the jOOQ query object model.

4.4.1. Design

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The model API (Query Object Model or org.jooq.impl.QOM) is an auxiliary API implemented by each and every org.jooq.QueryPart allowing for users to get public access to jOOQ's internal query object model structure. For example:

```
// Create an expression using the DSL API:
Field<String> field = substring(BOOK.TITLE, 2, 4);

// Access the expression's internals using the model API
if (field instanceof QOM.Substring substring) {
    Field<String> string = substring.$string();
    Field<? extends Number> startingPosition = substring.$startingPosition();
    Field<? extends Number> length = substring.$length();
}
```

Every argument passed to the DSL API has a $ prefixed accessor method on the model API, exposing the wrapped argument. Using these accessor methods, users can traverse the expression tree manually or via the model API traversal API. More recent Java language features like pattern matching can be very helpful for such operations, especially as we're planning to seal the entire query object model API.

All of the model API is immutable, but new expressions can still be created using equivalent $ prefixed setter methods, which don't mutate the original expression but return a copy:

```
// Produces a substring(BOOK.TITLE, 2, 4) column expression
QOM.Substring substring = (QOM.Substring) substring(BOOK.TITLE, 2, 4);

// Produces a substring(BOOK.TITLE, 3, 5) column expression
substring.$startingPosition(val(3)).$length(val(5));
```

These basic operations allow for powerful SQL transformations on expression trees created with the DSL API but also with the SQL parser.

4.4.2. Traversal

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

While the accessor methods from the model API allow for traversing the expression tree manually, a more generic way to traverse the expression tree is using the org.jooq.Traverser API, which can traverse a org.jooq.QueryPart in a similar fashion as a java.util.stream.Collector can iterate a java.util.stream.Stream, collecting and aggregating data about the expression tree. A Traverser consists of this API:

```
public interface Traverser<A, R> {

    /**
     * A supplier for a temporary data structure to accumulate {@link QueryPart}
     * objects into during traversal.
     */
    Supplier<A> supplier();

    /**
     * An optional traversal abort condition to short circuit traversal e.g.
     * when the searched object has been found.
     */
    Predicate<A> abort();

    /**
     * An optional traversal abort condition to short circuit traversal e.g.
     * when the searched object has been found.
     */
    Predicate<QueryPart> recurse();

    /**
     * A callback that is invoked before recursing into a subtree.
     */
    BiFunction<A, QueryPart, A> before();

    /**
     * A callback that is invoked after recursing into a subtree.
     */
    BiFunction<A, QueryPart, A> after();

    /**
     * An optional transformation function to turn the temporary data structure
     * supplied by {@link #supplier()} into the final data structure.
     */
    Function<A, R> finisher();
}
```

Some elements are similar to that of a java.util.stream.Collector, others are specific to tree traversal. A simple illustration shows what can be done:

```
// Any ordinary QueryPart:
Condition condition = BOOK.ID.eq(1);
int count = condition.$traverse(

    // Supplier of the initial data structure: an int
    () -> 0,

    // Print all traversed QueryParts and increment the counter
    (r, q) -> {
        System.out.println("Part " + r + ": " + q);
        return r + 1;
    }
);
System.out.println("Count : " + count);
```

The above will print:

```
Part 0: "BOOK"."ID" = 1
Part 1: "BOOK"."ID"
Part 2: 1
Count : 3
```

Using the same traverser on a slightly more complex QueryPart

```
Condition condition = BOOK.ID.eq(1).or(BOOK.ID.eq(2));
// ...
```

```
Part 0: ("BOOK"."ID" = 1 or "BOOK"."ID" = 2)
Part 1: "BOOK"."ID" = 1
Part 2: "BOOK"."ID"
Part 3: 1
Part 4: "BOOK"."ID" = 2
Part 5: "BOOK"."ID"
Part 6: 2
Count : 7
```

Re-using your JDK collectors

Any Collector can be turned into a Traverser using Traversers.collecting(Collector). For example, if you want to count all QueryPart items in an expression, instead of the above hand-written traverser, just use the JDK Collectors.counting():

```
// Contains 3 query parts
long count1 = BOOK.ID.eq(1)
    .$traverse(Traversers.collecting(Collectors.counting());

// Contains 7 query parts
long count2 = BOOK.ID.eq(1).or(BOOK.ID.eq(2))
    .$traverse(Traversers.collecting(Collectors.counting());
```

4.4.3. Replacement

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

A very powerful way to transform your SQL is to replace specific org.jooq.QueryPart elements in any expression tree by something else using the QueryPart.replace() API. This API treats the expression tree as a persistent data structure, i.e. the resulting tree may consist of parts of the existing tree, but the existing tree is not modified.

Let's assume you wish to implement an optimisation engine that removes redundant SQL clauses. For example, an expression NOT(NOT(p)) can be replaced by p in standard SQL (it may not be the exact same thing in some "clever" dialects without standard BOOLEAN type support):

```
// Contains redundant operators
Condition c = not(not(BOOK.ID.eq(1)));
System.out.println(c);
System.out.println(c.$replace(q ->
    q instanceof QOM.Not n1 && n1.$arg1() instanceof QOM.Not n2
        ? n2.$arg1()
        : q
));
```

The above prints:

```
not (not ("BOOK"."ID" = 1))
"BOOK"."ID" = 1
```

The replacement algorithm will attempt to run the replacement function recursively on your tree until it no longer affects the tree. This means two things:

- You can implement all of your replacement logic in a single function, for various rules. The order of application of the rules is the one you define in your function.
- The algorithm stops only when no more rules apply. If two rules turn A > B and B > A, then the algorithm may never stop.

Here's a more complex example that logs the replacements with println() calls:

```
// Contains redundant operators
Condition c = not(not(not(BOOK.ID.ne(1))));
QueryPart result = c.$replace(q -> {
    if (q instanceof QOM.Not n1 && n1.$arg1() instanceof QOM.Not n2) {
        System.out.println("Replacing NOT(NOT(p)) by NOT(p): " + q);
        return n2.$arg1();
    }
    else if (q instanceof QOM.Not n1 && n1.$arg1() instanceof QOM.Ne<?> n2) {
        System.out.println("Replacing NOT(x != y) by x = y: " + q);
        return n2.transform(Field::eq);
    }

    return q;
}));
System.out.println("Result: " + result);
```

The output is:

```
Replacing NOT(x != y) by x = y: not ("BOOK"."ID" <> 1)
Replacing NOT(NOT(p)) by NOT(p): not (not ("BOOK"."ID" = 1))
Result: "BOOK"."ID" = 1
```

As you can see:

- The replacement function is invoked several times.
- The second invocation can work on the result of the first invocation, where the NOT (x != y) predicate has already been improved.
- The replacement works recursively, depth first, and bottom up.
- It stops when no more replacements take place.

This obviously also works when you use jOOQ's parser, and is extremely useful when used via the parsing connection, e.g. to optimise any type of JDBC or R2DBC based application!

```
// Contains redundant operators
Condition c = create.parser().parseCondition("not not not book.id != 1");
QueryPart result = c.$replace(q -> {
    if (q instanceof QOM.Not n1 && n1.$arg1() instanceof QOM.Not n2) {
        System.out.println("Replacing NOT(NOT(p)) by NOT(p): " + q);
        return n2.$arg1();
    }
    else if (q instanceof QOM.Not n1 && n1.$arg1() instanceof QOM.Ne<?> n2) {
        System.out.println("Replacing NOT(x != y) by x = y: " + q);
        return n2.transform(Field::eq);
    }

    return q;
}));
System.out.println("Result: " + result);
```

The result is exactly the same:

```
Replacing NOT(x != y) by x = y: not (book.id <> 1)
Replacing NOT(NOT(p)) by NOT(p): not (not (book.id = 1))
Result: book.id = 1
```

Pattern based replacement

Note that jOOQ offers a lot of out-of-the-box pattern based replacements like the above examples. Please look at the sections about pattern based transformation for more details.

4.4.4. The historic 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));
```

This API is completely mutable, and for historic reasons, early DSL API elements have inherited this mutability.

4.5. 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.

4.5.1. 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();
```

4.5.2. 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.

4.5.3. 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

```sql
-- 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
```

```java
// 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:

```java
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:

```java
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.

4.5.3.1. 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.

4.5.3.1.1. 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.

4.5.3.1.2. 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:

- org.jooq.Field: Every column expression can automatically be projected in SELECT as you would expect.
- org.jooq.Row: nested records can be projected in SELECT
- org.jooq.Table: tables can be projected as type safe nested records in SELECT
- MULTISET and other means of projecting nested collections can be projected as well

4.5.3.1.3. 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.

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.

4.5.3.1.4. 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.

Dialect support

This example using jOOQ:

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

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
SELECT LANGUAGE.CD, LANGUAGE.DESCRIPTION
FROM LANGUAGE

-- BIGQUERY
SELECT * EXCEPT (ID)
FROM LANGUAGE

-- H2
SELECT * EXCEPT (LANGUAGE.ID)
FROM LANGUAGE

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

4.5.3.1.5. 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
```

```
Select<?> select1 =
 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
```

4.5.3.1.6. 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.

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, COCKROACHDB, H2, POSTGRES, YUGABYTEDB
SELECT DISTINCT ON (BOOK.LANGUAGE_ID) BOOK.LANGUAGE_ID, BOOK.TITLE
FROM BOOK
ORDER BY BOOK.LANGUAGE_ID, BOOK.TITLE

-- DB2, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE,
-- SQLSERVER, SYBASE, TERADATA, 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

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, HSQLDB
/* UNSUPPORTED */
```

4.5.3.1.7. 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();
```

4.5.3.2. 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();
```

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:

4.5.3.3. 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.

4.5.3.4. Implicit 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. 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.

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.

Future versions of jOOQ may choose to generate correlated subqueries or inner joins where this may seem more appropriate, if the query semantics doesn't change through that.

Known limitations

- 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

4.5.3.5. 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.

4.5.3.6. 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();
```

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 and pseudo-columns are these (available from the DSL):

- LEVEL
- CONNECT_BY_IS_CYCLE
- CONNECT_BY_IS_LEAF
- CONNECT_BY_ROOT
- SYS_CONNECT_BY_PATH
- PRIOR

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();
```

4.5.3.7. 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. For instance, you can group books by BOOK.AUTHOR_ID:

```
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, as defined in the SQL standard, when grouping, you may no longer project any columns that are not a formal part of the GROUP BY clause, or aggregate functions.

Empty GROUP BY clauses

jOOQ supports empty GROUP BY () clause as well. This will result in SELECT statements that return only one record.

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

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

ROLLUP(), CUBE() and GROUPING SETS()

Some databases support the SQL standard grouping functions and some extensions thereof. See the manual's section about grouping functions for more details.

4.5.3.8. 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();
```

4.5.3.9. 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(PEOPLE.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/

4.5.3.10. 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:

```sql
SELECT AUTHOR_ID, count(*)
FROM BOOK
GROUP BY AUTHOR_ID
QUALIFY rank() OVER (ORDER BY count(*) DESC) <= 5
ORDER BY count(*) DESC
```

```java
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .qualify(rank().over(orderBy(count().desc())).le(5))
      .orderBy(count().desc())
      .fetch();
```

4.5.3.11. ORDER BY clause

Databases are allowed to return data in any arbitrary order, unless you explicitly declare that order in the ORDER BY clause. In jOOQ, this is straight-forward:

```sql
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY AUTHOR_ID ASC, TITLE DESC
```

```java
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.

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:

```sql
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY 1 ASC, 2 DESC
```

```java
create.select(BOOK.AUTHOR_ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(one().asc(), inline(2).desc())
      .fetch();
```

Note, how one() is used as a convenience short-cut for inline(1)

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 as follows

```
SELECT
  AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
FROM AUTHOR
ORDER BY LAST_NAME ASC,
         CASE WHEN FIRST_NAME IS NULL
              THEN 1 ELSE 0 END ASC,
         FIRST_NAME ASC
```

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(new HashMap<String, Integer>() {{
          put("1984", 1);
          put("Animal Farm", 13);
          put("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();
```

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.

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

4.5.3.12. 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.

4.5.3.13. 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 actor_id
FETCH NEXT 1 ROWS WITH TIES
```

```
DSL.using(configuration)
   .selectFrom(BOOK)
   .orderBy(BOOK.ACTOR_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.

Not all databases support WITH TIES. Oracle 12c supports the clause as specified in the SQL standard, and SQL Server knows TOP n WITH TIES without OFFSET support.

4.5.3.14. 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:

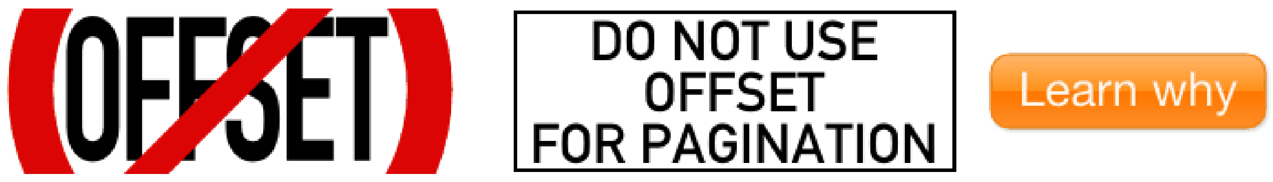
```
|  ID | VALUE |
|-----|-------|
| 640 |     2 |
| 776 |     2 |
| 815 |     2 |
| 947 |     2 |
|  37 |     3 |
```

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:



4.5.3.15. FOR clause

While both XML and JSON usage in SQL has been standardised in more recent versions of the SQL standard, SQL Server has always had some very convenient utilities at the end of a SELECT statement, which allow for converting SQL tables into the most common XML or JSON representations.

Starting with jOOQ 3.14, these syntaxes are supported in jOOQ as well, and if possible, emulated in other dialects which have native XML or JSON support.

FOR XML

Consider the following query

```sql
SELECT id, title
FROM book
ORDER BY id
FOR XML PATH ('book'), ROOT ('books')
```

```java
create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(BOOK.ID)
      .forXML().path("book").root("books")
      .fetch();
```

This query produces a document like this:

```
<books>
  <book><id>1</id><title>1984</title></book>
  <book><id>2</id><title>Animal Farm</title></book>
  <book><id>3</id><title>O Alquimista</title></book>
  <book><id>4</id><title>Brida</title></book>
</books>
```

FOR JSON

JSON is just XML with less syntax and less features. So the FOR JSON syntax in SQL Server is almost the same as the above FOR XML syntax:

```
SELECT id, title
FROM book
ORDER BY id
FOR JSON PATH
```

```
create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(BOOK.ID)
      .forJSON().path()
      .fetch();
```

This query produces a document like this:

```
[
  {"id": 1, "title": "1984"},
  {"id": 2, "title": "Animal Farm"},
  {"id": 3, "title": "O Alquimista"},
  {"id": 4, "title": "Brida"}
]
```

4.5.3.16. 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

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.

4.5.3.17. 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

4.5.3.17.1. 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
```

4.5.3.17.2. 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. This is particularly interesting when applying converters, including ad-hoc converters or converters attached to generated code.

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.

4.5.3.17.3. 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, ASE, 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

-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, EXASOL, HANA, HSQLDB, 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

-- DB2
(
  SELECT BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
)
UNION (
  SELECT AUTHOR.ID
  FROM AUTHOR
  ORDER BY AUTHOR.ID
  FETCH NEXT 1 ROWS ONLY
)
ORDER BY 1

-- DERBY, H2, MARIADB, ORACLE, POSTGRES
(
  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

-- FIREBIRD

  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 BOOK.ID
  FROM BOOK
  ORDER BY BOOK.ID
)
UNION (
  SELECT *
  FROM (
    SELECT FIRST 1 AUTHOR.ID
    FROM AUTHOR
    ORDER BY AUTHOR.ID
  ) x
)
ORDER BY 1

-- MEMSQL
SELECT
  t.*
FROM (
  (
    SELECT BOOK.ID
    FROM BOOK
```

4.5.3.17.4. UNION

A UNION operation combines two subquery results of compatible row type into a single result. While UNION removes all duplicate records resulting from this combination, UNION ALL leaves subselect results as they are. 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 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).union(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB,
-- MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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

-- 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
```

4.5.3.17.5. 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 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).intersect(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

```
-- ASE, AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB, MYSQL, ORACLE, POSTGRES,
-- SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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

-- 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 */
```

4.5.3.17.6. 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 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).except(select(AUTHOR.ID).from(AUTHOR)).orderBy(BOOK.ID)
```

Translates to the following dialect specific expressions:

```
-- ASE, AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB, MYSQL, POSTGRES, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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

-- 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 */
```

4.5.3.18. 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
- The SELECT clause: Only now, the projection is evaluated. In case of a SELECT DISTINCT statement, data is further reduced to remove duplicates
- 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 clause: Transformation to XML or JSON
- 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 idea at first, it only complicates translation to more advanced SQL statements while impairing readability for those users that are used to writing SQL. jOOQ is designed to look just like 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.

For these reasons, the jOOQ DSL API is modelled in SQL's lexical order.

4.5.4. 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.

4.5.4.1. 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

```sql
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse'),
       (101, 'Alfred', 'Döblin');
```

```java
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:

```sql
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME)
SELECT 100, 'Hermann', 'Hesse' FROM DUAL UNION ALL
SELECT 101, 'Alfred', 'Döblin' FROM DUAL;
```

```java
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(100, "Hermann", "Hesse")
      .values(101, "Alfred", "Döblin")
      .execute();
```

If your inserted rows or records are dynamic, you can use valuesOfRows() or valuesOfRecords() instead:

```sql
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse'),
       (101, 'Alfred', 'Döblin');
```

```java
List<Record3<Integer, String, String>> records = ...
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .valuesOfRecords(records)
      .execute();
```

4.5.4.2. 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:

```sql
INSERT INTO AUTHOR
DEFAULT VALUES;
```

```java
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:

```sql
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME, ...)
VALUES (
 DEFAULT,
 DEFAULT,
 DEFAULT, ...);
```

```java
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.

4.5.4.3. 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")
      .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.

4.5.4.4. 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();
```

4.5.4.5. INSERT .. ON DUPLICATE KEY

The synthetic ON DUPLICATE KEY UPDATE clause

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();
```

The synthetic ON DUPLICATE KEY IGNORE clause

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:

Emulating IGNORE with MERGE

The above jOOQ statement can be emulated with the following, equivalent SQL statement:

```
MERGE INTO AUTHOR
USING (SELECT 1 FROM DUAL)
ON (AUTHOR.ID = 3)
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
  VALUES (3, 'Koontz')
```

Emulating IGNORE with INSERT .. SELECT WHERE NOT EXISTS

The above jOOQ statement can be emulated with the following, equivalent SQL statement:

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 3, 'Koontz'
WHERE NOT EXISTS (
  SELECT 1
  FROM AUTHOR
  WHERE AUTHOR.ID = 3
)
```

4.5.4.6. 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.

4.5.5. 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 will be implemented in the near 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();
```

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:

```
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();
```

Using row value expressions in an UPDATE statement

jOOQ supports formal row value expressions in various contexts, among which the UPDATE statement. Only one row value expression can be updated at a time. Here's an example:

```
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 subselects:

```
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.

UPDATE .. FROM

Some databases, including 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.

UPDATE .. RETURNING

The Firebird and Postgres databases support a RETURNING clause 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);
```

The UPDATE .. RETURNING clause is emulated for DB2 using the SQL standard SELECT .. FROM FINAL TABLE(UPDATE ..) construct, and in Oracle, using the PL/SQL UPDATE .. RETURNING statement.

4.5.6. 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 will be implemented in the near future. An example delete query is given here:

```
DELETE AUTHOR
 WHERE ID = 100;
```

```
create.delete(AUTHOR)
      .where(AUTHOR.ID.eq(100))
      .execute();
```

4.5.7. The MERGE statement

The MERGE statement is one of the most advanced standardised SQL constructs, which is supported by DB2, HSQLDB, Oracle, SQL Server and Sybase (MySQL has the similar INSERT .. ON DUPLICATE KEY UPDATE construct)

The point of the standard MERGE statement is to take a TARGET table, and merge (INSERT, UPDATE) data from a SOURCE table into it. DB2, Oracle, SQL Server and Sybase also allow for DELETING some data and for adding many additional clauses. With jOOQ 3.17.8, only Oracle's MERGE extensions are supported. Here is an example:

```
-- Check if there is already an author called 'Hitchcock'
-- If there is, rename him to John. If there isn't add him.
MERGE INTO AUTHOR
USING (SELECT 1 FROM DUAL)
ON (LAST_NAME = 'Hitchcock')
WHEN MATCHED THEN UPDATE SET FIRST_NAME = 'John'
WHEN NOT MATCHED THEN INSERT (LAST_NAME) VALUES ('Hitchcock');
```

```
create.mergeInto(AUTHOR)
      .using(create.selectOne())
      .on(AUTHOR.LAST_NAME.eq("Hitchcock"))
      .whenMatchedThenUpdate()
      .set(AUTHOR.FIRST_NAME, "John")
      .whenNotMatchedThenInsert(AUTHOR.LAST_NAME)
      .values("Hitchcock")
      .execute();
```

Typesafety of VALUES() for degrees up to 22

Much like the INSERT statement, the MERGE statement's VALUES() clause provides typesafety for degrees up to 22, in both the standard syntax variant as well as the H2 variant.

4.6. 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.

4.6.1. The ALTER statement

ALTER statements are used to alter properties of existing objects in the database catalog.

4.6.1.1. ALTER DATABASE

The only property of a database that can be changed, currently, is its name. In order to alter an database's name, use:

```
// Renaming the database
create.alterDatabase("old_database").renameTo("new_database").execute();
```

4.6.1.2. ALTER DOMAIN

The ALTER DOMAIN statement allows for altering DOMAIN types for use as data types in table columns.

```
// Alter the default of a domain
create.alterDomain("d").setDefault(1).execute();
create.alterDomain("d").dropDefault().execute();

// Alter the NOT NULL constraint of a domain
create.alterDomain("d").setNotNull().execute();
create.alterDomain("d").dropNotNull().execute();

// Add / remove CHECK constraints to a domain
create.alterDomain("d").add(constraint("c").check(value(INTEGER).gt(0))).execute();
create.alterDomain("d").dropConstraint("c").execute();

// Rename the domain
create.alterDomain("d").renameTo("e").execute();

// Rename a constraint
create.alterDomain("d").renameConstraint("c1").to("c1").execute();
```

4.6.1.3. ALTER INDEX

The only property of an index that can be changed, currently, is its name. In order to alter an index's name, use:

```
// Renaming the index
create.alterIndex("old_index").renameTo("new_index").execute();
```

4.6.1.4. ALTER SCHEMA

The only property of a schema that can be changed, currently, is its name. In order to alter an schema's name, use:

```
// Renaming the schema
create.alterSchema("old_schema").renameTo("new_schema").execute();
```

4.6.1.5. ALTER SEQUENCE

The following types of statements are supported when altering a sequence:

Alter sequence properties

jOOQ supports a variety of sequence properties through meta data and DDL.

```
// Cache a number of values for the sequence, typically on a per session basis.
create.alterSequence("sequence").cache(200).execute();
create.alterSequence("sequence").noCache().execute();

// Specify whether the sequence should cycle when it reaches the MAXVALUE
create.alterSequence("sequence").cycle().execute();
create.alterSequence("sequence").noCycle().execute();

// The increment by which a sequence should produce the next value
create.alterSequence("sequence").incrementBy(10).execute();

// The MAXVALUE before which the sequence should cycle if applicable
create.alterSequence("sequence").maxvalue(1000).execute();
create.alterSequence("sequence").noMaxvalue.execute();

// The MINVALUE from which the sequence should cycle if applicable
create.alterSequence("sequence").minvalue(1).execute();
create.alterSequence("sequence").noMinvalue.execute();

// Let the sequence restart with MINVALUE or with a specific value
create.alterSequence(S_AUTHOR_ID).restart().execute();
create.alterSequence(S_AUTHOR_ID).restartWith(1).execute();

// Let the sequence start with a specific value
create.alterSequence(S_AUTHOR_ID).startWith(1).execute();
```

RENAME

Like most object types, sequences can be renamed:

```
// Renaming the sequence
create.alterSequence("old_sequence").renameTo("new_sequence").execute();
```

4.6.1.6. 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. The following types of statements are supported when altering a table:

ADD

In most dialects, tables can contain two types of objects:

- Columns
- Constraints

These types of objects can be added to a table using the following API:

```
// Adding a single column to a table
create.alterTable("table").add("column", INTEGER).execute();

// Adding several columns to a table in one go
create.alterTable("table").add(field(name("column1"), INTEGER), field(name("column2"), INTEGER)).execute();

// Adding an unnamed constraint to a table
create.alterTable("table").add(primaryKey("id")).execute();
create.alterTable("table").add(unique("user_name")).execute();
create.alterTable("table").add(foreignKey("author_id").references("author")).execute();
create.alterTable("table").add(check(length(field(name("user_name"), VARCHAR)).gt(5))).execute();

// Adding a named constraint to a table
create.alterTable("table").add(constraint("pk").primaryKey("id")).execute();
create.alterTable("table").add(constraint("uk").unique("user_name")).execute();
create.alterTable("table").add(constraint("fk").foreignKey("author_id").references("author")).execute();
create.alterTable("table").add(constraint("ck").check(length(field(name("user_name"), VARCHAR)).gt(5))).execute();
```

There exists alternative API representing optional keywords, such as e.g. addColumn(), which have been omitted from the examples.

It is possible to specify the column ordering when adding new columns, where this is supported:

```
// Adding a single column and specify its position
create.alterTable("table").add("column", INTEGER).after("other_column").execute();
create.alterTable("table").add("column", INTEGER).before("other_column").execute();
create.alterTable("table").add("column", INTEGER).first().execute();
```

Note that some dialects also consider indexes to be a part of a table, but jOOQ does not yet support ALTER TABLE subclauses modifying indexes. Consider CREATE INDEX, ALTER INDEX, or DROP INDEX, instead.

ALTER

Both of the above objects can be altered in a table using the following API:

```
// Specify a new default value for a column
create.alterTable("table").alter("column").default_(1).execute();
create.alterTable("table").alter("column").dropDefault().execute();

// Specify the not null constraint on a column
create.alterTable("table").alter("column").setNotNull().execute();
create.alterTable("table").alter("column").dropNotNull().execute();

// Set a new data type on the column
create.alterTable("table").alter("column").set(VARCHAR(50)).execute();

// Set the enforced flag on a constraint
create.alterTable("table").alterConstraint("uk").enforced().execute();
create.alterTable("table").alterConstraint("uk").notEnforced().execute();
```

There exists alternative API representing optional keywords, such as e.g. alterColumn(), which have been omitted from the examples.

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();
```

DROP

Both columns and constraints can also be dropped from tables using this API:

```
// Drop a single column
create.alterTable("table").drop("column").execute();

// Drop several columns in one go
create.alterTable("table").drop("column1", "column2").execute();

// Add CASCADE or RESTRICT clauses when dropping columns (or constraints)
create.alterTable("table").drop("column").cascade().execute();
create.alterTable("table").drop("column").restrict().execute();

// Drop a constraint
create.alterTable("table").dropConstraint("uk").execute();

// Drop specific types of constraints (e.g. if the above syntax is not supported by the dialect)
create.alterTable("table").dropPrimaryKey().execute();
create.alterTable("table").dropUnique("uk").execute();
create.alterTable("table").dropForeignKey("fk").execute();
```

RENAME

Like most object types, tables, columns, and constraints can be renamed:

```
// Rename a table
create.alterTable("old_table").renameTo("new_table").execute();

// Rename a column
create.alterTable("table").renameColumn("old_column").to("new_column").execute();

// Rename a constraint
create.alterTable("table").renameConstraint("old_constraint").to("new_constraint").execute();

// Rename a index (as a convenience for the ALTER INDEX statement)
create.alterTable("table").renameIndex("old_index").to("new_index").execute();
```

4.6.1.7. ALTER TYPE

The following types of statements are supported when altering a type:

RENAME

Like most object types, types can be renamed. This is independent of the object type:

```
// Renaming the sequence
create.alterType("old_type").renameTo("new_type").execute();
```

Enum type alterations

Some alterations are specific to enum types, e.g. in PostgreSQL. These include:

```
// Adding an enum value to an existing type
create.alterType("type").addValue("new_enum_value").execute();

// Renaming an enum value of an existing type to a new value
create.alterType("type").renameValue("old_enum_value").to("new_enum_value").execute();

// Move an enum type to a new schema
create.alterType("type").setSchema("new_schema").execute();
```

4.6.1.8. ALTER VIEW

The following types of statements are supported when altering a view:

COMMENT

For convenience, jOOQ supports MySQL's COMMENT syntax also on views, despite MySQL currently not supporting this. It can be emulated using the COMMENT ON VIEW statement

```
// Renaming the sequence
create.alterView("old_view").comment("a comment describing the view").execute();
```

RENAME

Like most object types, views can be renamed:

```
// Renaming the sequence
create.alterView("old_view").renameTo("new_name").execute();
```

4.6.2. The COMMENT statement

The COMMENT statement can be used to store a description for an object from the database catalog.

These comments will be picked up by the code generator and are generated as Javadoc on generated classes.

```
// Commenting a table
create.commentOnTable("table").is("a comment describing the table").execute();

// Commenting a view
create.commentOnView("view").is("a comment describing the view").execute();

// Commenting a column
create.commentOnColumn(name("table", "column")).is("a comment describing the column").execute();
```

4.6.3. The CREATE statement

The CREATE statement is the most important DDL statement. It allows for creating new objects in the database catalog.

4.6.3.1. CREATE DATABASE

The CREATE DATABASE statement is used to create a new database (catalog).

```
// Create a database
create.createDatabase("new_database").execute();
```

4.6.3.2. CREATE DOMAIN

The CREATE DOMAIN statement allows for creating DOMAIN types for use as data types in table columns.

Depending on the dialect, a DOMAIN combines the following features:

- A qualified name
- A base data type
- A DEFAULT value
- A NOT NULL constraint
- A COLLATION
- A set of CHECK constraints

Domains can be created in jOOQ using:

```
// Create a domain on a base type
create.createDomain("d1").as(INTEGER).execute();

// Create a domain on a base type and add a DEFAULT expression
create.createDomain("d2").as(INTEGER).default_(1).execute();

// Create a domain on a base type and add a CHECK constraint
create.createDomain("d3").as(INTEGER).constraints(check(value(INTEGER).gt(0))).execute();
```

4.6.3.3. CREATE FUNCTION

The CREATE FUNCTION statement allows for creating stored functions in your catalog.

A stored function, as opposed to a stored procedure, can be used in SQL statements as a scalar column expression, specifically a user defined function, or as a table expression, specifically a table valued functions.

4.6.3.3.1. Scalar functions

The most common type of user defined function is a scalar function, i.e. a function that returns a single scalar value. Such functions can be used in the SELECT clause, the WHERE clause, the GROUP BY clause, the HAVING clause, the ORDER BY clause, and elsewhere, where column expressions can be used.

A simple example for creating such a function is:

```
// Create a function that always return 1
create.createFunction("one")
      .returns(INTEGER)
      .as(return_(1))
      .execute();

// Create a function that returns the sum of its inputs
Parameter<Integer> i1 = in("i1", INTEGER);
Parameter<Integer> i2 = in("i2", INTEGER);

create.createFunction("my_sum")
      .parameters(i1, i2)
      .returns(INTEGER)
      .as(return_(i1.plus(i2)))
      .execute();
```

Once you've created the above functions, you can either run code generation to get a type safe stub for calling them, or use plain SQL (specifically, DSL.function()) from within a SELECT statement:

```
// Call the previously created functions with generated code:
create.select(one(), mySum(1, 2)).fetchOne();

// ...or with plain SQL
create.select(
  function(name("one"), INTEGER),
  function(name("my_sum"), INTEGER, val(1), val(2))
).fetchOne();
```

Both yielding:

```
+-----+--------+
| ONE | MY_SUM |
+-----+--------+
|   1 |      3 |
+-----+--------+
```

4.6.3.3.2. CREATE OR REPLACE FUNCTION

In most cases, you will want to CREATE OR REPLACE a function, not CREATE [OR FAIL] when the function already exists. For this, just use the auxiliary OR REPLACE clause, which can be emulated by jOOQ via an additional DROP FUNCTION statement if this syntax is not available in your dialect.

```
// Create a function that always return 1
create.createOrReplaceFunction("one")
      .returns(INTEGER)
      .as(return_(1))
      .execute();
```

4.6.3.3.3. SQL data access characteristics

Some dialects require the explicit specification of a few characteristics of a function, defining what kind of content a function is allowed (and expected) to have. These act both as contracts for your development (similar to the java.lang.FunctionalInterface annotation), as well as hints to the database regarding whether a function is expected to have side-effects (and thus maybe cannot be used in a SELECT statement), or whether it depends on data, or is purely deterministic (see also the DETERMINISTIC characteristic). The SQL data access characteristics include:

- NO SQL
- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

While the semantics seem pretty clear, please refer to your database manual for the details, as there may be subtle differences, e.g. regarding what particular procedural statement constitues "SQL".

If a characteristic is not supported by your dialect, you can still specify it, and jOOQ will simply ignore it in generated SQL.

```
create.createFunction("f1").returns(INTEGER).noSQL().as(return_(1)).execute();
create.createFunction("f2").returns(INTEGER).containsSQL().as(return_(select(val(1)))).execute();
create.createFunction("f3").returns(INTEGER).readsSQLData().as(return_(selectCount().from(BOOK))).execute();
create.createFunction("f4").returns(INTEGER).modifiesSQLData().as(
  insertInto(LOGS).columns(LOGS.TEXT).values("Function F4 was called"),
  return_(1)
).execute();
```

This works just like SQL data access characteristics for procedures

4.6.3.3.4. DETERMINISTIC characteristic

Some dialects require the explicit specification of a few characteristics of a function. The DETERMINISTIC characteristic can be used to tell the database that contents of a function are guaranteed by the user to be "deterministic" (or "IMMUTABLE" in PostgreSQL), meaning that the result of a function is purely defined by the function arguments (never by data or session values or the current time or random number generators, etc.) such that a function expression can be replaced by the result value at the call site. Such a function is also said to be side-effect free, and pure. Some dialects (e.g. Oracle) allow for using DETERMINISTIC functions in function based indexes.

If DETERMINISTIC is not supported by your dialect, you can still specify it, and jOOQ will simply ignore it in generated SQL.

```
create.createFunction("f1").returns(INTEGER).deterministic().as(return_(1)).execute();
create.createFunction("f2").returns(INTEGER).notDeterministic().as(return_(rand())).execute();
```

4.6.3.3.5. ON NULL INPUT characteristic

This characteristic both determines a contract for use by optimisers, as well as influences the behaviour of a function.

Most built-in SQL functions return NULL as soon as any of the arguments are NULL. For example SUBSTRING:

```
create.select(
  substring("abc"            , 2)                 .as("s1"),
  substring("abc"            , val(null, INTEGER)).as("s2"),
  substring(val(null, VARCHAR), 2)                 .as("s3"),
  substring(val(null, VARCHAR), val(null, INTEGER)).as("s4"),
).fetchOne();
```

Yielding:

```
+----+--------+--------+--------+
| s1 | s2     | s3     | s4     |
+----+--------+--------+--------+
| bc | {null} | {null} | {null} |
+----+--------+--------+--------+
```

While it is easy to implement this manually, it is both convenient, and helpful for optimisers, to just use the characteristic to achieve this standard behaviour, and possibly even to prevent calling the function, preventing the overhead from the context switches, etc.

If not natively supported by your dialect, jOOQ will simply wrap your function body in a IF statement checking for argument value nullability.

```
Parameter<Integer> i1 = in("i1", INTEGER);
Parameter<Integer> i2 = in("i2", INTEGER);

// The function always returns NULL if any argument value is NULL
create.createFunction("my_sum")
      .parameters(i1, i2)
      .returns(INTEGER)
      .returnsNullOnNullInput()
      .as(return_(i1.plus(i2)))
      .execute();

// The function may not return NULL if any argument value is NULL
create.createFunction("my_null_safe_sum")
      .parameters(i1, i2)
      .returns(INTEGER)
      .calledOnNullInput()
      .as(return_(coalesce(i1, 0).plus(coalesce(i2, 0))))
      .execute();
```

4.6.3.4. 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();
```

4.6.3.5. CREATE PROCEDURE

The CREATE PROCEDURE statement allows for creating stored procedures in your catalog.

A stored procedure, as opposed to a stored function, is expected to have side effects of some sort, and can thus not be used in SQL statements.

A simple example for creating such a procedure is:

```
// Create a procedure that inserts a log message in a table
create.createProcedure("log")
      .as(insertInto(LOG).columns(LOG.TEXT).values("Log called"))
      .execute();
```

Once you've created the above procedure, you can either run code generation to get a type safe stub for calling them, or use CALL in an anonymous block, or directly:

```
// Call the previously created procedure with generated code:
log(configuration);

// ...or with the CALL statement in an anonymous block
create.begin(call(name("log"))).execute();

// ...or with the CALL statement directly
call(name("log")).execute();
```

4.6.3.5.1. CREATE OR REPLACE PROCEDURE

In most cases, you will want to CREATE OR REPLACE a procedure, not CREATE [OR FAIL] when the procedure already exists. For this, just use the auxiliary OR REPLACE clause, which can be emulated by jOOQ via an additional DROP PROCEDURE statement if this syntax is not available in your dialect.

```
// Create a procedure that inserts a log message in a table
create.createOrReplaceProcedure("log")
      .as(insertInto(LOG).columns(LOG.TEXT).values("Log called"))
      .execute();
```

4.6.3.5.2. SQL data access characteristics

Some dialects require the explicit specification of a few characteristics of a procedure, defining what kind of content a procedure is allowed (and expected) to have. These act both as contracts for your development (similar to the java.lang.FunctionalInterface annotation), as well as hints to the database regarding whether a procedure is expected to have side-effects (and thus maybe cannot be used indirectly via a stored function in a SELECT statement), or whether it depends on data, or is purely deterministic. The SQL data access characteristics include:

While procedures are generally expected to yield side effects, it may be useful to use a procedure as a "function with quirky syntax" to be consumed by other functions, because it can return several OUT parameters, instead of just a single RETURN value, like function, hence the utility of these characteristics also in procedures.

- NO SQL
- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

While the semantics seem pretty clear, please refer to your database manual for the details, as there may be subtle differences, e.g. regarding what particular procedural statement constitues "SQL".

If a characteristic is not supported by your dialect, you can still specify it, and jOOQ will simply ignore it in generated SQL.

```
Parameter<Integer> o = out("o", INTEGER);

create.createProcedure("p1")
      .parameters(o)
      .noSQL()
      .as(o.set(1))
      .execute();

create.createProcedure("p2")
      .parameters(o)
      .containsSQL()
      .as(o.set(select(val(1))))
      .execute();

create.createProcedure("p3")
      .parameters(o)
      .readsSQLData()
      .as(o.set(selectCount().from(BOOK)))
      .execute();

create.createProcedure("p4")
      .parameters(o)
      .modifiesSQLData()
      .as(
        insertInto(LOGS).columns(LOGS.TEXT).values("Function F4 was called"),
        o.set(1)
      )
      .execute();
```

This works just like SQL data access characteristics for functions

4.6.3.6. 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();
```

4.6.3.7. 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();
```

Sequence flags

Many dialects support standard SQL sequence flag in CREATE SEQUENCE and also ALTER SEQUENCE statements:

All of these flags can be combined in a single CREATE SEQUENCE statement.

```
// Cache a number of values for the sequence, typically on a per session basis.
create.createSequence("sequence").cache(200).execute();
create.createSequence("sequence").noCache().execute();

// Specify whether the sequence should cycle when it reaches the MAXVALUE
create.createSequence("sequence").cycle().execute();
create.createSequence("sequence").noCycle().execute();

// The increment by which a sequence should produce the next value
create.createSequence("sequence").incrementBy(10).execute();

// The MAXVALUE before which the sequence should cycle if applicable
create.createSequence("sequence").maxvalue(1000).execute();
create.createSequence("sequence").noMaxvalue.execute();

// The MINVALUE from which the sequence should cycle if applicable
create.createSequence("sequence").minvalue(1).execute();
create.createSequence("sequence").noMinvalue.execute();

// Let the sequence start with a specific value
create.createSequence(S_AUTHOR_ID).startWith(1).execute();
```

4.6.3.8. 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.

4.6.3.8.1. 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, EXASOL, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, VERTICA, YUGABYTEDB
CREATE TABLE table (
  col1 int
)

-- BIGQUERY
CREATE TABLE table (
  col1 int64
)

-- COCKROACHDB
CREATE TABLE table (
  col1 int4
)

-- ORACLE, SNOWFLAKE
CREATE TABLE table (
  col1 number(10)
)

-- SQLITE
CREATE TABLE "table" (
  col1 int
)
```

4.6.3.8.2. 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.null_())
      .column("explicit_not_nullable", INTEGER.notNull())
      .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, EXASOL, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLSERVER,
-- VERTICA, YUGABYTEDB
CREATE TABLE table (
  vendor_specific_default int,
  explicit_nullable int NULL,
  explicit_not_nullable int NOT NULL
)

-- BIGQUERY
CREATE TABLE table (
  vendor_specific_default int64,
  explicit_nullable int64,
  explicit_not_nullable int64 NOT NULL
)

-- COCKROACHDB
CREATE TABLE table (
  vendor_specific_default int4,
  explicit_nullable int4 NULL,
  explicit_not_nullable int4 NOT NULL
)

-- 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
)

-- ORACLE, SNOWFLAKE
CREATE TABLE table (
  vendor_specific_default number(10),
  explicit_nullable number(10) NULL,
  explicit_not_nullable number(10) NOT NULL
)

-- SQLITE
CREATE TABLE "table" (
  vendor_specific_default int,
  explicit_nullable int NULL,
  explicit_not_nullable int NOT NULL
)
```

4.6.3.8.3. 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, EXASOL, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, VERTICA, YUGABYTEDB
CREATE TABLE table (
  column1 int DEFAULT 1
)

-- COCKROACHDB
CREATE TABLE table (
  column1 int4 DEFAULT 1
)

-- 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
)

-- BIGQUERY
/* UNSUPPORTED */
```

4.6.3.8.4. 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 integer DEFAULT (unique_rowid() % 2 ^ 31) NOT NULL
)

-- 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
)

-- 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
/* UNSUPPORTED */
```

4.6.3.8.5. Computed columns

Computed columns, sometimes also called "virtual" columns, are columns that are generated from an expression based on other columns of the same row directly in the database. They cannot be written to, but may be used in projections, filters, and even indexes, as a complement or replacement of function based indexes.

Like any other data type modifying flag, the generator expression can be passed to the data type in jOOQ when creating such a table with computed columns:

Dialect support

This example using jOOQ:

```
createTable(name("x"))
   .column(name("interest"), DOUBLE)
   .column(name("interest_percent"), VARCHAR.generatedAlwaysAs(field(name("interest"), DOUBLE).times(100.0).concat(" %")))
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES
CREATE TABLE x (
  interest double precision,
  interest_percent varchar GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar) || ' %'))
)

-- COCKROACHDB
CREATE TABLE x (
  interest double precision,
  interest_percent string GENERATED ALWAYS AS ((CAST((interest * CAST(1E2 AS double precision)) AS string) || ' %')) STORED
)

-- DB2, HSQLDB
CREATE TABLE x (
  interest double,
  interest_percent varchar(32672) GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar(32672)) || ' %'))
)

-- DERBY
CREATE TABLE x (
  interest double,
  interest_percent varchar(32672) GENERATED ALWAYS AS ((TRIM(CAST(CAST((interest * 1E2) AS char(38)) AS varchar(32672))) || ' %'))
)

-- FIREBIRD
CREATE TABLE x (
  interest double precision,
  interest_percent varchar(4000) GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar(4000)) || ' %'))
)

-- H2
CREATE TABLE x (
  interest double,
  interest_percent varchar AS ((CAST((interest * CAST(1E2 AS double)) AS varchar) || ' %'))
)

-- HANA
CREATE TABLE x (
  interest double,
  interest_percent varchar GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar) || ' %'))
)

-- MARIADB, MYSQL
CREATE TABLE x (
  interest double,
  interest_percent text GENERATED ALWAYS AS (concat(
    CAST((interest * 1E2) AS char),
    ' %'
  ))
)

-- ORACLE
CREATE TABLE x (
  interest float,
  interest_percent varchar2(4000) GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar2(4000)) || ' %'))
)

-- POSTGRES
CREATE TABLE x (
  interest double precision,
  interest_percent varchar GENERATED ALWAYS AS ((CAST((interest * 1E2) AS varchar) || ' %')) STORED
)

-- SQLSERVER
CREATE TABLE x (
  interest float,
  interest_percent AS (CAST((interest * 1E2) AS varchar(max)) + ' %')
)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, EXASOL, INFORMIX, MEMSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE,
-- TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.6.3.8.6. 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)
      .primaryKey("column1")
      .execute();

// Equivalent to the above
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, SYBASE
CREATE TABLE table (
  column1 int NULL,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- AURORA_MYSQL, AURORA_POSTGRES, DERBY, EXASOL, H2, HSQLDB, POSTGRES, REDSHIFT, YUGABYTEDB
CREATE TABLE table (
  column1 int,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- BIGQUERY
CREATE TABLE table (
  column1 int64
)

-- COCKROACHDB
CREATE TABLE table (
  column1 int4,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- DB2, TERADATA
CREATE TABLE table (
  column1 integer NOT NULL,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- INFORMIX
CREATE TABLE table (
  column1 integer,
  PRIMARY KEY (column1) CONSTRAINT pk
)

-- MARIADB, MEMSQL, MYSQL, SQLSERVER
CREATE TABLE table (
  column1 int NOT NULL,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- 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
)

-- SQLITE
CREATE TABLE "table" (
  column1 int,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)

-- VERTICA
CREATE TABLE table (
  column1 int,
  dummy int,
  CONSTRAINT pk
    PRIMARY KEY (column1)
)
```

4.6.3.8.7. 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)
      .unique("column1")
      .unique("column2", "column3")
      .execute();

// Equivalent to the above
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, EXASOL, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLSERVER,
-- VERTICA, YUGABYTEDB
CREATE TABLE table (
  column1 int,
  CONSTRAINT uk
    UNIQUE (column1)
)

-- BIGQUERY
CREATE TABLE table (
  column1 int64
)

-- COCKROACHDB
CREATE TABLE table (
  column1 int4,
  CONSTRAINT uk
    UNIQUE (column1)
)

-- INFORMIX
CREATE TABLE table (
  column1 integer,
  UNIQUE (column1) CONSTRAINT uk
)

-- ORACLE, SNOWFLAKE
CREATE TABLE table (
  column1 number(10),
  CONSTRAINT uk
    UNIQUE (column1)
)

-- SQLDATAWAREHOUSE
CREATE TABLE table (
  column1 int,
  CONSTRAINT uk
    UNIQUE (column1) NOT ENFORCED
)

-- SQLITE
CREATE TABLE "table" (
  column1 int,
  CONSTRAINT uk
    UNIQUE (column1)
)
```

4.6.3.8.8. 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 joins and 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, EXASOL, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, VERTICA, YUGABYTEDB
CREATE TABLE table (
  column1 int,
  CONSTRAINT fk
    FOREIGN KEY (column1)
    REFERENCES other_table (other_column1)
)

-- BIGQUERY
CREATE TABLE table (
  column1 int64
)

-- COCKROACHDB
CREATE TABLE table (
  column1 int4,
  CONSTRAINT fk
    FOREIGN KEY (column1)
    REFERENCES other_table (other_column1)
)

-- 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)
)

-- SQLITE
CREATE TABLE "table" (
  column1 int,
  CONSTRAINT fk
    FOREIGN KEY (column1)
    REFERENCES other_table (other_column1)
)
```

4.6.3.8.9. 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 (or even a DOMAIN that contains a CHECK constraint).

```
// Create a new table with columns and unnamed constraints
create.createTable("table")
      .column("column1", INTEGER)
      .check(field(name("column1"), INTEGER).gt(0))
      .execute();

// Equivalent to the above
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)
)

-- AURORA_POSTGRES, DERBY, H2, HSQLDB, MARIADB, MYSQL, POSTGRES, SQLSERVER, VERTICA, YUGABYTEDB
CREATE TABLE table (
  column1 int,
  CONSTRAINT chk
    CHECK (column1 > 0)
)

-- 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)
)

-- SQLITE
CREATE TABLE "table" (
  column1 int,
  CONSTRAINT chk
    CHECK (column1 > 0)
)

-- AURORA_MYSQL, BIGQUERY, EXASOL, MEMSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE
/* UNSUPPORTED */
```

4.6.3.8.10. 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:

```
-- ACCESS, ASE, 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

-- COCKROACHDB, H2, MARIADB, MYSQL, SNOWFLAKE
CREATE TABLE book_archive
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

-- HANA, HSQLDB
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
)
WITH NO DATA

-- TERADATA
CREATE TABLE book_archive
AS (
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
  WHERE 1 = 0
)
WITH DATA

-- BIGQUERY, FIREBIRD, INFORMIX, SYBASE
/* UNSUPPORTED */
```

4.6.3.8.11. Temporary tables

Many dialects support different notions of "temporary" tables, i.e. tables whose data and/or meta data is stored only temporarily. The details of these temporary are implementation specific. jOOQ supports the following syntaxes, both with explicit column lists or as CREATE TABLE AS SELECT:

```
// Create a new temporary table
create.createTemporaryTable("book_archive")
      .column("column1", INTEGER)
      .execute();

// Create a new temporary table
create.createGlobalTemporaryTable("book_archive")
      .column("column1", INTEGER)
      .execute();
```

Dialect support

This example using jOOQ:

```
createTemporaryTable("book_archive")
      .column("column1", INTEGER)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, YUGABYTEDB
CREATE TEMPORARY TABLE book_archive (
  column1 int
)

-- COCKROACHDB
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 int4
)

-- FIREBIRD, HANA, TERADATA
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 integer
)

-- ORACLE, SNOWFLAKE
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 number(10)
)

-- VERTICA
CREATE GLOBAL TEMPORARY TABLE book_archive (
  column1 int
)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, H2, HSQLDB, INFORMIX, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE
/* UNSUPPORTED */
```

4.6.3.9. CREATE TRIGGER

Most dialects support triggers, which is SQL or procedural logic that executes at certain events to perform actions including:

- Changing the data about to be inserted or updated (e.g. when a column DEFAULT in a table definition doesn't do the trick)
- Performing additional integrity checks prior to any DML statement (e.g. when a CHECK constraint doesn't suffice)
- Logging extra data, such as audit data.
- Etc.

4.6.3.9.1. Events

Most dialects supporting triggers can fire ...

- BEFORE
- AFTER
- INSTEAD OF

... a certain event, including ...

- INSERT
- UPDATE
- DELETE

Note that the above refer to events, not the statements, meaning that a trigger may fire also for the MERGE statement.

Some examples illustrating possible triggers:

```
create.createTrigger("trg1")
      .beforeInsert()
      .on(BOOK)
      .forEachRow()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row inserted in BOOK"))
      .execute();

create.createTrigger("trg")
      .beforeUpdate()
      .on(BOOK)
      .forEachRow()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row updated in BOOK"))
      .execute();

create.createTrigger("trg")
      .beforeDelete()
      .on(BOOK)
      .forEachRow()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row deleted in BOOK"))
      .execute();

create.createTrigger("trg")
      .beforeInsert().orUpdate().orDelete()
      .on(BOOK)
      .forEachRow()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row deleted in BOOK"))
      .execute();
```

4.6.3.9.2. REFERENCING clause

A trigger is executed while a data mutation is being executed. During this time, it is possible for a trigger to see a row or table's state before (OLD pseudo table) or after (NEW pseudo table) the modification. Specifically:

- INSERT: Only NEW is available
- UPDATE: Both OLD and NEW are available
- DELETE: Only OLD is available

In the rare event when the default OLD or NEW pseudo table identifiers conflict with actual tables in the schema, the REFERENCING clause can be used to rename these identifiers for the scope of a trigger. In some dialects, REFERENCING is always mandatory, and in others, it's not supported at all.

An example would be

```
 // A trigger that prevents the update of NULL titles in BOOK
create.createTrigger("trg")
      .beforeUpdate().of(BOOK.TITLE)
      .on(BOOK)
      .referencingOldAs("o")
      .referencingNewAs("n")
      .forEachRow()
      .as(if_(BOOK.as("o").TITLE.isNull()).then(
            variable(BOOK.as("n").TITLE.getQualifiedName(), BOOK.TITLE.getDataType()).setNull()
      ))
      .execute();
```

4.6.3.9.3. STATEMENT vs ROW triggers

A trigger can choose whether it fires only once for entire sets of data being changed (FOR EACH STATEMENT), or for each individual row (FOR EACH ROW). The specific semantics of each of these clauses, and what's possible to do with each is vendor specific. Please refer to your database manual to see what's possible here.

An example of such triggers:

```
create.createTrigger("trg")
      .beforeInsert()
      .on(BOOK)
      .forEachRow()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row inserted in BOOK"))
      .execute();

create.createTrigger("trg")
      .beforeInsert()
      .on(BOOK)
      .forEachStatement()
      .as(insertInto(LOG).columns(LOG.TEXT).values("Rows inserted in BOOK"))
      .execute();
```

4.6.3.9.4. WHEN clause

A trigger can specify a filter, which helps avoid context switching, etc. when the trigger does not need to be fired. This can be achieved with the WHEN clause, which can access the OLD and NEW pseudo table columns, see also the REFERENCING clause.

```
create.createTrigger("trg")
      .beforeInsert()
      .on(BOOK)
      .referencingNewAs("n")
      .forEachRow()
      .when(BOOK.as("n").TITLE.isNotNull())
      .as(insertInto(LOG).columns(LOG.TEXT).values("Row inserted in BOOK"))
      .execute();
```

4.6.3.10. 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();
```

4.6.3.11. 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();
```

Dialect support

This example using jOOQ:

```
createView("a", "id").as(select(AUTHOR.ID).from(AUTHOR))
```

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB,
-- INFORMIX, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR

-- MEMSQL
CREATE VIEW a
AS
SELECT t.id
FROM (
  SELECT AUTHOR.ID id
  FROM AUTHOR
) t
```

4.6.3.11.1. 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, COCKROACHDB, DERBY, EXASOL, H2, HSQLDB, MEMSQL, REDSHIFT, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLITE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.6.3.11.2. 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, SQLITE, SQLSERVER, 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, COCKROACHDB, H2, MARIADB, MYSQL, POSTGRES, SNOWFLAKE, YUGABYTEDB
CREATE VIEW a(id)
AS
SELECT AUTHOR.ID
FROM AUTHOR
UNION
SELECT NULL
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

-- 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

-- 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 NULL
FROM (VALUES(1)) AS dual(dual)
WHERE FALSE
```

4.6.4. The DROP statement

The DROP statement is used to drop objects from the database catalog.

4.6.4.1. DROP DATABASE

This statement is used to drop a DATABASE.

```
// Drop a database
create.dropDatabase("database").execute();
```

4.6.4.2. DROP DOMAIN

The DROP DOMAIN statement allows for droping DOMAIN types.

```
create.dropDomain("d").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();
```

Please refer to your database manual for an understanding of the semantics of CASCADE (e.g. in PostgreSQL, all referencing columns are dropped!)

4.6.4.3. DROP FUNCTION

This statement is used to drop a FUNCTION from the database catalog.

```
// Drop a function
create.dropFunction("func").execute();

// Drop a function if it exists
create.dropFunctionIfExists("func").execute();
```

4.6.4.4. 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();
```

4.6.4.5. DROP PROCEDURE

This statement is used to drop a PROCEDURE from the database catalog.

```
// Drop a procedure
create.dropProcedure("proc").execute();

// Drop a procedure if it exists
create.dropProcedureIfExists("procedure").execute();
```

4.6.4.6. DROP SCHEMA

This statement is used to drop an 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();
```

4.6.4.7. DROP SEQUENCE

This statement is used to drop an SEQUENCE from the database catalog.

```
// Drop a sequence
create.dropSequence("sequence").execute();
```

4.6.4.8. DROP TABLE

This statement is used to drop an TABLE from the database catalog.

```
// Drop a table
create.dropTable("table").execute();
```

CASCADE

It is possible to supply a CASCADE or RESTRICT clause, explicitly

```
// Specify the CASCADE / RESTRICT clauses explicitly
create.dropTable("table").cascade().execute();
create.dropTable("table").restrict().execute();
```

4.6.4.9. DROP TRIGGER

This statement is used to drop a TRIGGER from the database catalog.

```
// Drop a trigger
create.dropTrigger("trg").execute();

// Drop a trigger if it exists
create.dropTriggerIfExists("trg").execute();
```

4.6.4.10. DROP TYPE

This statement is used to drop an 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();
```

4.6.4.11. DROP VIEW

This statement is used to drop an VIEW from the database catalog.

```
// Drop a view
create.dropView("view").execute();
```

4.6.5. 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();
```

4.6.6. 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();
```

4.6.7. The SET statement

Most databases support a variety of SET statements to set session specific environment variables. jOOQ supports two of these set statements that are particularly useful when running DDL scripts:

```
SET CATALOG catalogname;
SET SCHEMA schemaname;
```

```
create.setCatalog("catalogname").execute();
create.setSchema("schemaname").execute();
```

Depending on whether your database supports catalogs and schemas, the above SET statements may be supported in your database.

In MariaDB, MySQL, SQL Server, the SET CATALOG statement is emulated using:

```
USE catalogname;
```

In Oracle, the SET SCHEMA statement is emulated using:

```
ALTER SESSION SET CURRENT_SCHEMA = schemaname;
```

4.6.8. 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 Ingres and SQLite. jOOQ will execute a DELETE FROM AUTHOR statement instead.

4.6.9. 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"."T_AUTHOR"(
  "ID" int not null,
  "FIRST_NAME" varchar(50) null,
  "LAST_NAME" varchar(50) not null,
  ...
  constraint "PK_T_AUTHOR"
    primary key ("ID")
)
create table "PUBLIC"."T_BOOK"(
  "ID" int not null,
  "AUTHOR_ID" int not null,
  "TITLE" varchar(400) not null,
  ...
  constraint "PK_T_BOOK"
    primary key ("ID")
)
...
alter table "PUBLIC"."T_BOOK"
  add constraint "FK_T_BOOK_AUTHOR_ID"
    foreign key ("AUTHOR_ID")
    references "T_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.

4.7. 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.

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.

4.7.1. 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, 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, 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_1677918343891_2316969 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_1677918343891_2316969();
DROP ALIAS block_1677918343891_2316969;

-- 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;

-- MYSQL

CREATE PROCEDURE block_1677918343896_9112928()
MODIFIES SQL DATA
BEGIN
  DELETE FROM BOOK;
  DELETE FROM AUTHOR;
END;
CALL block_1677918343896_9112928();
DROP PROCEDURE block_1677918343896_9112928;

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE
/* UNSUPPORTED */
```

4.7.2. CALL statement

When using CREATE PROCEDURE statements, for greater composability, it is essential to be able to call a procedure from another procedure. This is done using the CALL statement.

```
// Create a procedure that inserts a log message in a table
Parameter<String> message = in("message", VARCHAR);

create.createProcedure("log")
      .parameters(message)
      .as(insertInto(LOG).columns(LOG.TEXT).values(message))
      .execute();

create.createProcedure("some_other_procedure")
      .as(
      // ...
      call("log").args(val("My first message")),
      // ...
      call("log").args(val("My second message"))
      // ...
      )
      .execute();
```

Dialect support

This example using jOOQ:

```
call("log").args(val("message"))
```

Translates to the following dialect specific expressions:

```
-- BIGQUERY, DB2, HANA, HSQLDB, MARIADB, MYSQL, POSTGRES, YUGABYTEDB
CALL log('message')

-- FIREBIRD, INFORMIX
EXECUTE PROCEDURE log('message')

-- ORACLE
BEGIN
  log('message');
END;

-- SQLDATAWAREHOUSE, SQLSERVER
EXEC log 'message'

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, DERBY, EXASOL, H2, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE,
-- SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.3. 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
LOOP
  SET 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

-- INFORMIX
LOOP
  LET i = (i + 1);
  IF i > 10 THEN
    CONTINUE;
  END IF;
END LOOP

-- ORACLE, POSTGRES, YUGABYTEDB
LOOP
  i := (i + 1);
  CONTINUE WHEN i > 10;
END LOOP

-- SQLDATAWAREHOUSE, SQLSERVER
WHILE 1 = 1 BEGIN
  SET @i = (@i + 1);
  IF @i > 10
    CONTINUE;
END

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, EXASOL, FIREBIRD, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA,
-- VERTICA
/* UNSUPPORTED */
```

4.7.4. 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:

```
-- BIGQUERY, DB2, HANA, MARIADB, ORACLE
EXECUTE IMMEDIATE 'create table t (i int)'

-- FIREBIRD
EXECUTE STATEMENT 'create table t (i int)'

-- MYSQL

CREATE PROCEDURE block_1677918344407_1258750()
MODIFIES SQL DATA
BEGIN
  PREPARE s FROM 'create table t (i int)';
  EXECUTE s;
  DEALLOCATE PREPARE s;
END;
CALL block_1677918344407_1258750();
DROP PROCEDURE block_1677918344407_1258750;

-- POSTGRES, YUGABYTEDB
EXECUTE 'create table t (i int)'

-- SQLSERVER
EXECUTE ('create table t (i int)')

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, DERBY, EXASOL, H2, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.5. 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
LOOP
  SET 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

-- ORACLE, POSTGRES, YUGABYTEDB
LOOP
  i := (i + 1);
  EXIT WHEN i > 10;
END LOOP

-- SQLDATAWAREHOUSE, SQLSERVER
WHILE 1 = 1 BEGIN
  SET @i = (@i + 1);
  IF @i > 10
    BREAK;
END

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, EXASOL, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.6. 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, 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;

-- 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;

-- 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;

-- 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
```

4.7.7. 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;

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB,
-- MEMSQL, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.7.8. 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, DB2, EXASOL, HANA, HSQLDB, 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

-- 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();
  }
}

-- SQLDATAWAREHOUSE, SQLSERVER
IF @i = 0
  DELETE FROM BOOK;
ELSE
  DELETE FROM AUTHOR

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.9. 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, INFORMIX, ORACLE, POSTGRES, YUGABYTEDB
<<l>>
DELETE FROM BOOK

-- DB2, FIREBIRD, H2, HSQLDB, MARIADB, MYSQL, SQLDATAWAREHOUSE, SQLSERVER
l:
DELETE FROM BOOK

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, EXASOL, HANA, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE,
-- TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.10. 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, POSTGRES, 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

-- 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

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.11. 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
<<alias_2>>
LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  SET 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

-- 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

-- ORACLE, POSTGRES, YUGABYTEDB
<<alias_2>>
LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
  i := (i + 1);
  EXIT alias_2 WHEN i > 10;
END LOOP

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER,
-- SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.12. SIGNAL

The standard SQL way to raise exceptions is via the SIGNAL statement, which is supported natively in a few dialects, and can be emulated in some others.

Some example SIGNAL invocations.

```
begin(signalSQLState("45000")).execute();
begin(signalSQLState("45000").setMessageText("Custom message")).execute();
```

Dialect support

This example using jOOQ:

```
signalSQLState("45000").setMessageText("Custom message")
```

Translates to the following dialect specific expressions:

```
-- DB2, HSQLDB, MARIADB, MYSQL
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Custom message'

-- HANA
SIGNAL SQL_ERROR_CODE '45000' SET MESSAGE_TEXT = 'Custom message'

-- POSTGRES, YUGABYTEDB
RAISE SQLSTATE '45000' USING MESSAGE = 'Custom message'

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, INFORMIX, MEMSQL,
-- ORACLE, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.13. 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.

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))
```

The above is equivalent to this:

```
-- PL/SQL syntax
SELECT MAX(col) INTO i FROM t;
```

```
// All dialects
select(max(T.COL)).into(i).from(T)
```

Row assignment

Some dialects support row assignment of variables, which other languages call "destructuring". This is particularly useful when assigning multiple values from a query:

```
-- HSQLDB syntax
SET (i, j) = (SELECT MIN(col), MAX(col) FROM t);
```

```
// All dialects
row(i, j).set(select(min(T.COL), max(T.COL)).from(T))
```

The above is equivalent to this:

```
-- PL/SQL syntax
SELECT MIN(col), MAX(col) INTO i, j FROM t;
```

```
// All dialects
select(min(T.COL), max(T.COL)).into(i, j).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
DO $$
DECLARE
  DECLARE i int;
BEGIN
  SET i = 1;
END;
$$

-- BIGQUERY
BEGIN
  DECLARE i int64;
  SET 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_1677918346900_3672946 AS $$
  void x(Connection c) throws SQLException {
    Integer i = null;
    i = 1;
  }
$$;
CALL block_1677918346900_3672946();
DROP ALIAS block_1677918346900_3672946;

-- 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;

-- MYSQL

CREATE PROCEDURE block_1677918346902_1852844()
MODIFIES SQL DATA
BEGIN
  DECLARE i int;
  SET i = 1;
END;
CALL block_1677918346902_1852844();
DROP PROCEDURE block_1677918346902_1852844;

-- ORACLE
DECLARE
  i number(10);
BEGIN
  i := 1;
END;

-- POSTGRES, YUGABYTEDB
DO $$
DECLARE
  i int;
BEGIN
  i := 1;
END;
$$

-- SQLDATAWAREHOUSE, SQLSERVER
BEGIN
  DECLARE @i int;
  SET @i = 1;
END;

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.7.14. 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, EXASOL, INFORMIX, ORACLE, POSTGRES, YUGABYTEDB
WHILE i <= 10 LOOP
  DELETE FROM BOOK
  WHERE BOOK.ID = i;
END LOOP

-- BIGQUERY, DB2, HANA, HSQLDB, 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();
  }
}

-- SQLDATAWAREHOUSE, SQLSERVER
WHILE @i <= 10 BEGIN
  DELETE FROM BOOK
  WHERE BOOK.ID = @i;
END

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, MEMSQL, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.8. 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

4.9. Table expressions

The following sections explain the various types of table expressions supported by jOOQ

4.9.1. 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

4.9.2. Aliased 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.

Dereferencing columns from other table expressions

Only few table expressions provide the SQL syntax typesafety as shown above, where generated tables are used. Most tables, however, expose their fields through 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");
```

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 (as supported by Postgres, for instance):

```
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();
```

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.

Rendering declarations or references

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.

4.9.3. 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

4.9.3.1. 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.

4.9.3.2. 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.

4.9.3.3. 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.

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.

4.9.3.4. 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.

4.9.3.5. 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.

4.9.3.6. 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.

4.9.3.7. ON KEY 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 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.

4.9.3.8. USING clause

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.

4.9.3.9. NATURAL clause

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.

4.9.3.10. 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, DB2, FIREBIRD, MYSQL, ORACLE, POSTGRES, SNOWFLAKE, SYBASE, 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

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, REDSHIFT,
-- SQLITE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.9.3.11. 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, DB2, FIREBIRD, POSTGRES, SNOWFLAKE, 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

-- BIGQUERY, 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

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL, REDSHIFT,
-- SQLITE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.9.3.12. 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();
```

4.9.4. 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.

The above statement is emulated by jOOQ for those databases that do not support the VALUES() constructor, natively (actual emulations may vary):

```
-- If derived column expressions are supported:
SELECT a, b
FROM (
  SELECT 1, 'a' FROM DUAL UNION ALL
  SELECT 2, 'b' FROM DUAL
) t(a, b)

-- If derived column expressions are not supported:
SELECT a, b
FROM (

  -- An empty dummy record is added to provide column names for the emulated derived column expression
  SELECT NULL a, NULL b FROM DUAL WHERE 1 = 0 UNION ALL

  -- Then, the actual VALUES() constructor is emulated
  SELECT 1,      'a'    FROM DUAL             UNION ALL
  SELECT 2,      'b'    FROM DUAL
) t
```

4.9.5. 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();
```

4.9.6. The Oracle 11g 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.

4.9.7. jOOQ's relational division syntax

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/

4.9.8. 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.

4.9.9. 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();
```

4.9.10. JSON_TABLE

Some dialects ship with a built-in standard SQL table-valued function called JSON_TABLE, which can be used to unnest a JSON data structure into a SQL table.

```
SELECT *
FROM json_table(
  '[{"a":5,"b":{"x":10}},{"a":7,"b":{"y":20}}]',
  '$[*]'
  COLUMNS (
    id FOR ORDINALITY,
    a INT,
    x INT PATH '$.b.x',
    y INT PATH '$.b.y'
  )
) AS t
```

```
create.select()
      .from(jsonTable(
          JSON.valueOf("[{\"a\":5,\"b\":{\"x\":10}},"
                      + "{\"a\":7,\"b\":{\"y\":20}}]"),
          "$[*]")
        .column("id").forOrdinality()
        .column("a", INTEGER)
        .column("x", INTEGER).path("$.b.x")
        .column("y", INTEGER).path("$.b.y")
        .as("t"))
      .fetch();
```

The result would look like this:

```
+----+---+----+----+
| ID | A |  X |  Y |
+----+---+----+----+
|  1 | 5 | 10 |    |
|  2 | 7 |    | 20 |
+----+---+----+----+
```

4.9.11. XMLTABLE

Some dialects ship with a built-in standard SQL table-valued function called XMLTABLE, which can be used to unnest an XML data structure into a SQL table.

```
SELECT *
FROM xmltable('//row'
  PASSING
    '<rows>
       <row><a>5</a><b><x>10</x></b></row>
       <row><a>7</a><b><y>20</y></b></row>
     </rows>'
  COLUMNS
    id FOR ORDINALITY,
    a INT,
    x INT PATH 'b/x',
    y INT PATH 'b/y'
)
```

```
create.select()
      .from(xmltable("//row")
        .passing(
          "<rows>"
        + "<row><a>5</a><b><x>10</x></b></row>"
        + "<row><a>7</a><b><y>20</y></b></row>"
        + "</rows>"
        )
        .column("id").forOrdinality()
        .column("a", INTEGER)
        .column("x", INTEGER).path("b/x")
        .column("y", INTEGER).path("b/y"))
      .fetch();
```

The result would look like this:

```
+----+---+----+----+
| ID | A |  X |  Y |
+----+---+----+----+
|  1 | 5 | 10 |    |
|  2 | 7 |    | 20 |
+----+---+----+----+
```

4.9.12. 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

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, COCKROACHDB, EXASOL, H2, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, VERTICA, YUGABYTEDB
SELECT 1

-- AURORA_MYSQL, MEMSQL, ORACLE
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"
```

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.

4.9.13. Temporal tables

SQL:2011 standardised a feature called temporal validity, which comes in two flavours implemented through temporal tables:

- System versioned tables
- Application versioned tables

A few dialects, including DB2, MariaDB, Oracle, SQL Server implement one or the other, or both types of temporal tables through standard or vendor specific syntax.

System versioned tables

A system versioned table can be used for automatic backups or audit logging of all content in a table. Each "version" of a record has a validity period, until the record is updated or deleted, when a new "version" of the record is created.

Consider the following table (please consider your database manual for actual syntax. There are restrictions on data types and on how the history table is managed.):

```
CREATE TABLE product_price (
  product_id BIGINT NOT NULL PRIMARY KEY,
  price DECIMAL NOT NULL,
  start_ts TIMESTAMP GENERATED ALWAYS AS ROW START,
  end_ts TIMESTAMP GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME (start_ts, end_ts)
);
```

A table that is defined using the above (simplified) syntax can now be used in DML statements as follows:

```
-- At time T1, a new product is created:
INSERT INTO product (product_id, price)
VALUES (1, 100.00);

-- Later, at time T2, the price is updated:
UPDATE product
SET price = 200.00
WHERE product_id = 1;
```

```
create.insertInto(PRODUCT, PRODUCT.PRODUCT_ID, PRODUCT.PRICE)
      .values(1, new BigDecimal("100.00"))
      .execute();

create.update(PRODUCT)
      .set(PRODUCT.PRICE, new BigDecimal("200.00"))
      .where(PRODUCT.PRODUCT_ID.eq(1))
      .execute();
```

Thanks to system versioning, the UPDATE statement is no longer "destructive", meaning that the original row containing the (1, 100.00) price information is still available from the archive. We can query the PRODUCT table and its archive as follows (see example query results further down):

```
-- 1. Get the current version by default
SELECT * FROM product;

-- 2. Get the version at a given time
SELECT * FROM product
  FOR system_time AS OF :t1;

-- 3. Get several versions overlapping a time range [t1, t2]
SELECT * FROM product
  FOR system_time BETWEEN :t1 AND :t2;

-- 4. Get several versions overlapping a time range [t1, t2)
SELECT * FROM product
  FOR system_time FROM :t1 TO :t2;

-- 5. Get several versions included in a time range [t1, t2]
SELECT * FROM product
  FOR system_time CONTAINED IN (:t1, :t2);

-- 6. Get all versions
SELECT * FROM product
  FOR system_time ALL;
```

```
// Get the current version by default
create.selectFrom(product).fetch();

create.selectFrom(product.for_(
  systemTime().asOf(t1)
)).fetch();

create.selectFrom(product.for_(
  systemTime().between(t1).and(t2))
)).fetch();

create.selectFrom(product.for_(
  systemTime().from(t1).to(t2)
)).fetch();

create.selectFrom(product.for_(
  systemTime().containedIn(t1, t2)
)).fetch();

create.selectFrom(product.for_(
  systemTime().all()
)).fetch();
```

The results of the above queries might look like this:

```
+----------+------------+-------+----------+--------+
| QUERY_NO | PRODUCT_ID | PRICE | START_TS | END_TS |
+----------+------------+-------+----------+--------+
|        1 |          1 |   200 | T2       |        |
+----------+------------+-------+----------+--------+

+----------+------------+-------+----------+--------+
| QUERY_NO | PRODUCT_ID | PRICE | START_TS | END_TS |
+----------+------------+-------+----------+--------+
|  2, 4, 5 |          1 |   100 | T1       | T2     |
+----------+------------+-------+----------+--------+

+----------+------------+-------+----------+--------+
| QUERY_NO | PRODUCT_ID | PRICE | START_TS | END_TS |
+----------+------------+-------+----------+--------+
|     3, 6 |          1 |   100 | T1       | T2     |
|     3, 6 |          1 |   200 | T2       |        |
+----------+------------+-------+----------+--------+
```

jOOQ 3.13 only supports the above syntaxes if they are natively supported by the underlying dialect as well. Future jOOQ versions may emulate the syntax also in other dialects, or where a specific clause is not supported.

Application versioned tables

While system versioned tables allow for implementing backups and audit logs, some data is naturally versioned from a business perspective as well. Perhaps, instead of just archiving the pricing information, we may wish to specify a validity range for which a given price was valid on a given product. That way, we can accurately restore the old price on an old period in case we need it for reporting or accounting reasons. The (simplified) syntax is almost the same as with system versioned tables, except that instead of using a "magic" SYSTEM_TIME period name, we can now use our own (or in case of DB2, use BUSINESS_TIME). Please look up your dialect's manual again, for exact syntax:

```
CREATE TABLE product_price (
  product_id BIGINT NOT NULL PRIMARY KEY,
  price DECIMAL NOT NULL,
  start_ts TIMESTAMP,
  end_ts TIMESTAMP,
  PERIOD FOR validity (start_ts, end_ts)
);
```

A table that is defined using the above (simplified) syntax can now be used in DML statements as follows:

```
-- A new product is created
INSERT INTO product (product_id, price)
VALUES (1, 100.00);

-- For the time between [t1, t2], a discount is applied
UPDATE product
  FOR PORTION OF validity FROM t1 TO t2
SET price = 50.00
WHERE product_id = 1;
```

```
create.insertInto(PRODUCT, PRODUCT.PRODUCT_ID, PRODUCT.PRICE)
      .values(1, new BigDecimal("100.00"))
      .execute();

create.update(PRODUCT.forPortionOf(
  period(unquotedName("validity")).from(t1).to(t2)))
      .set(PRODUCT.PRICE, new BigDecimal("50.00"))
      .where(PRODUCT.PRODUCT_ID.eq(1))
      .execute();
```

If not combined with system versioning, this is again a destructive UPDATE statement, which is effectively transformed into several statements. The resulting table content now looks like this:

```
+------------+-------+----------+--------+
| PRODUCT_ID | PRICE | START_TS | END_TS |
+------------+-------+----------+--------+
|          1 | 100.0 |          | T1     |
|          1 |  50.0 | T1       | T2     |
|          1 | 100.0 | T2       |        |
+------------+-------+----------+--------+
```

Depending on your dialect, you can reuse the previous FOR clauses in SELECT statements, for example:

```
-- 2. Get the version at a given time
SELECT * FROM product
  FOR validity AS OF :t1;
```

```
create.selectFrom(product.for_(
  period(unquotedName("validity")).asOf(t1)
)).fetch();
```

Which will produce the value of your attribute(s) given their validity at a given timestamp T1:

```
+------------+-------+----------+--------+
| PRODUCT_ID | PRICE | START_TS | END_TS |
+------------+-------+----------+--------+
|          1 |  50.0 | T1       | T2     |
+------------+-------+----------+--------+
```

4.9.14. Data change delta tables

The SQL standard specifies how to turn a DML statement into a table expression that can be used in the FROM clause of a SELECT statement. Other dialects support a RETURNING or OUTPUT clause of some sort to produce the same behaviour, though less powerful.

A data change delta table has two parts:

- The result option (OLD, NEW, FINAL)
- The data change statement, which includes DELETE, INSERT, MERGE, UPDATE

You can thus express a query like the following to return all the inserted data (including DEFAULT and TRIGGER generated values):

```
SELECT *
FROM FINAL TABLE (
  INSERT INTO BOOK
    (ID, TITLE)
  VALUES
    (1, 'The Book')
)
```

```
create.select()
      .from(finalTable(
          insertInto(BOOK)
          .columns(BOOK.ID, BOOK.TITLE)
          .values(1, "The Book")
      ))
      .fetch();
```

Following the restrictions implemented by your dialect, the results of such tables can be further processed, projected, etc.

The semantics of the result options are:

- OLD: Access the row data as it was prior to being modified by the data change statement. This does not work for INSERT
- NEW: Access the row data as it is after being modified by the data change statement, but before any AFTER TRIGGERS are fired. This does not work for DELETE
- FINAL: Access the row data as it is after being modified by the data change statement, and all triggers. The data is in its "final" form. This does not work for DELETE

Dialect support

This example using jOOQ:

```
select(BOOK.ID).from(finalTable(insertInto(BOOK).columns(BOOK.ID, BOOK.TITLE).values(1, "The Book")))
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES
WITH
  BOOK AS (
    INSERT INTO BOOK (ID, TITLE)
    VALUES (
      1,
      'The Book'
    )
    RETURNING
      BOOK.ID,
      BOOK.AUTHOR_ID,
      BOOK.TITLE,
      BOOK.PUBLISHED_IN,
      BOOK.LANGUAGE_ID
  )
SELECT BOOK.ID
FROM BOOK BOOK

-- DB2, H2
SELECT BOOK.ID
FROM FINAL TABLE (
  INSERT INTO BOOK (ID, TITLE)
  VALUES (
    1,
    'The Book'
  )
) BOOK

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.9.15. Tables as SelectField

An org.jooq.Table expression extends the org.jooq.SelectField type, and as such, can be used in the SELECT clause directly, as well as everywhere else a SelectField is accepted, e.g. in nested records. This is specifically useful for (generated) table references. The following shows how to project a nested org.jooq.TableRecord:

```
Result<Record2<AuthorRecord, BookRecord>> result =
create.select(AUTHOR, BOOK)
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .fetch();
```

This plays very well together with implicit joins:

```
Result<Record2<AuthorRecord, BookRecord>> result =
create.select(BOOK.author(), BOOK)
      .from(BOOK)
      .fetch();
```

Behind the scenes, the implementation may either be native in dialects that support this kind of projection (e.g. PostgreSQL), or emulated using the usual nested records emulations.

4.9.16. Tables as GroupField

An org.jooq.Table expression extends the org.jooq.GroupField type, and as such, can be used in the GROUP BY clause directly. This is specifically useful for (generated) table references. The following two statements are equivalent, although their generated SQL may differ, depending on native support:

```
// Ordinary grouping
create.select(AUTHOR.ID, count())
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
  .groupBy(AUTHOR)
  .fetch();

// Convenient grouping by the entire table
create.select(AUTHOR.ID, count())
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
  .groupBy(AUTHOR.ID)
  .fetch();
```

4.10. 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);
```

4.10.1. 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. 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.

See the manual's section about generated tables for more information about what is really generated by the code generator

4.10.2. 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.

4.10.3. 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 same thing can be achieved by casting a Field directly to String.class, as VARCHAR is the default SQLDataType to map to Java's String

```
create.select(AUTHOR.LAST_NAME.cast(String.class)).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);
}
```

4.10.4. 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.

4.10.5. Readonly columns

jOOQ's code generator may decide that a column is readonly, in case of which using it in various DML statements may subtly change. The following Settings govern this behaviour:

- Settings.readonlyInsert: Inclusion in an INSERT statement, or in the INSERT clause of a MERGE statement.
- Settings.readonlyUpdate: Inclusion in an UPDATE statement, or in the UPDATE clause of a MERGE statement.
- Settings.readonlyTableRecordInsert: Inclusion in a TableRecord.insert() operation, or the INSERT part or execution of TableRecord.store() or UpdatableRecord.merge(). If this is deactivated, Settings.readonlyInsert still applies
- Settings.readonlyUpdatableRecordUpdate: Inclusion in a UpdatableRecord.update() operation, or the UPDATE part or execution of TableRecord.store() or UpdatableRecord.merge(). If this is deactivated, Settings.readonlyUpdate still applies

Each one of these flags is of type org.jooq.conf.WriteIfReadonly with these permitted states:

- WRITE: Write to the column as if it weren't readonly. This effectively turns off the feature.
- IGNORE: Ignore the column in a relevant statement. This is the default.
- THROW: Throw an exception if the column is included in a relevant DML statement.

The default behaviour IGNORE is particularly useful when loading POJO data into org.jooq.UpdatableRecord and storing it, while ignoring IDENTITY columns, computed columns, synthetic columns (such as the synthetic ROWIDs), and more:

```
// If BOOK.ID is an auto-generated identity, we don't want to load NULL values into the record.
create.newRecord(BOOK, bookPojo);
```

4.10.6. Computed columns

Computed columns, sometimes also called "virtual" columns, are columns that are generated from an expression based on other columns of the same row directly in the database. They cannot be written to, but may be used in projections, filters, and even indexes, as a complement or replacement of function based indexes.

jOOQ's code generator picks up computed columns like any other, but marks them as read only for your convenience, such that they are excluded from DML statements, by default.

See the section about computed columns in CREATE TABLE for more details.

4.10.7. 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();
```

4.10.8. 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

4.10.9. 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();
```

4.10.10. 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();
```

4.10.11. 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.

4.10.11.1. 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, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, ORACLE, POSTGRES,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
CASE 1
  WHEN 1 THEN 'a'
  WHEN 2 THEN 'b'
END

-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
elt(1, 'a', 'b')

-- DERBY
CASE
  WHEN 1 = 1 THEN 'a'
  WHEN 1 = 2 THEN 'b'
END

-- SQLSERVER
choose(1, 'a', 'b')
```

4.10.11.2. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
coalesce(NULL, NULL, 1)

-- DERBY
coalesce(?, ?, 1)

-- INFORMIX
nvl(
  nvl(
    NULL,
    NULL
  ),
  1
)
```

4.10.11.3. 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, FIREBIRD, HSQLDB, POSTGRES, SNOWFLAKE, SQLSERVER, 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

-- DB2, EXASOL, H2, INFORMIX, MEMSQL, ORACLE, TERADATA, VERTICA
decode(
  AUTHOR.FIRST_NAME,
  'Paulo',
  'BR',
  'George',
  'EN',
  'unknown'
)

-- 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'
)

-- REDSHIFT
CASE
  WHEN NOT (AUTHOR.FIRST_NAME IS DISTINCT FROM 'Paulo') THEN 'BR'
  WHEN NOT (AUTHOR.FIRST_NAME IS DISTINCT FROM 'George') THEN 'EN'
  ELSE 'unknown'
END

-- SQLITE
CASE
  WHEN (AUTHOR.FIRST_NAME IS 'Paulo') THEN 'BR'
  WHEN (AUTHOR.FIRST_NAME IS 'George') THEN 'EN'
  ELSE 'unknown'
END

-- 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'
```

4.10.11.4. 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, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, ORACLE,
-- POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
CASE
  WHEN 1 = 2 THEN 3
  ELSE 4
END

-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
if(1 = 2, 3, 4)
```

4.10.11.5. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
nullif(1, 2)
```

4.10.11.6. 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, EXASOL, FIREBIRD, HANA, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
coalesce(
  NULL,
  1
)

-- AURORA_MYSQL, BIGQUERY, MARIADB, MEMSQL, MYSQL, SQLITE
ifnull(
  NULL,
  1
)

-- DB2, H2, HSQLDB, INFORMIX, ORACLE
nvl(
  NULL,
  1
)

-- DERBY
coalesce(
  ?,
  1
)
```

4.10.11.7. 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, COCKROACHDB, DERBY, FIREBIRD, HANA, MEMSQL, MYSQL, POSTGRES,
-- SQLDATAWAREHOUSE, SQLITE, SYBASE, YUGABYTEDB
CASE
  WHEN 1 IS NOT NULL THEN 2
  ELSE 3
END

-- DB2, EXASOL, H2, HSQLDB, INFORMIX, MARIADB, ORACLE, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA
nvl2(1, 2, 3)
```

4.10.12. 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.

4.10.12.1. 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)
```

4.10.12.2. 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 |
+------------+
```

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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
acos(0)

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.3. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
asin(1)

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.4. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
atan(1)

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.5. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
atan2(1, 1)

-- ACCESS
/* UNSUPPORTED */
```

4.10.12.6. 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, SQLDATAWAREHOUSE, SQLSERVER
ceiling(1.7E0)

-- AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
ceil(1.7E0)

-- COCKROACHDB
ceil(CAST(1.7E0 AS double precision))

-- H2
ceiling(CAST(1.7E0 AS double))
```

4.10.12.7. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
cos(3.14159265359E0)

-- COCKROACHDB
cos(CAST(3.14159265359E0 AS double precision))

-- H2
cos(CAST(3.14159265359E0 AS double))
```

4.10.12.8. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
((exp((1 * 2)) + 1) / (exp(1) * 2))

-- BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, INFORMIX, ORACLE, SNOWFLAKE, SQLITE, TERADATA
cosh(1)

-- COCKROACHDB
((exp(CAST((1 * 2) AS numeric)) + 1) / (exp(CAST(1 AS numeric)) * 2))
```

4.10.12.9. 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, INFORMIX, ORACLE, SQLITE, TERADATA
(cos(1.5707963268E0) / sin(1.5707963268E0))

-- ASE, AURORA_MYSQL, AURORA_POSTGRES, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
cot(1.5707963268E0)

-- COCKROACHDB
cot(CAST(1.5707963268E0 AS double precision))

-- H2
cot(CAST(1.5707963268E0 AS double))
```

4.10.12.10. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
((exp((1 * 2)) + 1) / (exp((1 * 2)) - 1))

-- COCKROACHDB
((exp(CAST((1 * 2) AS numeric)) + 1) / (exp(CAST((1 * 2) AS numeric)) - 1))
```

4.10.12.11. 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, AURORA_MYSQL, AURORA_POSTGRES, DB2, DERBY, EXASOL, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
degrees(3.14159265359E0)

-- BIGQUERY
((CAST(3.14159265359E0 AS decimal) * 180) / acos(-1))

-- COCKROACHDB
degrees(CAST(3.14159265359E0 AS double precision))

-- FIREBIRD
((CAST(3.14159265359E0 AS numeric) * 180) / pi())

-- H2
degrees(CAST(3.14159265359E0 AS double))

-- HANA
((CAST(3.14159265359E0 AS numeric) * 180) / acos(-1))

-- ORACLE
((CAST(3.14159265359E0 AS number) * 180) / acos(-1))
```

4.10.12.12. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
exp(1)

-- COCKROACHDB
exp(CAST(1 AS numeric))
```

4.10.12.13. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
exp(1)

-- COCKROACHDB
exp(CAST(1 AS numeric))
```

4.10.12.14. 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, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL,
-- MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
floor(1.7E0)

-- COCKROACHDB
floor(CAST(1.7E0 AS double precision))

-- H2
floor(CAST(1.7E0 AS double))
```

4.10.12.15. 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, COCKROACHDB, DB2, EXASOL, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, ORACLE,
-- POSTGRES, REDSHIFT, SNOWFLAKE, SQLSERVER, TERADATA, VERTICA, YUGABYTEDB
greatest(2, 3)

-- FIREBIRD
maxvalue(2, 3)

-- SQLITE
max(2, 3)
```

4.10.12.16. 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, COCKROACHDB, DB2, EXASOL, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, ORACLE,
-- POSTGRES, REDSHIFT, SNOWFLAKE, SQLSERVER, TERADATA, VERTICA, YUGABYTEDB
least(2, 3)

-- FIREBIRD
minvalue(2, 3)

-- SQLITE
min(2, 3)
```

4.10.12.17. 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:

```
-- ACCESS, ASE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER
log(1)

-- AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL,
-- ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
ln(1)

-- COCKROACHDB
ln(CAST(1 AS numeric))

-- INFORMIX
logn(1)
```

4.10.12.18. 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:

```
-- ACCESS, ASE, SQLITE
(log(8) / log(2))

-- AURORA_MYSQL, AURORA_POSTGRES, EXASOL, FIREBIRD, H2, HANA, MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT,
-- SNOWFLAKE, VERTICA, YUGABYTEDB
log(2, 8)

-- BIGQUERY, SQLDATAWAREHOUSE, SQLSERVER
log(8, 2)

-- COCKROACHDB
(ln(CAST(8 AS numeric)) / ln(CAST(2 AS numeric)))

-- DB2, DERBY, HSQLDB, SYBASE, TERADATA
(ln(8) / ln(2))

-- INFORMIX
(logn(8) / logn(2))
```

4.10.12.19. LOG10

The LOG10() function calculates the logarithm of a numeric value, given base 10.

```
SELECT log10(100);
```

```
create.select(log10(100)).fetch();
```

The result being

```
+-------+
| log10 |
+-------+
|     2 |
+-------+
```

Dialect support

This example using jOOQ:

```
log10(100)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, POSTGRES, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA
log10(100)

-- COCKROACHDB
(ln(CAST(100 AS numeric)) / ln(CAST(10 AS numeric)))

-- HANA, ORACLE, SNOWFLAKE, YUGABYTEDB
log(10, 100)

-- REDSHIFT, TERADATA
log(100)
```

4.10.12.20. 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
```

4.10.12.21. 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, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
pi()

-- BIGQUERY, DB2, HANA, INFORMIX, ORACLE, TERADATA
acos(-1)
```

4.10.12.22. 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, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
power(2, 3)

-- DERBY
exp((ln(2) * 3))

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.23. 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, COCKROACHDB, DB2, DERBY, EXASOL, H2, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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)
```

4.10.12.24. 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, DB2, FIREBIRD, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, SQLDATAWAREHOUSE, SQLSERVER,
-- SYBASE
rand()

-- AURORA_POSTGRES, COCKROACHDB, DERBY, 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 */
```

4.10.12.25. 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, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, TERADATA, VERTICA, YUGABYTEDB
round(1.7E0)

-- ASE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE
round(1.7E0, 0)

-- COCKROACHDB
round(CAST(CAST(1.7E0 AS double precision) AS numeric))

-- DERBY
CASE
  WHEN (1.7E0 - floor(1.7E0)) < 5E-1 THEN floor(1.7E0)
  ELSE ceil(1.7E0)
END

-- H2
round(CAST(1.7E0 AS double))
```

4.10.12.26. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
sign(3)

-- SQLITE
CASE
  WHEN 3 > 0 THEN 1
  WHEN 3 < 0 THEN -1
  WHEN 3 = 0 THEN 0
END
```

4.10.12.27. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
sin(3.14159265359E0)

-- COCKROACHDB
sin(CAST(3.14159265359E0 AS double precision))

-- H2
sin(CAST(3.14159265359E0 AS double))
```

4.10.12.28. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
((exp((1 * 2)) - 1) / (exp(1) * 2))

-- BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, INFORMIX, ORACLE, SNOWFLAKE, SQLITE, TERADATA
sinh(1)

-- COCKROACHDB
((exp(CAST((1 * 2) AS numeric)) - 1) / (exp(CAST(1 AS numeric)) * 2))
```

4.10.12.29. 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, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
sqrt(4)

-- COCKROACHDB
sqrt(CAST(4 AS numeric))
```

4.10.12.30. SQUARE

The SQUARE() function calculates the value of a number squared.

```
SELECT square(3);
```

```
create.select(square(3)).fetch();
```

The result being

```
+--------+
| square |
+--------+
|      9 |
+--------+
```

Dialect support

This example using jOOQ:

```
square(3)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
(3 * 3)

-- ASE, SQLDATAWAREHOUSE, SQLSERVER
square(3)
```

4.10.12.31. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
tan(3.14159265359E0)

-- COCKROACHDB
tan(CAST(3.14159265359E0 AS double precision))

-- H2
tan(CAST(3.14159265359E0 AS double))

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.32. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLDATAWAREHOUSE,
-- SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
((exp((1 * 2)) - 1) / (exp((1 * 2)) + 1))

-- BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, INFORMIX, ORACLE, SQLITE, TERADATA
tanh(1)

-- COCKROACHDB
((exp(CAST((1 * 2) AS numeric)) - 1) / (exp(CAST((1 * 2) AS numeric)) + 1))

-- SNOWFLAKE
/* UNSUPPORTED */
```

4.10.12.33. 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, MARIADB, MEMSQL, MYSQL
truncate(1.7E0, 0)

-- AURORA_POSTGRES, POSTGRES
CAST(trunc(
  CAST(1.7E0 AS numeric),
  0
) AS double precision)

-- DB2, FIREBIRD, HSQLDB, INFORMIX, ORACLE, TERADATA, VERTICA
trunc(1.7E0, 0)

-- DERBY
CASE
  WHEN sign(1.7E0) >= 0 THEN (floor((1.7E0 * 1)) / 1)
  ELSE (ceil((1.7E0 * 1)) / 1)
END

-- H2
truncate(CAST(1.7E0 AS double), 0)

-- HANA
round(1.7E0, 0, round_down)

-- SQLDATAWAREHOUSE, SQLSERVER
round(1.7E0, 0, 1)

-- SYBASE
truncnum(1.7E0, 0)

-- ACCESS, BIGQUERY, COCKROACHDB, EXASOL, REDSHIFT, SNOWFLAKE, SQLITE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.12.34. 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, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, 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, COCKROACHDB, ORACLE, POSTGRES, SNOWFLAKE, TERADATA, YUGABYTEDB
width_bucket(15, 0, 100, 10)
```

4.10.13. 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"

4.10.13.1. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
(5 & 4)

-- DB2, H2, HANA, HSQLDB, INFORMIX, ORACLE, SNOWFLAKE, TERADATA
bitand(5, 4)

-- EXASOL
bit_and(5, 4)

-- FIREBIRD
bin_and(5, 4)

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.2. 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, 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)

-- 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)

-- 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))

-- 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)

-- ACCESS, ASE, DB2, DERBY, EXASOL
/* UNSUPPORTED */
```

4.10.13.3. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
~((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)

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.4. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
~((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)

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.5. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
~5

-- DB2, H2, HANA, INFORMIX, SNOWFLAKE, TERADATA
bitnot(5)

-- EXASOL
bit_not(5)

-- FIREBIRD
bin_not(5)

-- HSQLDB, ORACLE
((0 - 5) - 1)

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.6. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
(5 | 2)

-- DB2, H2, HANA, HSQLDB, INFORMIX, SNOWFLAKE, TERADATA
bitor(5, 2)

-- EXASOL
bit_or(5, 2)

-- FIREBIRD
bin_or(5, 2)

-- ORACLE
((5 + 2) - bitand(5, 2))

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.7. 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))

-- DB2, HANA, INFORMIX, SNOWFLAKE, TERADATA
bitnot(bitxor(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)))

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.8. BIT_XOR

The BIT_XOR() function produces the bitwise XOR (exclusive OR) operation.

```sql
SELECT bit_xor(5, 3);
```

```java
create.select(bitXor(5, 3)).fetch();
```

The result being

```
+---------+
| bit_xor |
+---------+
|       6 |
+---------+
```

Dialect support

This example using jOOQ:

```java
bitXor(5, 3)
```

Translates to the following dialect specific expressions:

```sql
-- ASE, AURORA_MYSQL, BIGQUERY, MARIADB, MEMSQL, MYSQL, SQLDATAWAREHOUSE, SQLSERVER, SYBASE
(5 ^ 3)

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, REDSHIFT, VERTICA, YUGABYTEDB
(5 # 3)

-- DB2, H2, HANA, HSQLDB, INFORMIX, SNOWFLAKE, TERADATA
bitxor(5, 3)

-- EXASOL
bit_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))

-- ACCESS, DERBY
/* UNSUPPORTED */
```

4.10.13.9. SHL

The SHL() function produces the bitwise shift left operation.

```sql
SELECT shl(1, 4);
```

```java
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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLITE, SQLSERVER,
-- VERTICA, YUGABYTEDB
(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)

-- TERADATA
shiftleft(1, 4)

-- ACCESS, DERBY, HANA
/* UNSUPPORTED */
```

4.10.13.10. 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, MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SQLITE, SQLSERVER,
-- VERTICA, YUGABYTEDB
(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)

-- TERADATA
shiftright(16, 4)

-- ACCESS, DERBY, HANA
/* UNSUPPORTED */
```

4.10.14. 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.

4.10.14.1. 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, COCKROACHDB, DB2, EXASOL, H2, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL,
-- MYSQL, ORACLE, POSTGRES, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
ascii('A')

-- FIREBIRD
ascii_val('A')

-- DERBY, REDSHIFT, SQLITE
/* UNSUPPORTED */
```

4.10.14.2. CHR

The CHR() or CHAR() function calculates the character representation of an ASCII code, or unicode in some dialects.

```
SELECT chr(64);
```

```
create.select(chr(65)).fetch();
```

The result being

```
+-----+
| chr |
+-----+
| A   |
+-----+
```

Dialect support

This example using jOOQ:

```
chr(65)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, BIGQUERY, COCKROACHDB, EXASOL, H2, INFORMIX, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA
chr(65)

-- ASE, DB2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, SQLITE, SQLSERVER, SYBASE, YUGABYTEDB
char(65)

-- FIREBIRD
ascii_char(65)

-- AURORA_MYSQL, AURORA_POSTGRES, DERBY, SQLDATAWAREHOUSE
/* UNSUPPORTED */
```

4.10.14.3. CONCAT

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' & ' ')

-- ASE, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, ORACLE,
-- POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
(('hello' || ' ') || 'world')

-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
concat('hello', ' ', 'world')

-- SQLDATAWAREHOUSE, SQLSERVER
(('hello' + ' ') + 'world')
```

4.10.14.4. DIGITS

The DIGITS() function allows for turning numbers to 0-padded string that makes sorting those strings according to numeric values easier when concatenated to other strings. The padding depends on the number's data type, precision, and scale.

```
SELECT digits(cast(1234.5 as decimal(8, 2)));
```

```
create.select(digits(cast(val(1234.5), DECIMAL(8, 2)))).fetch();
```

The result being

```
+----------+
| digits   |
+----------+
| 00123450 |
+----------+
```

Dialect support

This example using jOOQ:

```
digits(cast(val(1234.5), DECIMAL(7, 2)))
```

Translates to the following dialect specific expressions:

```
-- ACCESS
(replace(space(7 - len(cstr(cdec(abs((cdec(1.2345E3) * 100)))))), ' ', '0') & cstr(cdec(abs((cdec(1.2345E3) * 100)))))

-- ASE
(replicate(
  '0',
  (7 - char_length(CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7))))
) || CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7)))

-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS char(7)),
  7,
  '0'
)

-- AURORA_POSTGRES, EXASOL, FIREBIRD, HANA, HSQLDB, POSTGRES, TERADATA, VERTICA, YUGABYTEDB
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7)),
  7,
  '0'
)

-- BIGQUERY
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS decimal) * 100)) AS decimal) AS string),
  7,
  '0'
)

-- COCKROACHDB
lpad(
  CAST(CAST(abs((CAST(CAST(1.2345E3 AS double precision) AS decimal(7, 2)) * 100)) AS decimal(7)) AS string(7)),
  7,
  '0'
)

-- DB2
digits(CAST(1.2345E3 AS decimal(7, 2)))

-- H2
lpad(
  CAST(CAST(abs((CAST(CAST(1.2345E3 AS double) AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7)),
  7,
  '0'
)

-- INFORMIX
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS lvarchar(7)),
  7,
  '0'
)

-- ORACLE
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar2(7)),
  7,
  '0'
)

-- SNOWFLAKE
lpad(
  CAST(CAST(abs((CAST(1.2345E3 AS number(7, 2)) * 100)) AS number(7)) AS varchar(7)),
  7,
  '0'
)

-- SQLDATAWAREHOUSE, SQLSERVER
(replicate(
  '0',
  (7 - len(CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7))))
) + CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7)))

-- SQLITE
substr("replace"(hex(zeroblob(7)), '00', '0'), 1, 7 - length(CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS
 varchar(7)))) || CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7))

-- SYBASE
(repeat(
  '0',
  (7 - length(CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7))))
) || CAST(CAST(abs((CAST(1.2345E3 AS decimal(7, 2)) * 100)) AS decimal(7)) AS varchar(7)))

-- DERBY, REDSHIFT
/* UNSUPPORTED */
```

4.10.14.5. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
left('hello world', 5)

-- DERBY, ORACLE, SQLITE
substr('hello world', 1, 5)
```

4.10.14.6. 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, EXASOL, FIREBIRD, H2, HSQLDB, INFORMIX, MARIADB, MEMSQL,
-- MYSQL, POSTGRES, REDSHIFT, VERTICA, YUGABYTEDB
char_length('hello')

-- DB2, DERBY, HANA, ORACLE, SNOWFLAKE, SQLITE, SYBASE, TERADATA
length('hello')
```

4.10.14.7. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
lower('HELLO')
```

4.10.14.8. 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, HANA, HSQLDB, 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, DERBY, EXASOL, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.14.9. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
ltrim('  hello  ')

-- FIREBIRD
trim(LEADING FROM '  hello  ')
```

4.10.14.10. 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, COCKROACHDB, MARIADB, MEMSQL, MYSQL, POSTGRES, 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))

-- ACCESS, ASE, DB2, DERBY, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA
/* UNSUPPORTED */
```

4.10.14.11. MID

The MID() function is an alias for the substring function

4.10.14.12. OVERLAY

The OVERLAY() function takes a string and "overlays it on top of another string".

```
SELECT overlay('abcdefg', 'xxx', 2);
```

```
create.select(overlay(val("abcdefg"), "xxx", 2)).fetch();
```

The result being

```
+---------+
| overlay |
+---------+
| axxxefg |
+---------+
```

Dialect support

This example using jOOQ:

```
overlay(val("abcdefg"), "xxx", 2)
```

Translates to the following dialect specific expressions:

```
-- ACCESS
((mid(
  'abcdefg',
  1,
  (2 - 1)
) & 'xxx') & mid(
  'abcdefg',
  (2 + len('xxx'))
))

-- ASE
((substring(
  'abcdefg',
  1,
  (2 - 1)
) || 'xxx') || substring(
  'abcdefg',
  (2 + char_length('xxx')),
  2147483647
))

-- AURORA_MYSQL, EXASOL, H2, MARIADB, MYSQL
insert(
  'abcdefg',
  2,
  char_length('xxx'),
  'xxx'
)

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, FIREBIRD, POSTGRES, VERTICA, YUGABYTEDB
overlay('abcdefg' PLACING 'xxx' FROM 2)

-- DB2
overlay('abcdefg' PLACING 'xxx' FROM 2 FOR length('xxx'))

-- DERBY, ORACLE, SQLITE
((substr(
  'abcdefg',
  1,
  (2 - 1)
) || 'xxx') || substr(
  'abcdefg',
  (2 + length('xxx'))
))

-- HANA, SYBASE
((substring(
  'abcdefg',
  1,
  (2 - 1)
) || 'xxx') || substring(
  'abcdefg',
  (2 + length('xxx'))
))

-- HSQLDB, REDSHIFT
((substring(
  'abcdefg',
  1,
  (2 - 1)
) || 'xxx') || substring(
  'abcdefg',
  (2 + char_length('xxx'))
))

-- INFORMIX
((substr(
  'abcdefg',
  1,
  (2 - 1)
) || 'xxx') || substr(
  'abcdefg',
  (2 + char_length('xxx'))
))

-- MEMSQL
concat(
  concat(
    substring(
      'abcdefg',
      1,
      (2 - 1)
    ),
    'xxx'
  ),
  substring(
    'abcdefg',
    (2 + char_length('xxx'))
  )
)

-- SNOWFLAKE
insert(
  'abcdefg',
  2,
  length('xxx'),
  'xxx'
)

-- SQLDATAWAREHOUSE, SQLSERVER
((substring(
  'abcdefg',
  1,
  (2 - 1)
) + 'xxx') + substring(
  'abcdefg',
  (2 + len('xxx')),
  2147483647
```

4.10.14.13. 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, EXASOL, FIREBIRD, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES, SNOWFLAKE,
-- TERADATA, VERTICA, YUGABYTEDB
position('e' IN 'hello')

-- BIGQUERY, ORACLE, SQLITE
instr('hello', 'e')

-- DB2, DERBY
locate('e', 'hello')

-- HANA, SYBASE
locate('hello', 'e')

-- ACCESS, INFORMIX, REDSHIFT
/* UNSUPPORTED */
```

4.10.14.14. REGEXP_REPLACE

The REGEXP_REPLACE() function searches a string for a regular expression pattern, and replaces all or the first occurrence of that string.

Vendors offer different versions of this function, so jOOQ standardises them as two synthetic functions:

- REGEXP_REPLACE_ALL()
- REGEXP_REPLACE_FIRST()

For example:

```
SELECT
  regexp_replace_all('hello', 'l', ''),
  regexp_replace_first('hello', 'l', '');
```

```
create.select(
  regexpReplaceAll(val("hello"), "l", ""),
  regexpReplaceFirst(val("hello"), "l", "")).fetch();
```

The result being

```
+--------------------+----------------------+
| regexp_replace_all | regexp_replace_first |
+--------------------+----------------------+
| heo                | helo                 |
+--------------------+----------------------+
```

Dialect support

This example using jOOQ:

```
regexpReplaceAll(val("hello"), "l", "")
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, DB2, H2, HSQLDB, MARIADB, MYSQL, ORACLE, TERADATA, VERTICA
regexp_replace('hello', 'l', '')

-- AURORA_POSTGRES, COCKROACHDB, MEMSQL, POSTGRES, YUGABYTEDB
regexp_replace('hello', 'l', '', 'g')

-- HANA
replace_regexpr('l' IN 'hello' WITH '')

-- INFORMIX
regex_replace('hello', 'l', '')

-- ACCESS, ASE, BIGQUERY, DERBY, EXASOL, FIREBIRD, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE
/* UNSUPPORTED */
```

4.10.14.15. 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, COCKROACHDB, DB2, EXASOL, H2, HSQLDB, MARIADB, MYSQL, POSTGRES, SNOWFLAKE,
-- SYBASE, VERTICA, YUGABYTEDB
repeat('abc', 3)

-- FIREBIRD, MEMSQL
rpad(
  'abc',
  (char_length('abc') * 3),
  'abc'
)

-- HANA, ORACLE, TERADATA
rpad(
  'abc',
  (length('abc') * 3),
  'abc'
)

-- SQLITE
"replace"(hex(zeroblob(3)), '00', 'abc')

-- ACCESS, DERBY, INFORMIX, REDSHIFT
/* UNSUPPORTED */
```

4.10.14.16. 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, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
replace('hello world', 'llo', 'y')

-- ASE
str_replace('hello world', 'llo', 'y')

-- SQLITE
"replace"('hello world', 'llo', 'y')

-- TERADATA
oreplace('hello world', 'llo', 'y')

-- DERBY, REDSHIFT
/* UNSUPPORTED */
```

4.10.14.17. 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, COCKROACHDB, EXASOL, HSQLDB, MARIADB, MYSQL, ORACLE, POSTGRES, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLSERVER, TERADATA, YUGABYTEDB
reverse('hello')

-- ACCESS, DB2, DERBY, FIREBIRD, H2, HANA, INFORMIX, MEMSQL, REDSHIFT, SQLITE, SYBASE, VERTICA
/* UNSUPPORTED */
```

4.10.14.18. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
right('hello world', 5)

-- DERBY
substr(
  'hello world',
  (length('hello world') + (1 - 5))
)

-- ORACLE, SQLITE
substr(
  'hello world',
  -5
)
```

4.10.14.19. 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, HANA, HSQLDB, 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, DERBY, EXASOL, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.14.20. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
rtrim('  hello  ')

-- FIREBIRD
trim(TRAILING FROM '  hello  ')
```

4.10.14.21. 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, DB2, EXASOL, H2, MARIADB, MYSQL, SNOWFLAKE, SQLDATAWAREHOUSE, SQLSERVER, SYBASE, VERTICA
space(3)

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, HSQLDB, POSTGRES, YUGABYTEDB
repeat(' ', 3)

-- FIREBIRD, HANA, INFORMIX, MEMSQL, ORACLE, TERADATA
rpad(' ', 3, ' ')

-- SQLITE
' ' || substr("replace"(hex(zeroblob(3)), '00', ' '), 1, 3 - length(' '))

-- ACCESS, DERBY, REDSHIFT
/* UNSUPPORTED */
```

4.10.14.22. SPLIT_PART

The SPLIT_PART() function splits a string into substrings and retrieves the nth part, starting from 1.

```
SELECT split_part('a,b,c', ',', 2);
```

```
create.select(splitPart(val("a,b,c"), ",", 2)).fetch();
```

The result being

```
+------------+
| split_part |
+------------+
| b          |
+------------+
```

Dialect support

This example using jOOQ:

```
splitPart(val("a,b,c"), ",", 2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
substring(
  substring_index('a,b,c', ',', 2),
  CASE 2
    WHEN 1 THEN 1
    ELSE (char_length(substring_index(
      'a,b,c',
      ',',
      (2 - 1)
    )) + char_length(',') + 1)
  END
)

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, REDSHIFT, SNOWFLAKE, VERTICA, YUGABYTEDB
split_part('a,b,c', ',', 2)

-- BIGQUERY
split('a,b,c', ',')[ORDINAL(2)]

-- DB2, ORACLE
coalesce(
  substr(
    'a,b,c',
    nullif(
      decode(
        2,
        1,
        1,
        (nullif(
          instr(
            'a,b,c',
            ',',
            1,
            nullif(
              (2 - 1),
              0
            )
          ),
          0
        ) + length(','))
      ),
      (length('a,b,c') + 1)
    ),
    coalesce(
      (nullif(
        instr('a,b,c', ',', 1, 2),
        0
      ) - decode(
        2,
        1,
        1,
        (nullif(
          instr(
            'a,b,c',
            ',',
            1,
            nullif(
              (2 - 1),
              0
            )
          ),
          0
        ) + length(','))
      )),
      ((length('a,b,c') - nullif(
        instr(
          'a,b,c',
          ',',
          1,
          nullif(
            (2 - 1),
            0
          )
        ),
        0
      )) - (length(',') - 1))
    )
  ),
  ''
)

-- SQLSERVER
coalesce(
  (
    SELECT value
    FROM string_split('a,b,c', ',', 1)
    WHERE ordinal = 2
  ),
  ''
)

-- TERADATA
strtok('a,b,c', ',', 2)

-- ACCESS, ASE, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, SQLDATAWAREHOUSE, SQLITE, SYBASE
/* UNSUPPORTED */
```

4.10.14.23. 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, COCKROACHDB, EXASOL, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, POSTGRES,
-- REDSHIFT, SNOWFLAKE, SYBASE, VERTICA, YUGABYTEDB
substring('hello world', 7)

-- DB2, DERBY, INFORMIX, ORACLE, SQLITE
substr('hello world', 7)

-- FIREBIRD, TERADATA
substring('hello world' FROM 7)
```

4.10.14.24. SUBSTRING_INDEX

The SUBSTRING_INDEX() function gets a substring of a string, from the beginning until the nth occurrence of a delimiter.

```
SELECT
  substring_index('a,b,c,d', ',', 2),
  substring_index('a,b,c,d', ',', 3);
```

```
create.select(
  substringIndex(val("a,b,c,d"), ",", 2),
  substringIndex(val("a,b,c,d"), ",", 3)).fetch();
```

The result being

```
+-----------------+-----------------+
| substring_index | substring_index |
+-----------------+-----------------+
| a,b             | a,b,c           |
+-----------------+-----------------+
```

Dialect support

This example using jOOQ:

```
substringIndex(val("a,b,c,d"), ",", 3)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, MARIADB, MEMSQL, MYSQL
substring_index('a,b,c,d', ',', 3)

-- DB2, ORACLE
coalesce(
  substr(
    'a,b,c,d',
    1,
    (nullif(
      instr('a,b,c,d', ',', 1, 3),
      0
    ) - 1)
  ),
  'a,b,c,d'
)

-- VERTICA
coalesce(
  substring(
    'a,b,c,d',
    1,
    (nullif(
      instr('a,b,c,d', ',', 1, 3),
      0
    ) - 1)
  ),
  'a,b,c,d'
)

-- ACCESS, ASE, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, POSTGRES,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.14.25. TO_CHAR

The TO_CHAR() function converts a value to a string value using a vendor-specific format mask.

```
SELECT to_char(date '2000-01-01, 'YYYY/MM/DD');
```

```
create.select(toChar(Date.valueOf("2000-01-01"), "YYYY/MM/
DD")).fetch();
```

The result being

```
+------------+
| to_char    |
+------------+
| 2000/01/01 |
+------------+
```

Dialect support

This example using jOOQ:

```
toChar(Date.valueOf("2000-01-01"), "YYYY/MM/DD")
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, DB2, H2, ORACLE, POSTGRES, REDSHIFT
to_char(DATE '2000-01-01', 'YYYY/MM/DD')

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL,
-- MYSQL, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.14.26. TO_HEX

The TO_HEX() function translates a numeric value to its hexadecimal string counterpart.

```
SELECT to_hex(255);
```

```
create.select(toHex(255)).fetch();
```

The result being

```
+--------+
| to_hex |
+--------+
| ff     |
+--------+
```

Dialect support

This example using jOOQ:

```
toHex(255)
```

Translates to the following dialect specific expressions:

```
-- COCKROACHDB, DB2, POSTGRES, REDSHIFT, VERTICA, YUGABYTEDB
to_hex(255)

-- H2, ORACLE
trim(to_char(255, 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'))

-- MARIADB, MYSQL
hex(255)

-- SQLITE
printf('%X', 255)

-- SQLSERVER
format(255, 'X')

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL,
-- SNOWFLAKE, SQLDATAWAREHOUSE, SYBASE, TERADATA
/* UNSUPPORTED */
```

4.10.14.27. 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, COCKROACHDB, EXASOL, H2, HSQLDB, ORACLE, POSTGRES, SNOWFLAKE, SQLSERVER, VERTICA,
-- YUGABYTEDB
translate('1 * [2 + 3]', '[]', '()')

-- DB2
translate('1 * [2 + 3]', '()', '[]')

-- TERADATA
otranslate('1 * [2 + 3]', '[]', '()')

-- ACCESS, ASE, AURORA_MYSQL, DERBY, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, REDSHIFT, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE
/* UNSUPPORTED */
```

4.10.14.28. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA,
-- YUGABYTEDB
trim('  hello  ')

-- ASE, SQLDATAWAREHOUSE
ltrim(rtrim('  hello  '))
```

4.10.14.29. 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, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
upper('hello')
```

4.10.14.30. UUID

The UUID() function generates a new random UUID

```
SELECT uuid();
```

```
create.select(uuid()).fetch();
```

The result being

```
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 1fc454e5-b9f6-4d55-b783-5987fe76cb45 |
+--------------------------------------+
```

Dialect support

This example using jOOQ:

```
uuid()
```

Translates to the following dialect specific expressions:

```
-- ACCESS
genguid()

-- ASE
newid(-1)

-- BIGQUERY
generate_uuid()

-- COCKROACHDB, POSTGRES
gen_random_uuid()

-- DB2
CAST(regexp_replace((hex(rand()) || hex(generate_unique())), '(.{8})(.{4})(.{4})(.{4})(.{12}).*', '$1-$2-$3-$4-$5') AS char(36))

-- FIREBIRD
uuid_to_char(gen_uuid())

-- H2
random_uuid()

-- HANA
CAST(replace_regexpr('(.{8})(.{4})(.{4})(.{4})(.{12}).*' IN CAST(sysuuid AS char(36)) WITH '\1-\2-\3-\4-\5') AS char(36))

-- HSQLDB, MARIADB, MYSQL
uuid()

-- ORACLE
CAST(regexp_replace(rawtohex(sys_guid()), '(.{8})(.{4})(.{4})(.{4})(.{12}).*', '\1-\2-\3-\4-\5') AS varchar2(36))

-- SNOWFLAKE
uuid_string()

-- SQLITE
(
  SELECT ((((substr(u, 1, 8) || '-') || (substr(u, 9, 4) || '-')) || (substr(u, 13, 4) || '-')) || (substr(u, 17, 4) || '-')) ||
 substr(u, 21))
  FROM (
    SELECT lower(hex(randomblob(16))) u
  ) t
)

-- SQLSERVER
newid()

-- VERTICA
uuid_generate()

-- AURORA_MYSQL, AURORA_POSTGRES, DERBY, EXASOL, INFORMIX, MEMSQL, REDSHIFT, SQLDATAWAREHOUSE, SYBASE, TERADATA,
-- YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15. 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)

4.10.15.1. 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, MEMSQL, MYSQL
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))

-- 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))

-- COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, REDSHIFT, SNOWFLAKE, TERADATA, 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))

-- 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))

-- 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))
```

4.10.15.2. 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, MARIADB, MEMSQL, MYSQL, SNOWFLAKE
current_date()

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, POSTGRES, REDSHIFT, SQLITE,
-- TERADATA, VERTICA, YUGABYTEDB
CURRENT_DATE

-- INFORMIX
CURRENT YEAR TO DAY

-- ORACLE
trunc(current_date)

-- SQLDATAWAREHOUSE, SQLSERVER
convert(DATE, current_timestamp)

-- SYBASE
CURRENT DATE
```

4.10.15.3. 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, MARIADB, MEMSQL, MYSQL, SNOWFLAKE
current_date()

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, POSTGRES, REDSHIFT, SQLITE,
-- TERADATA, VERTICA, YUGABYTEDB
CURRENT_DATE

-- INFORMIX
CURRENT YEAR TO DAY

-- ORACLE
trunc(current_date)

-- SQLDATAWAREHOUSE, SQLSERVER
convert(DATE, current_timestamp)

-- SYBASE
CURRENT DATE
```

4.10.15.4. 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, MARIADB, MEMSQL, MYSQL, SNOWFLAKE
current_timestamp()

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, ORACLE, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, TERADATA, VERTICA, YUGABYTEDB
CURRENT_TIMESTAMP

-- INFORMIX
CURRENT YEAR TO FRACTION (5)

-- SYBASE
CURRENT TIMESTAMP
```

4.10.15.5. 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, FIREBIRD, H2, HANA, HSQLDB, POSTGRES, REDSHIFT, SQLITE, TERADATA,
-- VERTICA, YUGABYTEDB
CURRENT_TIME

-- EXASOL, ORACLE
current_timestamp

-- INFORMIX
CURRENT HOUR TO SECOND

-- SQLDATAWAREHOUSE, SQLSERVER
convert(TIME, current_timestamp)

-- SYBASE
CURRENT TIME
```

4.10.15.6. 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, MARIADB, MEMSQL, MYSQL
CAST(current_timestamp() AS timestamp with time zone)

-- AURORA_POSTGRES, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, ORACLE, POSTGRES, REDSHIFT, SQLITE,
-- TERADATA, VERTICA, YUGABYTEDB
CAST(CURRENT_TIMESTAMP AS timestamp with time zone)

-- 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)

-- SQLDATAWAREHOUSE, SQLSERVER
CAST(CURRENT_TIMESTAMP AS datetimeoffset)

-- SYBASE
CAST(CURRENT TIMESTAMP AS timestamp with time zone)
```

4.10.15.7. 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, FIREBIRD, H2, HANA, HSQLDB, POSTGRES, REDSHIFT, SQLITE, TERADATA,
-- VERTICA, YUGABYTEDB
CAST(CURRENT_TIME AS time with time zone)

-- 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)
```

4.10.15.8. 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, FIREBIRD, H2, HANA, HSQLDB, POSTGRES, REDSHIFT, SQLITE, TERADATA,
-- VERTICA, YUGABYTEDB
CURRENT_TIME

-- EXASOL, ORACLE
current_timestamp

-- INFORMIX
CURRENT HOUR TO SECOND

-- SQLDATAWAREHOUSE, SQLSERVER
convert(TIME, current_timestamp)

-- SYBASE
CURRENT TIME
```

4.10.15.9. 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, MARIADB, MEMSQL, MYSQL, SNOWFLAKE
current_timestamp()

-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, ORACLE, POSTGRES, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, TERADATA, VERTICA, YUGABYTEDB
CURRENT_TIMESTAMP

-- INFORMIX
CURRENT YEAR TO FRACTION (5)

-- SYBASE
CURRENT TIMESTAMP
```

4.10.15.10. 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, MEMSQL, MYSQL
{d '2020-02-03'}

-- AURORA_POSTGRES, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, TERADATA, VERTICA
DATE '2020-02-03'

-- DERBY
DATE('2020-02-03')

-- INFORMIX
DATETIME(2020-02-03) YEAR TO DAY

-- SQLDATAWAREHOUSE, SQLSERVER
CAST('2020-02-03' AS date)

-- BIGQUERY, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.11. 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, MEMSQL, MYSQL
date_add({d '2020-02-03'}, INTERVAL 3 DAY)

-- AURORA_POSTGRES, COCKROACHDB, 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)

-- 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)

-- MARIADB
date_add(DATE '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)
```

4.10.15.12. 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, MEMSQL, MYSQL
datediff({d '2020-02-03'}, {d '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)

-- 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')) }

-- EXASOL
CAST((DATE '2020-02-03' - DATE '2020-02-01') AS int)

-- 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)

-- MARIADB
datediff(DATE '2020-02-03', DATE '2020-02-01')

-- REDSHIFT
datediff('day', DATE '2020-02-01', DATE '2020-02-03')

-- 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

-- TERADATA
CAST((DATE '2020-02-03' - DATE '2020-02-01') AS integer)
```

SQL Server 3 argument version

In SQL Server, there is a 3 argument verison of the DATEDIFF() function, where the result produces the number of date part periods between the two dates, with the dates being TRUNC-ed to the relevant date part. The argument order is in the order of the interval notation: [start_date, end_date]

```
SELECT DATEDIFF(
  MONTH
  DATE '2020-02-03',
  DATE '2020-04-01');
```

```
create.select(dateDiff(
  DatePart.MONTH,
  Date.valueOf("2020-02-03"),
  Date.valueOf("2020-04-01"))).fetch();
```

The result being

```
+------------+
| datediff   |
+------------+
|          2 |
+------------+
```

Notice the truncation happening prior to calculating the difference. The result is the same as for:

```
SELECT DATEDIFF(
  MONTH
  DATE '2020-02-01',
  DATE '2020-04-01');
```

```
create.select(dateDiff(
  DatePart.MONTH,
  Date.valueOf("2020-02-01"),
  Date.valueOf("2020-04-01"))).fetch();
```

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, MEMSQL, MYSQL
(((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'})))

-- AURORA_POSTGRES, COCKROACHDB, HANA, MARIADB, 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')

-- 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))

-- ACCESS, ASE, EXASOL, INFORMIX, SQLITE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.15.13. 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).

```sql
SELECT DATE '2020-02-03' - 2;
```

```java
create.select(dateSub(Date.valueOf("2020-02-03"), 2)).fetch();
```

The result being

```
+------------+
| date_sub   |
+------------+
| 2020-02-01 |
+------------+
```

Dialect support

This example using jOOQ:

```java
dateSub(Date.valueOf("2020-02-03"), 2)
```

Translates to the following dialect specific expressions:

```sql
-- ACCESS
dateadd('d', -2, #2020/02/03#)

-- ASE, SYBASE
dateadd(DAY, -2, '2020-02-03')

-- AURORA_MYSQL, MEMSQL, MYSQL
date_add({d '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)

-- DB2, HSQLDB
(DATE '2020-02-03' - (2) day)

-- DERBY
CAST({fn timestampadd(SQL_TSI_DAY, -2, DATE('2020-02-03')) } AS DATE)

-- EXASOL, H2, ORACLE, VERTICA
(DATE '2020-02-03' - 2)

-- 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)

-- MARIADB
date_add(DATE '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)
```

4.10.15.14. 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, MEMSQL, MYSQL
extract(DAY FROM {ts '2020-02-03 00:00:00.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)

-- 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)
```

4.10.15.15. 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, MEMSQL, MYSQL
dayofyear({ts '2020-02-03 00:00:00.0'})

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES
extract(DOY FROM TIMESTAMP '2020-02-03 00:00:00.0')

-- DB2, HANA, MARIADB
dayofyear(TIMESTAMP '2020-02-03 00:00:00.0')

-- H2, HSQLDB
extract(DAY_OF_YEAR FROM TIMESTAMP '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, DERBY, EXASOL, FIREBIRD, INFORMIX, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.16. 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, MEMSQL, MYSQL
floor((extract(YEAR FROM {ts '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))

-- COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA
floor((extract(YEAR FROM TIMESTAMP '2020-02-03 00:00:00.0') / 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))

-- 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))
```

4.10.15.17. 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, MEMSQL, MYSQL
UNIX_TIMESTAMP({ts '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')

-- HSQLDB, MARIADB
UNIX_TIMESTAMP(TIMESTAMP '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, DERBY, EXASOL, FIREBIRD, INFORMIX, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.18. 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, MEMSQL, MYSQL
extract(MONTH FROM {ts '2020-02-03 00:00:00.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)

-- 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)
```

4.10.15.19. 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, MEMSQL, MYSQL
extract(HOUR FROM {ts '2020-02-03 15:30:45.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- DB2
HOUR(TIMESTAMP '2020-02-03 15:30:45.0')

-- 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)))

-- 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)
```

4.10.15.20. 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, MEMSQL, MYSQL
weekday({ts '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)

-- HSQLDB
(mod(
  (EXTRACT(DAY_OF_WEEK FROM TIMESTAMP '2020-02-03 00:00:00.0') + 5),
  7
) + 1)

-- MARIADB
weekday(TIMESTAMP '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, DERBY, EXASOL, FIREBIRD, INFORMIX, REDSHIFT, SNOWFLAKE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.21. 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, MEMSQL, MYSQL
{d '2020-02-03'}

-- AURORA_POSTGRES, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, TERADATA, VERTICA
DATE '2020-02-03'

-- DERBY
DATE('2020-02-03')

-- INFORMIX
DATETIME(2020-02-03) YEAR TO DAY

-- SQLDATAWAREHOUSE, SQLSERVER
CAST('2020-02-03' AS date)

-- BIGQUERY, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.22. 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, MEMSQL, MYSQL
date_add({d '2020-02-03'}, INTERVAL 3 DAY)

-- AURORA_POSTGRES, COCKROACHDB, 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)

-- 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)

-- MARIADB
date_add(DATE '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)
```

4.10.15.23. 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, MEMSQL, MYSQL
date_add({d '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)

-- DB2, HSQLDB
(DATE '2020-02-03' - (2) day)

-- DERBY
CAST({fn timestampadd(SQL_TSI_DAY, -2, DATE('2020-02-03')) } AS DATE)

-- EXASOL, H2, ORACLE, VERTICA
(DATE '2020-02-03' - 2)

-- 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)

-- MARIADB
date_add(DATE '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)
```

4.10.15.24. 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, MEMSQL, MYSQL
{ts '2020-02-03 15:30:45.0'}

-- AURORA_POSTGRES, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, TERADATA, VERTICA
TIMESTAMP '2020-02-03 15:30:45.0'

-- DERBY
TIMESTAMP('2020-02-03 15:30:45.0')

-- INFORMIX
DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION

-- SQLDATAWAREHOUSE, SQLSERVER
CAST('2020-02-03 15:30:45.0' AS DATETIME2)

-- BIGQUERY, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.25. 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, MEMSQL, MYSQL
date_add({ts '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)

-- 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)

-- EXASOL, H2, ORACLE, VERTICA
(TIMESTAMP '2020-02-03 15:30:45.0' + 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)

-- MARIADB
date_add(TIMESTAMP '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))

-- 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)
```

4.10.15.26. 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, MEMSQL, MYSQL
date_add({ts '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)

-- DB2, HSQLDB
(TIMESTAMP '2020-02-03 15:30:45.0' - (2) day)

-- DERBY
CAST({fn timestampadd(SQL_TSI_DAY, -2, TIMESTAMP('2020-02-03 15:30:45.0')) } AS TIMESTAMP)

-- EXASOL, H2, ORACLE, VERTICA
(TIMESTAMP '2020-02-03 15:30:45.0' - 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)

-- MARIADB
date_add(TIMESTAMP '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)
```

4.10.15.27. 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, AURORA_MYSQL, MEMSQL, MYSQL
{t '15:30:45'}

-- ASE, SQLITE, SYBASE
'15:30:45'

-- AURORA_POSTGRES, COCKROACHDB, DB2, FIREBIRD, H2, HANA, HSQLDB, MARIADB, 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, EXASOL, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.28. 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, MEMSQL, MYSQL
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))

-- 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))

-- COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, REDSHIFT, SNOWFLAKE, TERADATA, 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))

-- 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))

-- 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))
```

4.10.15.29. 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, MEMSQL, MYSQL
extract(MINUTE FROM {ts '2020-02-03 15:30:45.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)))

-- SQLDATAWAREHOUSE, SQLSERVER
datepart(mi, CAST('2020-02-03 15:30:45.0' AS DATETIME2))

-- SQLITE
CAST(strftime('%M', '2020-02-03 15:30:45.0') AS int)
```

4.10.15.30. 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, MEMSQL, MYSQL
extract(MONTH FROM {ts '2020-02-03 00:00:00.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)

-- 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)
```

4.10.15.31. 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, MEMSQL, MYSQL
quarter({ts '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))

-- DB2, H2, HSQLDB, MARIADB
quarter(TIMESTAMP '2020-02-03 00:00:00.0')

-- DERBY
floor(((MONTH(TIMESTAMP('2020-02-03 00:00:00.0')) + 2) / 3))

-- EXASOL, FIREBIRD, HANA, REDSHIFT, TERADATA, VERTICA
floor(((extract(MONTH FROM 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))

-- 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))
```

4.10.15.32. 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, MEMSQL, MYSQL
extract(SECOND FROM {ts '2020-02-03 15:30:45.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)))

-- 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)
```

4.10.15.33. 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, AURORA_MYSQL, MEMSQL, MYSQL
{t '15:30:45'}

-- ASE, SQLITE, SYBASE
'15:30:45'

-- AURORA_POSTGRES, COCKROACHDB, DB2, FIREBIRD, H2, HANA, HSQLDB, MARIADB, 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, EXASOL, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.34. 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, MEMSQL, MYSQL
{ts '2020-02-03 15:30:45.0'}

-- AURORA_POSTGRES, COCKROACHDB, DB2, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, TERADATA, VERTICA
TIMESTAMP '2020-02-03 15:30:45.0'

-- DERBY
TIMESTAMP('2020-02-03 15:30:45.0')

-- INFORMIX
DATETIME(2020-02-03 15:30:45.0) YEAR TO FRACTION

-- SQLDATAWAREHOUSE, SQLSERVER
CAST('2020-02-03 15:30:45.0' AS DATETIME2)

-- BIGQUERY, REDSHIFT, SNOWFLAKE, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.35. 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, MEMSQL, MYSQL
date_add({ts '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)

-- 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')) }

-- EXASOL, H2, ORACLE, VERTICA
(TIMESTAMP '2020-02-03 15:30:45.0' + 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)

-- MARIADB
date_add(TIMESTAMP '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))

-- 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)
```

4.10.15.36. 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_subd       |
+---------------------+
| 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, MEMSQL, MYSQL
date_add({ts '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)

-- DB2, HSQLDB
(TIMESTAMP '2020-02-03 15:30:45.0' - (2) day)

-- DERBY
{fn timestampadd(SQL_TSI_DAY, -2, TIMESTAMP('2020-02-03 15:30:45.0')) }

-- EXASOL, H2, ORACLE, VERTICA
(TIMESTAMP '2020-02-03 15:30:45.0' - 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)

-- MARIADB
date_add(TIMESTAMP '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)
```

4.10.15.37. 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, EXASOL, HSQLDB, ORACLE, POSTGRES, VERTICA, YUGABYTEDB
to_date('20200203', 'YYYYMMDD')

-- SQLDATAWAREHOUSE, SQLSERVER
convert(
  date,
  '20200203',
  112
)

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, FIREBIRD, H2, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, REDSHIFT,
-- SNOWFLAKE, SQLITE, SYBASE, TERADATA
/* UNSUPPORTED */
```

4.10.15.38. 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')

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.39. 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')

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.40. 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, EXASOL, HSQLDB, ORACLE, POSTGRES, SQLDATAWAREHOUSE, SQLSERVER, VERTICA, YUGABYTEDB
to_timestamp('20200203153045', 'YYYYMMDDHH24MISS')

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DERBY, FIREBIRD, H2, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- REDSHIFT, SNOWFLAKE, SQLITE, SYBASE, TERADATA
/* UNSUPPORTED */
```

4.10.15.41. 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')

-- ACCESS, ASE, AURORA_MYSQL, DERBY, EXASOL, FIREBIRD, HANA, MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.15.42. 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, MEMSQL, MYSQL
extract(YEAR FROM {ts '2020-02-03 00:00:00.0'})

-- AURORA_POSTGRES, COCKROACHDB, EXASOL, FIREBIRD, H2, HANA, HSQLDB, MARIADB, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE,
-- TERADATA, 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')

-- 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)

-- 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)
```

4.10.16. 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.

4.10.16.1. ARRAY_GET

The ARRAY_GET function allows for accessing array elements by 1-based ordinal.

```
SELECT (ARRAY[1, 2])[1]
```

```
create.select(arrayGet(array(1, 2), 1)).fetch();
```

The result would look like this:

```
+-----------+
| array_get |
+-----------+
|         1 |
+-----------+
```

Dialect support

This example using jOOQ:

```
arrayGet(array(1, 2), 1)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, H2, POSTGRES, YUGABYTEDB
(ARRAY[1, 2])[1]

-- HSQLDB
CASE
  WHEN cardinality(ARRAY[1, 2]) >= 1 THEN (ARRAY[1, 2])[1]
END

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.16.2. 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, COCKROACHDB, H2, HSQLDB, POSTGRES, YUGABYTEDB
ARRAY[1, 2]

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.16.3. ARRAY constructor from subquery

In order to construct an ad-hoc ARRAY type from a subquery, the ARRAY constructor can be used.

```
SELECT
  T_AUTHOR.ID,
  ARRAY(
    SELECT ID
    FROM T_BOOK
    WHERE T_BOOK.AUTHOR_ID = T_AUTHOR.ID
  )
FROM T_AUTHOR
```

```
create.select(
  T_AUTHOR.ID,
  array(
    select(T_BOOK.ID)
    .from(T_BOOK)
    .where(T_BOOK.AUTHOR_ID.eq(T_AUTHOR.ID))))
  .from(T_AUTHOR)
  .fetch();
```

Unlike ARRAY_AGG, this ARRAY constructor does not act as an aggregate function, and thus does not produce aggregate semantics, such as requiring an explicit (or producing an implicit) GROUP BY clause.

The result would look like this:

```
+----+----------+
| ID |          |
+----+----------+
|  1 | [ 1, 2 ] |
|  2 | [ 3, 4 ] |
+----+----------+
```

Dialect support

This example using jOOQ:

```
array(select(val(1)))
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, YUGABYTEDB
ARRAY(
  SELECT 1
)

-- H2
(
  SELECT array_agg(t.c)
  FROM (
    SELECT 1
  ) t (c)
)

-- HSQLDB
ARRAY(
  SELECT 1
  FROM (VALUES(1)) AS dual(dual)
)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.16.4. CARDINALITY

The CARDINALITY function allows for getting the size of an array.

```
SELECT CARDINALITY(ARRAY[1, 2])
```

```
create.select(cardinality(array(1, 2))).fetch();
```

The result would look like this:

```
+-------------+
| cardinality |
+-------------+
|           2 |
+-------------+
```

Dialect support

This example using jOOQ:

```
cardinality(array(1, 2))
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, H2, HSQLDB, POSTGRES, YUGABYTEDB
cardinality(ARRAY[1, 2])

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- ORACLE, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.17. 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, but there are also conditions, aggregate functions, and table valued functions for JSON usage.

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.

4.10.17.1. JSON_ARRAY

The JSON_ARRAY function is used to produce simple JSON arrays from scalar values, without aggregation (see also JSON_ARRAYAGG)

```
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))

-- COCKROACHDB
json_build_array(CAST(1 AS int4), CAST(2 AS int4))

-- DB2, H2, MARIADB, MYSQL, SQLITE
json_array(1, 2)

-- ORACLE
json_array(nvl(NULL, 1), nvl(NULL, 2))

-- 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
)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SQLDATAWAREHOUSE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.17.2. JSON_OBJECT

The JSON_OBJECT function is used to produce simple JSON objects from scalar values, without aggregation (see also JSON_OBJECTAGG)

```
SELECT json_object(
  KEY 'firstName' VALUE author.first_name,
  KEY 'lastName'  VALUE author.last_name
)
FROM author
```

```
create.select(jsonObject(
          key("firstName").value(AUTHOR.FIRST_NAME),
          key("lastName").value(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)

-- DB2, H2, ORACLE
json_object(KEY 'firstName' VALUE AUTHOR.FIRST_NAME)

-- MARIADB, MYSQL, SQLITE
json_object('firstName', 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
  )
)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SQLDATAWAREHOUSE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.17.3. JSON_VALUE

The JSON_VALUE function is used to extract content from JSON documents using a JSON path expression.

```
SELECT json_value(
  '{"a":[1,2,3]}',
  '$.a[1]'
)
FROM dual
```

```
create.select(jsonValue(
          val(JSON.valueOf("{\"a\":[1,2,3]}")),
          "$.a[1]"
      )
      .fetch();
```

The result would look like this:

```
+------------+
| json_value |
+------------+
| 2          |
+------------+
```

If the value does not matter, but you just want to check for a value's existence, use the JSON_EXISTS predicate.

Dialect support

This example using jOOQ:

```
jsonValue(val(json("[1,2]")), "$[*]")
```

Translates to the following dialect specific expressions:

```
-- DB2, MARIADB, ORACLE
json_value('[1,2]', '$[*]')

-- MYSQL, SQLITE
json_extract('[1,2]', '$[*]')

-- POSTGRES, YUGABYTEDB
jsonb_path_query_first(
  CAST('[1,2]' AS jsonb),
  cast('$[*]' as jsonpath)
)

-- SQLSERVER
(
  SELECT c
  FROM openjson('[1,2]', '$') WITH (c varchar(max) '$[*]')
)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MEMSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.18. XML functions

jOOQ 3.14 introduced support for the org.jooq.XML type, which are used to wrap string based XML documents in a type safe way. With this type, there are also a few standard XML functions that have been added to the API. This section describes scalar functions, but there are also conditions, aggregate functions, and table valued functions for XML usage.

Notice that XML is case sensitive in all dialects, but SQL identifiers may not be. jOOQ by default produces quoted identifiers (see the RenderQuotedNames Setting), which maintains the case you provide in Java.

4.10.18.1. XMLATTRIBUTES

The XMLATTRIBUTES() function is used to create attributes inside of XMLELEMENT().

```
SELECT xmlelement(
  NAME element,
  xmlattributes('value' AS attr)
)
```

```
create.select(xmlelement("element",
        xmlattributes(val("value").as("attr"))
      ))
      .fetch();
```

The result would look like this:

```
+------------------------+
| xmlelement             |
+------------------------+
| <element attr="value"/> |
+------------------------+
```

Dialect support

This example using jOOQ:

```
xmlelement("e", xmlattributes(val("value").as("attr")))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlelement(
  NAME e,
  xmlattributes('value' AS attr)
)

-- SQLSERVER
(
  SELECT
    1 tag,
    NULL parent,
    'value' [e!1!attr]
  FOR XML EXPLICIT, TYPE
)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.2. XMLCOMMENT

The XMLCOMMENT() function is used to create comments inside XML documents, at arbitrary positions.

```
SELECT xmlcomment('comment')
```

```
create.select(xmlcomment("comment"))
      .fetch();
```

The result would look like this:

```
+----------------+
| xmlcomment     |
+----------------+
| <!--comment--> |
+----------------+
```

Dialect support

This example using jOOQ:

```
xmlcomment("comment")
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlcomment('comment')

-- SQLSERVER
CAST((('<!--' + 'comment') + '-->') AS xml)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.3. XMLCONCAT

The XMLCONCAT() function is used to concatenate two XML fragments of arbitrary type

```
SELECT xmlconcat(
  xmlelement(NAME e1),
  xmlelement(NAME e2)
)
```

```
create.select(xmlconcat(
         xmlelement("e1"),
         xmlelement("e2")))
      .fetch();
```

The result would look like this:

```
+------------+
| xmlconcat  |
+------------+
| <e1/><e2/> |
+------------+
```

Dialect support

This example using jOOQ:

```
xmlconcat(xmlelement("e1"), xmlelement("e2"))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlconcat(
  xmlelement(NAME e1),
  xmlelement(NAME e2)
)

-- SQLSERVER
(
  SELECT
    (
      SELECT
        1 tag,
        NULL parent,
        NULL [e1!1]
      FOR XML EXPLICIT, TYPE
    ),
    (
      SELECT
        1 tag,
        NULL parent,
        NULL [e2!1]
      FOR XML EXPLICIT, TYPE
    )
  FOR XML PATH (''), TYPE
)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.4. XMLDOCUMENT

The XMLDOCUMENT() function is used to produce an XML document from document contents, useful in dialects where this is required in some cases.

```
SELECT xmldocument(xmlelement(NAME e1))
FROM sysibm.dual
```

```
create.select(xmldocument(xmlelement("e1")))
      .fetch();
```

The result would look like this:

```
+--------------+
| xmldocument  |
+--------------+
| <e1/>        |
+--------------+
```

Dialect support

This example using jOOQ:

```
xmldocument(xmlelement("e1"))
```

Translates to the following dialect specific expressions:

```
-- DB2, TERADATA
xmldocument(xmlelement(NAME e1))

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA,
-- YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.5. XMLELEMENT

The XMLELEMENT() function is used to create XML elements, possibly with attributes (see XMLATTRIBUTES())

```
SELECT
  xmlelement(NAME e1) AS e1,
  xmlelement(NAME e2, xmlattributes('1' AS a)) AS e2,
  xmlelement(NAME e3, 'text-content') AS e3,
  xmlelement(NAME e4, xmlelement(NAME nested)) AS e4
```

```
create.select(
         xmlelement("e1").as("e1"),
         xmlelement("e2",
 xmlattributes(val("1").as("a"))).as("e2"),
         xmlelement("e3", val("text-content")).as("e3"),
         xmlelement("e4", xmlelement("nested")).as("e4"))
      .fetch();
```

The result would look like this:

```
+-------+-------------+-----------------------+--------------------+
| e1    | e2          | e3                    | e4                 |
+-------+-------------+-----------------------+--------------------+
| <e1/> | <e2 a="1"/> | <e3>text-content</e3> | <e4><nested/></e4> |
+-------+-------------+-----------------------+--------------------+
```

Dialect support

This example using jOOQ:

```
xmlelement("e1", val("text-content"))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlelement(NAME e1, 'text-content')

-- SQLSERVER
(
  SELECT
    1 tag,
    NULL parent,
    'text-content' [e1!1]
  FOR XML EXPLICIT, TYPE
)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.6. XMLFOREST

The XMLFOREST() function is used to create XML forests from other elements

```
SELECT xmlforest(
  xmlelement(NAME e1) AS w1,
  xmlelement(NAME e2) AS w2
)
```

```
create.select(xmlforest(
         xmlelement("e1").as("w1"),
         xmlelement("e2").as("w2")))
      .fetch();
```

The result would look like this:

```
+------------------------------+
| xmlforest                    |
+------------------------------+
| <w1><e1/></w1><w2><e2/></w2> |
+------------------------------+
```

Dialect support

This example using jOOQ:

```
xmlforest(xmlelement("e1").as("w1"), xmlelement("e2").as("w2"))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlforest(
  xmlelement(NAME e1) AS w1,
  xmlelement(NAME e2) AS w2
)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.7. XMLPARSE

The XMLPARSE() function allows for explicit parsing of XML documents or elements for further processing, when implicit conversion from strings is not possible, or when it leads to more clarity.

```
SELECT
  xmlparse(document '<d/>') AS d,
  xmlparse(element '<e/>') AS e
```

```
create.select(
         xmlparseDocument("<d/>").as("d"),
         xmlparseElement("<e/>").as("e"))
      .fetch();
```

The result would look like this:

```
+------+------+
| d    | e    |
+------+------+
| <d/> | <e/> |
+------+------+
```

Dialect support

This example using jOOQ:

```
xmlparseDocument("<d/>")
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES
xmlparse(DOCUMENT '<d/>')

-- TERADATA
xmlparse(DOCUMENT '<d/>' PRESERVE WHITESPACE)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.8. XMLPI

The XMLPI() function produces XML processing instructions.

```
SELECT xmlpi(NAME "php")
```

```
create.select(xmlpi("php"))
      .fetch();
```

The result would look like this:

```
+---------+
| xmlpi   |
+---------+
| <?php?> |
+---------+
```

Dialect support

This example using jOOQ:

```
xmlpi("php")
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlpi(NAME php)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.9. XMLQUERY

The XMLQUERY() function allows for extracting content from an XML document using XQuery or XPath

```
SELECT xmlquery('/doc/x'
  PASSING xmlparse(DOCUMENT '<doc><x>content</x></doc>')
  RETURNING CONTENT
)
FROM dual
```

```
create.select(xmlquery("/doc/x")
        .passing(
            XML.xml("<doc><x>content</x></doc>")
        ))
      .fetch();
```

The result would look like this:

```
+----------+
| xmlquery |
+----------+
| content  |
+----------+
```

Dialect support

This example using jOOQ:

```
xmlquery("/doc/x").passing(xml("<doc><x>content</x></doc>"))
```

Translates to the following dialect specific expressions:

```
-- DB2, TERADATA
xmlquery(
  '/doc/x'
  PASSING '<doc><x>content</x></doc>'
)

-- ORACLE
xmlquery(
  '/doc/x'
  PASSING '<doc><x>content</x></doc>'
  RETURNING CONTENT
)

-- POSTGRES
(SELECT xmlagg(x)
FROM UNNEST(xpath('/doc/x', CAST('<doc><x>content</x></doc>' AS xml))) t (x))

-- SQLSERVER
'<doc><x>content</x></doc>'.QUERY('/doc/x')

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.18.10. XMLSERIALIZE

The XMLSERIALIZE() function allows for serializing XML content into a string type

```
SELECT xmlserialize(CONTENT xmlelement(NAME 'a') AS VARCHAR)
FROM dual
```

```
create.select(xmlserializeContent(xmlelement("a"), VARCHAR))
      .fetch();
```

The result would look like this:

```
+--------------+
| xmlserialize |
+--------------+
| <a/>         |
+--------------+
```

Dialect support

This example using jOOQ:

```
xmlserializeContent(xmlelement("a"), VARCHAR(1000))
```

Translates to the following dialect specific expressions:

```
-- DB2, POSTGRES, TERADATA
xmlserialize(CONTENT xmlelement(NAME a) AS varchar(1000))

-- ORACLE
xmlserialize(CONTENT xmlelement(NAME a) AS varchar2(1000))

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.19. System functions

Some system functions are supported by jOOQ.

4.10.19.1. 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

-- DERBY
CURRENT SCHEMA

-- FIREBIRD, SQLITE
''

-- H2
SCHEMA()

-- ORACLE
user

-- SNOWFLAKE, VERTICA
CURRENT_SCHEMA()

-- SQLDATAWAREHOUSE, SQLSERVER
schema_name()

-- TERADATA
DATABASE

-- ACCESS, ASE, BIGQUERY, EXASOL, HANA, INFORMIX, REDSHIFT, SYBASE
/* UNSUPPORTED */
```

4.10.19.2. 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, H2, MARIADB, MEMSQL, MYSQL, SNOWFLAKE
current_user()

-- AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, FIREBIRD, HANA, HSQLDB, POSTGRES, SQLDATAWAREHOUSE, SQLSERVER, SYBASE,
-- TERADATA, YUGABYTEDB
current_user

-- SQLITE
''

-- ACCESS, BIGQUERY, EXASOL, REDSHIFT, VERTICA
/* UNSUPPORTED */
```

4.10.20. Spatial functions

A few databases have implemented the ISO/IEC 13249-3 SQL standard spatial extensions (or vendor specific adaptations thereof) to calculate geometric or geographic sets.

These extensions are very useful, and SQL is a perfect match for them. Starting with jOOQ 3.16, we support quite a few of these spatial functions, spatial aggregate and window functions, spatial predicates, and more.

Before moving on with the following chapters of the jOOQ manual, get yourself acquainted with the various standards around spatial data, the WKT (Well-known text representation of geometry), and other related topics, to see how a polygon like:

```
POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))
```

... can produce the equivalent:

4.10.20.1. ST_Area

This function calculates the area of a polygonal geometry. For example:

```
create.select(stArea(stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"))).fetch();
```

The result being, for example

```
+---------+
| ST_Area |
+---------+
| 4       |
+---------+
```

Dialect support

This example using jOOQ:

```
stArea(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_area(geometry)

-- ORACLE
sdo_geom.sdo_area(geometry, tol => null)

-- SQLSERVER
geometry.STArea()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.2. ST_AsText

This function produces the WKT (Well-known text representation of geometry). For example:

```
create.select(stAsText(stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"))).fetch();
```

The result being, for example

```
+------------------------------------------+
| ST_GeomFromText                          |
+------------------------------------------+
| POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1)) |
+------------------------------------------+
```

Dialect support

This example using jOOQ:

```
stAsText(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_astext(geometry)

-- ORACLE
(geometry).Get_WKT()

-- SQLSERVER
geometry.STAsText()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.3. ST_Centroid

This function calculates the geometric center of mass of a geometry. For example:

```
create.select(stCentroid(stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"))).fetch();
```

The result being, for example

```
+-------------+
| ST_Centroid |
+-------------+
| POINT (0 0) |
+-------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stCentroid(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_centroid(geometry)

-- ORACLE
sdo_geom.sdo_centroid(geometry, tol => null)

-- SQLSERVER
geometry.STCentroid()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.4. ST_Difference

This function subtracts a geometry from another. For example:

```
create.select(stDifference(
  stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"),
  stGeomFromText("POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))"),
)).fetch();
```

The result being, for example

```
+----------------------------------------------------+
| ST_Difference                                      |
+----------------------------------------------------+
| POLYGON ((-1 1, 0 1, 0 0, 1 0, 1 -1, -1 -1, -1 1)) |
+----------------------------------------------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stDifference(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_difference(geometry1, geometry2)

-- ORACLE
sdo_geom.sdo_difference(geometry1, geometry2, tol => null)

-- SQLSERVER
geometry1.STDifference(geometry2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.5. ST_Distance

This function calculates the distance between two geometries. For example:

```
create.select(stDistance(
  stGeomFromText("POINT (0 0)"),
  stGeomFromText("POINT (1 0)"),
)).fetch();
```

The result being, for example

```
+-------------+
| ST_Distance |
+-------------+
| 1.0         |
+-------------+
```

Dialect support

This example using jOOQ:

```
stDistance(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_distance(geometry1, geometry2)

-- ORACLE
sdo_geom.sdo_distance(geometry1, geometry2, tol => null)

-- SQLSERVER
geometry1.STDistance(geometry2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.6. ST_EndPoint

This function returns the last point of a linestring. For example:

```
create.select(stEndPoint(stGeomFromText("LINESTRING (0 0, 1 1, 2 0, 3 1)"))).fetch();
```

The result being, for example

```
+-------------+
| ST_EndPoint |
+-------------+
| POINT (3 1) |
+-------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stEndPoint(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_endpoint(geometry)

-- SQLSERVER
geometry.STEndPoint()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.7. ST_ExteriorRing

This function returns a linestring corresponding to the exterior ring of a polygon geometry. For example:

```
create.select(stExteriorRing(
  stGeomFromText("""
    POLYGON (
      (-3 -3, 3 -3, 3 3, -3 3, -3 -3),
      (-2 -2, 2 -2, 2 2, -2 2, -2 -2),
      (-1 -1, 1 -1, 1 1, -1 1, -1 -1)
    )
  """)
)).fetch();
```

The result being, for example

```
+-------------------------------------------+
| ST_ExteriorRing                           |
+-------------------------------------------+
| LINESTRING (-3 -3, 3 -3, 3 3, -3 3, -3 -3) |
+-------------------------------------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stExteriorRing(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_exteriorring(geometry)

-- ORACLE
sdo_util.extract(geometry, 1, 1)

-- SQLSERVER
geometry.STExteriorRing()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.8. ST_GeometryN

This function returns the N-th geometry (1-based) from a geometry that contains multiple nested geometries. For example:

```
create.select(stGeometryN(stGeomFromText("MULTIPOINT ((-1 -1), (1 -1), (1 1), (-1 1))"), 2)).fetch();
```

The result being, for example

```
+--------------+
| ST_GeometryN |
+--------------+
| POINT (1 -1) |
+--------------+
```

Or, visually:

To get the total number of geometries, use ST_NumGeometries.

Dialect support

This example using jOOQ:

```
stGeometryN(geometry, 2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_geometryn(geometry, 2)

-- SQLSERVER
geometry.STGeometryN(2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.9. ST_GeometryType

This function returns the type name of a geometry (which may be vendor specific). For example:

```
create.select(stGeometryType(stGeomFromText("POINT (1 1)"))).fetch();
```

The result being, for example

```
+-----------------+
| ST_GeometryType |
+-----------------+
| ST_Point        |
+-----------------+
```

Dialect support

This example using jOOQ:

```
stGeometryType(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_geometrytype(geometry)

-- ORACLE
decode(
  mod(
    (geometry).sdo_gtype,
    100
  ),
  1,
  'POINT',
  2,
  'LINE',
  3,
  'POLYGON',
  4,
  'COLLECTION',
  5,
  'MULTIPOINT',
  6,
  'MULTILINE',
  7,
  'MULTIPOLYGON',
  8,
  'SOLID',
  9,
  'MULTISOLID',
  'UNKNOWN_GEOMETRY'
)

-- SQLSERVER
geometry.STGeometryType()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.10. ST_GeomFromText

This function parses a WKT (Well-known text representation of geometry). For example:

```
create.select(stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))")).fetch();
```

The result being, for example

```
+-------------------------------------------+
| ST_GeomFromText                           |
+-------------------------------------------+
| POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1)) |
+-------------------------------------------+
```

... which is a simple square:

Dialect support

This example using jOOQ:

```
stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))")
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_geomfromtext('POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))', 0)

-- ORACLE
sdo_geometry(wkt => 'POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))', srid => NULL)

-- SQLSERVER
geometry::STGeomFromText('POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))', 0)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.11. ST_InteriorRingN

This function returns a linestring corresponding to the Nth interior ring (1-based) of a polygon geometry. For example:

```
create.select(stInteriorRingN(
  stGeomFromText("""
    POLYGON (
      (-3 -3, 3 -3, 3 3, -3 3, -3 -3),
      (-2 -2, 2 -2, 2 2, -2 2, -2 -2),
      (-1 -1, 1 -1, 1 1, -1 1, -1 -1)
    )
  """)
), 1).fetch();
```

The result being, for example

```
+--------------------------------------------+
| ST_InteriorRingN                           |
+--------------------------------------------+
| LINESTRING (-2 -2, 2 -2, 2 2, -2 2, -2 -2) |
+--------------------------------------------+
```

Or, visually:

To get the total number of interior rings, use ST_NumInteriorRings.

Dialect support

This example using jOOQ:

```
stInteriorRingN(geometry, 1)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_interiorringn(geometry, 1)

-- ORACLE
sdo_util.extract(geometry, 1, 1 + 1)

-- SQLSERVER
geometry.STInteriorRingN(1)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.12. ST_Intersection

This function intersects two geometries. For example:

```
create.select(stIntersection(
  stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"),
  stGeomFromText("POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))"),
)).fetch();
```

The result being, for example

```
+------------------------------------+
| ST_Difference                      |
+------------------------------------+
| POLYGON ((1 1, 1 0, 0 0, 0 1, 1 1)) |
+------------------------------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stIntersection(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_intersection(geometry1, geometry2)

-- ORACLE
sdo_geom.sdo_intersection(geometry1, geometry2, tol => null)

-- SQLSERVER
geometry1.STIntersection(geometry2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.13. ST_Length

This function returns the length of a geometry. For example:

```
create.select(stLength(stGeomFromText("LINESTRING (0 0, 0 1, 1 1, 1 2)"))).fetch();
```

The result being, for example

```
+--------------------------------+
| ST_Length                      |
+--------------------------------+
| 3                              |
+--------------------------------+
```

Dialect support

This example using jOOQ:

```
stLength(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_length(geometry)

-- ORACLE
sdo_geom.sdo_length(geometry, tol => null)

-- SQLSERVER
geometry.STLength()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.14. ST_NumGeometries

This function returns the number of interior rings of a polygon geometry. For example:

```
create.select(stNumGeometries(stGeomFromText("MULTIPOINT ((-1 -1), (1 -1), (1 1), (-1 1))"))).fetch();
```

The result being, for example

```
+-----------------+
| ST_NumGeometries |
+-----------------+
| 4               |
+-----------------+
```

To access the Nth geometry (1-based), use ST_GeometryN.

Dialect support

This example using jOOQ:

```
stNumGeometries(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_numgeometries(geometry)

-- ORACLE
sdo_util.getnumelem(geometry)

-- SQLSERVER
geometry.STNumGeometries()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.15. ST_NumInteriorRings

This function returns the number of interior rings of a polygon geometry. For example:

```
create.select(stNumInteriorRings(
  stGeomFromText("""
    POLYGON (
      (-3 -3, 3 -3, 3 3, -3 3, -3 -3),
      (-2 -2, 2 -2, 2 2, -2 2, -2 -2),
      (-1 -1, 1 -1, 1 1, -1 1, -1 -1)
    )
  """)
)).fetch();
```

The result being, for example

```
+---------------------+
| ST_NumInteriorRings |
+---------------------+
| 2                   |
+---------------------+
```

To access the Nth interior ring (1-based), use ST_InteriorRingN.

Dialect support

This example using jOOQ:

```
stNumInteriorRings(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_numinteriorring(geometry)

-- MARIADB
st_numinteriorrings(geometry)

-- ORACLE
(
  SELECT (nullif(
    count(*),
    0
  ) - 1)
  FROM table(sdo_util.extract_all(geometry))
)

-- SQLSERVER
geometry.STNumInteriorRing()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.16. ST_NumPoints

This function returns the number of points on a linestring geometry. For example:

```
create.select(stNumPoints(stGeomFromText("LINESTRING (0 0, 0 1, 1 1, 1 2)"))).fetch();
```

The result being, for example

```
+--------------+
| ST_NumPoints |
+--------------+
| 4            |
+--------------+
```

To access the Nth point (1-based), use ST_PointN.

Dialect support

This example using jOOQ:

```
stNumPoints(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_numpoints(geometry)

-- SQLSERVER
geometry.STNumPoints()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.17. ST_PointN

This function returns the Nth point (1-based) of a linestring geometry. For example:

```
create.select(stPointN(stGeomFromText("LINESTRING (0 0, 0 1, 1 1, 1 2)"), 2)).fetch();
```

The result being, for example

```
| ST_PointN   |
+-------------+
| POINT (0 1) |
+-------------+
```

Or, visually:

To get the total number of points, use ST_NumPoints.

Dialect support

This example using jOOQ:

```
stPointN(geometry, 2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_pointn(geometry, 2)

-- SQLSERVER
geometry.STPointN(2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.18. ST_SRID

This function returns the SRID of a geometry. For example:

```
create.select(stSrid(stGeomFromText("POINT (1 0)", 4326))).fetch();
```

The result being, for example

```
+---------+
| ST_SRID |
+---------+
| 4326    |
+---------+
```

Dialect support

This example using jOOQ:

```
stSrid(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_srid(geometry)

-- ORACLE
(geometry).sdo_srid

-- SQLSERVER
geometry.STSrid

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.19. ST_StartPoint

This function returns the first point of a linestring. For example:

```
create.select(stStartPoint(stGeomFromText("LINESTRING (0 0, 1 1, 2 0, 3 1)"))).fetch();
```

The result being, for example

```
+---------------+
| ST_StartPoint |
+---------------+
| POINT (0 0)   |
+---------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stStartPoint(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_startpoint(geometry)

-- SQLSERVER
geometry.STStartPoint()

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.20. ST_Union

This function combines a geometry with another. For example:

```
create.select(stUnion(
  stGeomFromText("POLYGON ((-1 -1, 1 -1, 1 1, -1 1, -1 -1))"),
  stGeomFromText("POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))"),
)).fetch();
```

The result being, for example

```
+-------------------------------------------------------------+
| ST_Union                                                    |
+-------------------------------------------------------------+
| POLYGON ((1 -1, -1 -1, -1 1, 0 1, 0 2, 2 2, 2 0, 1 0, 1 -1)) |
+-------------------------------------------------------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stUnion(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_union(geometry1, geometry2)

-- COCKROACHDB
(
  SELECT st_union(v)
  FROM (
    VALUES
      (geometry1),
      (geometry2)
  ) t (v)
)

-- ORACLE
sdo_geom.sdo_union(geometry1, geometry2, tol => null)

-- SQLSERVER
geometry1.STUnion(geometry2)

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.21. ST_X

This function extracts the X coordinate of a point. For example:

```
create.select(stX(stGeomFromText("POINT (1 0)"))).fetch();
```

The result being, for example

```
+------+
| ST_X |
+------+
| 1    |
+------+
```

Dialect support

This example using jOOQ:

```
stX(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_x(geometry)

-- ORACLE
(geometry).sdo_point.x

-- SQLSERVER
geometry.STX

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.22. ST_Y

This function extracts the Y coordinate of a point. For example:

```
create.select(stY(stGeomFromText("POINT (0 1)"))).fetch();
```

The result being, for example

```
+------+
| ST_Y |
+------+
| 1    |
+------+
```

Dialect support

This example using jOOQ:

```
stY(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_y(geometry)

-- ORACLE
(geometry).sdo_point.y

-- SQLSERVER
geometry.STY

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.20.23. ST_Z

This function extracts the Z coordinate of a point. For example:

```
create.select(stZ(stGeomFromText("POINT (0 0 1)"))).fetch();
```

The result being, for example

```
+------+
| ST_Z |
+------+
| 1    |
+------+
```

Dialect support

This example using jOOQ:

```
stZ(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, REDSHIFT, SNOWFLAKE
st_z(geometry)

-- ORACLE
(geometry).sdo_point.z

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB, MEMSQL, MYSQL,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.21. 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.

4.10.21.1. 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.

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 |
+------+
```

4.10.21.2. 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.

4.10.21.3. 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, FIREBIRD, H2, HSQLDB, POSTGRES, SQLITE, YUGABYTEDB
count(*) FILTER (WHERE BOOK.TITLE LIKE 'A%')

-- BIGQUERY
countif((BOOK.TITLE LIKE 'A%'))

-- SNOWFLAKE
count_if((BOOK.TITLE LIKE 'A%'))
```

4.10.21.4. 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).
- JSON_ARRAYAGG, which aggregates data into a JSON array.
- 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.
- XMLAGG, which aggregates data into an XML element.

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.

4.10.21.5. 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 |
+-----------------+
```

4.10.21.6. 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)
```

4.10.21.7. ANY_VALUE

The ANY_VALUE() aggregate function produces any random value from the group, non-deterministically

```
SELECT any_value(ID)
FROM BOOK
```

```
create.select(anyValue(BOOK.ID))
      .from(BOOK)
```

Producing (for example):

```
+-----------+
| any_value |
+-----------+
|         3 |
+-----------+
```

Dialect support

This example using jOOQ:

```
anyValue(BOOK.ID)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_POSTGRES, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB, POSTGRES,
-- SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
min(BOOK.ID)

-- AURORA_MYSQL, BIGQUERY, MEMSQL, MYSQL, ORACLE, REDSHIFT, SNOWFLAKE
any_value(BOOK.ID)
```

4.10.21.8. 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] |
+--------------+--------------+
```

Unlike the MULTISET_AGG function, this:

- Produces an array, instead of a org.jooq.Result type
- Allows for projecting only a single column (though that column may contain a nested record)

Dialect support

This example using jOOQ:

```
arrayAgg(BOOK.ID)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, BIGQUERY, COCKROACHDB, H2, HSQLDB, POSTGRES, YUGABYTEDB
array_agg(BOOK.ID)

-- ACCESS, ASE, AURORA_MYSQL, DB2, DERBY, EXASOL, FIREBIRD, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE, REDSHIFT,
-- SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.9. 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)
```

4.10.21.10. BIT_AND_AGG

An aggregate function to perform the equivalent of the BIT_AND function on a data set. In other words, the resulting bits are:

- 1 at position p if the argument is 1 at position p for every row in the group.
- 0 at position p if the argument is 0 at position p for at least one row in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_and_agg(ID),
  bit_and_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitAndAgg(BOOK.ID),
         bitAndAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+-------------+-------------+
| bit_and_agg | bit_and_agg |
+-------------+-------------+
|           0 |           0 |
+-------------+-------------+
```

Dialect support

This example using jOOQ:

```
bitAndAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, MEMSQL, SQLDATAWAREHOUSE, SQLITE, SQLSERVER
(CASE min(CASE (BOOK.ID & 1)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 2)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 4)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 8)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 16)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 32)
  WHEN 0 THEN 0
  WHEN 32 THEN 32
END)
  WHEN 32 THEN 32
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 64)
  WHEN 0 THEN 0
  WHEN 64 THEN 64
END)
  WHEN 64 THEN 64
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & -128)
  WHEN 0 THEN 0
  WHEN -128 THEN -128
END)
  WHEN -128 THEN -128
  WHEN 0 THEN 0
END)

-- AURORA_MYSQL, AURORA_POSTGRES, H2, ORACLE, SNOWFLAKE
bit_and_agg(BOOK.ID)

-- BIGQUERY, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SYBASE, YUGABYTEDB
bit_and(BOOK.ID)

-- DB2, HANA, HSQLDB, INFORMIX, TERADATA
(CASE min(CASE bitand(
  BOOK.ID,
  1
)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  2
)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  4
)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  8
)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  16
)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  32
)
  WHEN 0 THEN 0
```

4.10.21.11. BIT_NAND_AGG

An aggregate function to perform the equivalent of the BIT_NAND function on a data set. In other words, the resulting bits are:

- 0 at position p if the argument is 1 at position p for every row in the group.
- 1 at position p if the argument is 0 at position p for at least one row in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_nand_agg(ID),
  bit_nand_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitNandAgg(BOOK.ID),
         bitNandAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+--------------+--------------+
| bit_nand_agg | bit_nand_agg |
+--------------+--------------+
|           -1 |           -1 |
+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
bitNandAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, MEMSQL, SQLDATAWAREHOUSE, SQLITE, SQLSERVER
~((CASE min(CASE (BOOK.ID & 1)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 2)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 4)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 8)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 16)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 32)
  WHEN 0 THEN 0
  WHEN 32 THEN 32
END)
  WHEN 32 THEN 32
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & 64)
  WHEN 0 THEN 0
  WHEN 64 THEN 64
END)
  WHEN 64 THEN 64
  WHEN 0 THEN 0
END + CASE min(CASE (BOOK.ID & -128)
  WHEN 0 THEN 0
  WHEN -128 THEN -128
END)
  WHEN -128 THEN -128
  WHEN 0 THEN 0
END))

-- AURORA_MYSQL, AURORA_POSTGRES
~(bit_and_agg(BOOK.ID))

-- BIGQUERY, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SYBASE, YUGABYTEDB
~(bit_and(BOOK.ID))

-- DB2, HANA, INFORMIX, TERADATA
bitnot((CASE min(CASE bitand(
  BOOK.ID,
  1
)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  2
)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  4
)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  8
)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  16
)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE min(CASE bitand(
  BOOK.ID,
  32
)
  WHEN 0 THEN 0
```

4.10.21.12. BIT_NOR_AGG

An aggregate function to perform the equivalent of the BIT_NOR function on a data set. In other words, the resulting bits are:

- 0 at position p if the argument is 1 at position p for at least one row in the group.
- 1 at position p if the argument is 0 at position p for every row in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_nor_agg(ID),
  bit_nor_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitNorAgg(BOOK.ID),
         bitNorAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+-------------+--------------+
| bit_nor_agg | bit_nor_agg  |
+-------------+--------------+
|          -8 |           -4 |
+-------------+--------------+
```

Dialect support

This example using jOOQ:

```
bitNorAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, MEMSQL, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER
~((CASE max(CASE (BOOK.ID & 1)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 2)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 4)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 8)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 16)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 32)
  WHEN 0 THEN 0
  WHEN 32 THEN 32
END)
  WHEN 32 THEN 32
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 64)
  WHEN 0 THEN 0
  WHEN 64 THEN 64
END)
  WHEN 64 THEN 64
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & -128)
  WHEN 0 THEN 0
  WHEN -128 THEN -128
END)
  WHEN -128 THEN -128
  WHEN 0 THEN 0
END))

-- AURORA_MYSQL, AURORA_POSTGRES
~(bit_or_agg(BOOK.ID))

-- BIGQUERY, COCKROACHDB, MARIADB, MYSQL, POSTGRES, SYBASE, YUGABYTEDB
~(bit_or(BOOK.ID))

-- DB2, HANA, INFORMIX, TERADATA
bitnot((CASE max(CASE bitand(
  BOOK.ID,
  1
)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  2
)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  4
)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  8
)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  16
)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  32
)
  WHEN 0 THEN 0
```

4.10.21.13. BIT_OR_AGG

An aggregate function to perform the equivalent of the BIT_OR function on a data set. In other words, the resulting bits are:

- 1 at position p if the argument is 1 at position p for at least one row in the group.
- 0 at position p if the argument is 0 at position p for every row in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_or_agg(ID),
  bit_or_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitOrAgg(BOOK.ID),
         bitOrAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+------------+------------+
| bit_or_agg | bit_or_agg |
+------------+------------+
|          7 |          3 |
+------------+------------+
```

Dialect support

This example using jOOQ:

```
bitOrAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, MEMSQL, REDSHIFT, SQLDATAWAREHOUSE, SQLITE, SQLSERVER
(CASE max(CASE (BOOK.ID & 1)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 2)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 4)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 8)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 16)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 32)
  WHEN 0 THEN 0
  WHEN 32 THEN 32
END)
  WHEN 32 THEN 32
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & 64)
  WHEN 0 THEN 0
  WHEN 64 THEN 64
END)
  WHEN 64 THEN 64
  WHEN 0 THEN 0
END + CASE max(CASE (BOOK.ID & -128)
  WHEN 0 THEN 0
  WHEN -128 THEN -128
END)
  WHEN -128 THEN -128
  WHEN 0 THEN 0
END)

-- AURORA_MYSQL, AURORA_POSTGRES, H2, ORACLE, SNOWFLAKE
bit_or_agg(BOOK.ID)

-- BIGQUERY, COCKROACHDB, MARIADB, MYSQL, POSTGRES, SYBASE, YUGABYTEDB
bit_or(BOOK.ID)

-- DB2, HANA, HSQLDB, INFORMIX, TERADATA
(CASE max(CASE bitand(
  BOOK.ID,
  1
)
  WHEN 0 THEN 0
  WHEN 1 THEN 1
END)
  WHEN 1 THEN 1
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  2
)
  WHEN 0 THEN 0
  WHEN 2 THEN 2
END)
  WHEN 2 THEN 2
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  4
)
  WHEN 0 THEN 0
  WHEN 4 THEN 4
END)
  WHEN 4 THEN 4
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  8
)
  WHEN 0 THEN 0
  WHEN 8 THEN 8
END)
  WHEN 8 THEN 8
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  16
)
  WHEN 0 THEN 0
  WHEN 16 THEN 16
END)
  WHEN 16 THEN 16
  WHEN 0 THEN 0
END + CASE max(CASE bitand(
  BOOK.ID,
  32
)
  WHEN 0 THEN 0
```

4.10.21.14. BIT_XOR_AGG

An aggregate function to perform the equivalent of the BIT_XOR function on a data set. In other words, the resulting bits are:

- 1 at position p if the argument is 1 at position p for an odd number of rows in the group.
- 0 at position p if the argument is 0 at position p for an even number of rows in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_xor_agg(ID),
  bit_xor_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitXorAgg(BOOK.ID),
         bitXorAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+-------------+-------------+
| bit_xor_agg | bit_xor_agg |
+-------------+-------------+
|           4 |           0 |
+-------------+-------------+
```

Dialect support

This example using jOOQ:

```
bitXorAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, REDSHIFT, SQLDATAWAREHOUSE, SQLSERVER
(CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 1) = 1 THEN 1
  END) % 2) = 1 THEN 1
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 2) = 2 THEN 1
  END) % 2) = 1 THEN 2
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 4) = 4 THEN 1
  END) % 2) = 1 THEN 4
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 8) = 8 THEN 1
  END) % 2) = 1 THEN 8
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 16) = 16 THEN 1
  END) % 2) = 1 THEN 16
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 32) = 32 THEN 1
  END) % 2) = 1 THEN 32
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 64) = 64 THEN 1
  END) % 2) = 1 THEN 64
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & -128) = -128 THEN 1
  END) % 2) = 1 THEN -128
  ELSE 0
END)

-- AURORA_MYSQL, ORACLE, SNOWFLAKE
bit_xor_agg(BOOK.ID)

-- AURORA_POSTGRES, COCKROACHDB, YUGABYTEDB
(CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 1) = 1),
    2
  ) = 1 THEN 1
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 2) = 2),
    2
  ) = 1 THEN 2
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 4) = 4),
    2
  ) = 1 THEN 4
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 8) = 8),
    2
  ) = 1 THEN 8
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 16) = 16),
    2
  ) = 1 THEN 16
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 32) = 32),
    2
  ) = 1 THEN 32
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 64) = 64),
    2
  ) = 1 THEN 64
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & -128) = -128),
    2
  ) = 1 THEN -128
  ELSE 0
END)

-- BIGQUERY, MARIADB, MYSQL, POSTGRES, SYBASE
bit_xor(BOOK.ID)

-- DB2, HANA, INFORMIX
(CASE
  WHEN mod(
    count(CASE
      WHEN bitand(
        BOOK.ID,
        1
      ) = 1 THEN 1
```

4.10.21.15. BIT_XNOR_AGG

An aggregate function to perform the equivalent of the BIT_XNOR function on a data set. In other words, the resulting bits are:

- 0 at position p if the argument is 1 at position p for an odd number of rows in the group.
- 1 at position p if the argument is 0 at position p for an even number of rows in the group.

As with most aggregate functions, NULL values are not aggregated.

```
SELECT
  bit_xnor_agg(ID),
  bit_xnor_agg(AUTHOR_ID)
FROM BOOK
```

```
create.select(
         bitXNorAgg(BOOK.ID),
         bitXNorAgg(BOOK.AUTHOR_ID))
      .from(BOOK)
```

Producing:

```
+--------------+--------------+
| bit_xnor_agg | bit_xnor_agg |
+--------------+--------------+
|           -5 |           -1 |
+--------------+--------------+
```

Dialect support

This example using jOOQ:

```
bitXNorAgg(BOOK.ID.coerce(TINYINT))
```

Translates to the following dialect specific expressions:

```
-- ASE, REDSHIFT, SQLDATAWAREHOUSE, SQLSERVER
~((CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 1) = 1 THEN 1
  END) % 2) = 1 THEN 1
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 2) = 2 THEN 1
  END) % 2) = 1 THEN 2
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 4) = 4 THEN 1
  END) % 2) = 1 THEN 4
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 8) = 8 THEN 1
  END) % 2) = 1 THEN 8
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 16) = 16 THEN 1
  END) % 2) = 1 THEN 16
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 32) = 32 THEN 1
  END) % 2) = 1 THEN 32
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & 64) = 64 THEN 1
  END) % 2) = 1 THEN 64
  ELSE 0
END + CASE
  WHEN (count(CASE
    WHEN (BOOK.ID & -128) = -128 THEN 1
  END) % 2) = 1 THEN -128
  ELSE 0
END))

-- AURORA_MYSQL
~(bit_xor_agg(BOOK.ID))

-- AURORA_POSTGRES, COCKROACHDB, YUGABYTEDB
~((CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 1) = 1),
    2
  ) = 1 THEN 1
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 2) = 2),
    2
  ) = 1 THEN 2
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 4) = 4),
    2
  ) = 1 THEN 4
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 8) = 8),
    2
  ) = 1 THEN 8
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 16) = 16),
    2
  ) = 1 THEN 16
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 32) = 32),
    2
  ) = 1 THEN 32
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & 64) = 64),
    2
  ) = 1 THEN 64
  ELSE 0
END + CASE
  WHEN mod(
    count(*) FILTER (WHERE (BOOK.ID & -128) = -128),
    2
  ) = 1 THEN -128
  ELSE 0
END))

-- BIGQUERY, MARIADB, MYSQL, POSTGRES, SYBASE
~(bit_xor(BOOK.ID))

-- DB2, HANA, INFORMIX
bitnot((CASE
  WHEN mod(
    count(CASE
      WHEN bitand(
        BOOK.ID,
        1
      ) = 1 THEN 1
```

4.10.21.16. 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, ORACLE, 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, DERBY, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, REDSHIFT, SQLITE
(min(CASE
  WHEN BOOK.ID < 4 THEN 1
  ELSE 0
END) = 1)

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, 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))
```

4.10.21.17. 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, ORACLE, 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, DERBY, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, REDSHIFT, SQLITE
(max(CASE
  WHEN BOOK.ID >= 4 THEN 1
  ELSE 0
END) = 1)

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, 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))
```

4.10.21.18. 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] |
+--------------+
```

4.10.21.19. 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)
```

4.10.21.20. 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)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.21. 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)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.22. 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, ORACLE, 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, DERBY, H2, HSQLDB, MARIADB, MEMSQL, MYSQL, REDSHIFT, SQLITE
(min(CASE
  WHEN BOOK.ID < 4 THEN 1
  ELSE 0
END) = 1)

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, 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))
```

4.10.21.23. 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, 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, ',')

-- 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)

-- ACCESS, ASE, DERBY, FIREBIRD, INFORMIX, SNOWFLAKE, SQLDATAWAREHOUSE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.21.24. JSON_ARRAYAGG

A data set can be aggregated into a org.jooq.JSON or org.jooq.JSONB array using JSON_ARRAYAGG

```
SELECT json_arrayagg(author.id)
FROM author
```

```
create.select(jsonArrayAgg(AUTHOR.ID))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+---------------+
| json_arrayagg |
+---------------+
| [1,2]         |
+---------------+
```

Ordering aggregation contents

When aggregating data into an array or JSON array, ordering may be relevant. For this, use the ORDER BY clause in JSON_ARRAYAGG

```
SELECT json_arrayagg(author.id ORDER BY author.id DESC)
FROM author
```

```
create.select(jsonArrayAgg(AUTHOR.ID).orderBy(AUTHOR.ID.desc())
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+---------------+
| json_arrayagg |
+---------------+
| [2,1]         |
+---------------+
```

NULL handling

Some dialects support the SQL standard NULL ON NULL and ABSENT ON NULL syntax, which allows for including / excluding NULL values from aggregation. By default, SQL aggregate functions always exclude NULL values, but in the context of JSON data types, NULL may have a different significance:

```
SELECT
  json_arrayagg(nullif(author.id, 1) NULL   ON NULL) AS c1,
  json_arrayagg(nullif(author.id, 1) ABSENT ON NULL) AS c2
FROM author
```

```
create.select(
        jsonArrayAgg(nullif(AUTHOR.ID, 1)).nullOnNull()
  .as("c1"),
        jsonArrayAgg(nullif(AUTHOR.ID,
 1)).absentOnNull().as("c2"))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+----------+-----+
| C1       | C2  |
+----------+-----+
| [null,2] | [2] |
+----------+-----+
```

The effect is similar to that of the FILTER clause.

Dialect support

This example using jOOQ:

```
jsonArrayAgg(AUTHOR.ID)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, YUGABYTEDB
json_agg(AUTHOR.ID)

-- DB2
CAST((('[' || listagg(
  AUTHOR.ID,
  ','
)) || ']') AS varchar(32672))

-- H2, ORACLE
json_arrayagg(AUTHOR.ID)

-- MARIADB, MYSQL
json_merge_preserve(
  '[]',
  concat(
    '[',
    group_concat(AUTHOR.ID SEPARATOR ','),
    ']'
  )
)

-- SNOWFLAKE
array_agg(coalesce(
  to_variant(AUTHOR.ID),
  parse_json('null')
))

-- SQLITE
json_group_array(AUTHOR.ID)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.25. JSON_OBJECTAGG

A data set can be aggregated into a org.jooq.JSON or org.jooq.JSONB object using JSON_OBJECTAGG

```
SELECT json_objectagg(
  CAST(author.id AS varchar(100)),
  first_name
)
FROM author
```

```
create.select(jsonObjectAgg(
          cast(AUTHOR.ID, VARCHAR(100)),
          AUTHOR.FIRST_NAME
       ))
       .from(AUTHOR)
       .fetch();
```

The result would look like this:

```
+----------------------------+
| json_objectagg             |
+----------------------------+
| {"1":"George","2":"Paulo"} |
+----------------------------+
```

NULL handling

Some dialects support the SQL standard NULL ON NULL and ABSENT ON NULL syntax, which allows for including / excluding NULL values from aggregation. By default, SQL aggregate functions always exclude NULL values, but in the context of JSON data types, NULL may have a different significance:

```
SELECT
  json_objectagg(
    CAST(author.id AS varchar(100)),
    nullif(first_name, 'George')
    NULL ON NULL
  ) AS c1,
  json_objectagg(
    CAST(author.id AS varchar(100)),
    nullif(first_name, 'George')
    ABSENT ON NULL
  ) AS c2
FROM author
```

```
create.select(
         jsonObjectAgg(
           cast(AUTHOR.ID, VARCHAR(100)),
           nullif(AUTHOR.FIRST_NAME, "George")
         ).nullOnNull().as("c1"),
         jsonObjectAgg(
           cast(AUTHOR.ID, VARCHAR(100)),
           nullif(AUTHOR.FIRST_NAME, "George")
         ).absentOnNull().as("c2")
       )
       .from(AUTHOR)
       .fetch();
```

The result would look like this:

```
+------------------------+---------------+
| C1                     | C2            |
+------------------------+---------------+
| {"1":null,"2":"Paulo"} | {"2":"Paulo"} |
+------------------------+---------------+
```

The effect is similar to that of the FILTER clause.

Dialect support

This example using jOOQ:

```
jsonObjectAgg(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, POSTGRES, YUGABYTEDB
json_object_agg(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)

-- COCKROACHDB
((('{' || string_agg(regexp_replace(CAST(json_build_object(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME) AS string), '^\{(.*)\}$', '\1', 'g'),
 ',')) || '}'))

-- DB2
((('{' || listagg(
  regexp_replace(CAST(json_object(KEY AUTHOR.FIRST_NAME VALUE AUTHOR.LAST_NAME) AS varchar(32672)), '^\{(.*)\}$', '\1'),
  ','
)) || '}'))

-- H2, ORACLE
json_objectagg(KEY AUTHOR.FIRST_NAME VALUE AUTHOR.LAST_NAME)

-- MARIADB, MYSQL
json_objectagg(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)

-- SNOWFLAKE
object_agg(coalesce(
  to_variant(AUTHOR.FIRST_NAME),
  parse_json('null')
), coalesce(
  to_variant(AUTHOR.LAST_NAME),
  parse_json('null')
))

-- SQLITE
json_group_object(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.26. 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, 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)

-- TERADATA
substring(xmlserialize(CONTENT xmlagg((',' || CAST(BOOK.AUTHOR_ID AS varchar(32000))) ORDER BY BOOK.ID) AS varchar(32000)) FROM 2)

-- ACCESS, ASE, DERBY, FIREBIRD, INFORMIX, MEMSQL, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.21.27. 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)
```

4.10.21.28. 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, BIGQUERY, POSTGRES, TERADATA, YUGABYTEDB
percentile_cont(0.5) WITHIN GROUP (ORDER BY BOOK.ID)

-- DB2, EXASOL, H2, HANA, HSQLDB, MARIADB, ORACLE, REDSHIFT, SNOWFLAKE, SYBASE
median(BOOK.ID)

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, FIREBIRD, INFORMIX, MEMSQL, MYSQL, SQLDATAWAREHOUSE, SQLITE, SQLSERVER,
-- VERTICA
/* UNSUPPORTED */
```

4.10.21.29. 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)
```

4.10.21.30. 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, H2, POSTGRES, YUGABYTEDB
mode() WITHIN GROUP (ORDER BY BOOK.AUTHOR_ID)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.31. MULTISET_AGG

The synthetic MULTISET_AGG() aggregate function collects group contents into a nested collection, just like the MULTISET value constructor (learn about other synthetic sql syntaxes).

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  MULTISET_AGG(
    BOOK.ID,
    BOOK.TITLE,
    LANGUAGE.CD
  )
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
JOIN LANGUAGE ON BOOK.LANGUAGE_ID = LANGUAGE.ID
GROUP BY
  AUTHOR.ID,
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
ORDER BY AUTHOR.ID
```

```
create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,
          multisetAgg(
              BOOK.ID,
              BOOK.TITLE,
              BOOK.language().CD
          ).as("books"))
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .groupBy(
          AUTHOR.ID,
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME)
      .orderBy(AUTHOR.ID)
      .fetch()
```

The result being:

```
+----------+---------+--------------------------------------+
|first_name|last_name|books                                 |
+----------+---------+--------------------------------------+
|George    |Orwell   |[(2, Animal Farm, en), (1, 1984, en)]  |
|Paulo     |Coelho   |[(4, Brida, de), (3, O Alquimista, pt)]|
+----------+---------+--------------------------------------+
```

Unlike the ARRAY_AGG function, this:

- Produces a more convenient org.jooq.Result type, instead of an array
- Allows for projecting multiple columns in a type safe way, instead of just a single column

Dialect support

This example using jOOQ:

```
multisetAgg(BOOK.ID, BOOK.TITLE)
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, YUGABYTEDB
jsonb_agg(jsonb_build_array(BOOK.ID, BOOK.TITLE))

-- DB2
xmlelement(
  NAME result,
  xmlagg(xmlelement(
    NAME record,
    xmlelement(NAME v0, BOOK.ID),
    xmlelement(
      NAME v1,
      xmlattributes(
        CASE
          WHEN BOOK.TITLE IS NULL THEN 'true'
        END AS xsi:nil
      ),
      BOOK.TITLE
    )
  ))
)

-- H2
json_arrayagg(json_array(BOOK.ID, BOOK.TITLE NULL ON NULL))

-- MARIADB, MYSQL
json_merge_preserve(
  '[]',
  concat(
    '[',
    group_concat(json_array(BOOK.ID, BOOK.TITLE) SEPARATOR ','),
    ']'
  )
)

-- ORACLE
json_arrayagg(json_array(BOOK.ID, BOOK.TITLE NULL ON NULL RETURNING clob) FORMAT JSON RETURNING clob)

-- SNOWFLAKE
array_agg(array_construct(coalesce(
  to_variant(BOOK.ID),
  parse_json('null')
), coalesce(
  to_variant(BOOK.TITLE),
  parse_json('null')
)))

-- SQLITE
json_group_array(json_array(BOOK.ID, BOOK.TITLE))

-- TERADATA
xmlelement(
  NAME "result",
  xmlagg(xmlelement(
    NAME record,
    xmlelement(NAME v0, BOOK.ID),
    xmlelement(
      NAME v1,
      xmlattributes(
        CASE
          WHEN BOOK.TITLE IS NULL THEN 'true'
        END AS nil
      ),
      BOOK.TITLE
    )
  ))
)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT,
-- SQLDATAWAREHOUSE, SQLSERVER, SYBASE, VERTICA
/* UNSUPPORTED */
```

4.10.21.32. 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)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.33. 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, BIGQUERY, DB2, EXASOL, MARIADB, MEMSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE,
-- SQLSERVER, TERADATA, YUGABYTEDB
percentile_cont(0E0) WITHIN GROUP (ORDER BY BOOK.ID)

-- H2
percentile_cont(CAST(0E0 AS double)) WITHIN GROUP (ORDER BY BOOK.ID)

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, FIREBIRD, HANA, HSQLDB, INFORMIX, MYSQL, SQLITE, SYBASE, VERTICA
/* UNSUPPORTED */
```

4.10.21.34. 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, BIGQUERY, DB2, EXASOL, MARIADB, MEMSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE,
-- SQLSERVER, TERADATA, YUGABYTEDB
percentile_disc(0E0) WITHIN GROUP (ORDER BY BOOK.ID)

-- H2
percentile_disc(CAST(0E0 AS double)) WITHIN GROUP (ORDER BY BOOK.ID)

-- ACCESS, ASE, AURORA_MYSQL, COCKROACHDB, DERBY, FIREBIRD, HANA, HSQLDB, INFORMIX, MYSQL, SQLITE, SYBASE, VERTICA
/* UNSUPPORTED */
```

4.10.21.35. PRODUCT

The PRODUCT() aggregate function is a synthetic aggregate function that calculates the product of all values in the group, similar to how the SUM function calculates the sum (learn about other synthetic sql syntaxes).

```
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, SQLDATAWAREHOUSE, SQLSERVER
(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, DB2, FIREBIRD, H2, HANA, HSQLDB, MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES,
-- SNOWFLAKE, SYBASE, 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) > 0 THEN 0
  WHEN mod(
    sum(CASE
      WHEN BOOK.ID < 0 THEN -1
    END),
    2
  ) < 0 THEN -1
  ELSE 1
END * exp(sum(ln(CAST(abs(nullif(BOOK.ID, 0)) AS numeric)))))

-- 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))))))

-- 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))))))

-- 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))))))

-- 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 */
```

4.10.21.36. 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)

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, VERTICA
/* UNSUPPORTED */
```

4.10.21.37. 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)
```

4.10.21.38. XMLAGG

A data set can be aggregated into a org.jooq.XML element using XMLAGG

```
SELECT xmlelement(
  NAME ids,
  xmlagg(xmlelement(NAME id, id))
)
FROM author
```

```
create.select(xmlelement("ids",
        xmlagg(xmlelement("id", AUTHOR.ID))
      ))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+---------------------------------+
| xmlelement                      |
+---------------------------------+
| <ids><id>1</id><id>2</id></ids> |
+---------------------------------+
```

Ordering aggregation contents

When aggregating data into XML, ordering may be relevant. For this, use the ORDER BY clause in XMLAGG

```
SELECT xmlelement(
  NAME ids,
  xmlagg(xmlelement(NAME id, id) ORDER BY id DESC)
)
FROM author
```

```
create.select(xmlelement("ids",
        xmlagg(xmlelement("id", AUTHOR.ID))
          .orderBy(AUTHOR.ID.desc())))
      .from(AUTHOR)
      .fetch();
```

The result would look like this:

```
+---------------------------------+
| xmlelement                      |
+---------------------------------+
| <ids><id>2</id><id>1</id></ids> |
+---------------------------------+
```

Dialect support

This example using jOOQ:

```
xmlagg(xmlelement("id", AUTHOR.ID))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, POSTGRES, TERADATA
xmlagg(xmlelement(NAME id, AUTHOR.ID))

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.10.22. Window functions

Most major RDBMS support the concept of window functions.

As previously discussed, any org.jooq.AggregateFunction can be transformed into a window function using the over() method. See the chapter about aggregate functions for details. In addition to those, there are also some more window functions supported by jOOQ, as declared in the DSL:

```
// Ranking functions
    WindowOverStep<Integer>    rowNumber();
    WindowOverStep<Integer>    rank();
    WindowOverStep<Integer>    denseRank();
    WindowOverStep<BigDecimal> percentRank();

// Windowing functions
<T> WindowIgnoreNullsStep<T>   firstValue(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lastValue(Field<T> field);
<T> WindowIgnoreNullsStep<T>   nthValue(Field<T> field, int nth);
<T> WindowIgnoreNullsStep<T>   nthValue(Field<T> field, Field<Integer> nth);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset, T defaultValue);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset, Field<T> defaultValue);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset, T defaultValue);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset, Field<T> defaultValue);

// Statistical functions
    WindowOverStep<BigDecimal> cumeDist();
    WindowOverStep<Integer>    ntile(int number);

// Inverse distribution functions
    OrderedAggregateFunction<BigDecimal> percentileCont(Number number);
    OrderedAggregateFunction<BigDecimal> percentileCont(Field<? extends Number> number);
    OrderedAggregateFunction<BigDecimal> percentileDisc(Number number);
    OrderedAggregateFunction<BigDecimal> percentileDisc(Field<? extends Number> number);
```

SQL distinguishes between various window function types (e.g. "ranking functions"). Depending on the function, SQL expects mandatory PARTITION BY or ORDER BY clauses within the OVER() clause. jOOQ does not enforce those rules for two reasons:

- Your JDBC driver or database already checks SQL syntax semantics
- Not all databases behave correctly according to the SQL standard

If possible, however, jOOQ tries to render missing clauses for you, if a given SQL dialect is more restrictive.

Some examples

Here are some simple examples of window functions with jOOQ:

```sql
SELECT

  -- Sample uses of ROW_NUMBER()
  ROW_NUMBER() OVER (),
  ROW_NUMBER() OVER (ORDER BY BOOK.ID),
  ROW_NUMBER() OVER (PARTITION BY BOOK.AUTHOR_ID ORDER BY
 BOOK.ID),

  -- Sample uses of FIRST_VALUE
  FIRST_VALUE(BOOK.ID) OVER(),
  FIRST_VALUE(BOOK.ID IGNORE NULLS) OVER(),
  FIRST_VALUE(BOOK.ID RESPECT NULLS) OVER()
FROM BOOK
```

```java
create.select(

  // Sample uses of rowNumber()
  rowNumber().over(),
  rowNumber().over().orderBy(BOOK.ID),

 rowNumber().over().partitionBy(BOOK.AUTHOR_ID).orderBy(BOOK.ID),

  // Sample uses of firstValue()
  firstValue(BOOK.ID).over(),
  firstValue(BOOK.ID).ignoreNulls().over(),
  firstValue(BOOK.ID).respectNulls().over())
.from(BOOK).fetch();
```

An advanced window function example

Window functions can be used for things like calculating a "running total". The following example fetches transactions and the running total for every transaction going back to the beginning of the transaction table (ordered by booked_at). Window functions are accessible from the previously seen org.jooq.AggregateFunction type using the over() method:

```sql
SELECT booked_at, amount,
   SUM(amount) OVER (PARTITION BY 1
                     ORDER BY booked_at
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS total
  FROM transactions
```

```java
create.select(t.BOOKED_AT, t.AMOUNT,
         sum(t.AMOUNT).over().partitionByOne()
                      .orderBy(t.BOOKED_AT)
                      .rowsBetweenUnboundedPreceding()
                      .andCurrentRow().as("total")
      .from(TRANSACTIONS.as("t"))
      .fetch();
```

Window functions created from ordered-set aggregate functions

In the previous chapter about aggregate functions, we have seen the concept of "ordered-set aggregate functions", such as Oracle's LISTAGG(). These functions have a window function / analytical function variant, as well. For example:

```sql
SELECT   LISTAGG(TITLE, ', ')
         WITHIN GROUP (ORDER BY TITLE)
         OVER (PARTITION BY BOOK.AUTHOR_ID)
FROM     BOOK
```

```java
create.select(listAgg(BOOK.TITLE, ", ")
      .withinGroupOrderBy(BOOK.TITLE)
      .over().partitionBy(BOOK.AUTHOR_ID))
      .from(BOOK)
      .fetch();
```

Window functions created from Oracle's FIRST and LAST aggregate functions

In the previous chapter about aggregate functions, we have seen the concept of "FIRST and LAST aggregate functions". These functions have a window function / analytical function variant, as well. For example:

```sql
SUM(BOOK.AMOUNT_SOLD)
  KEEP(DENSE_RANK FIRST ORDER BY BOOK.AUTHOR_ID)
  OVER(PARTITION BY 1)
```

```java
sum(BOOK.AMOUNT_SOLD)
  .keepDenseRankFirstOrderBy(BOOK.AUTHOR_ID)
  .over().partitionByOne();
```

Window functions created from user-defined aggregate functions

User-defined aggregate functions also implement org.jooq.AggregateFunction, hence they can also be transformed into window functions using over(). This is supported by Oracle in particular. See the manual's section about user-defined aggregate functions for more details.

4.10.23. Grouping functions

ROLLUP() explained in SQL

The SQL standard defines special functions that can be used in the GROUP BY clause: the grouping functions. These functions can be used to generate several groupings in a single clause. This can best be explained in SQL. Let's take ROLLUP() for instance:

```
-- ROLLUP() with one argument
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY ROLLUP(AUTHOR_ID)


-- ROLLUP() with two arguments
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY ROLLUP(AUTHOR_ID, PUBLISHED_IN)
```

```
-- The same query using UNION ALL:
  SELECT AUTHOR_ID, COUNT(*) FROM BOOK GROUP BY AUTHOR_ID
UNION ALL
  SELECT NULL, COUNT(*) FROM BOOK GROUP BY ()
ORDER BY 1 NULLS LAST

-- The same query using UNION ALL:
  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 ()
ORDER BY 1 NULLS LAST, 2 NULLS LAST
```

In English, the ROLLUP() grouping function provides N+1 groupings, when N is the number of arguments to the ROLLUP() function. Each grouping has an additional group field from the ROLLUP() argument field list. The results of the second query might look something 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 ()
+-----------+--------------+----------+
```

CUBE() explained in SQL

CUBE() is different from ROLLUP() in the way that it doesn't just create N+1 groupings, it creates all 2^N possible combinations between all group fields in the CUBE() function argument list. Let's re-consider our second query from before:

```
-- CUBE() with two arguments
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY CUBE(AUTHOR_ID, PUBLISHED_IN)
```

```
-- The same query using UNION ALL:
  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, PUBLISHED_IN, COUNT(*)
  FROM BOOK GROUP BY PUBLISHED_IN
UNION ALL
  SELECT NULL, NULL, COUNT(*)
  FROM BOOK GROUP BY ()
ORDER BY 1 NULLS FIRST, 2 NULLS FIRST
```

The results would then hold:

```
+-----------+--------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+--------------+----------+
|      NULL |         NULL |        2 | <- 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)
+-----------+--------------+----------+
```

GROUPING SETS()

GROUPING SETS() are the generalised way to create multiple groupings. From our previous examples

- ROLLUP(AUTHOR_ID, PUBLISHED_IN) corresponds to GROUPING SETS((AUTHOR_ID, PUBLISHED_IN), (AUTHOR_ID), ())
- CUBE(AUTHOR_ID, PUBLISHED_IN) corresponds to GROUPING SETS((AUTHOR_ID, PUBLISHED_IN), (AUTHOR_ID), (PUBLISHED_IN), ())

This is nicely explained in the SQL Server manual pages about GROUPING SETS() and other grouping functions:
https://msdn.microsoft.com/en-us/library/bb510427(v=sql.105)

jOOQ's support for ROLLUP(), CUBE(), GROUPING SETS()

jOOQ fully supports all of these functions, as well as the utility functions GROUPING() and GROUPING_ID(), used for identifying the grouping set ID of a record. The DSL API thus includes:

```
// The various grouping function constructors
GroupField rollup(Field<?>... fields);
GroupField cube(Field<?>... fields);
GroupField groupingSets(Field<?>... fields);
GroupField groupingSets(Field<?>[]... fields);
GroupField groupingSets(Collection<? extends Field<?>>... fields);

// The utility functions generating IDs per GROUPING SET
Field<Integer> grouping(Field<?>);
Field<Integer> groupingId(Field<?>...);
```

MySQL's WITH ROLLUP syntax

MySQL historically did not know any grouping functions, but they support a WITH ROLLUP clause, that is equivalent to simple ROLLUP() grouping functions. jOOQ emulates ROLLUP() in MySQL, by rendering this WITH ROLLUP clause. The following two statements mean the same:

```
-- Statement 1: SQL standard
GROUP BY ROLLUP(A, B, C)

-- Statement 2: SQL standard
GROUP BY A, ROLLUP(B, C)
```

```
-- Statement 1: MySQL
GROUP BY A, B, C WITH ROLLUP

-- Statement 2: MySQL
-- This is not supported in MySQL
```

4.10.24. 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

4.10.25. 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();
```

4.10.26. 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.

4.10.27. 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

4.10.28. 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.

4.10.29. ARRAY value constructor

The ARRAY value constructor allows for collecting the results of a single-column, non scalar subquery into a single nested collection value with ARRAY data type semantics (ordinals are defined on elements).

For example, let's find:

- All authors.
- The languages in which that author has their books published.
- The book stores at which that author's books are available.

This can be done in a single query:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  ARRAY(
    SELECT DISTINCT LANGUAGE.CD
    FROM BOOK
    JOIN LANGUAGE ON BOOK.LANGUAGE_ID = LANGUAGE.ID
  ) AS BOOKS,
  ARRAY(
    SELECT DISTINCT BOOK_TO_BOOK_STORE.BOOK_STORE_NAME
    FROM BOOK_TO_BOOK_STORE
    JOIN BOOK ON BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) AS BOOK_STORES
FROM AUTHOR
ORDER BY AUTHOR.ID
```

```
var result = create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,
          array(
              selectDistinct(BOOK.language().CD)
              .from(BOOK)
              .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          ).as("books"),
          array(
              selectDistinct(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
              .from(BOOK_TO_BOOK_STORE)

 .where(BOOK_TO_BOOK_STORE.tBook().AUTHOR_ID.eq(AUTHOR.ID))
          ).as("book_stores"))
      .from(AUTHOR)
      .orderBy(AUTHOR.ID)
      .fetch();
```

The above var result is inferred to:

```
Result<Record4<String, String, String[], String[]>> result =
```

The result of the above query may look like this:

```
+----------+---------+--------+-------------------------------------------------+
|first_name|last_name|books   |book_stores                                      |
+----------+---------+--------+-------------------------------------------------+
|George    |Orwell   |[en]    |[Ex Libris, Orell Füssli]                        |
|Paulo     |Coelho   |[de, pt]|[Buchhandlung im Volkshaus, Ex Libris, Orell Fü...|
+----------+---------+--------+-------------------------------------------------+
```

4.10.30. MULTISET value constructor

The MULTISET value constructor is one of jOOQ's and standard SQL's most powerful features. It allows for collecting the results of a non scalar subquery into a single nested collection value with

MULTISET semantics (ordinals are not defined on elements, though jOOQ attempts to maintain ORDER BY produced ordering when projecting a MULTISET).

For example, let's find:

- All authors.
- The languages in which that author has their books published.
- The book stores at which that author's books are available.

This can be done in a single query:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  MULTISET(
    SELECT DISTINCT
      LANGUAGE.CD
      LANGUAGE.DESCRIPTION
    FROM BOOK
    JOIN LANGUAGE ON BOOK.LANGUAGE_ID = LANGUAGE.ID
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) AS BOOKS,
  MULTISET(
    SELECT DISTINCT BOOK_TO_BOOK_STORE.BOOK_STORE_NAME
    FROM BOOK_TO_BOOK_STORE
    JOIN BOOK ON BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  ) AS BOOK_STORES
FROM AUTHOR
ORDER BY AUTHOR.ID
```

```
var result = create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,
          multiset(
              selectDistinct(
                  BOOK.language().CD,
                  BOOK.language().DESCRIPTION)
              .from(BOOK)
              .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          ).as("books"),
          multiset(
              selectDistinct(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
              .from(BOOK_TO_BOOK_STORE)

 .where(BOOK_TO_BOOK_STORE.tBook().AUTHOR_ID.eq(AUTHOR.ID))
          ).as("book_stores"))
      .from(AUTHOR)
      .orderBy(AUTHOR.ID)
      .fetch();
```

Notice how the Java 10 var keyword really shines here. It is usually not desirable to denote the types arising from nesting records or collections in jOOQ. The above var result is inferred to:

```
Result<Record4<
    String,          // AUTHOR.FIRST_NAME
    String,          // AUTHOR.LAST_NAME
    Result<Record2<
        String,      // LANGUAGE.CD
        String       // LANGUAGE.DESCRIPTION
    >>,              // books
    Result<Record1<
        String       // BOOK_TO_BOOK_STORE.BOOK_STORE_NAME
    >>               // book_stores
>> result = ...
```

Notice also that in a lot of cases, using RecordMappers can be very helpful when nesting collections to DTO trees, especially when combined with ad hoc converters.

The result of the above query may look like this:

```
+----------+---------+-----------------------------+-------------------------------------------------+
|first_name|last_name|books                        |book_stores                                      |
+----------+---------+-----------------------------+-------------------------------------------------+
|George    |Orwell   |[(en, English)]              |[(Ex Libris), (Orell Füssli)]                    |
|Paulo     |Coelho   |[(de, Deutsch), (pt, {null})]|[(Buchhandlung im Volkshaus), (Ex Libris), (Ore...|
+----------+---------+-----------------------------+-------------------------------------------------+
```

Or, when exported as JSON (alternatively, use JSON_ARRAYAGG directly):

```
[
  {
    "first_name": "George",
    "last_name": "Orwell",
    "books": [
      {
        "cd": "en",
        "description": "English"
      }
    ],
    "book_stores": [
      { "book_store_name": "Ex Libris" },
      { "book_store_name": "Orell Füssli" }
    ]
  },
  {
    "first_name": "Paulo",
    "last_name": "Coelho",
    "books": [
      {
        "cd": "de",
        "description": "Deutsch"
      },
      {
        "cd": "pt",
        "description": null
      }
    ],
    "book_stores": [
      { "book_store_name": "Buchhandlung im Volkshaus" },
      { "book_store_name": "Ex Libris" },
      { "book_store_name": "Orell Füssli" }
    ]
  }
]
```

Or, when exported as XML (alternatively, use XMLAGG directly):

```
<result>
  <record>
    <first_name>George</first_name>
    <last_name>Orwell</last_name>
    <books>
      <result>
        <record>
          <cd>en</cd>
          <description>English</description>
        </record>
      </result>
    </books>
    <book_stores>
      <result>
        <record>
          <book_store_name>Ex Libris</book_store_name>
        </record>
        <record>
          <book_store_name>Orell Füssli</book_store_name>
        </record>
      </result>
    </book_stores>
  </record>
  <record>
    <first_name>Paulo</first_name>
    <last_name>Coelho</last_name>
    <books>
      <result>
        <record>
          <cd>de</cd>
          <description>Deutsch</description>
        </record>
        <record>
          <cd>pt</cd>
          <description/>
        </record>
      </result>
    </books>
    <book_stores>
      <result>
        <record>
          <book_store_name>Buchhandlung im Volkshaus</book_store_name>
        </record>
        <record>
          <book_store_name>Ex Libris</book_store_name>
        </record>
        <record>
          <book_store_name>Orell Füssli</book_store_name>
        </record>
      </result>
    </book_stores>
  </record>
</result>
```

Implementation

The bad news is, hardly any dialect supports the MULTISET constructor natively (e.g. Informix or Oracle do). In all other dialects, it has to be emulated using SQL/JSON or SQL/XML. The above query may look like this, in PostgreSQL:

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME,
  (
    SELECT COALESCE(
      JSONB_AGG(JSONB_BUILD_ARRAY(V0, V1)),
      JSONB_BUILD_ARRAY()
    )
    FROM (
      SELECT DISTINCT
        ALIAS_86077489.CD AS V0,
        ALIAS_86077489.DESCRIPTION AS V1
      FROM BOOK
        JOIN LANGUAGE AS ALIAS_86077489
          ON BOOK.LANGUAGE_ID = ALIAS_86077489.ID
      WHERE BOOK.AUTHOR_ID = AUTHOR.ID
    ) AS T
  ) AS BOOKS,
  (
    SELECT COALESCE(
      JSONB_AGG(JSONB_BUILD_ARRAY(V0)),
      JSONB_BUILD_ARRAY()
    )
    FROM (
      SELECT DISTINCT BOOK_TO_BOOK_STORE.BOOK_STORE_NAME AS V0
      FROM BOOK_TO_BOOK_STORE
        JOIN BOOK AS ALIAS_129518614
          ON BOOK_TO_BOOK_STORE.BOOK_ID = ALIAS_129518614.ID
      WHERE ALIAS_129518614.AUTHOR_ID = AUTHOR.ID
    ) AS T
  ) AS BOOK_STORES
FROM AUTHOR
ORDER BY AUTHOR.ID
```

As you might notice, this produces a slightly different JSON structure than what one might have created manually. It generates arrays of arrays, which look something like this in a formatted result:

```
|first_name|last_name|books                           |book_stores                                                      |
|----------|---------|--------------------------------|-----------------------------------------------------------------|
|George    |Orwell   |[["en", "English"]]             |[["Ex Libris"], ["Orell Füssli"]]                                |
|Paulo     |Coelho   |[["de", "Deutsch"], ["pt", null]]|[["Buchhandlung im Volkshaus"], ["Ex Libris"], ["Orell Füssli"]]|
```

The benefits are:

- Arrays take less space than objects in JSON, so the serialisation format is more optimal
- Arrays don't care about duplicate column names, which can cause issues with various JSON parsers (even if JSON supports it)
- Array elements have a well defined order, object keys do not, and index lookups are faster than name lookups

The resulting JSON or XML document will be parsed and mapped to a jOOQ org.jooq.Result and org.jooq.Record hierarchy.

By default, the "best" serialisation format is used (JSON, XML, or ARRAY in the future), but you can override it using Settings.emulateMultiset, which offers the following values:

- DEFAULT: Let jOOQ decide how to serialise nested collections
- XML: Use XML to serialise nested collections
- JSON: Use JSON to serialise nested collections
- JSONB: Use JSONB to serialise nested collections
- NATIVE: Generate a native syntax

Dialect support

This example using jOOQ:

```
multiset(select(BOOK.ID, BOOK.TITLE).from(BOOK))
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, YUGABYTEDB
(
  SELECT coalesce(
    jsonb_agg(jsonb_build_array(v0, v1)),
    jsonb_build_array()
  )
  FROM (
    SELECT
      BOOK.ID v0,
      BOOK.TITLE v1
    FROM BOOK
  ) t
)

-- DB2
(
  SELECT xmlelement(
    NAME result,
    xmlagg(xmlelement(
      NAME record,
      xmlelement(NAME v0, BOOK.ID),
      xmlelement(
        NAME v1,
        xmlattributes(
          CASE
            WHEN BOOK.TITLE IS NULL THEN 'true'
          END AS xsi:nil
        ),
        BOOK.TITLE
      )
    ))
  )
  FROM BOOK
)

-- H2
(
  SELECT coalesce(
    json_arrayagg(json_array(BOOK.ID, BOOK.TITLE NULL ON NULL)),
    json_array(NULL ON NULL)
  )
  FROM BOOK
)

-- INFORMIX
MULTISET(
  SELECT BOOK.ID, BOOK.TITLE
  FROM BOOK
)

-- MARIADB
(
  SELECT coalesce(
    json_merge_preserve(
      '[]',
      concat(
        '[',
        group_concat(json_array(BOOK.ID, BOOK.TITLE) SEPARATOR ','),
        ']'
      )
    ),
    json_array()
  )
  FROM BOOK
)

-- MYSQL
(
  SELECT coalesce(
    json_merge_preserve(
      '[]',
      concat(
        '[',
        group_concat(json_array(t.v0, t.v1) SEPARATOR ','),
        ']'
      )
    ),
    json_array()
  )
  FROM (
    SELECT
      BOOK.ID v0,
      BOOK.TITLE v1
    FROM BOOK
  ) t
)

-- ORACLE
(
  SELECT coalesce(
    json_arrayagg(json_array(t.v0, t.v1 NULL ON NULL RETURNING clob) FORMAT JSON RETURNING clob),
    json_array(RETURNING clob)
  )
  FROM (
    SELECT
      BOOK.ID v0,
      BOOK.TITLE v1
    FROM BOOK
  ) t
)

-- SNOWFLAKE
(
  SELECT coalesce(
    array_agg(array_construct(coalesce(
      to_variant(t.v0),
      parse_json('null')
    ), coalesce(
```

4.10.31. 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.

Projecting row value expressions

Row value expressions can be used to project nested records, which allows for powerful mapping of structure data directly in SQL.

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.

Dialect support

This example using jOOQ:

```
row(BOOK.ID, BOOK.TITLE)
```

Translates to the following dialect specific expressions:

```
-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB,
-- MARIADB, MEMSQL, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
(BOOK.ID, BOOK.TITLE)

-- INFORMIX
ROW (BOOK.ID, BOOK.TITLE)
```

4.10.32. Nested records

The DSL.row() constructor isn't only useful for different types of row value expression predicates, but also to project nested record types, in most cases even Record1 to Record22 types, which maintain column level type safety.

All org.jooq.Row1 to org.jooq.Row22 types as well as the org.jooq.RowN type extend org.jooq.SelectField, meaning they can be placed in the SELECT clause or the RETURNING clause. The T type variable in SelectField<T> is bound to the appropriate Record1 to Record22 type, which allows for easily projecting nested records:

```
SELECT
  ID,
  ROW(
    FIRST_NAME,
    LAST_NAME
  )
FROM AUTHOR
```

```
// Type inference via lambdas or var really shines here!
Result<Record2<Integer, Record2<String, String>>> result =
create.select(
         AUTHOR.ID,
         row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME))
      .from(AUTHOR)
      .fetch();
```

Combining nested records with arrays

If your RDBMS supports ARRAY types and ARRAY constructors, and if nested records are natively supported, chances are that you can combine the two features. For example, to find all books for an author, as a nested collection rather than a flat join:

```
SELECT
  ID,
  ROW(
    AUTHOR.FIRST_NAME,
    AUTHOR.LAST_NAME
  ),
  ARRAY(
    SELECT BOOK.ID, BOOK.TITLE
    FROM BOOK
    WHERE BOOK.AUTHOR_ID = AUTHOR.ID
  )
FROM AUTHOR
```

```
// Type inference via lambdas or var really shines here!
Result<Record3<
  Integer,
  Record2<String, String>,
  Record2<Integer, String>[]
>> result =
create.select(
         AUTHOR.ID,
         row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
         array(
           select(row(BOOK.ID, BOOK.TITLE))
           .from(BOOK)
           .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
         )
       )
      .from(AUTHOR)
      .fetch();
```

Attaching RecordMappers to nested records

Nested records help structure your result sets using structural typing, but they really shine when you attach a RecordMapper to them. A RecordMapper is a java.lang.FunctionalInterface that can convert a Record subtype to any user type E. By calling e.g. Row2.mapping(), you can attach an ad-hoc converter to the nested record type to turn the nested object into something much more meaningful:

```
// Especially useful using Java 16 record types!
record Name(String firstName, String lastName) {}
record Author(int id, Name name) {}

// The "scary" structural type has gone!
List<Author> authors =
create.select(
         AUTHOR.ID,
         row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).mapping(Name::new)
       )
      .from(AUTHOR)
      .fetch(Records.mapping(Author::new));
```

All of the above is type safe and uses no reflection! Try it out yourself - add or remove a column to the query or to the records, and observe the compilation errors that appear.

Now for the ARRAY example:

```
record Name(String firstName, String lastName) {}
record Book(int id, String title) {}
record Author(int id, Name name, Book[] books) {}

// Again, no structural typing here has gone!
List<Author> authors =
create.select(
         AUTHOR.ID,
         row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).mapping(Name::new),
         array(
           select(row(BOOK.ID, BOOK.TITLE).mapping(Book.class, Book::new)
           .from(BOOK)
           .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
         )
       )
      .from(AUTHOR)
      .fetch(Records.mapping(Author::new));
```

Again, everything is type safe. Unfortunately, reflection is needed in this case to construct a Book[] array. You must pass the Book.class reference to help jOOQ with that. If you prefer lists, no problem. You can wrap the array again using the same technique, using an explicit ad-hoc converter:

```
record Name(String firstName, String lastName) {}
record Book(int id, String title) {}
record Author(int id, Name name, List<Book> books) {} // Is now using a List<Book> instead of Book[]

// Again, no structural typing here has gone!
List<Author> authors =
create.select(
         AUTHOR.ID,
         row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME).mapping(Name::new),
         array(
           select(row(BOOK.ID, BOOK.TITLE).mapping(Book.class, Book::new)
           .from(BOOK)
           .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
         ).convertFrom(Arrays::asList) // Additional converter here
       )
      .from(AUTHOR)
      .fetch(Records.mapping(Author::new));
```

Dialect support

This example using jOOQ:

```
select(row(BOOK.ID, BOOK.TITLE))
```

Translates to the following dialect specific expressions:

```
-- ACCESS
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM (
  SELECT count(*) dual
  FROM MSysResources
) AS dual

-- ASE, BIGQUERY, EXASOL, H2, MARIADB, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, VERTICA
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE

-- AURORA_MYSQL, MEMSQL, ORACLE
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM DUAL

-- AURORA_POSTGRES, COCKROACHDB, POSTGRES, YUGABYTEDB
SELECT ROW (BOOK.ID, BOOK.TITLE) nested

-- DB2
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM SYSIBM.DUAL

-- DERBY
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM SYSIBM.SYSDUMMY1

-- FIREBIRD
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM RDB$DATABASE

-- HANA, SYBASE
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM SYS.DUMMY

-- HSQLDB
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM (VALUES(1)) AS dual(dual)

-- INFORMIX
SELECT ROW (BOOK.ID, BOOK.TITLE) nested
FROM (
  SELECT 1 AS dual
  FROM systables
  WHERE (tabid = 1)
) AS dual

-- TERADATA
SELECT BOOK.ID nested__ID,
BOOK.TITLE nested__TITLE
FROM (
  SELECT 1 AS "dual"
) AS "dual"
```

4.11. 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.

4.11.1. 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.

4.11.2. 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.

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.

4.11.3. 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();
```

4.11.4. 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"))
```

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)
```

4.11.5. 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")
```

4.11.6. 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
```

4.11.7. 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:

```
-- ACCESS, ASE, DERBY, EXASOL, FIREBIRD, HANA, MEMSQL, SQLDATAWAREHOUSE, SQLSERVER, SYBASE
(
  AUTHOR.FIRST_NAME = 'John'
  AND AUTHOR.LAST_NAME = 'Doe'
)

-- AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, H2, HSQLDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE,
-- SQLITE, VERTICA, YUGABYTEDB
(AUTHOR.FIRST_NAME, 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"
)
```

4.11.8. 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, COCKROACHDB, DB2, DERBY, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MARIADB, MYSQL,
-- ORACLE, POSTGRES, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
BOOK.AUTHOR_ID = ANY (
  SELECT AUTHOR.ID
  FROM AUTHOR
)

-- ACCESS, BIGQUERY, EXASOL, MEMSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE
/* UNSUPPORTED */
```

4.11.9. 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:

```
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'
```

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:

```
-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DB2, DERBY, FIREBIRD, H2, HANA, INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE,
-- REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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'
```

4.11.10. 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:

```
-- ACCESS, ASE, DERBY, 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, COCKROACHDB, DB2, H2, HSQLDB, POSTGRES, REDSHIFT, SNOWFLAKE, SQLITE, VERTICA, YUGABYTEDB
(BOOK.ID, BOOK.TITLE) BETWEEN (1, 'A') AND (10, 'Z')

-- 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"
  )
)
```

4.11.11. 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, FIREBIRD, H2, HSQLDB, POSTGRES, REDSHIFT, SNOWFLAKE, SQLSERVER,
-- YUGABYTEDB
AUTHOR.FIRST_NAME IS DISTINCT FROM AUTHOR.LAST_NAME

-- 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 */
```

4.11.12. DOCUMENT predicate

The DOCUMENT predicate can be used to check if a given expression is an XML document (as opposed to toher XML content).

```
SELECT 1
WHERE
  '<xml/>' IS DOCUMENT
  AND '<!-- comment -->' IS NOT DOCUMENT
```

```
create.selectOne()
      .where(val("<xml/>").isDocument())
      .and(val("<!-- comment -->").isNotDocument())
      .fetch();
```

Dialect support

This example using jOOQ:

```
AUTHOR.FIRST_NAME.isDocument()
```

Translates to the following dialect specific expressions:

```
-- POSTGRES
AUTHOR.FIRST_NAME IS DOCUMENT

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB,
-- INFORMIX, MARIADB, MEMSQL, MYSQL, ORACLE, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA,
-- VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.13. EXISTS predicate

Slightly less intuitive, yet more powerful than the previously discussed 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
)
```

4.11.14. 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
)
```

4.11.15. 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:

```
-- ACCESS, ASE, DERBY, 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, COCKROACHDB, H2, HSQLDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SQLITE, TERADATA,
-- 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)
)

-- ORACLE
('FIRST', 'LAST') IN ((
  SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  FROM AUTHOR
))
```

4.11.16. 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:

```
-- MARIADB, MYSQL
json_valid('{}')

-- ORACLE
'{}' IS JSON

-- SNOWFLAKE
(check_json('{}') IS NULL)

-- SQLDATAWAREHOUSE, SQLSERVER
(isjson('{}') = 1)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB,
-- INFORMIX, MEMSQL, POSTGRES, REDSHIFT, SQLITE, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.17. JSON_EXISTS predicate

The JSON_EXISTS predicate can be used to check whether a JSON path expression produces a value within a JSON document (see also the JSON_VALUE function)

```
SELECT 1
FROM dual
WHERE json_exists('{"a":1}', '$.a')
```

```
create.selectOne()
      .where(jsonExists(val(JSON.valueOf("{\"a\":1}")), "$.a"))
      .fetch();
```

Dialect support

This example using jOOQ:

```
jsonExists(val(json("{\"a\":1}")), "$.a")
```

Translates to the following dialect specific expressions:

```
-- AURORA_POSTGRES, COCKROACHDB
JSON_EXISTS(CAST('{"a":1}' AS json), '$.a')

-- DB2, MARIADB, ORACLE
JSON_EXISTS('{"a":1}', '$.a')

-- MYSQL
json_contains_path('{"a":1}', 'one', '$.a')

-- POSTGRES
jsonb_path_exists(CAST('{"a":1}' AS jsonb), CAST('$.a' AS jsonpath))

-- SQLITE
json_type('{"a":1}', '$.a') IS NOT NULL

-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, REDSHIFT, SNOWFLAKE,
-- SQLDATAWAREHOUSE, SQLSERVER, SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.18. 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.

Quantified LIKE predicate

In addition to the above, 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("%abc%")
```

Translates to the following dialect specific expressions:

```
-- All dialects
BOOK.TITLE LIKE '%abc%'
```

4.11.19. 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
```

4.11.20. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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
```

4.11.21. 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))
```

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
)
```

4.11.22. 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:

```
-- All dialects
BOOK.TITLE SIMILAR TO '%X%'
```

4.11.23. Spatial predicates

A few databases have implemented the ISO/IEC 13249-3 SQL standard spatial extensions (or vendor specific adaptations thereof) to calculate geometric or geographic sets.

Most functionality comes in the form of spatial functions, but some useful predicates can also be formed from geometries.

4.11.23.1. ST_Contains

This predicate checks if a geometry contains another geometry. For example:

```
create.select(
  field(stContains(
    stGeomFromText("POLYGON ((-3 -1, -1 -1, -1 1, -3 1, -3 -1))"),
    stGeomFromText("POINT (-2 0)")
  )),
  field(stContains(
    stGeomFromText("POLYGON ((3 -1, 1 -1, 1 1, 3 1, 3 -1))"),
    stGeomFromText("POINT (4 0)")
  ))
).fetch();
```

The result being, for example

```
+-------------+------------+
| ST_Contains | ST_Contains |
+-------------+------------+
| true        | false      |
+-------------+------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stContains(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_contains(geometry1, geometry2)

-- ORACLE
((st_contains(geometry1, geometry2) = 'TRUE'))

-- SQLSERVER
geometry1.STContains(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.2. ST_Crosses

This predicate checks if a geometry crosses another geometry. For example:

```
create.select(
  field(stCrosses(
    stGeomFromText("LINESTRING (-3 -1, -1 1)"),
    stGeomFromText("LINESTRING (-1 -1, -3 1)")
  )),
  field(stCrosses(
    stGeomFromText("LINESTRING (1 -1, 1 1)"),
    stGeomFromText("LINESTRING (3 -1, 3 1)")
  ))
).fetch();
```

The result being, for example

```
+------------+------------+
| ST_Crosses | ST_Crosses |
+------------+------------+
| true       | false      |
+------------+------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stCrosses(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_crosses(geometry1, geometry2)

-- SQLSERVER
geometry1.STCrosses(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.3. ST_Disjoint

This predicate checks if a geometry is disjoint from another geometry. For example:

```
create.select(field(stDisjoint(
  stGeomFromText("LINESTRING (-1 -1, -1 1)"),
  stGeomFromText("LINESTRING (1 -1, 1 1)")
))).fetch();
```

The result being, for example

```
+-------------+
| ST_Disjoint |
+-------------+
| true        |
+-------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stDisjoint(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_disjoint(geometry1, geometry2)

-- SQLSERVER
geometry1.STDisjoint(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.4. ST_Equals

This predicate checks if a geometry is "spatially equal" to another geometry (i.e. the ordering of points is irrelevant for this equality). For example:

```
create.select(field(stEquals(
  stGeomFromText("LINESTRING (-1 -1, 1 1)"),
  stGeomFromText("LINESTRING (1 1, -1 -1)")
))).fetch();
```

The result being, for example

```
+-----------+
| ST_Equals |
+-----------+
| true      |
+-----------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stEquals(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_equals(geometry1, geometry2)

-- ORACLE
((st_equal(geometry1, geometry2) = 'TRUE'))

-- SQLSERVER
geometry1.STEquals(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.5. ST_Intersects

This predicate checks if a geometry intersects another geometry. For example:

```
create.select(
  field(stIntersects(
    stGeomFromText("LINESTRING (-3 -1, -1 1)"),
    stGeomFromText("LINESTRING (-1 -1, -3 1)")
  )),
  field(stIntersects(
    stGeomFromText("LINESTRING (1 -1, 1 1)"),
    stGeomFromText("LINESTRING (3 -1, 3 1)")
  ))
).fetch();
```

The result being, for example

```
+---------------+---------------+
| ST_Intersects | ST_Intersects |
+---------------+---------------+
| true          | false         |
+---------------+---------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stIntersects(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_intersects(geometry1, geometry2)

-- ORACLE
(sdo_geom.sdo_intersection(geometry1, geometry2, tol => null)).sdo_gtype IS NOT NULL

-- SQLSERVER
geometry1.STIntersects(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.6. ST_IsClosed

This predicate checks if a linestring has the same start point and end point. For example:

```
create.select(
  field(stIsClosed(stGeomFromText("LINESTRING (-3 -1, -1 -1, -3 1, -3 -1)"))),
  field(stIsClosed(stGeomFromText("LINESTRING (3 1, 1 1, 3 -1)")))
).fetch();
```

The result being, for example

```
+-------------+-------------+
| ST_IsClosed | ST_IsClosed |
+-------------+-------------+
| true        | false       |
+-------------+-------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stIsClosed(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_isclosed(geometry)

-- SQLSERVER
geometry.STIsClosed() = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.7. ST_IsEmpty

This predicate checks if a geometry is empty. For example:

```
create.select(field(stIsEmpty(stGeomFromText("POLYGON EMPTY")))).fetch();
```

The result being, for example

```
+------------+
| ST_IsEmpty |
+------------+
| true       |
+------------+
```

Dialect support

This example using jOOQ:

```
stIsEmpty(geometry)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, ORACLE, POSTGRES, REDSHIFT, SNOWFLAKE
st_isempty(geometry)

-- SQLSERVER
geometry.STIsEmpty() = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.8. ST_Overlaps

This predicate checks if two geometries overlap, i.e. they intersect, but no geometry completely contains the other. For example:

```
create.select(
  field(stOverlaps(
    stGeomFromText("POLYGON ((-3 -1, -1 -1, -1 1, -3 1, -3 -1))"),
    stGeomFromText("POLYGON ((-4 -2, -2 -2, -2 0, -4 0, -4 -2))")
  )),
  field(stOverlaps(
    stGeomFromText("POLYGON ((3 -1, 1 -1, 1 1, 3 1, 3 -1))"),
    stGeomFromText("POLYGON ((2 -1, 1 -1, 1 0, 2 0, 2 -1))")
  ))
).fetch();
```

The result being, for example

```
+-------------+------------+
| ST_Overlaps | ST_Overlaps |
+-------------+------------+
| true        | false       |
+-------------+------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stOverlaps(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_overlaps(geometry1, geometry2)

-- ORACLE
((st_overlaps(geometry1, geometry2) = 'TRUE'))

-- SQLSERVER
geometry1.STOverlaps(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.9. ST_Touches

This predicate checks if two geometries touch. For example:

```
create.select(
  field(stTouches(
    stGeomFromText("LINESTRING (-1 0, -1 1, -2 1, -2 2)"),
    stGeomFromText("LINESTRING (-1 1, -2 0)")
  )),
  field(stTouches(
    stGeomFromText("LINESTRING (1 0, 1 1, 2 1, 2 2)"),
    stGeomFromText("LINESTRING (1 2, 0 1)")
  ))
).fetch();
```

The result being, for example

```
+------------+------------+
| ST_Touches | ST_Touches |
+------------+------------+
| true       | false      |
+------------+------------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stTouches(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_touches(geometry1, geometry2)

-- ORACLE
((st_touch(geometry1, geometry2) = 'TRUE'))

-- SQLSERVER
geometry1.STTouches(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.23.10. ST_Within

This predicate checks if one geometry is within another. This is the inverse of ST_Contains. For example:

```
create.select(
  field(stWithin(
    stGeomFromText("POINT (-2 0)"),
    stGeomFromText("POLYGON ((-3 -1, -1 -1, -1 1, -3 1, -3 -1))")
  )),
  field(stWithin(
    stGeomFromText("POINT (4 0)"),
    stGeomFromText("POLYGON ((3 -1, 1 -1, 1 1, 3 1, 3 -1))")
  ))
).fetch();
```

The result being, for example

```
+-----------+-----------+
| ST_Within | ST_Within |
+-----------+-----------+
| true      | false     |
+-----------+-----------+
```

Or, visually:

Dialect support

This example using jOOQ:

```
stWithin(geometry1, geometry2)
```

Translates to the following dialect specific expressions:

```
-- AURORA_MYSQL, AURORA_POSTGRES, COCKROACHDB, MARIADB, MYSQL, POSTGRES, REDSHIFT, SNOWFLAKE
st_within(geometry1, geometry2)

-- ORACLE
((sdo_inside(geometry1, geometry2) = 'TRUE'))

-- SQLSERVER
geometry1.STWithin(geometry2) = 1

-- ACCESS, ASE, BIGQUERY, DB2, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX, MEMSQL, SQLDATAWAREHOUSE, SQLITE,
-- SYBASE, TERADATA, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.24. 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:

```
-- ACCESS, ASE, AURORA_MYSQL, BIGQUERY, COCKROACHDB, DB2, DERBY, EXASOL, FIREBIRD, HANA, HSQLDB, INFORMIX, MARIADB,
-- MEMSQL, MYSQL, ORACLE, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SQLSERVER, SYBASE, TERADATA, 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
)
```

4.11.25. XMLEXISTS predicate

The XMLEXISTS predicate can be used to check whether an XQuery or XPath expression produces a value within an XML document (see also the XMLQUERY function)

```
SELECT 1
WHERE xmlexists('/a/b' PASSING '<a><b/></a>')
```

```
create.selectOne()
      .where(xmlexists("/a/b").passing(XML.valueOf("<a><b/></
a>")))
      .fetch();
```

Dialect support

This example using jOOQ:

```
xmlexists("/a/b").passing(xml("<a><b/></a>"))
```

Translates to the following dialect specific expressions:

```
-- DB2, ORACLE, TERADATA
XMLEXISTS(
  '/a/b'
  PASSING '<a><b/></a>'
)

-- POSTGRES
XMLEXISTS(
  '/a/b'
  PASSING CAST('<a><b/></a>' AS xml)
)

-- SQLSERVER
('<a><b/></a>'.EXIST('/a/b') = 1)

-- ACCESS, ASE, AURORA_MYSQL, AURORA_POSTGRES, BIGQUERY, COCKROACHDB, DERBY, EXASOL, FIREBIRD, H2, HANA, HSQLDB, INFORMIX,
-- MARIADB, MEMSQL, MYSQL, REDSHIFT, SNOWFLAKE, SQLDATAWAREHOUSE, SQLITE, SYBASE, VERTICA, YUGABYTEDB
/* UNSUPPORTED */
```

4.11.26. 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).

4.12. 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.
- MULTISET_AGG function: The MULTISET aggregate function is not natively supported anywhere, even if a native implementation for the MULTISET value constructor exists.
- 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.

4.13. 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

o Constructed dynamically
o 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.

4.13.1. Optional column expressions

A key capability when creating dynamic SQL queries is to be able to provide optional column expressions. For example, imagine you have a condition based on which you want to add an ORDER BY clause and a LIMIT clause. You can do this using DSL.noField():

```
boolean condition = ...

create.select(BOOK.ID)
      .from(BOOK)
      .orderBy(condition ? BOOK.ID : noField())
      .limit(condition ? val(10) : noField(INTEGER))
      .fetch();
```

The above query produces:

```
-- If condition is true
SELECT book.id FROM book ORDER BY book.id LIMIT 10

-- If condition is false
SELECT book.id FROM book
```

In clauses that do not project columns, the noField() 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, or nested records, as well as function calls, in case of which a NULL value will be projected.

Using a noField() as a conditional expression, e.g. by wrapping it with DSL.condition(noField()) will produce a DSL.noCondition().

4.13.2. 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, or nested records, 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.

Using a noCondition() as a column expression, e.g. by wrapping it with DSL.field(noCondition()) will produce a DSL.noField(BOOLEAN).

4.14. 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 a DSL is always expressed in a host language (Java in this case), which was not made for exactly the same purposes as its hosted DSL. If it were, then jOOQ would be implemented on a compiler-level, similar to LINQ in .NET. But it's not, and so, the DSL is limited by language constraints of its host language. We have seen many functionalities where the DSL becomes a bit verbose. This can be especially true for:

- aliasing
- derived tables
- arithmetic expressions
- casting

You'll probably find other examples. If verbosity scares you off, don't worry. The verbose use-cases for jOOQ are rather rare, and when they come up, you do have an option. Just write SQL the way you're used to!

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

The DSL plain SQL API

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")
Condition condition(String sql);

// Construct a condition with bind values
// Example: condition("a = ?", 1);
Condition condition(String sql, Object... bindings);

// Construct a condition taking other jOOQ object arguments
// Example: condition("a = {0}", val(1));
Condition condition(String sql, QueryPart... parts);
```

Both the bind value and the query part argument overloads make use of 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 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:

- jOOQ doesn't know what you're doing. You're on your own again!
- 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.

4.15. 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
- 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
- Quoted names
- Comments
- JDBC escape sequences
- 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.

4.16. 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:

4.16.1. 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.

4.16.1.1. 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()

4.16.1.2. 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()
```

4.16.1.3. 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()
```

4.16.2. 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.

4.16.3. 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.

4.16.3.1. 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()
```

4.16.3.2. 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.

4.17. 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).

4.17.1. 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 API 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.
- Parser.parseResultQuery(String): This produces the org.jooq.ResultQuery type, containing a single query.
- 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.

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;
```

4.17.2. 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-3.17.8.jar:reactive-streams-1.0.3.jar:r2dbc-spi-0.9.0.RELEASE.jar org.jooq.ParserCLI -h
Usage:
  -f / --formatted                                               Format output SQL
  -h / --help                                                    Display this help
  -k / --keyword                            <RenderKeywordCase>  Specify the output keyword case
 (org.jooq.conf.RenderKeywordCase)
  -i / --identifier                         <RenderNameCase>     Specify the output identifier case
 (org.jooq.conf.RenderNameCase)
  -Q / --quoted                             <RenderQuotedNames>  Specify the output identifier quoting
 (org.jooq.conf.RenderQuotedNames)
  -F / --from-dialect                       <SQLDialect>         Specify the input dialect (org.jooq.SQLDialect)
  -T / --to-dialect                         <SQLDialect>         Specify the output dialect (org.jooq.SQLDialect)
  -S / --schema                             <String>             Specify the input SQL schema
  -s / --sql                                <String>             Specify the input SQL string

Additional flags:
  --parse-date-format                       <String>
  --parse-locale                            <Locale>
  --parse-name-case                         <ParseNameCase>
  --parse-named-param-prefix                <String>
  --parse-retain-comments-between-queries
  --parse-set-commands
  --parse-timestamp-format                  <String>
  --parse-unknown-functions                 <ParseUnknownFunctions>
  --parse-unsupported-syntax                <ParseUnsupportedSyntax>
  --render-optional-inner-keyword           <RenderOptionalKeyword>
  --render-optional-outer-keyword           <RenderOptionalKeyword>
  --render-optional-as-keyword-for-field-aliases  <RenderOptionalKeyword>
  --render-optional-as-keyword-for-table-aliases  <RenderOptionalKeyword>

Commercial distribution only features:
  --render-coalesce-to-empty-string-in-concat
  --transform-patterns
  --transform-ansi-join-to-table-lists
  --transform-qualify                       <Transformation>
  --transform-rownum                        <Transformation>
  --transform-table-lists-to-ansi-join
  --transform-unneeded-arithmetic           <TransformUnneededArithmeticExpressions>

  -I / --interactive                                          Start interactive mode

$ java -cp jooq-3.17.8.jar:reactive-streams-1.0.3.jar:r2dbc-spi-0.9.0.RELEASE.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.

4.17.3. SQL Parser Listener

In order to implement custom parser behaviour, it is possible to provide your Configuration with a set of custom org.jooq.ParseListener implementations.

The current SPI offers hooking into the parsing of 3 types of syntactic elements (see also the parser grammar):

- term to parse org.jooq.Field expressions
- predicate to parse org.jooq.Condition expressions
- tableFactor to parse org.jooq.Table expressions

And also these two lifecycle events:

- Before the start of a parse call
- After the end of a parse call

The idea is that any listener implementation you provide the parser with may be able to parse functions, top-level-precedence operators, etc. without having to deal with all the lower level precedence operators, such as AND, OR, NOT, +, -, etc. etc.

For example, assuming you want to add support for a logical LOGICAL_XOR operator as a function:

```
Query query = configuration
    .derive(ParseListener.onParseCondition(ctx -> {
        if (ctx.parseFunctionNameIf("LOGICAL_XOR")) {
            ctx.parse('(');
            Condition c1 = ctx.parseCondition();
            ctx.parse(',');
            Condition c2 = ctx.parseCondition();
            ctx.parse(')');

            return c1.andNot(c2).or(c2.andNot(c1));
        }

        // Let the parser take over if we don't know the token
        return null;
    })
    .dsl()
    .parser()
    .parseQuery("select * from t where logical_xor(t.a = 1, t.b = 2)");
```

The above will just translate the convenience function LOGICAL_XOR(c1, c2) into its formal definition c1 AND NOT c2 OR c2 AND NOT c1. But we can do even better than this. If a dialect has native XOR support, why not support that?

```
Query query = configuration
    .derive(ParseListener.onParseCondition(ctx -> {
        if (ctx.parseFunctionNameIf("LOGICAL_XOR")) {
            ctx.parse('(');
            Condition c1 = ctx.parseCondition();
            ctx.parse(',');
            Condition c2 = ctx.parseCondition();
            ctx.parse(')');

            return CustomCondition.of(c -> {
                switch (c.family()) {
                    case MARIADB:
                    case MYSQL:
                        c.visit(condition("{0} xor {1}", c1, c2));
                        break;
                    default:
                        c.visit(c1.andNot(c2).or(c2.andNot(c1)));
                        break;
                }
            });
        }

        // Let the parser take over if we don't know the token
        return null;
    }))
    .dsl()
    .parser()
    .parseQuery("select * from t where logical_xor(t.a = 1, t.b = 2)");

System.out.println(DSL.using(SQLDialect.MYSQL).render(query));
System.out.println(DSL.using(SQLDialect.ORACLE).render(query));
```

The output of the above is now:

```
-- MYSQL:
select * from t where (t.a = 1 xor t.b = 2);

-- ORACLE:
select * from t where (t.a = 1 and not (t.b = 2)) or (t.b = 2 and not (t.a = 1));
```

This way, with a modest effort, you can parse and/or translate arbitrary column expressions, table expressions, or conditional expressions that jOOQ does not support natively, or override the default behaviour of the parser in this area.

4.17.4. 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 layout of the grammar and the grammar itself is still work in progress)

The diagrams have been created with the neat RRDiagram library by Christopher Deckers.

4.18. SQL interpreter

Starting with jOOQ 3.13, a SQL interpreter has been implemented, which can interpret a subset of the SQL language (mostly DDL statements) and maintain an up-to-date in-memory representation of your database meta model.

The interpreter is made available through a variety of DSLContext.meta() methods, which can be used for example as follows:

```
// Using the parser
Meta meta1 = create.meta(
  "create table t (i int)",
  "alter table t add primary key (i)",
  "create table u (i int references t)"
);

Meta meta2 = create.meta(
  createTable("t").column("i", INTEGER),
  alterTable("t").add(primaryKey("i")),
  createTable("u").column("i", INTEGER).constraint(foreignKey("i").references("t"))
);
```

Interpretation is incremental. On any pre-existing org.jooq.Meta model, apply() can be called to derive a new version of the model. This includes being able to combine different source of models, including JDBC java.sql.DatabaseMetaData based ones.

```
// Get live access to your current Connection's DatabaseMetaData
Meta meta1 = create.meta();

// Create a new model representation with an interpreted, additional table
// The query is not executed! The DDL is only interpreted in jOOQ, and a derived meta model is created from it
Meta meta2 = meta1.apply("create table t (i int)");

// Create another new model representation, with a table removed
// Again, none of these queries are executed.
Meta meta3 = meta2.apply("drop table u cascade");
```

Finally, if you want to export the meta model to one of the different supported formats, you can use:

```
// The JAXB annotated XML version of your "information schema":
InformationSchema is = meta.informationSchema();

// A set of DDL queries that can be used to reproduce your schema on any database and dialect:
Queries queries = meta.ddl();
```

Different sources for the meta model can be used, including jOOQ API built DDL statements, or a set of SQL strings that will be parsed using the SQL parser, dynamically. These sources could be database change management scripts, such as those managed by Flyway, or by jOOQ.

The resulting type is the runtime org.jooq.Meta model, which gives access to the ordinary jOOQ API, including catalogs and schemas or tables. These objects can then, in turn, be used with any other jOOQ API, for instance to count all the rows in all tables in a database:

```
for (Table<?> table : meta.getTables())
  System.out.println(table + " has " + create.fetchCount(table) + " rows");
```

For a set of interpreter configuration flags, please refer to the section about the interpreter settings.

4.19. Schema diff

Starting with jOOQ 3.13, and the capability of interpreting DDL, jOOQ is now able to create a "diff" between two versions of your schema. An example:

```
// We're using interpreted Meta objects. But any other type of Meta can be used, too
Meta m1 = create.meta("create table t (i int)");
Meta m2 = create.meta("create table t (i int, j int)");

// The diff is now printed in both directions:
System.out.println("-- Schema upgrade");
System.out.println(m1.migrateTo(m2));
System.out.println();
System.out.println("-- Schema downgrade");
System.out.println(m2.migrateTo(m1));
```

The output of the above program is:

```
-- Schema upgrade
alter table T add J int;

-- Schema downgrade
alter table T drop J;
```

The diff algorithm can non-ambiguously determine the following set of changes

- Catalog additions and removals
- Schema additions and removals
- Table additions and removals
- Column additions and removals
- Column type changes
- Constraint additions, removals, and renamings (including primary keys, unique keys, foreign keys, and check constraints)
- Index additions, removals, and renamings
- View additions, removals, and replacements
- Sequence additions, removals, and sequence flag changes
- Comment additions and removals

The following set of changes are difficult to detect. Some heuristics could be implemented, but are not yet supported in jOOQ:

- Catalog renamings
- Schema renamings, and moving between catalogs
- Table renamings, and moving between schemas
- Column renamings, and moving between tables
- View renamings

Another way to use this API is the https://www.jooq.org/diff website.

4.20. Schema diff CLI

The schema diff API can be used to generate a diff between schema versions programmatically, as we've seen in the previous section about the schema diff API. This functionality can also usefully be accessed on the command line as shown below:

```
$ java -cp jooq-3.17.8.jar:reactive-streams-1.0.3.jar:r2dbc-spi-0.9.0.RELEASE.jar org.jooq.DiffCLI -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)
  -1 / --sql1         <String>              Specify the input SQL string 1 (from SQL)
  -2 / --sql2         <String>              Specify the input SQL string 2 (to SQL)

$ java -cp jooq-3.17.8.jar:reactive-streams-1.0.3.jar:r2dbc-spi-0.9.0.RELEASE.jar org.jooq.DiffCLI -T POSTGRES -1 "create table t (i
 int);" -2 "create table t (i int, j int)"
alter table t add j int null;
```

Windows users: Please replace : by ; in the above examples.

Another way to use this API is the https://www.jooq.org/diff website.

4.21. 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.

4.22. 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.

4.22.1. 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.

4.22.2. 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"))));
```

4.22.3. Inlined parameters

Sometimes, you may wish to avoid rendering bind variables while still using custom values in SQL. 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();
```

4.22.4. 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.

4.23. 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.

4.23.1. 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.

4.23.2. 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.
- org.jooq.Parameter: Routine parameters can be declared in the CREATE FUNCTION or CREATE PROCEDURE statements.

4.23.3. 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.

4.23.4. 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.

4.23.5. 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

```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

4.23.6. 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 CustomStatement implements Statement {}
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 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));
    }
}
```

Alternatively, implement it using a simple lambda:

```
public class MyDSL {
    public static Field<String> toChar(Field<?> field, String format) {
        return CustomField.of("to_char", VARCHAR, ctx -> {
            switch (ctx.family()) {
                case ORACLE:
                    ctx.visit(DSL.field("TO_CHAR({0}, {1})", String.class, arg0, arg1));
                    break;

                case SQLSERVER:
                    ctx.visit(DSL.field("CONVERT(VARCHAR(8), {0}, {1})", String.class, arg0, arg1));
                    break;

                default:
                    throw new UnsupportedOperationException("Dialect not supported");
            }
        });
    }
}
```

4.23.7. 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) { ... }
```

4.23.8. 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

4.23.9. SQL transformation

jOOQ supports a few built in SQL transformations, which can be enabled through a variety of settings. These transformations will produce transformed SQL output regardless if the input was created using the DSL API, or the parser.

4.23.9.1. ANSI JOIN to table lists

Almost all SQL dialects support ANSI 92 JOIN syntax, but sometimes, users may wish to generate SQL that is compatible e.g. with older Oracle releases that did not yet support this syntax, or had limited optimiser support for them. As a workaround, the Settings.transformAnsiJoinToTableLists flag can be turned on to produce the following type of transformation:

```
-- Input
SELECT *
FROM a
JOIN b ON a.id = b.id
```

```
-- Output
SELECT *
FROM a, b
WHERE a.id = b.id
```

This also works for OUTER JOIN:

```
-- Input
SELECT *
FROM a
LEFT JOIN b ON a.id = b.id
```

```
-- Output
SELECT *
FROM a, b
WHERE a.id = b.id(+)
```

Example configuration

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

4.23.9.2. Table lists to ANSI JOIN

When upgrading from legacy join syntax to the "new" (1992!) ANSI JOIN syntax, the Settings.transformTableListsToAnsiJoin flag can be turned on to produce the following type of transformation:

```
-- Input
SELECT *
FROM a, b
WHERE a.id = b.id
```

```
-- Output
SELECT *
FROM a
JOIN b ON a.id = b.id
```

This also works for OUTER JOIN:

```
-- Input
SELECT *
FROM a, b
WHERE a.id = b.id(+)
```

```
-- Output
SELECT *
FROM a
LEFT JOIN b ON a.id = b.id
```

Example configuration

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

4.23.9.3. ROWNUM to LIMIT

Old Oracle SQL statements may use ROWNUM filtering to paginate, as Oracle has introduced support for FETCH FIRST only in Oracle 12c.

This transformation allows for transforming some of these syntaxes to equivalent, standard syntax using window functions or FETCH FIRST:

```
-- Input
SELECT * FROM t WHERE rownum = 1
```

```
-- Output
SELECT * FROM t LIMIT 1
```

Example configuration

```
Settings settings = new Settings()
    .withTransformRownum(Transformation.WHEN_NEEDED);
```

4.23.9.4. QUALIFY to derived table

Teradata introduced the useful QUALIFY clause, which has since been reproduced by a number of implementations. jOOQ can transform a query containing the QUALIFY clause to an equivalent query filtering on a derived table

This transformation allows for transforming some of these syntaxes to equivalent, standard syntax using window functions and derived tables:

```
-- Input
SELECT *
FROM
  t
QUALIFY
  row_number () OVER (ORDER BY id) <= 5
```

```
-- Output
SELECT *
FROM (
  SELECT *, row_number () OVER (ORDER BY id) AS rn FROM t
) AS t
WHERE t.rn <= 5
```

Example configuration

```
Settings settings = new Settings()
    .withTransformQualify(Transformation.WHEN_NEEDED);
```

4.23.9.5. IN condition subquery with LIMIT to derived table

Some dialects do not support a LIMIT clause in a subquery of an IN predicate. jOOQ can transform such a subquery to an equivalent derived table:

```
-- Input
SELECT *
FROM t
WHERE id IN (SELECT id FROM u ORDER BY id LIMIT 5)
```

```
-- Output
SELECT *
FROM t
WHERE id IN (SELECT * FROM (SELECT id FROM u ORDER BY id LIMIT 5)
 AS u)
```

Example configuration

```
Settings settings = new Settings()
    .setTransformInConditionSubqueryWithLimitToDerivedTable(Transformation.WHEN_NEEDED);
```

4.23.9.6. Unnecessary arithmetic expressions

The jOOQ expression tree may contain unnecessary arithmetic expressions, which may not be desirable in generated SQL output. These expressions may originate from user written jOOQ API, or from internal emulations.

```
-- Input
SELECT -1 - (1 + 1)
```

```
-- Output
SELECT -3
```

Example configuration

```
Settings settings = new Settings()
    .withTransformUnneededArithmeticExpressions(TransformUnneededArithmeticExpressions.ALWAYS);
```

These options are available:

- NEVER (default): Don't transform arithmetic expressions
- INTERNAL: Transform only arithmetic expressions produced by jOOQ's internals, e.g. for emulations
- ALWAYS: Always transform arithmetic expressions, if possible

4.23.9.7. Pattern based transformation

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Starting with jOOQ 3.16, a new Query Object Model (QOM) API has been added, which features traversal and replacement SPIs. One out of the box application of replacement are pattern transformations, where we recognise common patterns in a query's expressions and allow for replacing them by a better expression.

For example, the following transformations are possible:

```
-- With Settings.transformPatternsTrim active, this:
SELECT RTRIM(LTRIM(col)) FROM tab;

-- ... is transformed into the equivalent expression:
SELECT TRIM(col) FROM tab;
```

The whole feature set is deactivated by default using the Settings.transformPatterns flag as the traversal overhead does not warrant activation by default. Common use-cases for the activation of this feature include:

- SQL linting, including as part of an ExecuteListener, where the effort can be controlled in a utility run only during integration testing, for example.
- SQL cleanup, including as part of a ParsingConnection, where inputs/outputs are cached by default.
- Dialect migration, when moving from a less powerful dialect to a more powerful one, or to a new version, manually written emulations may become obsolete.
- Patching specific SQL features only

Individual features can either be turned on in bulk and opted out of individually, or each feature is turned on individually, depending on the use case.

Repetition of pattern replacement

As any replacement SPI implementation, the algorithm iterates over the jOOQ expression tree until it can no longer replace anything, i.e. until the transformations stabilise. For example, the following sequence of transformations is possible:

```
-- Original SQL
SELECT NOT(1 = 0) AND 1 = 1;

-- Step 1: Settings.transformPatternsNotComparison
SELECT 1 != 0 AND 1 = 1;

-- Step 2: Settings.transformPatternsTrivialPredicates
SELECT TRUE AND TRUE;

-- Step 3: Settings.transformPatternsTrivialPredicates
SELECT TRUE;
```

Dialect specific result

The following sections will make claims like 1 / TAN(x) or COS(x) / SIN(x) being replaced by COT(x), see e.g. Trigonometric functions. Some dialects may not have native support for COT(), so this is going to be emulated on those dialects, making the transformation effectively redundant. But these things happen at different times:

- This pattern transformation feature can either be invoked explicitly by users, or it is enabled by configuration and then happens ***before*** all the rendering takes place, only once for the top level org.jooq.QueryPart
- The rendering of SQL invokes emulations as the above, ***after*** the SQL has been transformed.

In other words, the two operations do not know about each other.

4.23.9.7.1. AND to NOT IN

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

A concatenation of AND predicates combining non-equalities that always use the same operand on one side can be replaced by a more concise, but equivalent NOT IN predicate.

Using Settings.transformPatternsAndNeToNotIn, the following transformations can be achieved:

```
-- With Settings.transformPatternsAndNeToNotIn active, this:
SELECT * FROM tab WHERE x != 1 AND x != 2 AND x != 3;

-- ... is transformed into the equivalent expression:
SELECT * FROM tab WHERE x NOT IN (1, 2, 3);
```

See also OR to IN

4.23.9.7.2. Arithmetic expressions

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some arithmetic expressions are unnecessary and may obscure the SQL query, possibly preventing optimisations that are otherwise possible. Sometimes, this is used as a trick, e.g. to prevent index access where such index access is undersirable

Using Settings.transformPatternsArithmeticExpressions, the following transformations can be achieved:

Identities

```
-- With Settings.transformPatternsArithmeticExpressions active, this:
SELECT 0 + x, 1 * x, POWER(x, 1)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT x, x, x FROM tab;
```

Negation

```
-- With Settings.transformPatternsArithmeticExpressions active, this:
SELECT
  0 - x,
  x + (-y),
  x - (-y),
  (-x) * (-y),
  (-x) / (-y),
  (-1) * x
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  -x,    -- 0 - x
  x - y, -- x + (-y)
  x + y, -- x - (-y)
  x * y, -- (-x) * (-y)
  x / y, -- (-x) / (-y)
  -x     -- (-1) * x
FROM tab;
```

Other

```
-- With Settings.transformPatternsArithmeticExpressions active, this:
SELECT
  1 / y * x,
  x * x,
  x + const,
  x * const
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x / y,     -- 1 / y * x
  SQUARE(x), -- x * x
  const + x, -- x + const
  const * x  -- x * const
FROM tab;
```

4.23.9.7.3. COUNT(*) scalar subquery comparison

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

When comparing a scalar subquery that calculates COUNT(*) with a single value, then chances are that weaker optimisers might be better off with an equivalent EXISTS predicate as can be seen in this blog post about COUNT(*) vs EXISTS.

This transformation is only applied under certain circumstances, including:

- In the absence of UNION and other set operations
- In the absence of GROUP BY and HAVING
- Only with COUNT(*), not with COUNT(expr) (see for that case)

Using Settings.transformPatternsScalarSubqueryCountAsteriskGtZero, the following transformations can be achieved:

```
-- With Settings.transformPatternsScalarSubqueryCountAsteriskGtZero active, this:
SELECT
  (SELECT COUNT(*) FROM tab) > 0;

-- ... is transformed into the equivalent expression:
SELECT
  EXISTS (SELECT 1 FROM tab);
```

4.23.9.7.4. Empty scalar subquery

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

When a scalar subquery is provably empty, we can replace it with NULL.

Using Settings.transformPatternsEmptyScalarSubquery, the following transformations can be achieved:

```
-- With Settings.transformPatternsEmptyScalarSubquery active, this:
SELECT
  (SELECT c FROM tab WHERE FALSE);

-- ... is transformed into the equivalent expression:
SELECT
  NULL -- (SELECT c FROM tab WHERE FALSE);
```

4.23.9.7.5. Hyperbolic functions

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some dialects may not support all hyperbolic functions, or they are derived from other, much more complex maths expressions.

Using Settings.transformPatternsHyperbolicFunctions, the following transformations can be achieved:

```
-- With Settings.transformPatternsTrigonometricFunctions active, this:
SELECT
  (EXP(x) - EXP(-x)) / 2
  (EXP(2 * x) - 1) / (2 * EXP(x))
  (1 - EXP(-2 * x)) / (2 * EXP(-x))
  (EXP(x) + EXP(-x)) / 2
  (EXP(2 * x) + 1) / (2 * EXP(x))
  (1 + EXP(-2 * x)) / (2 * EXP(-x))
  SINH(x) / COSH(x)
  1 / COTH(x)
  (EXP(x) - EXP(-x)) / (EXP(x) + EXP(-x))
  (EXP(2 * x) - 1) / (EXP(2 * x) + 1)
  COSH(x) / SINH(x)
  1 / TANH(x)
  (EXP(x) + EXP(-x)) / (EXP(x) - EXP(-x))
  (EXP(2 * x) + 1) / (EXP(2 * x) - 1)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  SINH(x), -- (EXP(x) - EXP(-x)) / 2,
  SINH(x), -- (EXP(2 * x) - 1) / (2 * EXP(x)),
  SINH(x), -- (1 - EXP(-2 * x)) / (2 * EXP(-x)),
  COSH(x), -- (EXP(x) + EXP(-x)) / 2,
  COSH(x), -- (EXP(2 * x) + 1) / (2 * EXP(x)),
  COSH(x), -- (1 + EXP(-2 * x)) / (2 * EXP(-x)),
  TANH(x), -- SINH(x) / COSH(x),
  TANH(x), -- 1 / COTH(x),
  TANH(x), -- (EXP(x) - EXP(-x)) / (EXP(x) + EXP(-x)),
  TANH(x), -- (EXP(2 * x) - 1) / (EXP(2 * x) + 1),
  COTH(x), -- COSH(x) / SINH(x),
  COTH(x), -- 1 / TANH(x),
  COTH(x), -- (EXP(x) + EXP(-x)) / (EXP(x) - EXP(-x)),
  COTH(x)  -- (EXP(2 * x) + 1) / (EXP(2 * x) - 1)
FROM tab;
```

These transformations in particular may not appear to be complete, as many other intermediary steps may be required to achieve the above documented starting expressions. More work might be implemented in the future.

4.23.9.7.6. Idempotent function repetition

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

When SQL is complex or generated, there may be accidental repetitions of functions that do not have any effects on the result. Such repetitions can be removed by a single function application.

Using Settings.transformPatternsIdempotentFunctionRepetition, the following transformations can be achieved:

```
-- With Settings.transformPatternsIdempotentFunctionRepetition active, this:
SELECT
  LTRIM(LTRIM(x)),
  RTRIM(RTRIM(x)),
  TRIM(TRIM(x)),
  TRIM(LTRIM(x)),
  TRIM(RTRIM(x)),
  RTRIM(TRIM(x)),
  LTRIM(TRIM(x)),
  UPPER(UPPER(x)),
  LOWER(LOWER(x)),
  ABS(ABS(x)),
  SIGN(SIGN(x)),
  CEIL(CEIL(x)),
  FLOOR(FLOOR(x)),
  ROUND(ROUND(x)),
  TRUNC(TRUNC(x)),
  CAST(CAST(x AS type) AS type)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  LTRIM(x),       -- LTRIM(LTRIM(x))
  RTRIM(x),       -- RTRIM(RTRIM(x))
  TRIM(x),        -- TRIM(TRIM(x))
  TRIM(x),        -- TRIM(LTRIM(x))
  TRIM(x),        -- TRIM(RTRIM(x))
  TRIM(x),        -- RTRIM(TRIM(x))
  TRIM(x),        -- LTRIM(TRIM(x))
  UPPER(x),       -- UPPER(UPPER(x))
  LOWER(x),       -- LOWER(LOWER(x))
  ABS(x),         -- ABS(ABS(x))
  SIGN(x),        -- SIGN(SIGN(x))
  CEIL(x),        -- CEIL(CEIL(x))
  FLOOR(x),       -- FLOOR(FLOOR(x))
  ROUND(x),       -- ROUND(ROUND(x))
  TRUNC(x),       -- TRUNC(TRUNC(x))
  CAST(x AS type) -- CAST(CAST(x AS type) AS type)
FROM tab;
```

4.23.9.7.7. Inverse hyperbolic functions

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some dialects may not support all trigonometric functions.

Using Settings.transformPatternsInverseHyperbolicFunctions, the following transformations can be achieved:

```
-- With Settings.transformPatternsInverseHyperbolicFunctions active, this:
SELECT
  LN(x + SQRT(SQUARE(x) + 1)),
  LN(x + SQRT(SQUARE(x) - 1)),
  LN((1 + x) / (1 - x)) / 2),
  LN((x + 1) / (x - 1)) / 2)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  ASINH(x), -- LN(x + SQRT(SQUARE(x) + 1))
  ACOSH(x), -- LN(x + SQRT(SQUARE(x) - 1))
  ATANH(x), -- LN((1 + x) / (1 - x)) / 2)
  ACOTH(x)  -- LN((x + 1) / (x - 1)) / 2)
FROM tab;
```

4.23.9.7.8. Logarithmic functions

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some dialects may not support all trigonometric functions.

Using Settings.transformPatternsLogarithmicFunctions, the following transformations can be achieved:

```
-- With Settings.transformPatternsLogarithmicFunctions active, this:
SELECT
  LN(value) / LN(base),
  EXP(1),
  LOG(e, x),
  LOG(10, x),
  POWER(e, x)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  LOG(base, value), -- LN(value) / LN(base)
  e,                -- EXP(1)
  LN(x),            -- LOG(e, x)
  LOG10(x),         -- LOG(10, x)
  EXP(x)            -- POWER(e, x)
FROM tab;
```

4.23.9.7.9. Merge AND predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

An AND predicate combining comparison predicates that share the same operands can often be merged into a single comparison predicate.

Using Settings.transformPatternsMergeAndComparison, the following transformations can be achieved:

```
-- With Settings.transformPatternsMergeAndComparison active, this:
SELECT
  x = a AND x >= a,
  x = a AND x <= a,
  x > a AND x < a,
  x > a AND x != a,
  x < a AND x != a
  -- And many more
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x = a,  -- x = a AND x >= a,
  x = a,  -- x = a AND x <= a,
  x != x, -- x > a AND x < a,
  x > a,  -- x > a AND x != a,
  x < a   -- x < a AND x != a
  -- And many more
FROM tab;
```

4.23.9.7.10. Merge BIT_NOT with BIT_NAND

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The BIT_NOT function bitwise inverts its argument value. If the argument value is a BIT_NAND function, we can simply merge the two functions.

Using Settings.transformPatternsBitNotNand, the following transformations can be achieved:

```
-- With Settings.transformPatternsBitNotNand active, this:
SELECT
  ~(bitnand(a, b)),
  ~(bitand(a, b))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  bitand(a, b), -- ~(bitnand(a, b))
  bitnand(a, b) -- ~(bitand(a, b))
FROM tab;
```

4.23.9.7.11. Merge BIT_NOT with BIT_NOR

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The BIT_NOT function bitwise inverts its argument value. If the argument value is a BIT_NOR function, we can simply merge the two functions.

Using Settings.transformPatternsBitNotNor, the following transformations can be achieved:

```
-- With Settings.transformPatternsBitNotNor active, this:
SELECT
  ~(bitnor(a, b)),
  ~(bitor(a, b))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  bitor(a, b), -- ~(bitnor(a, b))
  bitnor(a, b) -- ~(bitor(a, b))
FROM tab;
```

4.23.9.7.12. Merge BIT_NOT with BIT_XNOR

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The BIT_NOT function bitwise inverts its argument value. If the argument value is a BIT_XNOR function, we can simply merge the two functions.

Using Settings.transformPatternsBitNotXNor, the following transformations can be achieved:

```
-- With Settings.transformPatternsBitNotXNor active, this:
SELECT
  ~(bitxnor(a, b)),
  ~(bitxor(a, b))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  bitxor(a, b), -- ~(bitxnor(a, b))
  bitxnor(a, b) -- ~(bitxor(a, b))
FROM tab;
```

4.23.9.7.13. Merge IN predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

An AND predicate combining IN predicates that share the same operands can often be merged into a single IN predicate comparing the left operand with the intersection of the right IN lists.

Using Settings.transformPatternsMergeInPredicates, the following transformations can be achieved:

```
-- With Settings.transformPatternsMergeInPredicates active, this:
SELECT * FROM tab WHERE x IN (a, b, c) AND x IN (b, c, d);

-- ... is transformed into the equivalent expression:
SELECT * FROM tab WHERE x IN (b, c);
```

4.23.9.7.14. Merge NOT with comparison predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The NOT predicate inverts its argument predicate. If the argument predicate is a comparison predicate, we can simply merge the two operators to invert the argument into the inverse comparison predicate.

Using Settings.transformPatternsNotComparison, the following transformations can be achieved:

```
-- With Settings.transformPatternsNotComparison active, this:
SELECT
  NOT (a = b),
  NOT (a != b),
  NOT (a > b),
  NOT (a >= b),
  NOT (a < b),
  NOT (a <= b)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  a != b, -- NOT (a = b)
  a = b,  -- NOT (a != b)
  a <= b, -- NOT (a > b)
  a < b,  -- NOT (a >= b)
  a >= b, -- NOT (a < b)
  a > b   -- NOT (a <= b)
FROM tab;
```

4.23.9.7.15. Merge NOT with DISTINCT predicate

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The NOT predicate inverts its argument predicate. If the argument predicate is a DISTINCT predicate, we can simply merge the two operators to invert the argument into the inverse DISTINCT predicate.

Using Settings.transformPatternsNotNotDistinct, the following transformations can be achieved:

```
-- With Settings.transformPatternsNotNotDistinct active, this:
SELECT
  NOT (a IS NOT DISTINCT FROM b),
  NOT (a IS DISTINCT FROM b)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  a IS DISTINCT FROM b,    -- NOT (a IS NOT DISTINCT FROM b)
  a IS NOT DISTINCT FROM b -- NOT (a IS DISTINCT FROM b)
FROM tab;
```

4.23.9.7.16. Merge OR predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

An OR predicate combining comparison predicates that share the same operands can often be merged into a single comparison predicate.

Using Settings.transformPatternsMergeOrComparison, the following transformations can be achieved:

```
-- With Settings.transformPatternsMergeOrComparison active, this:
SELECT
  x = a OR x > a,
  x = a OR x < a,
  x > a OR x < a,
  x > a OR x != a,
  x < a OR x != a
  -- And many more
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x >= a, -- x = a OR x > a
  x <= a, -- x = a OR x < a
  x != a, -- x > a OR x < a
  x != a, -- x > a OR x != a
  x != a  -- x < a OR x != a
  -- And many more
FROM tab;
```

4.23.9.7.17. Merge range predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

An AND predicate combining range predicates that share the same operands can often be merged into a single BETWEEN predicate predicate comparing the left operand with the two right operands.

Using Settings.transformPatternsMergeRangePredicates, the following transformations can be achieved:

```
-- With Settings.transformPatternsMergeRangePredicates active, this:
SELECT * FROM tab WHERE x >= a AND x <= b;

-- ... is transformed into the equivalent expression:
SELECT * FROM tab WHERE x BETWEEN a AND b;
```

4.23.9.7.18. Normalise associative operations

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Associative operations are operations whose chaining can be done as (a op b) op c or a op (b op c) without changing the result. Examples of such operations are:

- +
- *
- AND
- OR

In order to simplify all of these pattern replacements, we normalise associative ops to always flatten tree structures into lists.

Using Settings.transformPatternsNormaliseAssociativeOps, the following transformations can be achieved:

```
-- With Settings.transformPatternsNormaliseAssociativeOps active, this:
SELECT
  (a + b) + (c + d),
  (a * b) * (c * d),
  (a AND b) AND (c AND d),
  (a OR b) OR (c OR d)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  ((a + b) + c) + d,       -- (a + b) + (c + d)
  ((a + b) + c) + d,       -- (a * b) * (c * d)
  ((a AND b) AND c) AND d, -- (a AND b) AND (c AND d)
  ((a OR b) OR c) OR d     -- (a OR b) OR (c OR d)
FROM tab;
```

Note that the unnecesary parentheses may not be generated in either case, but the in-memory data structure still looks as though the parentheses were there.

4.23.9.7.19. Normalise fields compared to values

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

When comparing column expressions with values, the various pattern replacements profit from a simplification where we normalise the order of operands to have the value on the right side of the operator.

Using Settings.transformPatternsNormaliseFieldCompareValue, the following transformations can be achieved:

```
-- With Settings.transformPatternsNormaliseFieldCompareValue active, this:
SELECT * FROM tab WHERE 1 = a;

-- ... is transformed into the equivalent expression:
SELECT * FROM tab WHERE a = 1;
```

4.23.9.7.20. Normalise IN list with single element to comparison

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

When comparing an IN predicate with a single value, then it may be beneficial for other pattern replacement if that expression is normalised to a comparison predicate.

Using Settings.transformPatternsNormaliseInListSingleElementToComparison, the following transformations can be achieved:

```
-- With Settings.transformPatternsNormaliseInListSingleElementToComparison active, this:
SELECT
  x IN (a),
  x NOT IN (a)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x = a, -- x IN (a),
  x != a -- x NOT IN (a)
FROM tab;
```

4.23.9.7.21. OR to IN

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

A concatenation of OR predicates combining equalities that always use the same operand on one side can be replaced by a more concise, but equivalent IN predicate.

Using Settings.transformPatternsOrEqToIn, the following transformations can be achieved:

```
-- With Settings.transformPatternsOrEqToIn active, this:
SELECT * FROM tab WHERE x = 1 OR x = 2 OR x = 3;

-- ... is transformed into the equivalent expression:
SELECT * FROM tab WHERE x IN (1, 2, 3);
```

See also AND to NOT IN

4.23.9.7.22. Repeated bitwise negation

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The bitwise negation unary operator reverses itself, when called repeatedly, meaning that redundant BIT_NOT() operators can be removed.

Using Settings.transformPatternsBitNotBitNot, the following transformations can be achieved:

```
-- With Settings.transformPatternsNegNeg active, this:
SELECT
  ~(~(x)),
  ~(~(~(x)))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x,  -- ~(~(x))
  ~x, -- ~(~(~(x)))
FROM tab;
```

4.23.9.7.23. Repeated logical negation

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The negation unary operator reverses itself, when called repeatedly, meaning that redundant NEG() operators can be removed.

Using Settings.transformPatternsNegNeg, the following transformations can be achieved:

```
-- With Settings.transformPatternsNegNeg active, this:
SELECT
  -(-(x)),
  -(-(-(x)))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x,  -- -(-(x))
  -x, -- -(-(-(x)))
FROM tab;
```

4.23.9.7.24. Repeated NOT

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The NOT unary operator reverses itself, when called repeatedly, meaning that redundant NOT operators can be removed.

Using Settings.transformPatternsNotNot, the following transformations can be achieved:

```
-- With Settings.transformPatternsNotNot active, this:
SELECT
  NOT (NOT (x = 1)),
  NOT (NOT (NOT (x = 1)))
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  x = 1,       -- NOT (NOT (x = 1))
  NOT (x = 1), -- NOT (NOT (NOT (x = 1)))
FROM tab;
```

4.23.9.7.25. Trigonometric functions

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some dialects may not support all trigonometric functions.

Using Settings.transformPatternsTrigonometricFunctions, the following transformations can be achieved:

```
-- With Settings.transformPatternsTrigonometricFunctions active, this:
SELECT
  SIN(x) / COS(x),
  COS(x) / SIN(x),
  1 / TAN(x),
  1 / COT(x)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  TAN(x), -- SIN(x) / COS(x)
  COT(x), -- COS(x) / SIN(x)
  COT(x), -- 1 / TAN(x)
  TAN(x)  -- 1 / COT(x)
FROM tab;
```

4.23.9.7.26. Trim

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Some dialects may not have supported TRIM() natively, so users emulated it by combining RTRIM() and LTRIM() manually.

Using Settings.transformPatternsTrim, the following transformations can be achieved:

```
-- With Settings.transformPatternsTrim active, this:
SELECT RTRIM(LTRIM(col)), LTRIM(RTRIM(col)) FROM tab;

-- ... is transformed into the equivalent expression:
SELECT TRIM(col), TRIM(col) FROM tab;
```

4.23.9.7.27. Trivial case abbreviations

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Case abbreviations like NULLIF() can be trivial, in case of which the function call can be removed.

Using Settings.transformPatternsTrivialCaseAbbreviation, the following transformations can be achieved:

```
-- With Settings.transformPatternsTrivialCaseAbbreviation active, this:
SELECT
  NVL(NULL, a),
  NULLIF(a, a),
  NULLIF(NULL, a),
  NULLIF(a, NULL)
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  a,    -- NVL(NULL, a),
  NULL, -- NULLIF(a, a),
  NULL, -- NULLIF(NULL, a),
  a     -- NULLIF(a, NULL)
FROM tab;
```

4.23.9.7.28. Trivial predicates

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

predicates can be trivial, in case of which the expression can be removed and replaced by a TRUE or FALSE condition.

Using Settings.transformPatternsTrivialPredicates, the following transformations can be achieved:

```
-- With Settings.transformPatternsTrivialPredicates active, this:
SELECT
  a IS NOT DISTINCT FROM a,
  a IS DISTINCT FROM a,
  a >= a,
  a <= a,
  a > a,
  a < a,
  const IS NOT NULL,
  const IS NULL,
  TRUE AND FALSE,
  TRUE OR FALSE,
  NOT TRUE,
  NOT FALSE
  -- and many more
FROM tab;

-- ... is transformed into the equivalent expression:
SELECT
  TRUE,   -- a IS NOT DISTINCT FROM a
  FALSE,  -- a IS DISTINCT FROM a
  a = a,  -- a >= a
  a = a,  -- a <= a
  a != a, -- a > a
  a != a, -- a < a
  TRUE,   -- const IS NOT NULL
  FALSE,  -- const IS NULL
  FALSE,  -- TRUE AND FALSE
  TRUE,   -- TRUE OR FALSE
  FALSE,  -- NOT TRUE
  TRUE    -- NOT FALSE
  -- and many more
FROM tab;
```

4.23.10. 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.

4.23.10.1. 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.

4.24. 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.

4.25. SQL building in Kotlin

jOOQ-kotlin is a maven module used for leveraging some advanced Kotlin features for those users that wish to use jOOQ with Kotlin.

You can integrate it using the following dependency:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-kotlin</artifactId>
  <version>3.17.8</version>
</dependency>
```

The following sections highlight a few useful extensions from that module.

Note that even without that module, you can take advantage of a lot of kotlin language features using vanilla jOOQ API, which has been designed with a strong kotlin focus, in mind. For more details, see this blog post.

4.25.1. Kotlin MULTISET Collectors

jOOQ's runtime API implements a few useful Collectors that can be used to fetch data into a map, for example:

```
public class Records {

    // [...]

    public static final <K, V, R extends Record2<K, V>> Collector<R, ?, Map<K, V>> intoMap() {
        return intoMap(Record2::value1, Record2::value2);
    }

    // [...]

}
```

These can be used with MULTISET or MULTISET_AGG as follows:

```
val map: Field<Map<Int, String>> =
multisetAgg(LANGUAGE.ID, LANGUAGE.CD).convertFrom { r -> r.collect(Records.intoMap()) }
```

The Collector leverages the query's type safety, such that it works only for queries projecting exactly 2 columns. Using the kotlin extensions module, a few useful extension functions are made available as follows:

```
package org.jooq.kotlin

inline fun <reified E> Field<Result<Record1<E>>>.intoArray(): Field<Array<E>> = collecting(Records.intoArray(E::class.java))
fun <K, V, R : Record2<K, V>> Field<Result<R>>.intoGroups(): Field<Map<K, List<V>>> = collecting(Records.intoGroups())
fun <E, R : Record1<E>> Field<Result<R>>.intoList(): Field<List<E>> = collecting(Records.intoList())
fun <K, V> Field<Result<Record2<K, V>>>.intoMap(): Field<Map<K, V>> = collecting(Records.intoMap())
fun <E, R : Record1<E>> Field<Result<R>>.intoSet(): Field<Set<E>> = collecting(Records.intoSet())

// [... and more]
```

This allows for the leaner version below:

```
val map: Field<Map<Int, String>> =
multisetAgg(LANGUAGE.ID, LANGUAGE.CD).intoMap()
```

4.25.2. Kotlin BOOLEAN value expressions

A BOOLEAN column (Field<Boolean>) and a conditional expression (Condition) are not really too different, especially when the dialect supports standard SQL BOOLEAN types.

In jOOQ, the two things can often be used exchangeably, but there are some exceptions, mainly when BOOLEAN operators should be used with (Field<Boolean>):

```
condition(BOOLEAN_COLUMN1).and(BOOLEAN_COLUMN2)
```

At least the left side of the expression has to be wrapped with DSL.condition(Field), which is cumbersome.

Using the kotlin extensions module, these operators are also made available on Field<Boolean> directly:

```
package org.jooq.kotlin

fun Field<Boolean>.and(other: Condition): Condition = condition(this).and(other)
fun Field<Boolean>.or(other: Condition): Condition = condition(this).or(other)
fun Field<Boolean>.not(): Condition = condition(this).not()

// [... and more]
```

This allows for the leaner version below:

```
BOOLEAN_COLUMN1.and(BOOLEAN_COLUMN2)
```

4.25.3. Kotlin ARRAY access

Array elements can be accessed using the ARRAY_GET function, which translates to the ARRAY subscript syntax:

```
SELECT (ARRAY[1, 2])[1]
```

```
create.select(arrayGet(array(1, 2), 1)).fetch();
```

Using the kotlin extensions module, these operators are also made available on Field<Array<T>> directly:

```
package org.jooq.kotlin

operator fun <T> Field<Array<T>?>.get(index: Int) = arrayGet(this, index)
operator fun <T> Field<Array<T>?>.get(index: Field<Int>) = arrayGet(this, index)

// [... and more]
```

This allows for the leaner version below:

```
create.select(array(1, 2)[1]).fetch();
```

4.25.4. Kotlin coroutine support

In kotlin, coroutines and suspending functions are a popular way to implement asynchronous, flow style logic. Starting from jOOQ 3.17, the jooq-kotlin-coroutines extension module allows for bridging between the reactive streams API and the coroutine APIs. For this, simply add the following dependencies:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-kotlin</artifactId>
  <version>3.17.8</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-kotlin-coroutines</artifactId>
  <version>3.17.8</version>
</dependency>
```

And now, you can use jOOQ in a kotlin coroutine style:

```
suspend fun findActor(id: Long): ActorRecord? {
    return create
        .selectFrom(ACTOR)
        .where(ACTOR.ACTOR_ID.eq(id))

        // Turn any reactive streams Publisher<T> into a suspension result using the
        // kotlinx-coroutines-reactive extensions
        .awaitFirstOrNull()
}
```

And wrap your transactional code in the org.jooq.kotlin.coroutines.transactionCoroutine() extension function:

```
suspend fun insertActorTransaction(): ActorRecord {
    return ctx.transactionCoroutine(::insertActor)
}

suspend fun insertActor(c: Configuration): ActorRecord = c.dsl()
    .insertInto(ACTOR)
    .columns(ACTOR.ACTOR_ID, ACTOR.FIRST_NAME, ACTOR.LAST_NAME)
    .values(201L, "A", "A")
    .returning()
    .awaitFirst()
```

*(!) While jOOQ implements the **reactive streams API** on top of JDBC by default (in a blocking way), we recommend you consider switching to R2DBC driver usage, if you want your coroutines to be truly non-blocking.*

4.26. 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:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>

  <!-- Scala 2.11 and 2.12 are also supported -->
  <artifactId>jooq-scala_2.13</artifactId>
  <version>3.17.8</version>
</dependency>
```

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!

5. 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.

5.1. 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.

5.2. 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();
```

5.3. 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 handler callbacks: This is an entirely different fetching paradigm. With Java 8's lambda expressions, this will become even more powerful.
- 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.

5.3.1. 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();
```

5.3.2. 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();
}
```

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.

5.3.3. 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.

5.3.4. RecordHandler

In a more functional operating mode, you might want to write callbacks that receive 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.RecordHandler classes and plug them into jOOQ's org.jooq.ResultQuery:

```
// Write callbacks to receive records from select statements
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch()
      .into(new RecordHandler<BookRecord>() {
          @Override
          public void next(BookRecord book) {
              Util.doThingsWithBook(book);
          }
      });

// Or more concisely
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetchInto(new RecordHandler<BookRecord>() {...});

// Or even more concisely with Java 8's lambda expressions:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetchInto(book -> { Util.doThingsWithBook(book); }; );
```

See also the manual's section about the RecordMapper, which provides similar features

5.3.5. 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();
          }
      });
```

An alternative utility is offered by org.jooq.Records, which is very useful when working with type safe Record[N] types and constructor references:

```
// Use Java 16 record types as simple DTOs
record Book (int id, String title) {}

List<Book> ids =
create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(BOOK.ID)
      .fetch(Records.mapping(Book::new));
```

Your custom RecordMapper types can be used automatically through jOOQ's POJO mapping APIs, by injecting a RecordMapperProvider into your Configuration.

See also the manual's section about the RecordHandler, which provides similar features

5.3.6. 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.

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);
```

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.

5.3.7. POJOs with RecordMappers

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)

5.3.8. Ad-hoc Converter

Sometimes, you want to attach an ad-hoc converter to some column, just for a single query or a few local queries. This is possible through a variety of ways. The most convenient one is to call Field.convert() and related methods. Assuming you have a converter like this to convert your SQL VARCHAR columns to Language:

```
enum Language { de, en, fr, it, pt; }
Converter<String, Language> converter = new EnumConverter<>(String.class, Language.class);
```

Now, instead of attaching them to your generated code, you may keep your generated code as Field<String>, and convert them on the fly to Field<Language> for the purpose of a single query:

```
Result<Record2<Integer, Language>> result =
create.select(LANGUAGE.ID, LANGUAGE.CD.convert(converter))
      .from(LANGUAGE)
      .fetch();
```

Alternatively, if you don't even have a Converter instance ready, you can attach conversion logic to fields like this:

```
Result<Record2<Integer, Language>> result =
create.select(LANGUAGE.ID, LANGUAGE.CD.convert(Language.class, Language::valueOf, Language::name))
      .from(LANGUAGE)
      .fetch();
```

Or, since in this case, conversions only happen from the database type (the T type) to the user type (the U type), you can omit the inverse conversion function using Field.convertFrom() ("from" as in "reading ***from*** the database"):

```
Result<Record2<Integer, Language>> result =
create.select(LANGUAGE.ID, LANGUAGE.CD.convertFrom(Language.class, Language::valueOf))
      .from(LANGUAGE)
      .fetch();
```

The inverse is possible too, e.g. when you only need to convert from the user type (the U type) to the database type (the T type) using Field.convertTo() ("to" as in "writing ***to*** the database"):

```
Result<Record2<Integer, Language>> result =
create.insertInto(LANGUAGE)
      .columns(LANGUAGE.ID, LANGUAGE.CD.convertTo(Language.class, Language::name))
      .values(5, Language.it)
      .execute();
```

Using ad hoc converters on nested collections

Ad-hoc converters are extremely powerful when used on nested collections, e.g. those constructed with the MULTISET value constructor

```
// Structurally typed result
Result<Record4<
    String,          // AUTHOR.FIRST_NAME
    String,          // AUTHOR.LAST_NAME
    Result<Record2<
        String,      // LANGUAGE.CD
        String       // LANGUAGE.DESCRIPTION
    >>,              // books
    Result<Record1<
        String       // BOOK_TO_BOOK_STORE.BOOK_STORE_NAME
    >>               // book_stores
>> result = create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,
          multiset(
              selectDistinct(
                  BOOK.language().CD,
                  BOOK.language().DESCRIPTION)
              .from(BOOK)
              .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          ).as("books"),
          multiset(
              selectDistinct(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
              .from(BOOK_TO_BOOK_STORE)
              .where(BOOK_TO_BOOK_STORE.tBook().AUTHOR_ID
                  .eq(AUTHOR.ID))
          ).as("book_stores"))
       .from(AUTHOR)
       .orderBy(AUTHOR.ID)
       .fetch();
```

For details about MULTISET, refer to the section about the MULTISET value constructor. Now, instead of the above structurally typed result, it may be desirable to map things into Java 16 record types instead, or some other form of DTO:

```
record Book(String cd, String description) {}
record BookStore(String name) {}
record Author(String firstName, String lastName, List<Book> books, List<BookStore> bookStores) {}
```

And now, using the static-import friendly org.jooq.Records utility:

```
// Nominally typed result, all type checked!
List<Author> result = create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,

          // This is now a Field<List<Book>>
          multiset(
              selectDistinct(
                  BOOK.language().CD,
                  BOOK.language().DESCRIPTION)
              .from(BOOK)
              .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          ).as("books").convertFrom(r -> r.map(Records.mapping(Book::new))),

          // This is now a Field<List<BookStore>>
          multiset(
              selectDistinct(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
              .from(BOOK_TO_BOOK_STORE)
              .where(BOOK_TO_BOOK_STORE.tBook().AUTHOR_ID.eq(AUTHOR.ID))
          ).as("book_stores").convertFrom(r -> r.map(Records.mapping(BookStore::new))))
       .from(AUTHOR)
       .orderBy(AUTHOR.ID)
       .fetch(Records.mapping(Author::new));
```

Try adding or removing a column from the projections, or adding or removing an attribute from your records, and the query no longer type-checks!

Of course, the usual reflective RecordMapper API can still be used just the same with these ad-hoc converters.

```
// Nominally typed result, but not strongly type checked
List<Author> result = create.select(
          AUTHOR.FIRST_NAME,
          AUTHOR.LAST_NAME,

          // This is now a Field<List<Book>>
          multiset(
              selectDistinct(
                  BOOK.language().CD,
                  BOOK.language().DESCRIPTION)
              .from(BOOK)
              .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          ).as("books").convertFrom(r -> r.into(Book.class)),

          // This is now a Field<List<BookStore>>
          multiset(
              selectDistinct(BOOK_TO_BOOK_STORE.BOOK_STORE_NAME)
              .from(BOOK_TO_BOOK_STORE)
              .where(BOOK_TO_BOOK_STORE.tBook().AUTHOR_ID.eq(AUTHOR.ID))
          ).as("book_stores").convertFrom(r -> r.into(BookStore.class)))
      .from(AUTHOR)
      .orderBy(AUTHOR.ID)
      .fetchInto(Author.class);
```

5.3.9. ConverterProvider

jOOQ supports some useful default data type conversion between common JDBC data types in org.jooq.tools.Convert. These conversions include, for example:

```
int i = Convert.convert("1", int.class); // Yields 1
Date d = Convert.convert("2000-01-01", Date.class); // Yields Date.valueOf("2000-01-01")
```

These auto-conversions are made available throughout the jOOQ API, for example when writing

```
Record record = create.fetchSingle(field("current_date"));
LocalDate d1 = record.get(0, LocalDate.class);
LocalDate d2 = create.fetchSingle(field("current_date"), LocalDate.class);
```

These auto-conversions are also applied implicitly when mapping POJOs as the previous sections have shown:

```
class POJO {
    LocalDate date;
}

POJO pojo = create.fetchSingle(field("current_date").as("date")).into(POJO.class);
```

Overriding the DefaultConverterProvider

Sometimes, it may be desireable to override the default behaviour provided by the org.jooq.impl.DefaultConverterProvider via a custom org.jooq.ConverterProvider. For example, assume you have an object like this:

```
public class Name {
    public String firstName;
    public String lastName;
}

public class Book {
    public String title;
}

public class Author {
    public Name name;
    public List<Book> books;
}
```

Now, imagine projecting some JSON functions or XML functions. You would probably want them to be mapped hierarchically to your above data structure:

```
List<Author> authors = create()
  .select(jsonObject(
     key("name").value(jsonObject(
       key("firstName").value(AUTHOR.FIRST_NAME),
       key("lastName").value(AUTHOR.LAST_NAME)
     )),
     key("books").value(jsonArrayAgg(
       jsonObject("title", BOOK.TITLE)
     ))
  ))
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
  .groupBy(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
  .orderBy(AUTHOR.ID)
  .fetchInto(Author.class);
```

If jOOQ finds Jackson or Gson on your classpath, the above works out of the box. If you want to override jOOQ's out of the box binding, you can easily provide your own by implementing a org.jooq.ConverterProvider as follows, e.g. using the Jackson library:

```
class JSONConverterProvider implements ConverterProvider {
    final ConverterProvider delegate = new DefaultConverterProvider();
    final ObjectMapper mapper = new ObjectMapper();

    @Override
    public <T, U> Converter<T, U> provide(Class<T> tType, Class<U> uType) {

        // Our specialised implementation can convert from JSON (optionally, add JSONB, too)
        if (tType == JSON.class) {
            return Converter.ofNullable(tType, uType,
                t -> {
                    try {
                        return mapper.readValue(((JSON) t).data(), uType);
                    }
                    catch (Exception e) {
                        throw new DataTypeException("JSON mapping error", e);
                    }
                },
                u -> {
                    try {
                        return (T) JSON.valueOf(mapper.writeValueAsString(u));
                    }
                    catch (Exception e) {
                        throw new DataTypeException("JSON mapping error", e);
                    }
                }
            );
        }

        // Delegate all other type pairs to jOOQ's default
        else
            return delegate.provide(tType, uType);
    }
}
```

Note: For best results with Jackson and kotlin, please also put the jackson-module-kotlin on the classpath.

5.3.10. 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.
- RecordHandler callbacks: You can use your own org.jooq.RecordHandler callbacks to receive lazily fetched records.
- 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.

5.3.11. 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
);
```

5.3.12. 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 straight-forward 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.

5.3.13. 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);
```

5.3.14. 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();
```

Out of the box, all jOOQ provided publishers will block on the underlying JDBC connection, but if you provide jOOQ with a io.r2dbc.spi.Connection or io.r2dbc.spi.ConnectionFactory, then the publishers will execute queries in a non-blocking fashion on an R2DBC driver.

5.3.15. 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.

5.3.16. 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);
```

5.3.17. Custom data type conversion

Apart from a few extra features (user-defined types), jOOQ only supports basic types as supported by the JDBC API. In your application, you may choose to transform these data types into your own ones, without writing too much boiler-plate code. This can be done using jOOQ's org.jooq.Converter types. A converter essentially allows for two-way conversion between two Java data types <T> and <U>. By convention, the <T> type corresponds to the type in your database whereas the <U> type corresponds to your own user type. The Converter API is given 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. Some examples have been illustrated in the manual's section about fetching.

A Converter for GregorianCalendar

Here is a some more elaborate example involving a Converter for java.util.GregorianCalendar:

```
// You may prefer Java Calendars over JDBC Timestamps
public class CalendarConverter implements Converter<Timestamp, GregorianCalendar> {

    @Override
    public GregorianCalendar from(Timestamp databaseObject) {
        GregorianCalendar calendar = (GregorianCalendar) Calendar.getInstance();
        calendar.setTimeInMillis(databaseObject.getTime());
        return calendar;
    }

    @Override
    public Timestamp to(GregorianCalendar userObject) {
        return new Timestamp(userObject.getTime().getTime());
    }

    @Override
    public Class<Timestamp> fromType() {
        return Timestamp.class;
    }

    @Override
    public Class<GregorianCalendar> toType() {
        return GregorianCalendar.class;
    }
}

// Now you can fetch calendar values from jOOQ's API:
List<GregorianCalendar> dates1 = create.selectFrom(BOOK).fetch().getValues(BOOK.PUBLISHING_DATE, new CalendarConverter());
List<GregorianCalendar> dates2 = create.selectFrom(BOOK).fetch(BOOK.PUBLISHING_DATE, new CalendarConverter());
```

Enum Converters

jOOQ ships with a built-in default org.jooq.impl.EnumConverter, that you can use to map VARCHAR values to enum literals or NUMBER values to enum ordinals (both modes are supported). Let's say, you want to map a YES / NO / MAYBE column to a custom Enum:

```
// Define your Enum
public enum YNM {
    YES, NO, MAYBE
}

// Define your converter
public class YNMConverter extends EnumConverter<String, YNM> {
    public YNMConverter() {
        super(String.class, YNM.class);
    }
}

// And you're all set for converting records to your custom Enum:
for (BookRecord book : create.selectFrom(BOOK).fetch()) {
    switch (book.getValue(BOOK.I_LIKE, new YNMConverter())) {
        case YES:    System.out.println("I like this book             : " + book.getTitle()); break;
        case NO:     System.out.println("I didn't like this book      : " + book.getTitle()); break;
        case MAYBE:  System.out.println("I'm not sure about this book : " + book.getTitle()); break;
    }
}
```

If you're using forcedTypes in your code generation configuration, you can configure the application of an EnumConverter by adding <enumConverter>true</enumConverter> to your <forcedType/> configuration.

Using Converters in generated source code

jOOQ also allows for generated source code to reference your own custom converters, in order to permanently replace a table column's <T> type by your own, custom <U> type. See the manual's section about custom data types for details.

5.3.18. Interning data

SQL result tables are not optimal in terms of used memory as they are not designed to represent hierarchical data as produced by JOIN operations. Specifically, FOREIGN KEY values may repeat themselves unnecessarily:

```
+----+-----------+--------------+
| ID | AUTHOR_ID | TITLE        |
+----+-----------+--------------+
|  1 |         1 | 1984         |
|  2 |         1 | Animal Farm  |
|  3 |         2 | O Alquimista |
|  4 |         2 | Brida        |
+----+-----------+--------------+
```

Now, if you have millions of records with only few distinct values for AUTHOR_ID, you may not want to hold references to distinct (but equal) java.lang.Integer objects. This is specifically true for IDs of type java.util.UUID or string representations thereof. jOOQ allows you to "intern" those values:

```
// Interning data after fetching
Result<?> r1 = create.select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
                     .from(BOOK)
                     .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                     .fetch()
                     .intern(BOOK.AUTHOR_ID);

// Interning data while fetching
Result<?> r1 = create.select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
                     .from(BOOK)
                     .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                     .intern(BOOK.AUTHOR_ID)
                     .fetch();
```

You can specify as many fields as you want for interning. The above has the following effect:

- If the interned Field is of type java.lang.String, then String.intern() is called upon each string
- If the interned Field is of any other type, then the call is ignored

Future versions of jOOQ will implement interning of data for non-String data types by collecting values in java.util.Set, removing duplicate instances.

Note, that jOOQ will not use interned data for identity comparisons: string1 == string2. Interning is used only to reduce the memory footprint of org.jooq.Result objects.

5.4. 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

5.5. 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.

5.6. 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.

5.7. 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.

5.8. 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.

5.9. 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.

5.9.1. 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.

5.9.2. 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.

5.10. 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

5.10.1. 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

5.10.2. 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}");
```

5.10.3. 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

5.10.4. Exporting HTML

```
// Fetch books and format them as HTML
String html = create.selectFrom(BOOK).fetch().formatHTML();
```

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>
```

5.10.5. 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

5.10.6. 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.

5.10.7. FormattingProvider

Most types of export formats discussed in the previous sections have an associated formatting configuration object, which can be passed on a per-formatting call, including:

- org.jooq.ChartFormat for formatting the chart export
- org.jooq.CSVFormat for formatting the CSV export
- org.jooq.JSONFormat for formatting the JSON export
- org.jooq.TXTFormat for formatting the text export
- org.jooq.XMLFormat for formatting the XML export

For example, when you want to specify the JSON record layout, as well as remove the header information you can write code like this:

```
System.out.println("Using the object layout");
System.out.println(result.formatJSON(new JSONFormat().header(false).recordFormat(JSONFormat.RecordFormat.OBJECT)));

System.out.println("Using the array layout");
System.out.println(result.formatJSON(new JSONFormat().header(false).recordFormat(JSONFormat.RecordFormat.ARRAY)));
```

The output is the following (see also JSON export for details):

```
Using the object layout
[{"col1": "string1", "col2": 1}, {"col1": "string2", "col2": 2}]

Using the array layout
[["string1", 1],["string2", 2]]
```

Instead of passing this JSONFormat object to ***every*** formatting call, you can also register the org.jooq.FormattingProvider SPI to your Configuration in order to specify the relevant formats that should be applied ***by default*** (in the absence of an explicit formatting configuration object).

For example, in order to turn off JSON headers and specify the array layout everywhere, do this:

```
JSONFormat format = new JSONFormat().header(false).recordFormat(JSONFormat.RecordFormat.ARRAY);
Configuration configuration = create.configuration().derive(FormattingProvider

    // This specifies the format to be used for Record.formatJSON() calls
    .onJsonFormatForRecords(() -> format)

    // This specifies the format to be used for Result.formatJSON() calls
    .onJsonFormatForResults(() -> format));

System.out.println(
    configuration.dsl()
        .select(BOOK.ID, BOOK.TITLE)
        .from(BOOK)
        .fetch()
        .formatJSON() // No need to pass the format here anymore
);
```

5.11. 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

5.11.1. 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

5.11.2. Import options

Prior to specifing data sources, data source independent loading options can be specified. These include throttling, duplicate handling, and error handling:

5.11.2.1. 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();
```

5.11.2.2. 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();
```

5.11.2.3. 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();
```

5.11.3. Import data sources

Different types of data sources are supported by jOOQ in the same formats as the export API. These include:

5.11.3.1. 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();

// Match target fields with source fields by "corresponding" name.
// This assumes that CSV row 1 is a header row containing the source field names.
create.loadInto(BOOK)
      .loadCSV(inputstream, encoding)
      .fieldsCorresponding()
      .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();
```

5.11.3.2. 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();

// Match target fields with source fields by "corresponding" name.
// This assumes that JSON content contains header information as exported by jOOQ
create.loadInto(BOOK)
      .loadJSON(inputstream, encoding)
      .fieldsCorresponding()
      .execute();
```

No other, JSON-specific options are currently available.

5.11.3.3. 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();

// Match target fields with source fields by "corresponding" name.
create.loadInto(BOOK)
      .loadRecords(result)
      .fieldsCorresponding()
      .execute();
```

No other, Record-specific options are currently available.

5.11.3.4. 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.

5.11.3.5. Importing XML

This is not yet supported

5.11.4. 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();
```

5.11.5. 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();
```

5.12. 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.

5.12.1. 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();
  }
}
```

5.12.2. 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.

5.12.3. 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.

5.12.4. 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.

5.12.5. 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.

5.12.6. 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 in the mean time
- 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.

5.12.7. 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.

5.12.8. 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 (see CREATE TRIGGER) if records should be changed, or a org.jooq.VisitListener if the SQL query should be changed independently of data.

5.13. 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);
```

5.14. 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.

Blocking (JDBC)

```
create.transaction((Configuration trx) -> {
    AuthorRecord author =
    trx.dsl()
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    trx.dsl()
       .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
       .values(author.getId(), "1984")
       .values(author.getId(), "Animal Farm")
       .execute();

    // Implicit commit executed here
});
```

Non blocking (R2DBC)

```
// Examples use reactor, but you can use any other RS API, too
create.transactionPublisher((Configuration trx) -> Mono
    .from(trx.dsl()
        .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
        .values("George", "Orwell")
        .returning())
    .flatMap((AuthorRecord author) -> trx.dsl()
        .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
        .values(author.getId(), "1984")
        .values(author.getId(), "Animal Farm"))

    // Implicit commit executed here
});
```

Note how the lambda expression receives a new, ***derived*** configuration which should be used within the local scope:

Blocking (JDBC)

```
create.transaction((Configuration trx) -> {

    // Wrap configuration in a new DSLContext:
    trx.dsl().insertInto(...);
    trx.dsl().insertInto(...);

    // Or, reuse the new DSLContext within the transaction scope:
    DSLContext ctx = trx.dsl();
    ctx.insertInto(...);
    ctx.insertInto(...);

    // ... but avoid using the scope from outside the transaction:
    create.insertInto(...);
    create.insertInto(...);
});
```

Non blocking (R2DBC)

```
create.transactionPublisher((Configuration trx) -> {

    // Wrap configuration in a new DSLContext:
    trx.dsl().insertInto(...);
    trx.dsl().insertInto(...);

    // Or, reuse the new DSLContext within the transaction scope:
    DSLContext ctx = trx.dsl();
    ctx.insertInto(...);
    ctx.insertInto(...);

    // ... but avoid using the scope from outside the transaction:
    create.insertInto(...);
    create.insertInto(...);
});
```

Rollbacks

Any uncaught checked or unchecked exception thrown from your transactional code (blocking), or unhandled error in a reactive stream (non-blocking) will rollback the transaction to the beginning of the transactional scope. This behaviour will allow for nesting transactions, if your configured org.jooq.TransactionProvider supports nesting of transactions. An example can be seen here:

Blocking (JDBC)

```
create.transaction((Configuration outer) -> {
    final AuthorRecord author =
    outer.dsl()
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    // Implicit savepoint created here
    try {
        outer.dsl()
           .transaction((Configuration nested) -> {
               nested.dsl()
                  .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
});
```

Non blocking (R2DBC)

```
// Examples use reactor, but you can use any other RS API, too
create.transactionPublisher((Configuration outer) -> Mono
    .from(outer.dsl()
        .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
        .values("George", "Orwell")
        .returning())
    .flatMap((AuthorRecord author) -> outer.dsl()

        // Implicit savepoint created here
        .transactionPublisher((Configuration nested) -> Mono
            .from(nested.dsl()
                .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
                .values(author.getId(), "1984")
                .values(author.getId(), "Animal Farm"))

            // Rolls back the nested transaction
            .<Integer>flatMap(i -> {
                throw new RuntimeException("Oops");
            }))

        // Implicit savepoint is discarded, but no commit is issued yet.
        .onErrorContinue((e, a) -> {
            log.info(e);
        }))

// Implicit commit executed here
);
```

Blocking only API considerations

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;
});
```

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;
    }
}
```

5.15. Exception handling

Checked vs. unchecked exceptions

This is an eternal and religious debate. Pros and cons have been discussed time and again, and it still is a matter of taste, today. In this case, 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).

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 all methods in jOOQ that may cause such a DataAccessException document this both in the Javadoc as well as in their method signature.

DataAccessException is subtyped several times as follows:

- 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.
- MappingException: Something went wrong when loading a record from a POJO or when mapping a record into a POJO

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.

5.16. 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.

5.17. 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-3.16.0.xsd)
- A generated code backed implementation.

The following sections document each one of the above.

5.17.1. JDBC meta data

This is the default implementation of Database meta data, which is backed by JDBC's java.sql.DatabaseMetaData. 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);
```

5.17.2. 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(
         "create table a (i int);",
         "create table b (j int);",
         "create table c (k int);"
       )
      .getTables()
      .forEach(System.out::println);

// Using the jOOQ API:
create.meta(
         createTable("a").columns("i", INTEGER),
         createTable("b").columns("j", INTEGER),
         createTable("c").columns("k", INTEGER)
       )
      .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 strings or 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()

5.17.3. XML meta data

jOOQ offers a JAXB serialisable XML representation of the standard SQL INFORMATION_SCHEMA: https://www.jooq.org/xsd/jooq-meta-3.16.0.xsd. This format can be imported and exported as org.jooq.util.xml.jaxb.InformationSchema, and converted to org.jooq.Meta:

```
// Using strings:
create.meta(
         "<information_schema>...</information_schema>"
       )
      .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).

5.17.4. 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, or create a schema diff, you can wrap your generated code using:

```
Meta meta = create.meta(BOOK, AUTHOR);
```

5.18. 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')");
    }
});
```

5.19. Batched Connection

jOOQ supports JDBC batching through dedicated API to explicitly batch some statements, for improved performance. If performance is an "after thought" in some areas of your application, or batching doesn't work well for all environments (or even makes things worse, depending on the dialect), the org.jooq.tools.jdbc.BatchedConnection can be used to transparently buffer all jOOQ (and other JDBC) statements and execute them lazily as batches.

Imagine you might have two services, which produce the following INSERT statements behind the scenes:

```
// "Regular code":
// ---------------
module1.insertSomething(configuration);
module2.insertSomethingElse(configuration);

// The above might generate... (each line is a statement)
// INSERT INTO something (A, B) VALUES (?, ?)
// INSERT INTO something (A, B) VALUES (?, ?)
// INSERT INTO something (A, B) VALUES (?, ?)
// INSERT INTO something (A, B, C) VALUES (?, ?, ?)
// INSERT INTO something (A, B, C) VALUES (?, ?, ?)
// INSERT INTO something (A, B) VALUES (?, ?)
// INSERT INTO something_else (X, Y) VALUES (?, ?)
// INSERT INTO something_else (X, Y) VALUES (?, ?)
// INSERT INTO something_else (X, Y) VALUES (?, ?)
```

The business logic is complex, and you don't want to touch it again, just to improve performance. You can now either wrap your own JDBC connection in a org.jooq.tools.jdbc.BatchedConnection, or let jOOQ do that for you by calling DSLContext.batched():

```
// "Batch-collecting code".
// Alternatively, use DSLContext.batchedResult() to return a result from the lambda.
DSL.using(configuration).batched(c -> {
    module1.insertSomething(c);
    module2.insertSomethingElse(c);
});

// The above might now generate... (each line is a batch statement)
// INSERT INTO something (A, B) VALUES (?, ?)       -- With 3 calls to PreparedStatement.addBatch()
// INSERT INTO something (A, B, C) VALUES (?, ?, ?) -- With 2 calls to PreparedStatement.addBatch()
// INSERT INTO something (A, B) VALUES (?, ?)       -- With 1 calls to PreparedStatement.addBatch()
// INSERT INTO something_else (X, Y) VALUES (?, ?)  -- With 2 calls to PreparedStatement.addBatch()
```

Without changing the SQL strings, or the execution sequence, this way, it is now possible to buffer consecutive identical SQL strings and collect their bind values in a single batch.

As soon as any of the following events happens, the buffered batch is executed and a new batch is created:

- The SQL string changes (including "irrelevant" whitespace changes).
- A query producing results is executed.
- A static statement (as opposed to a prepared statement) is created.
- The connection is closed, or the transaction committed, or any other such interaction with JDBC is invoked.
- The batch size threshold is reached.

Limitations

While this approach to batching is transparent to most use-cases, there are some limitations:

- With query execution being delayed, the JDBC PreparedStatement#executeUpdate() calls cannot produce the correct row count value yet. They always produce 0, instead, as no rows have (yet) been affected.
- If a query produces results (e.g. ordinary SELECT statements, or INSERT .. RETURNING statements), batching is prevented.

To track effectively batched statements, turn on DEBUG logging.

For more known limitations of this functionality, please refer to #10692.

5.20. 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.

5.21. 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

# Statements producing specific exceptions can indicate them as such
select * from t;
@ exception: ACCESS TO TABLE T FORBIDDEN
```

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();
```

5.22. 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
```

The parsing connection caches input/output SQL string pairs and bind variable index mappings according to the org.jooq.CacheProvider.

5.23. 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
}

// 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.

5.23.1. 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());
    }
}

// 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));
    }
}
```

5.23.2. 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.

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());
    }
}

// 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));
    }
}
```

5.23.3. 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());
    }
}

// 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.

5.23.4. 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());
    }
}

// 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.

5.23.5. 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());
    }
}

// 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));
    }
}
```

5.23.6. Trivial condition

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

The SPI methods handling these events are trivialCondition(). This diagnostic depends on the transform patterns feature.

This problem appears with JDBC, jOOQ or with any ORM. A predicate of arbitrary complexity can sometimes be reduced to a simple TRUE or FALSE condition.

Why is it bad?

A trivial condition is bad for your application for multiple reasons:

- It could just be a typo (e.g. a JOIN predicate of the form A.ID = A.ID instead of A.ID = B.ID), in case of which it's simply wrong.
- The redundant predicate might be a subtle cause for duplicate statements.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class TrivialCondition extends DefaultDiagnosticsListener {
    @Override
    public void trivialCondition(DiagnosticsContext ctx) {
        System.out.println("Trivial condition: " + ctx.part());
    }
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new TrivialCondition()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT * FROM author WHERE id = id")) {
        // ..
    }
}
```

5.24. 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.

5.25. 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();
}
```

5.26. 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.

5.27. 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.

5.27.1. 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.

5.27.2. 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() + "]";
    }
}
```

5.27.2.1. 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]));
```

5.27.2.2. 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);
    });
});
```

5.27.2.3. 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

6. 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.

6.1. 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-3.17.8.jar
  The main library that you will include in your application to run jOOQ
- jooq-meta-3.17.8.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-3.17.8.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.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.ignite.IgniteDatabase
               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.snowflake.SnowflakeDatabase
               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-hibernate dependency)

               org.jooq.meta.extensions.jpa.JPADatabase

           This value can be used to reverse-engineer schemas defined by JPA annotated entities
           (requires jooq-meta-extensions-liquibase dependency)

               org.jooq.meta.extensions.liquibase.LiquibaseDatabase

           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/>
```

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.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.ignite.IgniteDatabase
      // 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.snowflake.SnowflakeDatabase
      // 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-hibernate dependency)
      //
      // org.jooq.meta.extensions.jpa.JPADatabase
      //
      // This value can be used to reverse-engineer schemas defined by JPA annotated entities
      // (requires jooq-meta-extensions-liquibase dependency)
      //
      // org.jooq.meta.extensions.liquibase.LiquibaseDatabase
      //
      // 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]")
    )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {

  // 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.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.ignite.IgniteDatabase
      // 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.snowflake.SnowflakeDatabase
      // 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-hibernate dependency)
      //
      // org.jooq.meta.extensions.jpa.JPADatabase
      //
      // This value can be used to reverse-engineer schemas defined by JPA annotated entities
      // (requires jooq-meta-extensions-liquibase dependency)
      //
      // org.jooq.meta.extensions.liquibase.LiquibaseDatabase
      //
      // 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
```

See the configuration XSD andgradle 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-3.17.0.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-3.17.8.jar, jooq-meta-3.17.8.jar, jooq-codegen-3.17.8.jar, reactive-streams-1.0.3.jar, r2dbc-spi-0.9.0.RELEASE.jar
- The JDBC driver you configured

A command-line example (for Windows, OSX/Linux/etc. will be similar)

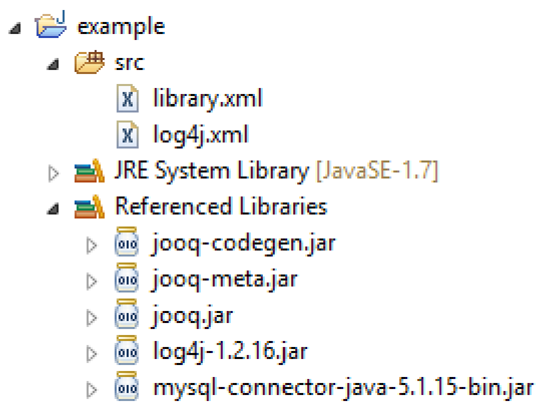
- Put the XML configuration file, jooq*.jar and the JDBC driver into a directory, e.g. C:\temp\jooq
- Go to C:\temp\jooq
- Run java -cp jooq-3.17.8.jar;jooq-meta-3.17.8.jar;jooq-codegen-3.17.8.jar;reactive-streams-1.0.3.jar;r2dbc-spi-0.9.0.RELEASE.jar;[JDBC-driver].jar org.jooq.codegen.GenerationTool <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

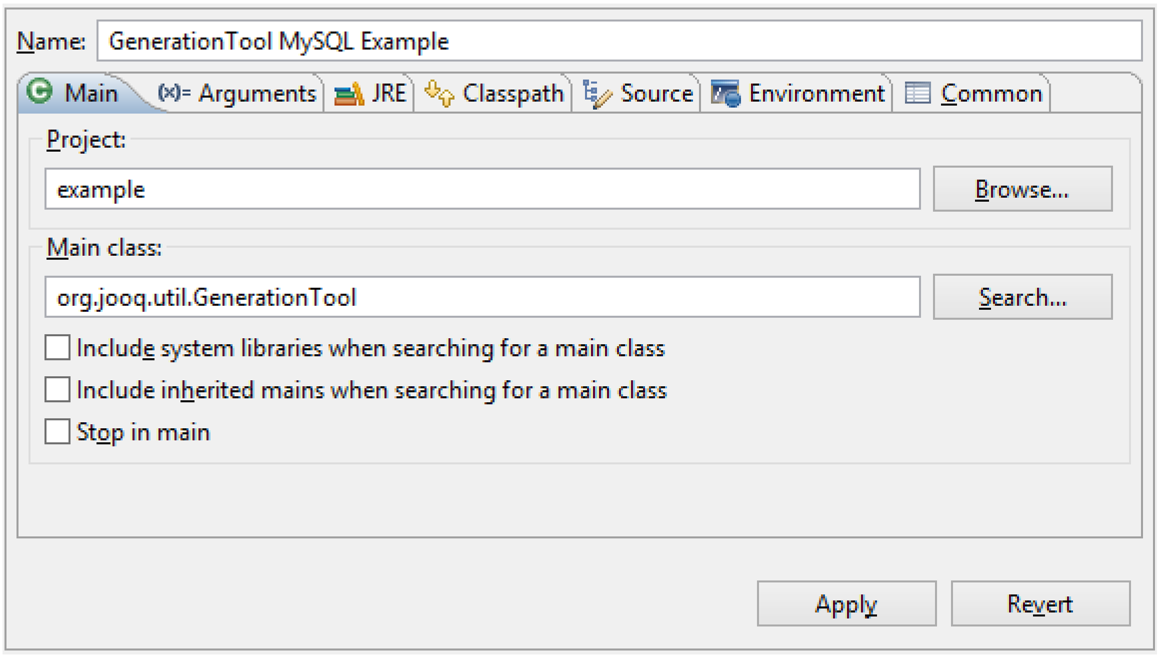
Run code generation from Eclipse

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-3.17.8.jar, jooq-meta-3.17.8.jar, jooq-codegen-3.17.8.jar artefacts may contain version numbers in the file names.


```
example
  src
    library.xml
    log4j.xml
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
    jooq-codegen.jar
    jooq-meta.jar
    jooq.jar
    log4j-1.2.16.jar
    mysql-connector-java-5.1.15-bin.jar
```

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.

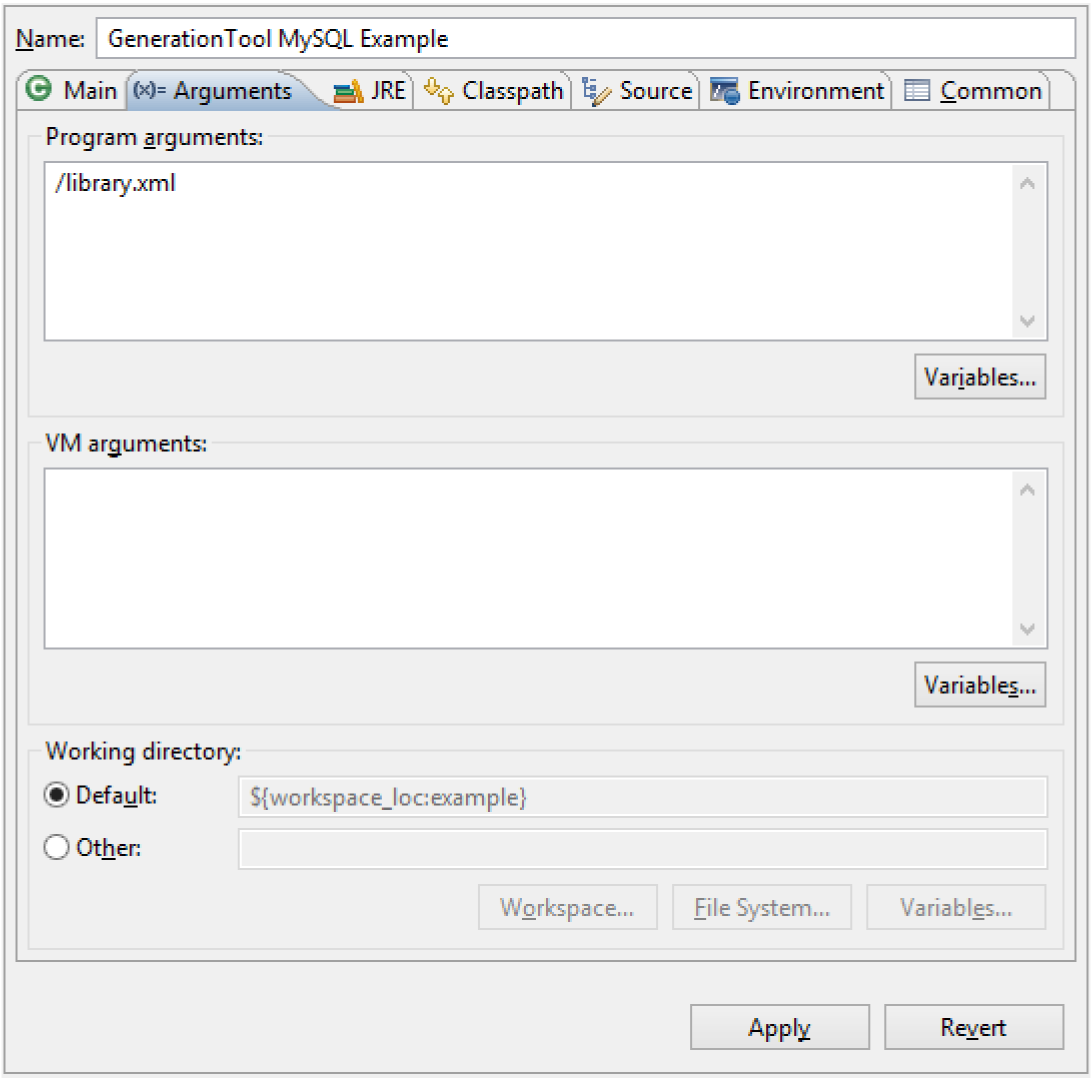

Name: GenerationTool MySQL Example

Project: example

Main class: org.jooq.util.GenerationTool

Main
Arguments
JRE
Classpath
Source
Environment
Common
Browse...
Search...
Include system libraries when searching for a main class
Include inherited mains when searching for a main class
Stop in main
Apply
Revert

With the XML file as an argument


Name: GenerationTool MySQL Example

Program arguments:

```
/library.xml
```

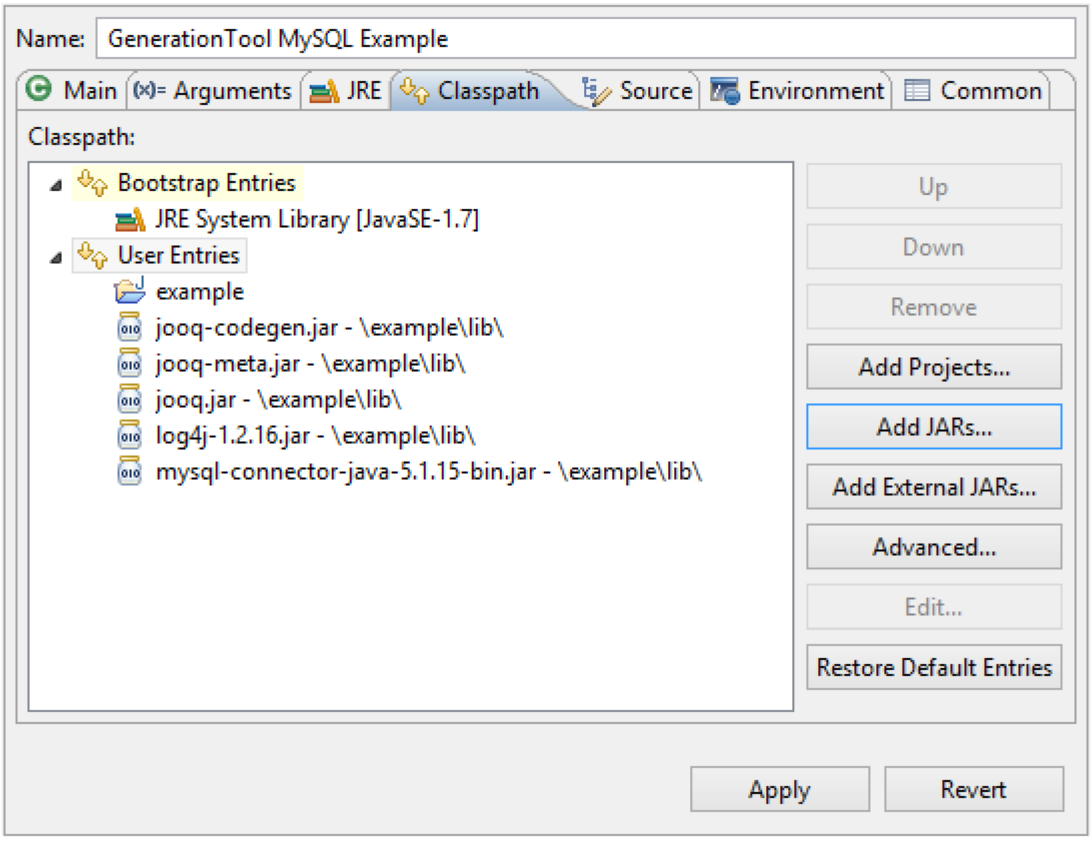
VM arguments:

Working directory:

● Default: ${workspace_loc:example}

○ Other:

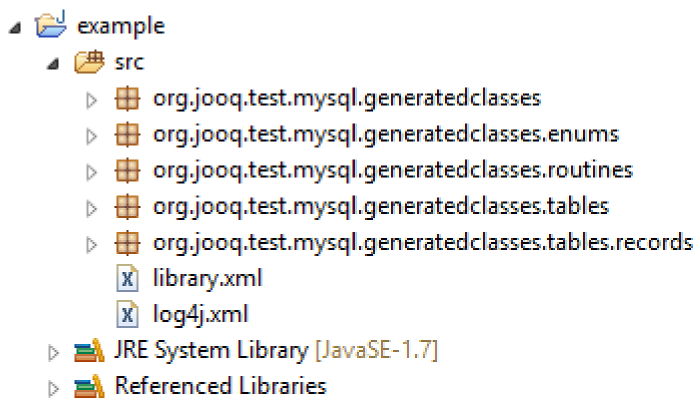
And the classpath set up correctly


Name: GenerationTool MySQL Example

Classpath:

```
Bootstrap Entries
    JRE System Library [JavaSE-1.7]
User Entries
    example
    jooq-codegen.jar - \example\lib\
    jooq-meta.jar - \example\lib\
    jooq.jar - \example\lib\
    log4j-1.2.16.jar - \example\lib\
    mysql-connector-java-5.1.15-bin.jar - \example\lib\
```

Finally, run the code generation and see your generated artefacts



```
example
    src
        org.jooq.test.mysql.generatedclasses
        org.jooq.test.mysql.generatedclasses.enums
        org.jooq.test.mysql.generatedclasses.routines
        org.jooq.test.mysql.generatedclasses.tables
        org.jooq.test.mysql.generatedclasses.tables.records
        library.xml
        log4j.xml
    JRE System Library [JavaSE-1.7]
    Referenced Libraries
```

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 with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.17.8</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>9.4.1212</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-3.17.8.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.

6.2. 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.

6.2.1. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  logging = 'WARN'
}
```

See the configuration XSD andgradle code generation for more details.
Available log levels are

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL

6.2.2. 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>

  <!-- Behaviour when encountering an unused configuration element. Defaults to LOG -->
  <onUnused>LOG</onUnused>
</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)

  // Behaviour when encountering an unused configuration element. Defaults to LOG
  .withOnUnused(OnError.LOG)
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {

  // Behaviour when encountering an exception. Defaults to FAIL
  onError = 'FAIL'

  // Behaviour when encountering an unused configuration element. Defaults to LOG
  onUnused = 'LOG'
}
```

See the configuration XSD andgradle 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

6.2.3. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  jdbc {
    autoCommit = true
  }
}
```

See the configuration XSD andgradle code generation for more details.

Init scripts

You can optionally provide a script to run after creating the JDBC connection, and before running the code generator.

XML (standalone and maven)

```
<configuration>
  <jdbc>
    <initScript>CREATE SCHEMA X//SET SCHEMA X</initScript>

    <!-- The separator between statements, defaulting to ";" -->
    <initSeparator>//</initSeparator>
  </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()
    .withInitScript("CREATE SCHEMA X//SET SCHEMA X")

    // The separator between statements, defaulting to ";"
    .withInitSeparator("//")
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  jdbc {
    initScript = 'CREATE SCHEMA X//SET SCHEMA X'

    // The separator between statements, defaulting to ";"
    initSeparator = '//'
  }
}
```

See the configuration XSD andgradle 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.

6.2.4. 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. Available generators:

         - org.jooq.codegen.JavaGenerator
         - org.jooq.codegen.KotlinGenerator
         - org.jooq.codegen.ScalaGenerator

         Defaults to org.jooq.codegen.JavaGenerator -->
    <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. Available generators:
    //
    // - org.jooq.codegen.JavaGenerator
    // - org.jooq.codegen.KotlinGenerator
    // - org.jooq.codegen.ScalaGenerator
    //
    // Defaults to org.jooq.codegen.JavaGenerator
    .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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {

    // Optional: The fully qualified class name of the code generator. Available generators:
    //
    // - org.jooq.codegen.JavaGenerator
    // - org.jooq.codegen.KotlinGenerator
    // - org.jooq.codegen.ScalaGenerator
    //
    // Defaults to org.jooq.codegen.JavaGenerator
    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 andgradle 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.

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.

6.2.5. 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:

6.2.5.1. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.ignite.IgniteDatabase
- 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.snowflake.SnowflakeDatabase
- 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.

6.2.5.2. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      regexFlags = 'COMMENTS DOTALL'
    }
  }
}
```

See the configuration XSD andgradle 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.

6.2.5.3. 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. Remember, these expressions are regular expressions with default flags, so multiple names need to be separated with the pipe symbol "|", not with commas, etc. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Identifier scope

A special, additional set of options allows for specifying whether the above two regular expressions should also match nested objects such as table columns or package routines. The following example will hide an INVISIBLE_COL in any table (and also tables and other objects called this way, of course), as well as INVISIBLE_ROUTINE:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>INVISIBLE_COL|INVISIBLE_ROUTINE</excludes>
      <includeExcludeColumns>true</includeExcludeColumns>
      <includeExcludePackageRoutines>true</includeExcludePackageRoutines>
    </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|INVISIBLE_ROUTINE")
      .withIncludeExcludeColumns(true)
      .withIncludeExcludePackageRoutines(true)
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includes = '.*'
      excludes = 'INVISIBLE_COL|INVISIBLE_ROUTINE'
      includeExcludeColumns = true
      includeExcludePackageRoutines = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Dynamic regular expression

Sometimes, you will like to dynamically generate the regular expression(s) based on more complex meta data than just the identifier of an object. For example, you might want to exclude all views from being generated. This can be done by appending <includeSql/< and <excludeSql/< elements, whose generated expressions are combined with <includes/< and <excludes/<. For example, if you're excluding views in PostgreSQL:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>STATIC_OBJECTS</excludes>
      <excludeSql>
        select table_schema || '\.' || table_name
        from information_schema.tables
        where table_type != 'VIEW'
      </excludeSql>
    </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("STATIC_OBJECTS")
      .withExcludeSql("" +
      "select table_schema || '\\.' || table_name" +
      "from information_schema.tables" +
      "where table_type != 'VIEW'" +
      "")
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includes = '.*'
      excludes = 'STATIC_OBJECTS'
      excludeSql = '''
        select table_schema || '\.' || table_name
        from information_schema.tables
        where table_type != 'VIEW'
      '''
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.4. 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)

```xml
<configuration>
  <generator>
    <database>
      <includeTables>true</includeTables>
      <includeSystemTables>false</includeSystemTables>
      <includeInvisibleColumns>true</includeInvisibleColumns>
      <includeEmbeddables>true</includeEmbeddables>
      <includeRoutines>true</includeRoutines>
      <includePackages>true</includePackages>
      <includePackageRoutines>true</includePackageRoutines>
      <includePackageUDTs>true</includePackageUDTs>
      <includePackageConstants>true</includePackageConstants>
      <includeUDTs>true</includeUDTs>
      <includeDomains>true</includeDomains>
      <includeSequences>false</includeSequences>
      <includeSystemSequences>false</includeSystemSequences>
      <includePrimaryKeys>false</includePrimaryKeys>
      <includeUniqueKeys>false</includeUniqueKeys>
      <includeForeignKeys>false</includeForeignKeys>
      <includeCheckConstraints>false</includeCheckConstraints>
      <includeSystemCheckConstraints>false</includeSystemCheckConstraints>
      <includeIndexes>false</includeIndexes>
      <includeSystemIndexes>false</includeSystemIndexes>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```java
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withIncludeTables(true)
      .withIncludeSystemTables(false)
      .withIncludeInvisibleColumns(true)
      .withIncludeEmbeddables(true)
      .withIncludeRoutines(true)
      .withIncludePackages(true)
      .withIncludePackageRoutines(true)
      .withIncludePackageUDTs(true)
      .withIncludePackageConstants(true)
      .withIncludeUDTs(true)
      .withIncludeDomains(true)
      .withIncludeSequences(false)
      .withIncludeSystemSequences(false)
      .withIncludePrimaryKeys(false)
      .withIncludeUniqueKeys(false)
      .withIncludeForeignKeys(false)
      .withIncludeCheckConstraints(false)
      .withIncludeSystemCheckConstraints(false)
      .withIncludeIndexes(false)
      .withIncludeSystemIndexes(false)
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```groovy
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includeTables = true
      includeSystemTables = false
      includeInvisibleColumns = true
      includeEmbeddables = true
      includeRoutines = true
      includePackages = true
      includePackageRoutines = true
      includePackageUDTs = true
      includePackageConstants = true
      includeUDTs = true
      includeDomains = true
      includeSequences = false
      includeSystemSequences = false
      includePrimaryKeys = false
      includeUniqueKeys = false
      includeForeignKeys = false
      includeCheckConstraints = false
      includeSystemCheckConstraints = false
      includeIndexes = false
      includeSystemIndexes = false
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

By default, most of these flags are set to true, with the exception of:

- includeSystemCheckConstraints: Some databases produce auxiliary CHECK constraints for other constraints like NOT NULL constraints. The redundant information is usually undesirable, which is why these are turned off by default.
- includeSystemIndexes: Some databases produce auxiliary INDEX objects for other constraints like FOREIGN KEY constraints. These indexes are not independent from the key, and the redundant information is usually undesirable, which is why these are turned off by default.
- includeSystemSequences: Some database produce auxiliary SEQUENCE objects to implement identities of tables. These sequences are usually not interesting to client code, which is why they are excluded by default.
- includeSystemTables: Some databases produce auxiliary TABLE objects to implement other types of tables, such as "virtual tables" (e.g. in SQLite). These implementation tables are usually not interesting for client code, which is why these are excluded by default.
- 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.

6.2.5.5. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      recordVersionFields = 'REC_VERSION'
      recordTimestampFields = 'REC_TIMESTAMP'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.6. Comments

jOOQ's code generator will pick up comments on schema objects created with the COMMENT statement and generate Javadocs accordingly.

If your dialect does not support the COMMENT statement, or your schema doesn't have comments, or you want to amend / replace the schema comments in the code generator, this section will show you how to add "synthetic" comments to your schema objects.

XML (standalone and maven)

```xml
<configuration>
  <generator>
    <database>
      <comments>
        <comment>

          <!-- Regular expression matching all objects that have this comment. -->
          <expression>CONFIGURED_COMMENT_TABLE</expression>

          <!-- Whether the comment is a deprecation notice. Defaults to false. -->
          <deprecated>true</deprecated>

          <!-- Whether the original schema comment should be included. Defaults to true. -->
          <includeSchemaComment>false</includeSchemaComment>

          <!-- The actual comment text. Defaults to no message. -->
          <message>Do not use this table.</message>
        </comment>
      </comments>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```java
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withComments(
        new CommentType()

          // Regular expression matching all objects that have this comment.
          .withExpression("CONFIGURED_COMMENT_TABLE")

          // Whether the comment is a deprecation notice. Defaults to false.
          .withDeprecated(true)

          // Whether the original schema comment should be included. Defaults to true.
          .withIncludeSchemaComment(false)

          // The actual comment text. Defaults to no message.
          .withMessage("Do not use this table.")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      comments {
        comment {

          // Regular expression matching all objects that have this comment.
          expression = 'CONFIGURED_COMMENT_TABLE'

          // Whether the comment is a deprecation notice. Defaults to false.
          deprecated = true

          // Whether the original schema comment should be included. Defaults to true.
          includeSchemaComment = false

          // The actual comment text. Defaults to no message.
          message = 'Do not use this table.'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7. Synthetic objects

Some legacy schemas may be lacking meta data such as foreign keys, but applications would still like to use them. Alternatively, updatable views may reflect underlying tables, but the code generator cannot access the meta data through the database's dictionary views. In those cases, "synthetic objects" can be configured, i.e. meta data that is generated purely by the code generator without being there in the meta data source.

6.2.5.7.1. Synthetic columns

There exist use cases where you want jOOQ's code generator to produce more columns than your meta data source suggests there are, per table. For example:

- Your code generation user doesn't have access to some columns, but your runtime user will have access.
- Your code generation database does not yet / anymore have those columns, but your production system does.
- You're generating client side computed columns, and you want them to have VIRTUAL semantics (computation on read), so the columns don't really exist in the schema.

In those cases, you can add synthetic columns to your schema like this:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <columns>
          <column>

            <!-- Optional regular expression matching all tables that have this identity. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- The name of the column -->
            <name>COLUMN</name>

            <!-- The type of the column -->
            <type>INTEGER</type>
          </column>
        </columns>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withColumns(
          new SyntheticColumnType()

            // Optional regular expression matching all tables that have this identity.
            .withTables("SCHEMA\\.TABLE")

            // The name of the column
            .withName("COLUMN")

            // The type of the column
            .withType("INTEGER")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        columns {
          column {

            // Optional regular expression matching all tables that have this identity.
            tables = 'SCHEMA\\.TABLE'

            // The name of the column
            name = 'COLUMN'

            // The type of the column
            type = 'INTEGER'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7.2. Synthetic readonly columns

jOOQ's code generator recognises readonly columns if it is configured to do so. Some databases do not support "real" readonly columns, but allow for emulating them, e.g. through triggers. If a column is a known "readonly column" without formally being one, users can specify regular expressions that match all tables and columns, which will be treated as if they were formal readonly columns. For example:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <readonlyColumns>
          <readonlyColumn>

            <!-- Optional regular expression matching all tables that have this identity. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- List all columns that are readonly -->
            <fields>ID</fields>
          </readonlyColumn>
        </readonlyColumns>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withReadonlyColumns(
          new SyntheticReadonlyColumnType()

            // Optional regular expression matching all tables that have this identity.
            .withTables("SCHEMA\\.TABLE")

            // List all columns that are readonly
            .withFields("ID")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        readonlyColumns {
          readonlyColumn {

            // Optional regular expression matching all tables that have this identity.
            tables = 'SCHEMA\\.TABLE'

            // List all columns that are readonly
            fields = 'ID'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7.3. Synthetic readonly ROWIDs

A few database products support a ROWID type, which models a physical location of a row on the disk. A ROWID can act like a primary key, e.g. in the absence of a formal primary key, although being a physical address, rather than a logical one, there is usually no guarantee of a ROWID to never change. For short-lived record access, this may be irrelevant (e.g. within a single query, for faster self-joins).

jOOQ's code generator allows for specifying a synthetic ROWID configuration that produces such ROWID columns on all tables that it matches. Combine this with the synthetic primary key feature, and you can replace a potentially existing primary key for all interactions with the row, including e.g. CRUD with UpdatableRecords (some vendor specific limitations may apply).

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <readonlyRowids>
          <readonlyRowid>

            <!-- Optional name of the column in generated code. -->
            <name>ROWID</name>

            <!-- Regular expression matching all tables that have this synthetic ROWID. -->
            <tables>SCHEMA\.TABLE</tables>
          </readonlyRowid>
        </readonlyRowids>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withReadonlyRowids(
          new SyntheticReadonlyRowidType()

            // Optional name of the column in generated code.
            .withName("ROWID")

            // Regular expression matching all tables that have this synthetic ROWID.
            .withTables("SCHEMA\\.TABLE")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        readonlyRowids {
          readonlyRowid {

            // Optional name of the column in generated code.
            name = 'ROWID'

            // Regular expression matching all tables that have this synthetic ROWID.
            tables = 'SCHEMA\\.TABLE'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7.4. 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 regular expressions that match all tables and columns, which will be treated as if they were formal identities. For example:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <identities>
          <identity>

            <!-- Optional regular expression matching all tables that have this identity. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- List all columns that are identities -->
            <fields>ID</fields>
          </identity>
        </identities>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withIdentities(
          new SyntheticIdentityType()

            // Optional regular expression matching all tables that have this identity.
            .withTables("SCHEMA\\.TABLE")

            // List all columns that are identities
            .withFields("ID")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        identities {
          identity {

            // Optional regular expression matching all tables that have this identity.
            tables = 'SCHEMA\\.TABLE'

            // List all columns that are identities
            fields = 'ID'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7.5. 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>
      <syntheticObjects>
        <primaryKeys>
          <primaryKey>

            <!-- Optional name of the primary key, if tables matches only a single table. -->
            <name>PK_TABLE</name>

            <!-- Optional regular expression matching all tables that have this primary key. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- List multiple fields in the key order -->
            <fields>
              <field>COLUMN1</field>
              <field>COLUMN2</field>
            </fields>

            <!-- Alternatively, instead of listing columns above, promote a unique key to a primary key, by name. -->
            <key>UK_TABLE</key>
          </primaryKey>
        </primaryKeys>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withPrimaryKeys(
          new SyntheticPrimaryKeyType()

            // Optional name of the primary key, if tables matches only a single table.
            .withName("PK_TABLE")

            // Optional regular expression matching all tables that have this primary key.
            .withTables("SCHEMA\\.TABLE")

            // List multiple fields in the key order
            .withFields(
              String.COLUMN1,
              String.COLUMN2
            )

            // Alternatively, instead of listing columns above, promote a unique key to a primary key, by name.
            .withKey("UK_TABLE")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        primaryKeys {
          primaryKey {

            // Optional name of the primary key, if tables matches only a single table.
            name = 'PK_TABLE'

            // Optional regular expression matching all tables that have this primary key.
            tables = 'SCHEMA\\.TABLE'

            // List multiple fields in the key order
            fields {
              field = 'COLUMN1'
              field = 'COLUMN2'
            }

            // Alternatively, instead of listing columns above, promote a unique key to a primary key, by name.
            key = 'UK_TABLE'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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.

6.2.5.7.6. Synthetic unique keys

jOOQ's code generator recognises unique 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 unique key. If a composite synthetic unique key is desired, the regular expression should match all columns of that table that are part of the unique key. For example, a composite synthetic unique key consists of (COLUMN1, COLUMN2) in table SCHEMA.TABLE:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <uniqueKeys>
          <uniqueKey>

            <!-- Optional name of the unique key, if tables matches only a single table. -->
            <name>UK_TABLE</name>

            <!-- Optional regular expression matching all tables that have this unique key. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- List multiple fields in the key order -->
            <fields>
              <field>COLUMN1</field>
              <field>COLUMN2</field>
            </fields>
          </uniqueKey>
        </uniqueKeys>
      </syntheticObjects>
    </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()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withUniqueKeys(
          new SyntheticUniqueKeyType()

            // Optional name of the unique key, if tables matches only a single table.
            .withName("UK_TABLE")

            // Optional regular expression matching all tables that have this unique key.
            .withTables("SCHEMA\\.TABLE")

            // List multiple fields in the key order
            .withFields(
              String.COLUMN1,
              String.COLUMN2
            )
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        uniqueKeys {
          uniqueKey {

            // Optional name of the unique key, if tables matches only a single table.
            name = 'UK_TABLE'

            // Optional regular expression matching all tables that have this unique key.
            tables = 'SCHEMA\\.TABLE'

            // List multiple fields in the key order
            fields {
              field = 'COLUMN1'
              field = 'COLUMN2'
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.7.7. Synthetic foreign keys

jOOQ's code generator recognises foreign 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 foreign key. If a composite synthetic foreign key is desired, the regular expression should match all columns of that table that are part of the foreign key. For example, a composite synthetic foreign key consists of (COLUMN1, COLUMN2) in table SCHEMA.TABLE:

XML (standalone and maven)

```xml
<configuration>
  <generator>
    <database>
      <syntheticObjects>
        <foreignKeys>
          <foreignKey>

            <!-- Optional name of the foreign key, if tables matches only a single table. -->
            <!-- This is useful to disambiguate navigational methods and implicit join methods! -->
            <name>FK_TABLE</name>

            <!-- Optional regular expression matching all tables that have this foreign key. -->
            <tables>SCHEMA\.TABLE</tables>

            <!-- List multiple fields in the key order -->
            <fields>
              <field>COLUMN1</field>
              <field>COLUMN2</field>
            </fields>

            <!-- Specify the table that is being referenced by this foreign key -->
            <referencedTable>SCHEMA\.OTHER_TABLE</referencedTable>

            <!-- Optional: The primary or unique key columns that are being referenced -->
            <referencedFields>
              <field>COLUMN1</field>
              <field>COLUMN2</field>
            </referencedFields>

            <!-- Optional: reference a primary or unique key by name -->
            <!-- If the referenced fields or key are not specified, the referencedTable's primary key is used -->
            <referencedKey>UK_TABLE</referencedKey>
          </foreignKey>
        </foreignKeys>
      </syntheticObjects>
    </database>
  </generator>
</configuration>
```

See the configuration XSD, standalone code generation, and maven code generation for more details.
Programmatic

```java
new org.jooq.meta.jaxb.Configuration()
  .withGenerator(new Generator()
    .withDatabase(new Database()
      .withSyntheticObjects(new SyntheticObjectsType()
        .withForeignKeys(
          new SyntheticForeignKeyType()

            // Optional name of the foreign key, if tables matches only a single table.
            .withName("FK_TABLE")

            // Optional regular expression matching all tables that have this foreign key.
            .withTables("SCHEMA\\.TABLE")

            // List multiple fields in the key order
            .withFields(
              String.COLUMN1,
              String.COLUMN2
            )

            // Specify the table that is being referenced by this foreign key
            .withReferencedTable("SCHEMA\\.OTHER_TABLE")

            // Optional: The primary or unique key columns that are being referenced
            .withReferencedFields(new ()
              .withField(String.COLUMN1)
              .withField(String.COLUMN2)
            )

            // Optional: reference a primary or unique key by name
            .withReferencedKey("UK_TABLE")
        )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticObjects {
        foreignKeys {
          foreignKey {

            // Optional name of the foreign key, if tables matches only a single table.
            name = 'FK_TABLE'

            // Optional regular expression matching all tables that have this foreign key.
            tables = 'SCHEMA\\.TABLE'

            // List multiple fields in the key order
            fields {
              field = 'COLUMN1'
              field = 'COLUMN2'
            }

            // Specify the table that is being referenced by this foreign key
            referencedTable = 'SCHEMA\\.OTHER_TABLE'

            // Optional: The primary or unique key columns that are being referenced
            referencedFields {
              field = 'COLUMN1'
              field = 'COLUMN2'
            }

            // Optional: reference a primary or unique key by name
            referencedKey = 'UK_TABLE'
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.5.8. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      dateAsTimestamp = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
This flag will apply before any other data type related flags are applied, including forced types.

6.2.5.9. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      ignoreProcedureReturnValues = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.5.10. Readonly columns

There are various reasons why a column could be readonly, including:

- The column is formally marked as READONLY, if the database product supports this.
- The column is from a view, and either the entire view, or just that particular column is not updatable.
- The user lacks INSERT and/or UPDATE grants to the column.
- The column is computed using [GENERATED ALWAYS] AS <expression> (DEFAULT columns aren't readonly)
- There is a trigger preventing writing to the column.

jOOQ's code generator can detect some of these, and mark columns as readonly for you, meaning that the column will not be taken into consideration in DML statements, such as INSERT or UPDATE, or UpdatableRecord CRUD calls. To configure the runtime behaviour of such readonly columns, please see the relevant section about readonly columns.

It's also possible to manually mark columns as readonly using the synthetic readonly columns configuration.

6.2.5.11. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      unsignedTypes = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.5.12. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputSchema = 'my_schema'
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_catalog'
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_catalog'
      inputSchema = 'my_schema'
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      schemata {
        schema {
          inputSchema = 'schema1'
        }
        schema {
          inputSchema = 'schema2'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = 'catalog1'
        }
        catalog {
          inputCatalog = 'catalog2'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = 'catalog'
          schemata {
            schema {
              inputSchema = 'schema1'
            }
            schema {
              inputSchema = 'schema2'
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.

Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_input_catalog'
      outputCatalog = 'my_output_catalog'
      inputSchema = 'my_input_schema'
      outputSchema = 'my_output_schema'
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_input_catalog'
      outputCatalogToDefault = true
      inputSchema = 'my_input_schema'
      outputSchemaToDefault = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
For more information about the catalog and schema mapping feature, please refer to the relevant section of the manual.

6.2.5.13. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogVersionProvider = 'com.example.MyCatalogVersionProvider'
      schemaVersionProvider = 'com.example.MySchemaVersionProvider'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogVersionProvider = 'SELECT :catalog_name || '_' || MAX("version") FROM "schema_version"'
      schemaVersionProvider = 'SELECT :schema_name || '_' || MAX("version") FROM "schema_version"'
    }
  }
}
```

See the configuration XSD andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogVersionProvider = 1
      schemaVersionProvider = 2
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.5.14. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      orderProvider = 'com.example.CaseInsensitiveOrderProvider'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
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
    }
}
```

6.2.5.15. 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.

6.2.5.15.1. 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 <priority/> and then 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:

- <priority>: Any number (default 0) indicating the priority of the forced type. Higher numbers have higher priority.

Matching predicates:

- <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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.

6.2.5.15.2. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.

6.2.5.15.3. 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 or ad-hoc converters. 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.

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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);
```

6.2.5.15.4. 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 andprogrammatic code generation for more details.

Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.

6.2.5.15.5. Lambda converters

In addition to inlining your complete converter definition, you can simplify usage of Converter.of() and Converter.ofNullable() using a lambdaConverter, specifying only the two lambda expressions, where from translates "from the database type" and to translates "to the database type".

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. -->
          <lambdaConverter>
            <from>Year::of</from>
            <to>Year::getValue</to>
          </lambdaConverter>

          <!-- 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.
          .withLambdaConverter(new LambdaConverter()
            .withFrom("Year::of")
            .withTo("Year::getValue")
          )

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.YEAR.*")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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.
          lambdaConverter {
            from = 'Year::of'
            to = 'Year::getValue'
          }

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = '.*\\.YEAR.*'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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.

6.2.5.15.6. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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.

6.2.5.15.7. Jackson converters

In case your database column is of type org.jooq.JSON or org.jooq.JSONB, you may want to attach a custom data type binding that is backed by the popular Jackson library. It is easy to achieve manually using a hand-written Converter, but you can also use the following convenience 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>com.example.MyType</userType>

          <!-- Apply the built in converter of type

               - org.jooq.jackson.extensions.converters.JSONtoJacksonConverter or
               - org.jooq.jackson.extensions.converters.JSONBtoJacksonConverter. -->
          <jsonConverter>true</jsonConverter>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.JSON_COLUMN</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.MyType")

          // Apply the built in converter of type
          //
          // - org.jooq.jackson.extensions.converters.JSONtoJacksonConverter or
          // - org.jooq.jackson.extensions.converters.JSONBtoJacksonConverter.
          .withJsonConverter(true)

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.JSON_COLUMN")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.

Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          userType = 'com.example.MyType'

          // Apply the built in converter of type
          //
          // - org.jooq.jackson.extensions.converters.JSONtoJacksonConverter or
          // - org.jooq.jackson.extensions.converters.JSONBtoJacksonConverter.
          jsonConverter = true

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = '.*\\.JSON_COLUMN'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

In order to access these converters, just add the following dependency to your project:

```
<dependency>
    <!-- Use org.jooq                for the Open Source Edition
             org.jooq.pro            for commercial editions with Java 17 support,
             org.jooq.pro-java-11    for commercial editions with Java 11 support,
             org.jooq.pro-java-8     for commercial editions with Java 8 support,
             org.jooq.trial          for the free trial edition with Java 17 support,
             org.jooq.trial-java-11  for the free trial edition with Java 11 support,
             org.jooq.trial-java-8   for the free trial edition with Java 8 support

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-jackson-extensions</artifactId>
    <version>3.17.8</version>
</dependency>
```

For more details about how to match columns, please refer to the section about matching columns for forced types.

6.2.5.15.8. JAXB converters

In case your database column is of type org.jooq.XML, you may want to attach a custom data type binding that is backed by JAXB. It is easy to achieve manually using a hand-written Converter, but you can also use the following convenience 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>com.example.MyType</userType>

          <!-- Apply the built in converter of type org.jooq.impl.XMLtoJAXBConverter. -->
          <xmlConverter>true</xmlConverter>

          <!-- A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions. --
>
          <includeExpression>.*\.XML_COLUMN</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.MyType")

          // Apply the built in converter of type org.jooq.impl.XMLtoJAXBConverter.
          .withXmlConverter(true)

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          .withIncludeExpression(".*\\.XML_COLUMN")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Specify the Java type of your custom type. This corresponds to the Converter's <U> type.
          userType = 'com.example.MyType'

          // Apply the built in converter of type org.jooq.impl.XMLtoJAXBConverter.
          xmlConverter = true

          // A Java regex matching fully-qualified columns, attributes, parameters. Use the pipe to separate several expressions.
          includeExpression = '.*\\.XML_COLUMN'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

For more details about how to match columns, please refer to the section about matching columns for forced types.

6.2.5.15.9. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 =    // [...]
    // [...]

}
```

6.2.5.15.10. Client side computed columns

```
This is experimental functionality, and as such subject to change. Use at your own risk!
```

Computed columns are a powerful feature on the server side, where a column can be guaranteed to always contain a value according to an expression based on the other columns of a table. Many dialects support them in at least one of two flavours:

- STORED: The computed value is calculated on write, and stored in the table
- VIRTUAL: The computed value is calculated on read (although it may still be stored in indexes)

There are various reasons why you may wish to compute columns, but not on the server side, i.e. not in the database directly, but let jOOQ do that for you on the client side. The forced type configuration can be used again to match columns, and apply an org.jooq.Generator:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>

          <!-- Provide an expression of type org.jooq.Generator, or a class reference
               com.example.X for a class of type org.jooq.Generator, which can be
               instantiated using new com.example.X() -->
          <generator>ctx -&gt; org.jooq.impl.DSL.val(1)</generator>
          <includeExpression>(?i:ALWAYS_ONE)</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()

          // Provide an expression of type org.jooq.Generator, or a class reference
          // com.example.X for a class of type org.jooq.Generator, which can be
          // instantiated using new com.example.X()
          .withGenerator("ctx -> org.jooq.impl.DSL.val(1)")
          .withIncludeExpression("(?i:ALWAYS_ONE)")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Provide an expression of type org.jooq.Generator, or a class reference
          // com.example.X for a class of type org.jooq.Generator, which can be
          // instantiated using new com.example.X()
          generator = 'ctx -> org.jooq.impl.DSL.val(1)'
          includeExpression = '(?i:ALWAYS_ONE)'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
The behaviour is as follows:

- If the forced type matches an actual column from your database, then the semantics is that of STORED computed columns, i.e. all of your DML statements will be transformed by jOOQ to ensure the correct computation of the value, which will be written to your schema. Other database clients, including non-jOOQ based ones, can then see those computed values.
- If the forced type matches a synthetic column, then the semantics is that of VIRTUAL computed columns, i.e. the column does not exist in your schema, but the computation will be added to all of your SELECT statements, if you include the column in your projections and other clauses.

Unlike server side computed columns, these computational expressions can include any type of column expression, including scalar subqueries (even correlated ones), or implicit joins, making client side computed columns an extremely powerful jOOQ feature!

The exact time a computation takes place is not specified for both VIRTUAL and STORED client side computed columns. While it may seem reasonable to expect the computation to take place when the column is rendered to SQL, future implementations may defer or even cache the computation. As such, a computation is recommended to be pure and side effect free. While side effects, such as producing a "current timestamp" are not forbidden, it is important to take into account the non-determinism of when the computation takes place. For example, caching a pre-computed transaction timestamp may be more reliable than computing the timestamp from within the Generator.

This is experimental functionality and may be subject to change.

6.2.5.15.11. Audit columns

A common use-case for (STORED) client side computed columns are audit columns. There exist many ways to implement auditing, including:

- Journalling all changes using a trigger that stores the complete change into a separate table
- SQL:2011 temporal tables, which are more powerful than mere auditing
- Using a trigger to fill in a few technical columns, such as CREATED_AT, MODIFIED_AT, etc.

While jOOQ recommends you use a trigger or other out-of-the-box, standard SQL features, in order to implement auditing directly inside of your database (to make sure no one can bypass it), there are valid use-cases where you may not want to do this in the database itself, e.g. because you cannot create triggers for some privilege or other technical reasons. For those cases, jOOQ offers basic support for audit columns.

For example, you can set up your code generation configuration like this:

XML (standalone and maven)

```xml
<configuration>
    <generator>
        <database>
            <forcedTypes>
                <forcedType>
                    <auditInsertTimestamp>true</auditInsertTimestamp>
                    <includeExpression>CREATED_AT</includeExpression>
                </forcedType>
                <forcedType>
                    <auditInsertUser>true</auditInsertUser>
                    <includeExpression>CREATED_BY</includeExpression>
                </forcedType>
                <forcedType>
                    <auditUpdateTimestamp>true</auditUpdateTimestamp>
                    <includeExpression>MODIFIED_AT</includeExpression>
                </forcedType>
                <forcedType>
                    <auditUpdateUser>true</auditUpdateUser>
                    <includeExpression>MODIFIED_BY</includeExpression>
                </forcedType>
                <forcedType>
                    <auditInsertTimestamp>true</auditInsertTimestamp>
                    <auditUpdateTimestamp>true</auditUpdateTimestamp>
                    <includeExpression>CREATED_OR_MODIFIED_AT</includeExpression>
                </forcedType>
                <forcedType>
                    <auditInsertUser>true</auditInsertUser>
                    <auditUpdateUser>true</auditUpdateUser>
                    <includeExpression>CREATED_OR_MODIFIED_BY</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()
          .withAuditInsertTimestamp(true)
          .withIncludeExpression("CREATED_AT"),
        new ForcedType()
          .withAuditInsertUser(true)
          .withIncludeExpression("CREATED_BY"),
        new ForcedType()
          .withAuditUpdateTimestamp(true)
          .withIncludeExpression("MODIFIED_AT"),
        new ForcedType()
          .withAuditUpdateUser(true)
          .withIncludeExpression("MODIFIED_BY"),
        new ForcedType()
          .withAuditInsertTimestamp(true)
          .withAuditUpdateTimestamp(true)
          .withIncludeExpression("CREATED_OR_MODIFIED_AT"),
        new ForcedType()
          .withAuditInsertUser(true)
          .withAuditUpdateUser(true)
          .withIncludeExpression("CREATED_OR_MODIFIED_BY")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {
          auditInsertTimestamp = true
          includeExpression = 'CREATED_AT'
        }
        forcedType {
          auditInsertUser = true
          includeExpression = 'CREATED_BY'
        }
        forcedType {
          auditUpdateTimestamp = true
          includeExpression = 'MODIFIED_AT'
        }
        forcedType {
          auditUpdateUser = true
          includeExpression = 'MODIFIED_BY'
        }
        forcedType {
          auditInsertTimestamp = true
          auditUpdateTimestamp = true
          includeExpression = 'CREATED_OR_MODIFIED_AT'
        }
        forcedType {
          auditInsertUser = true
          auditUpdateUser = true
          includeExpression = 'CREATED_OR_MODIFIED_BY'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
Using the above flags, you can specify an org.jooq.impl.AuditGenerator to be applied to your columns, which is just convenience for a hand-rolled (STORED) client side computed columns. All such audit columns compute their actual value from Configuration.auditProvider(), which allows for overriding the org.jooq.impl.DefaultAuditProvider behaviour, which is:

- Using CURRENT_TIMESTAMP, CURRENT_TIME, CURRENT_DATE, or a similar function, if the data type is temporal, and we're auditing timestamps.
- Using CURRENT_USER if the data type is a String, and we're auditing users.

The two types of org.jooq.AuditType flags (USER, TIMESTAMP) are mutually exclusive, but the two types of org.jooq.GeneratorStatementType (INSERT, UPDATE) can be combined in order to trigger writing to the column on either or both types of operations.

As any other (STORED) client side computed columns, this transforms any jOOQ generated DML statement, irrespective of whether it originates from within jOOQ (e.g. via the UpdatableRecord API or DAO API), or whether you hand-roll it using the jOOQ DSL. For example, assuming a table like this, which will be generated using the above <forcedTypes/> configuration:

```
CREATE TABLE t_audit (
  id INTEGER NOT NULL,
  val INTEGER NOT NULL,
  created_at TIMESTAMP NOT NULL,
  created_by VARCHAR(100) NOT NULL,
  modified_at TIMESTAMP,
  modified_by VARCHAR(100),
  created_or_modified_at TIMESTAMP NOT NULL,
  created_or_modified_by VARCHAR(100),

  CONSTRAINT pk_t_audit PRIMARY KEY (id)
);
```

Now, the following jOOQ statement:

```
create.insertInto(T_AUDIT)
      .columns(T_AUDIT.ID, T_AUDIT.VAL)
      .values(1, 1)
      .execute();
```

Might produce a SQL statement like this:

```
INSERT INTO public.t_audit (
  id,
  val,
  created_at,
  created_by,
  created_or_modified_at,
  created_or_modified_by
)
SELECT
  id,
  val,
  current_timestamp,
  current_user(),
  current_timestamp,
  current_user()
FROM (
  SELECT 1, 1
) AS t (id, val)
```

Combining this feature with embeddable types is particularly useful.

6.2.5.15.12. Visibility Modifier (per forced type)

There are a few cases where the visibility of individual columns should be reduced, e.g. to private in order to hide a technical column from user code. For those cases, the forced type configuration can be used again to match columns, and apply a visibility:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>
          <!-- Possible values for visibilityModifier
             - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
             - NONE     : Do not generate visibility modifiers
             - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
             - INTERNAL : Generate explicit internal modifiers (Kotlin)
             - PRIVATE  : Generate explicit private modifiers (Java, Kotlin, Scala) -->
          <visibilityModifier>PRIVATE</visibilityModifier>
          <includeExpression>(?i:CREATED|MODIFIED)_(?i:AT|BY)</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()

          // Possible values for visibilityModifier
          // - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
          // - NONE     : Do not generate visibility modifiers
          // - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
          // - INTERNAL : Generate explicit internal modifiers (Kotlin)
          // - PRIVATE  : Generate explicit private modifiers (Java, Kotlin, Scala)
          .withVisibilityModifier(VisibilityModifier.PRIVATE)
          .withIncludeExpression("(?i:CREATED|MODIFIED)_(?i:AT|BY)")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {

          // Possible values for visibilityModifier
          // - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
          // - NONE     : Do not generate visibility modifiers
          // - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
          // - INTERNAL : Generate explicit internal modifiers (Kotlin)
          // - PRIVATE  : Generate explicit private modifiers (Java, Kotlin, Scala)
          visibilityModifier = 'PRIVATE'
          includeExpression = '(?i:CREATED|MODIFIED)_(?i:AT|BY)'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
Such private columns can now no longer be referred by user queries explicitly, although they are still a part of the table, and can be projected when projecting all columns. Custom code sections in generated code can still access these columns, and so can inline converters.

It may make sense to combine this feature with client side computed columns, e.g. to compute a column A based on a column B, and then hide B from user code to make sure users always use A, instead.

Not all values make sense for all languages.

See also global visibility modifiers for further options.

6.2.5.16. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      tableValuedFunctions = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6. 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:

6.2.6.1. 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-)

- Nullable annotations: When using alternative JVM languages like Kotlin, it may be desireable to have some hints related to nullability on generated code. When jOOQ encounters a nullable column, for instance, a JSR-305 @Nullable annotation could warn Kotlin users about well-known nullable columns. @Nonnull columns are more treacherous, as there are numerous reasons why a jOOQ Record could contain a null value in such a column, e.g. when the record was initialised without any values, or when the record originates from a UNION or OUTER JOIN.
  The nullableAnnotationType and nonnullAnnotationType configurations allow for specifying an alternative, qualified annotation name other than the JSR-305 types below.

  * javax.annotation.Nullable When a column is nullable
  * javax.annotation.Nonnull When a column is non-nullable

- 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:

  * jakarta.persistence.Column
  * jakarta.persistence.Entity
  * jakarta.persistence.GeneratedValue
  * jakarta.persistence.GenerationType
  * jakarta.persistence.Id
  * jakarta.persistence.Index (JPA 2.1 and later)
  * jakarta.persistence.Table
  * jakarta.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.
- Bean annotations: A set of JavaBeans annotations can be added to the generated code to facilitate interoperability with the JavaBeans specification

  * java.beans.ConstructorProperties

- 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
  * org.springframework.transaction.annotation.Transactional (if <springDao/> is set)

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>
      <generatedAnnotationDate>true</generatedAnnotationDate>

      <nullableAnnotation>true</nullableAnnotation>
      <nullableAnnotationType>javax.annotation.Nullable</nullableAnnotationType>
      <nonnullAnnotation>true</nonnullAnnotation>
      <nonnullAnnotationType>javax.annotation.Nonnull</nonnullAnnotationType>

      <jpaAnnotations>true</jpaAnnotations>
      <jpaVersion>2.2</jpaVersion>

      <validationAnnotations>true</validationAnnotations>

      <!-- The springDao flag enables the generation of @Transactional annotations on a
           generated, Spring-specific DAO -->
      <springAnnotations>true</springAnnotations>
      <springDao>true</springDao>

      <kotlinSetterJvmNameAnnotationsOnIsPrefix>true</kotlinSetterJvmNameAnnotationsOnIsPrefix>

      <constructorPropertiesAnnotation>true</constructorPropertiesAnnotation>
      <constructorPropertiesAnnotationOnPojos>true</constructorPropertiesAnnotationOnPojos>
      <constructorPropertiesAnnotationOnRecords>true</constructorPropertiesAnnotationOnRecords>
    </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)
      .withGeneratedAnnotationDate(true)
      .withNullableAnnotation(true)
      .withNullableAnnotationType("javax.annotation.Nullable")
      .withNonnullAnnotation(true)
      .withNonnullAnnotationType("javax.annotation.Nonnull")
      .withJpaAnnotations(true)
      .withJpaVersion(2.2)
      .withValidationAnnotations(true)

      // The springDao flag enables the generation of @Transactional annotations on a
      // generated, Spring-specific DAO
      .withSpringAnnotations(true)
      .withSpringDao(true)
      .withKotlinSetterJvmNameAnnotationsOnIsPrefix(true)
      .withConstructorPropertiesAnnotation(true)
      .withConstructorPropertiesAnnotationOnPojos(true)
      .withConstructorPropertiesAnnotationOnRecords(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Possible values for generatedAnnotationType
      // - DETECT_FROM_JDK
      // - JAVAX_ANNOTATION_GENERATED
      // - JAVAX_ANNOTATION_PROCESSING_GENERATED
      generatedAnnotation = true
      generatedAnnotationType = 'DETECT_FROM_JDK'
      generatedAnnotationDate = true
      nullableAnnotation = true
      nullableAnnotationType = 'javax.annotation.Nullable'
      nonnullAnnotation = true
      nonnullAnnotationType = 'javax.annotation.Nonnull'
      jpaAnnotations = true
      jpaVersion = 2.2
      validationAnnotations = true

      // The springDao flag enables the generation of @Transactional annotations on a
      // generated, Spring-specific DAO
      springAnnotations = true
      springDao = true
      kotlinSetterJvmNameAnnotationsOnIsPrefix = true
      constructorPropertiesAnnotation = true
      constructorPropertiesAnnotationOnPojos = true
      constructorPropertiesAnnotationOnRecords = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.2. Extended types

In addition to all the data types specified by the JDBC specification such as SMALLINT, INTEGER, BIGINT, VARCHAR, and many more, jOOQ includes support for additional standard SQL data types, which JDBC ignores.

These data types are very convenient when they work out of the box, although you may prefer to roll your own, e.g. using converters or bindings.

Specifically, in the case of spatial types, which are available to the commercial editions only, you may want to opt out of jOOQ's code generation support when you're using the jOOQ Open Source Edition.

Support for these four data types can be configured in code generation:

- interval types (i.e. org.jooq.types.Interval and subtypes)
- json types (i.e. org.jooq.JSON and org.jooq.JSONB)
- spatial types (i.e. org.jooq.Spatial and subtypes)
- xml types (i.e. org.jooq.XML)

By default, all of the above data types are supported, and support is enabled. To disable support, use:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <intervalTypes>false</intervalTypes>
      <jsonTypes>false</jsonTypes>
      <spatialTypes>false</spatialTypes>
      <xmlTypes>false</xmlTypes>
    </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()
      .withIntervalTypes(false)
      .withJsonTypes(false)
      .withSpatialTypes(false)
      .withXmlTypes(false)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      intervalTypes = false
      jsonTypes = false
      spatialTypes = false
      xmlTypes = false
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.3. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      fluentSetters = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.4. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      fullyQualifiedTypes = '.*\\.MY_TABLE'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

6.2.6.5. 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>
      <globalDomainReferences>true</globalDomainReferences>
      <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)
      .withGlobalDomainReferences(true)
      .withGlobalUDTReferences(true)
      .withGlobalRoutineReferences(true)
      .withGlobalQueueReferences(true)
      .withGlobalLinkReferences(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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
      globalDomainReferences = true
      globalUDTReferences = true
      globalRoutineReferences = true
      globalQueueReferences = true
      globalLinkReferences = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.6. 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>

      <!-- Influencing how the DefaultGeneratorStrategy generates identifiers. The default is true.

           When a child table has only one FK towards a parent table, then that path is "unambiguous."
           In that case, the DefaultGeneratorStrategy uses the parent table name instead of the FK name. -->
      <implicitJoinPathsUseTableNameForUnambiguousFKs>true</implicitJoinPathsUseTableNameForUnambiguousFKs>

      <!-- Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true. -->
      <implicitJoinPathsAsKotlinProperties>true</implicitJoinPathsAsKotlinProperties>
    </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)

      // Influencing how the DefaultGeneratorStrategy generates identifiers. The default is true.
      //
      // When a child table has only one FK towards a parent table, then that path is "unambiguous."
      // In that case, the DefaultGeneratorStrategy uses the parent table name instead of the FK name.
      .withImplicitJoinPathsUseTableNameForUnambiguousFKs(true)

      // Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true.
      .withImplicitJoinPathsAsKotlinProperties(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Allowing to turn off the feature entirely. The default is true.
      implicitJoinPathsToOne = true

      // Influencing how the DefaultGeneratorStrategy generates identifiers. The default is true.
      //
      // When a child table has only one FK towards a parent table, then that path is "unambiguous."
      // In that case, the DefaultGeneratorStrategy uses the parent table name instead of the FK name.
      implicitJoinPathsUseTableNameForUnambiguousFKs = true

      // Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true.
      implicitJoinPathsAsKotlinProperties = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.7. 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)

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      javaTimeTypes = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.8. Serial Version UID

All jOOQ QueryPart types, including the generated ones, are are serializable. It is thus a good practice to generate a serialVersionUID value in all generated classes, for example:

```
private static final long serialVersionUID = -2074134614;
```

By default, this value is 1L to prevent compilation warnings and to minimise noisy diffs should you choose to check in generated sources to version control.
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Possible values for generatedSerialVersionUID
         - CONSTANT (default): Always generate 1L
         - OFF               : Don't generate a serialVersionUID
         - HASH              : Calculate a unique-ish value based on the hash code of the content -->
      <generatedSerialVersionUID>CONSTANT</generatedSerialVersionUID>
    </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 generatedSerialVersionUID
      // - CONSTANT (default): Always generate 1L
      // - OFF               : Don't generate a serialVersionUID
      // - HASH              : Calculate a unique-ish value based on the hash code of the content
      .withGeneratedSerialVersionUID(GeneratedSerialVersionUID.CONSTANT)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Possible values for generatedSerialVersionUID
      // - CONSTANT (default): Always generate 1L
      // - OFF               : Don't generate a serialVersionUID
      // - HASH              : Calculate a unique-ish value based on the hash code of the content
      generatedSerialVersionUID = 'CONSTANT'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.9. Sources

With jOOQ 3.13, source code for views is generated by the code generator if available. This feature can be turned off using:

- sources: The generation of all types of source code can be turned off globally
- sourcesOnViews: The generation of source code on views can be turned off

The flags governing the generation of these annotations are:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <sources>true</sources>
      <sourcesOnViews>true</sourcesOnViews>
    </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()
      .withSources(true)
      .withSourcesOnViews(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      sources = true
      sourcesOnViews = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.10. Text blocks

By default, jOOQ's code generator produces Java 15 text blocks for generated source code (e.g. views, check constraints, etc.) if your Java version is up to date. In some cases, it may be desirable to turn that off:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Options include:
           - DETECT_FROM_JDK (default, generate text blocks if Java version supports them)
           - ON
           - OFF
        -->
      <textBlocks>OFF</textBlocks>
    </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()

      // Options include:
      // - DETECT_FROM_JDK (default, generate text blocks if Java version supports them)
      // - ON
      // - OFF
      .withTextBlocks(GeneratedTextBlocks.OFF)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Options include:
      // - DETECT_FROM_JDK (default, generate text blocks if Java version supports them)
      // - ON
      // - OFF
      textBlocks = 'OFF'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.11. Visibility Modifier (global)

By default, most generated code that is intended for use by users is generated using the explicit (Java) or implicit (Kotlin, Scala) public modifier.

In Kotlin there are valid reasons to deviate from this default, including:

- -Xexplicit-api=strict is enabled, and it is a compilation error to omit the otherwise optional visibility modifier
- The generated code should be generated as internal, not public

In the above cases, a configuration flag can help:

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <!-- Possible values for visibilityModifier
         - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
         - NONE     : Do not generate visibility modifiers
         - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
         - INTERNAL : Generate explicit internal modifiers (Kotlin) -->
      <visibilityModifier>INTERNAL</visibilityModifier>
    </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 visibilityModifier
      // - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
      // - NONE     : Do not generate visibility modifiers
      // - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
      // - INTERNAL : Generate explicit internal modifiers (Kotlin)
      .withVisibilityModifier(VisibilityModifier.INTERNAL)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Possible values for visibilityModifier
      // - DEFAULT  : The default per language (Java: public, Kotlin, Scala: implicit public)
      // - NONE     : Do not generate visibility modifiers
      // - PUBLIC   : Generate explicit public modifiers (Java, Kotlin)
      // - INTERNAL : Generate explicit internal modifiers (Kotlin)
      visibilityModifier = 'INTERNAL'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
Not all values make sense for all languages.

See also visibility modifiers per forced types for further options.

6.2.6.12. 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) and \s (whitespace)
- Newline: \r (carriage return) and \n (newline)

XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>
      <indentation>\s\t</indentation>
      <newline>\r\n</newline>

      <!-- The number of characters after which Javadoc is line-wrapped. 0 to turn off line wrapping. -->
      <printMarginForBlockComment>80</printMarginForBlockComment>
    </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("\\s\\t")
      .withNewline("\\r\\n")

      // The number of characters after which Javadoc is line-wrapped. 0 to turn off line wrapping.
      .withPrintMarginForBlockComment(80)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      indentation = '\\s\\t'
      newline = '\\r\\n'

      // The number of characters after which Javadoc is line-wrapped. 0 to turn off line wrapping.
      printMarginForBlockComment = 80
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.6.13. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forceIntegerTypesOnZeroScaleDecimals = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.2.7. 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>
      <locale>de</locale>
      <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")
      .withLocale("de")
      .withClean(true)
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    target {
      packageName = 'org.jooq.your.packagename'
      directory = '/path/to/your/dir'
      encoding = 'UTF-8'
      locale = 'de'
      clean = true
    }
  }
}
```

See the configuration XSD andgradle 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.
- locale: The locale that should be used for locale-specific operations, such as String.toUpperCase(), e.g. by the default generator strategy.
- clean: Whether the target package (<packageName/>) should be cleaned to contain only generated code after a generation run. Defaults to true.

6.3. Programmatic generator configuration

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-3.17.0.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-3.17.8.jar, jooq-meta-3.17.8.jar, and jooq-codegen-3.17.8.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 jakarta.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:

6.4. 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>

    <!-- The code generator implementation. -->
    <name>org.jooq.codegen.JavaGenerator</name>

    <!-- A programmatic naming strategy implementation, referened by class name. -->
    <strategy>
      <name>org.jooq.codegen.DefaultGeneratorStrategy</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()

    // The code generator implementation.
    .withName("org.jooq.codegen.JavaGenerator")

    // A programmatic naming strategy implementation, referened by class name.
    .withStrategy(new Strategy()
      .withName("org.jooq.codegen.DefaultGeneratorStrategy")
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {

    // The code generator implementation.
    name = 'org.jooq.codegen.JavaGenerator'

    // A programmatic naming strategy implementation, referened by class name.
    strategy {
      name = 'org.jooq.codegen.DefaultGeneratorStrategy'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
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. 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, not for Tables
 */
public class AsInDatabaseStrategy extends DefaultGeneratorStrategy {

    /**
     * Override this to specifiy what identifiers in Java should look like.
     * This will just take the identifier as defined in the database.
     */
    @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. This example will name
     * setters "set[NAME_IN_DATABASE]"
     */
    @Override
    public String getJavaSetterName(Definition definition, Mode mode) {
        return "set" + definition.getOutputName();
    }

    /**
     * Just like setters...
     */
    @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 mostly for convenience methods
     * when calling stored procedures and functions. This example shows how to
     * set a prefix to a CamelCase version of your procedure
     */
    @Override
    public String getJavaMethodName(Definition definition, Mode mode) {
        return "call" + org.jooq.tools.StringUtils.toCamelCase(definition.getOutputName());
    }

    /**
     * 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 POJOs and method arguments
     */
    @Override
    public String getJavaMemberName(Definition definition, Mode mode) {
        return definition.getOutputName();
    }

    /**
     * Override this method to define the base class for those artefacts that
     * allow for custom base classes
     */
    @Override
    public String getJavaClassExtends(Definition definition, Mode mode) {
        return Object.class.getName();
    }

    /**
     * Override this method to define the interfaces to be implemented by those
     * artefacts that allow for custom interface implementation
     */
    @Override
    public List<String> getJavaClassImplements(Definition definition, Mode mode) {
        return Arrays.asList(Serializable.class.getName(), Cloneable.class.getName());
    }

    /**
     * Override this method to define the suffix to apply to routines when
```

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)
```

An out-of-the-box strategy to keep names as they are

By default, jOOQ's generator strategy will convert your database's UNDER_SCORE_NAMES to PascalCaseNames as this is a more common idiom in the Java ecosystem. If, however, you want to retain the names and the casing exactly as it is defined in your database, you can use the org.jooq.codegen.KeepNamesGeneratorStrategy, which will retain all names exactly as they are.

More examples can be found here:

- org.jooq.codegen.example.JPrefixGeneratorStrategy
- org.jooq.codegen.example.JVMArgsGeneratorStrategy

6.5. 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 such:

- NOTE: 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()).
- NOTE: There had been an incompatible change between jOOQ 3.2 and jOOQ 3.3 in the configuration of these matcher strategies. See #3217 for details.

XML (standalone and maven)

```
<configuration>
  <!-- These properties can be added directly to the generator element: -->
  <generator>
    <strategy>
      <matchers>

        <!-- Specify 0..n catalog matchers to provide a strategy for naming objects created from catalogs. -->
        <catalogs>
          <catalog>

            <!-- Match unqualified or qualified catalog names. If left empty, this matcher applies to all catalogs. -->
            <expression>MY_CATALOG</expression>

            <!-- These elements influence the naming of a generated org.jooq.Catalog object. -->
            <catalogClass> see below MatcherRule specification </catalogClass>
            <catalogIdentifier> see below MatcherRule specification </catalogIdentifier>
            <catalogImplements>com.example.MyOptionalCustomInterface</catalogImplements>
          </catalog>
        </catalogs>

        <!-- 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> see below MatcherRule specification </schemaClass>
            <schemaIdentifier> see below MatcherRule specification </schemaIdentifier>
            <schemaImplements>com.example.MyOptionalCustomInterface</schemaImplements>
          </schema>
        </schemas>

        <!-- 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> see below MatcherRule specification </tableClass>
            <tableIdentifier> see below MatcherRule specification </tableIdentifier>
            <tableImplements>com.example.MyOptionalCustomInterface</tableImplements>

            <!-- These elements influence the naming of a generated org.jooq.Record object. -->
            <recordClass> see below 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> see below MatcherRule specification </interfaceClass>
            <interfaceImplements>com.example.MyOptionalCustomInterface</interfaceImplements>

            <!-- These elements influence the naming of a generated org.jooq.DAO object. -->
            <daoClass> see below MatcherRule specification </daoClass>
            <daoImplements>com.example.MyOptionalCustomInterface</daoImplements>

            <!-- These elements influence the naming of a generated POJO object.  -->
            <pojoClass> see below MatcherRule specification </pojoClass>
            <pojoExtends>com.example.MyOptionalCustomBaseClass</pojoExtends>
            <pojoImplements>com.example.MyOptionalCustomInterface</pojoImplements>
          </table>
        </tables>

        <!-- 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> see below MatcherRule specification </fieldIdentifier>
            <fieldMember> see below MatcherRule specification </fieldMember>
            <fieldSetter> see below MatcherRule specification </fieldSetter>
            <fieldGetter> see below MatcherRule specification </fieldGetter>
          </field>
        </fields>

        <!-- 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> see below MatcherRule specification </routineClass>
            <routineMethod> see below MatcherRule specification </routineMethod>
            <routineImplements>com.example.MyOptionalCustomInterface</routineImplements>
          </routine>
        </routines>

        <!-- 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> see below MatcherRule specification </sequenceIdentifier>
          </sequence>
        </sequences>

        <!-- Specify 0..n enum matchers to provide a strategy for naming objects created from enums. -->
```

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 catalog matchers to provide a strategy for naming objects created from catalogs.
        .withCatalogs(
          new MatchersCatalogType()

            // Match unqualified or qualified catalog names. If left empty, this matcher applies to all catalogs.
            .withExpression("MY_CATALOG")

            // These elements influence the naming of a generated org.jooq.Catalog object.
            .withCatalogClass(MatcherRule. see below MatcherRule specification )
            .withCatalogIdentifier(MatcherRule. see below MatcherRule specification )
            .withCatalogImplements("com.example.MyOptionalCustomInterface")
        )

        // 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. see below MatcherRule specification )
            .withSchemaIdentifier(MatcherRule. see below MatcherRule specification )
            .withSchemaImplements("com.example.MyOptionalCustomInterface")
        )

        // 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. see below MatcherRule specification )
            .withTableIdentifier(MatcherRule. see below MatcherRule specification )
            .withTableImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated org.jooq.Record object.
            .withRecordClass(MatcherRule. see below 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. see below MatcherRule specification )
            .withInterfaceImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated org.jooq.DAO object.
            .withDaoClass(MatcherRule. see below MatcherRule specification )
            .withDaoImplements("com.example.MyOptionalCustomInterface")

            // These elements influence the naming of a generated POJO object.
            .withPojoClass(MatcherRule. see below MatcherRule specification )
            .withPojoExtends("com.example.MyOptionalCustomBaseClass")
            .withPojoImplements("com.example.MyOptionalCustomInterface")
        )

        // 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. see below MatcherRule specification )
            .withFieldMember(MatcherRule. see below MatcherRule specification )
            .withFieldSetter(MatcherRule. see below MatcherRule specification )
            .withFieldGetter(MatcherRule. see below MatcherRule specification )
        )

        // 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. see below MatcherRule specification )
            .withRoutineMethod(MatcherRule. see below MatcherRule specification )
            .withRoutineImplements("com.example.MyOptionalCustomInterface")
        )

        // 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. see below MatcherRule specification )
        )

        // 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")
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {

  // These properties can be added directly to the generator element:
  generator {
    strategy {
      matchers {

        // Specify 0..n catalog matchers to provide a strategy for naming objects created from catalogs.
        catalogs {
          catalog {

            // Match unqualified or qualified catalog names. If left empty, this matcher applies to all catalogs.
            expression = 'MY_CATALOG'

            // These elements influence the naming of a generated org.jooq.Catalog object.
            catalogClass = ' see below MatcherRule specification '
            catalogIdentifier = ' see below MatcherRule specification '
            catalogImplements = 'com.example.MyOptionalCustomInterface'
          }
        }

        // 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 = ' see below MatcherRule specification '
            schemaIdentifier = ' see below MatcherRule specification '
            schemaImplements = 'com.example.MyOptionalCustomInterface'
          }
        }

        // 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 = ' see below MatcherRule specification '
            tableIdentifier = ' see below MatcherRule specification '
            tableImplements = 'com.example.MyOptionalCustomInterface'

            // These elements influence the naming of a generated org.jooq.Record object.
            recordClass = ' see below 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 = ' see below MatcherRule specification '
            interfaceImplements = 'com.example.MyOptionalCustomInterface'

            // These elements influence the naming of a generated org.jooq.DAO object.
            daoClass = ' see below MatcherRule specification '
            daoImplements = 'com.example.MyOptionalCustomInterface'

            // These elements influence the naming of a generated POJO object.
            pojoClass = ' see below MatcherRule specification '
            pojoExtends = 'com.example.MyOptionalCustomBaseClass'
            pojoImplements = 'com.example.MyOptionalCustomInterface'
          }
        }

        // 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 = ' see below MatcherRule specification '
            fieldMember = ' see below MatcherRule specification '
            fieldSetter = ' see below MatcherRule specification '
            fieldGetter = ' see below MatcherRule specification '
          }
        }

        // 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 = ' see below MatcherRule specification '
            routineMethod = ' see below MatcherRule specification '
            routineImplements = 'com.example.MyOptionalCustomInterface'
          }
        }

        // 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 = ' see below MatcherRule specification '
          }
        }
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

The above example used references to "MatcherRule", which is an XSD type that looks like this:

XML (standalone and maven)

```
<configuration>
  <generator>
    <strategy>
      <matchers>
        <schemas>
          <schema>
            <schemaClass>
              <!-- The optional transform element lets you apply a name transformation algorithm
                   to transform the actual database name into a more convenient form. Possible values are:

                   - AS_IS               : Leave the database name as it is             : MY_name => MY_name
                   - LOWER               : Transform the database name into lower case  : MY_name => my_name
                   - LOWER_FIRST_LETTER  : Transform the first letter into lower case   : MY_name => mY_name
                   - UPPER               : Transform the database name into upper case  : MY_name => MY_NAME
                   - UPPER_FIRST_LETTER  : Transform the first letter into upper case   : my_NAME => My_NAME
                   - CAMEL               : Transform the database name into camel case  : MY_name => myName
                   - PASCAL              : Transform the database name into pascal case : MY_name => MyName -->
              <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()
            .withSchemaClass(new MatcherRule()

              // The optional transform element lets you apply a name transformation algorithm
              // to transform the actual database name into a more convenient form. Possible values are:
              //
              // - AS_IS               : Leave the database name as it is             : MY_name => MY_name
              // - LOWER               : Transform the database name into lower case  : MY_name => my_name
              // - LOWER_FIRST_LETTER  : Transform the first letter into lower case   : MY_name => mY_name
              // - UPPER               : Transform the database name into upper case  : MY_name => MY_NAME
              // - UPPER_FIRST_LETTER  : Transform the first letter into upper case   : my_NAME => My_NAME
              // - CAMEL               : Transform the database name into camel case  : MY_name => myName
              // - PASCAL              : Transform the database name into pascal case : MY_name => MyName
              .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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    strategy {
      matchers {
        schemas {
          schema {
            schemaClass {

              // The optional transform element lets you apply a name transformation algorithm
              // to transform the actual database name into a more convenient form. Possible values are:
              //
              // - AS_IS               : Leave the database name as it is             : MY_name => MY_name
              // - LOWER               : Transform the database name into lower case  : MY_name => my_name
              // - LOWER_FIRST_LETTER  : Transform the first letter into lower case   : MY_name => mY_name
              // - UPPER               : Transform the database name into upper case  : MY_name => MY_NAME
              // - UPPER_FIRST_LETTER  : Transform the first letter into upper case   : my_NAME => My_NAME
              // - CAMEL               : Transform the database name into camel case  : MY_name => myName
              // - PASCAL              : Transform the database name into pascal case : MY_name => MyName
              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 andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

Some 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    strategy {
      matchers {
        tables {
          table {
            expression = '^(.*?)_BOOK_(.*)$'
            tableIdentifier {
              transform = 'UPPER'
              expression = '$1_BROCHURE_$2'
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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-3.17.0.xsd XSD file.

6.6. 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.

6.7. 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.

6.8. 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

Table generation can be deactivated using the tables flag

6.9. Generated records

Every table and view in your database will generate an org.jooq.Record 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 influencing generated records

These 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)

Flags controlling record generation

Record generation can be deactivated using the records flag

6.10. Generated POJOs

Every table and view in your database will generate a POJO implementation that looks like this:

```
// JPA annotations can be generated, optionally
@jakarta.persistence.Entity
@jakarta.persistence.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 influencing generated POJOs

These 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)
- pojosAsJavaRecordClasses: If you're using the JavaGenerator, this will generate POJOs as (immutable) Java 16 record types
- pojosAsScalaCaseClasses: If you're using the ScalaGenerator, this will generate POJOs as (mutable or immutable) Scala case classes
- pojosAsKotlinDataClasses: If you're using the KotlinGenerator, this will generate POJOs as (mutable or immutable) kotlin data classes
- pojosToString: Whether POJOs should have a generated toString() implementation.
- pojosEqualsAndHashCode: Whether POJOs should have generated equals() and hashCode() implementations.
- unsignedTypes: This influences all relevant getters and setters
- validationAnnotations: JSR-303 validation annotations are used on generated records (details here)

Flags controlling POJO generation

POJO generation can be activated using the pojos flag

6.11. Generated Interfaces

Note, there are numerous problems related to generated interfaces as can be seen in #10509. A future version of jOOQ might remove support for this functionality.

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.

6.12. Generated DAOs

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 Spring-specific DAOs containing @Transactional annotations -->
      <springDao>true</springDao>
    </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)

      // Generate Spring-specific DAOs containing @Transactional annotations
      .withSpringDao(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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

      // Generate Spring-specific DAOs containing @Transactional annotations
      springDao = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
See generation annotations for more information about the annotation specific flags, above.

6.13. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Generate the Sequences class
      sequences = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.14. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Generate the Routines class and a class for each individual routine
      routines = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.15. Generated domains

Every DOMAIN type in your database will generate an org.jooq.Domain reference in a single Domains class that looks like this:

```
public class Domains {

    /**
     * The domain <code>PUBLIC.EMAIL</code>.
     */
    public static final Domain<String> EMAIL = Internal.createDomain(
          schema()
        , DSL.name("EMAIL")
        , org.jooq.impl.CLOB.nullable(false)
        , Internal.createCheck(null, null, "(\"VALUE\" LIKE '%@%')")
    );

    /**
     * The domain <code>PUBLIC.YEAR</code>.
     */
    public static final Domain<Integer> YEAR = Internal.createDomain(
          schema()
        , DSL.name("YEAR")
        , org.jooq.impl.INTEGER.nullable(false)
        , Internal.createCheck(null, null, "((\"VALUE\" >= 1900) AND (\"VALUE\" <= 2050))")
    );
}
```

These domains specifications are referenced from all columns that use the respective domain type.

6.16. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Generate the UDTs class, records and UDT literals for each UDT
      udts = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.17. Data type extensions

jOOQ has SPIs to support custom data types via converters or bindings, which can be hand-written and attached to generated code using code generation configuration.

For some dialects and some of their more popular data types, jOOQ also supports opt-in extensions. The following subsections document these extensions:

6.17.1. PostgreSQL

The jooq-postgres-extensions module contains data types, converters, and bindings for the following data types:

- cidr is for IPv4 and IPv6 networks.
- citext is "case insensitive text".
- daterange is a date range type. Depending on the javaTimeTypes configuration, this translates to java.sql.Date or java.time.LocalDate ranges.
- hstore is a key-value store (i.e. a Map<String, String>).
- inet is for IPv4 and IPv6 hosts and networks.
- int4range is a 32 bit integer range type.
- int8range is a 64 bit integer range type.
- ltree is a label tree data structure.
- numrange is a numeric range type.
- tsrange is a timestamp range type. Depending on the javaTimeTypes configuration, this translates to java.sql.Timestamp or java.time.LocalDateTime ranges.
- tstzrange is a timestamptz range type.

In order to access these data types, just add the following dependency to your project:

```
<dependency>
    <!-- Use org.jooq                for the Open Source Edition
             org.jooq.pro            for commercial editions with Java 17 support,
             org.jooq.pro-java-11    for commercial editions with Java 11 support,
             org.jooq.pro-java-8     for commercial editions with Java 8 support,
             org.jooq.trial          for the free trial edition with Java 17 support,
             org.jooq.trial-java-11  for the free trial edition with Java 11 support,
             org.jooq.trial-java-8   for the free trial edition with Java 8 support

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-postgres-extensions</artifactId>
    <version>3.17.8</version>
</dependency>
```

Starting with jOOQ 3.17, the code generator will auto-register the following forced types, if it finds the jooq-postgres-extensions module on the classpath. These types are registered with lowest priority, such that custom forced types will take precedence:
XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <forcedTypes>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.BigDecimalRange</userType>
          <binding>org.jooq.postgres.extensions.bindings.BigDecimalRangeBinding</binding>
          <includeTypes>numrange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.BigDecimalRange[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.BigDecimalRangeArrayBinding</binding>
          <includeTypes>_numrange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Cidr</userType>
          <binding>org.jooq.postgres.extensions.bindings.CidrBinding</binding>
          <includeTypes>cidr</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Cidr[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.CidrArrayBinding</binding>
          <includeTypes>_cidr</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>java.lang.String</userType>
          <binding>org.jooq.postgres.extensions.bindings.CitextBinding</binding>
          <includeTypes>citext</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>java.lang.String[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.CitextArrayBinding</binding>
          <includeTypes>_citext</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Hstore</userType>
          <binding>org.jooq.postgres.extensions.bindings.HstoreBinding</binding>
          <includeTypes>hstore</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Hstore[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.HstoreArrayBinding</binding>
          <includeTypes>_hstore</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Inet</userType>
          <binding>org.jooq.postgres.extensions.bindings.InetBinding</binding>
          <includeTypes>inet</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.Inet[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.InetArrayBinding</binding>
          <includeTypes>_inet</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.IntegerRange</userType>
          <binding>org.jooq.postgres.extensions.bindings.IntegerRangeBinding</binding>
          <includeTypes>int4range</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.IntegerRange[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.IntegerRangeArrayBinding</binding>
          <includeTypes>_int4range</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>

        <!-- Depending on <javaTimeTypes/>, this may map to DateRange instead -->
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.LocalDateRange</userType>
          <binding>org.jooq.postgres.extensions.bindings.LocalDateRangeBinding</binding>
          <includeTypes>daterange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>

        <!-- Depending on <javaTimeTypes/>, this may map to DateRange[] instead -->
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.LocalDateRange[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.LocalDateRangeArrayBinding</binding>
          <includeTypes>_daterange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>

        <!-- Depending on <javaTimeTypes/>, this may map to TimestampRange instead -->
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.LocalDateTimeRange</userType>
          <binding>org.jooq.postgres.extensions.bindings.LocalDateTimeRangeBinding</binding>
          <includeTypes>tsrange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>

        <!-- Depending on <javaTimeTypes/>, this may map to TimestampRange[] instead -->
        <forcedType>
          <userType>org.jooq.postgres.extensions.types.LocalDateTimeRange[]</userType>
          <binding>org.jooq.postgres.extensions.bindings.LocalDateTimeRangeArrayBinding</binding>
          <includeTypes>_tsrange</includeTypes>
          <priority>-2147483648</priority>
        </forcedType>
```

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()
          .withUserType("org.jooq.postgres.extensions.types.BigDecimalRange")
          .withBinding("org.jooq.postgres.extensions.bindings.BigDecimalRangeBinding")
          .withIncludeTypes("numrange")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.BigDecimalRange[]")
          .withBinding("org.jooq.postgres.extensions.bindings.BigDecimalRangeArrayBinding")
          .withIncludeTypes("_numrange")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Cidr")
          .withBinding("org.jooq.postgres.extensions.bindings.CidrBinding")
          .withIncludeTypes("cidr")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Cidr[]")
          .withBinding("org.jooq.postgres.extensions.bindings.CidrArrayBinding")
          .withIncludeTypes("_cidr")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("java.lang.String")
          .withBinding("org.jooq.postgres.extensions.bindings.CitextBinding")
          .withIncludeTypes("citext")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("java.lang.String[]")
          .withBinding("org.jooq.postgres.extensions.bindings.CitextArrayBinding")
          .withIncludeTypes("_citext")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Hstore")
          .withBinding("org.jooq.postgres.extensions.bindings.HstoreBinding")
          .withIncludeTypes("hstore")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Hstore[]")
          .withBinding("org.jooq.postgres.extensions.bindings.HstoreArrayBinding")
          .withIncludeTypes("_hstore")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Inet")
          .withBinding("org.jooq.postgres.extensions.bindings.InetBinding")
          .withIncludeTypes("inet")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Inet[]")
          .withBinding("org.jooq.postgres.extensions.bindings.InetArrayBinding")
          .withIncludeTypes("_inet")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.IntegerRange")
          .withBinding("org.jooq.postgres.extensions.bindings.IntegerRangeBinding")
          .withIncludeTypes("int4range")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.IntegerRange[]")
          .withBinding("org.jooq.postgres.extensions.bindings.IntegerRangeArrayBinding")
          .withIncludeTypes("_int4range")
          .withPriority("-2147483648"),

        // Depending on <javaTimeTypes/>, this may map to DateRange instead
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LocalDateRange")
          .withBinding("org.jooq.postgres.extensions.bindings.LocalDateRangeBinding")
          .withIncludeTypes("daterange")
          .withPriority("-2147483648"),

        // Depending on <javaTimeTypes/>, this may map to DateRange[] instead
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LocalDateRange[]")
          .withBinding("org.jooq.postgres.extensions.bindings.LocalDateRangeArrayBinding")
          .withIncludeTypes("_daterange")
          .withPriority("-2147483648"),

        // Depending on <javaTimeTypes/>, this may map to TimestampRange instead
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LocalDateTimeRange")
          .withBinding("org.jooq.postgres.extensions.bindings.LocalDateTimeRangeBinding")
          .withIncludeTypes("tsrange")
          .withPriority("-2147483648"),

        // Depending on <javaTimeTypes/>, this may map to TimestampRange[] instead
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LocalDateTimeRange[]")
          .withBinding("org.jooq.postgres.extensions.bindings.LocalDateTimeRangeArrayBinding")
          .withIncludeTypes("_tsrange")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LongRange")
          .withBinding("org.jooq.postgres.extensions.bindings.LongRangeBinding")
          .withIncludeTypes("int8range")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.LongRange[]")
          .withBinding("org.jooq.postgres.extensions.bindings.LongRangeArrayBinding")
          .withIncludeTypes("_int8range")
          .withPriority("-2147483648"),
        new ForcedType()
          .withUserType("org.jooq.postgres.extensions.types.Ltree")
          .withBinding("org.jooq.postgres.extensions.bindings.LtreeBinding")
          .withIncludeTypes("ltree")
          .withPriority("-2147483648"),
        new ForcedType()
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forcedTypes {
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.BigDecimalRange'
          binding = 'org.jooq.postgres.extensions.bindings.BigDecimalRangeBinding'
          includeTypes = 'numrange'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.BigDecimalRange[]'
          binding = 'org.jooq.postgres.extensions.bindings.BigDecimalRangeArrayBinding'
          includeTypes = '_numrange'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Cidr'
          binding = 'org.jooq.postgres.extensions.bindings.CidrBinding'
          includeTypes = 'cidr'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Cidr[]'
          binding = 'org.jooq.postgres.extensions.bindings.CidrArrayBinding'
          includeTypes = '_cidr'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'java.lang.String'
          binding = 'org.jooq.postgres.extensions.bindings.CitextBinding'
          includeTypes = 'citext'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'java.lang.String[]'
          binding = 'org.jooq.postgres.extensions.bindings.CitextArrayBinding'
          includeTypes = '_citext'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Hstore'
          binding = 'org.jooq.postgres.extensions.bindings.HstoreBinding'
          includeTypes = 'hstore'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Hstore[]'
          binding = 'org.jooq.postgres.extensions.bindings.HstoreArrayBinding'
          includeTypes = '_hstore'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Inet'
          binding = 'org.jooq.postgres.extensions.bindings.InetBinding'
          includeTypes = 'inet'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.Inet[]'
          binding = 'org.jooq.postgres.extensions.bindings.InetArrayBinding'
          includeTypes = '_inet'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.IntegerRange'
          binding = 'org.jooq.postgres.extensions.bindings.IntegerRangeBinding'
          includeTypes = 'int4range'
          priority = '-2147483648'
        }
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.IntegerRange[]'
          binding = 'org.jooq.postgres.extensions.bindings.IntegerRangeArrayBinding'
          includeTypes = '_int4range'
          priority = '-2147483648'
        }

        // Depending on <javaTimeTypes/>, this may map to DateRange instead
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.LocalDateRange'
          binding = 'org.jooq.postgres.extensions.bindings.LocalDateRangeBinding'
          includeTypes = 'daterange'
          priority = '-2147483648'
        }

        // Depending on <javaTimeTypes/>, this may map to DateRange[] instead
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.LocalDateRange[]'
          binding = 'org.jooq.postgres.extensions.bindings.LocalDateRangeArrayBinding'
          includeTypes = '_daterange'
          priority = '-2147483648'
        }

        // Depending on <javaTimeTypes/>, this may map to TimestampRange instead
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.LocalDateTimeRange'
          binding = 'org.jooq.postgres.extensions.bindings.LocalDateTimeRangeBinding'
          includeTypes = 'tsrange'
          priority = '-2147483648'
        }

        // Depending on <javaTimeTypes/>, this may map to TimestampRange[] instead
        forcedType {
          userType = 'org.jooq.postgres.extensions.types.LocalDateTimeRange[]'
          binding = 'org.jooq.postgres.extensions.bindings.LocalDateTimeRangeArrayBinding'
          includeTypes = '_tsrange'
          priority = '-2147483648'
        }
```

See the configuration XSD andgradle code generation for more details.

6.18. Embeddable types

Only few SQL databases support ORDBMS extensions, such as user-defined types (UDTs). When working with strong domain semantics, it is often desired to wrap the primitive, technical database type (e.g. VARCHAR) in a more semantic type (e.g. com.example.Email).

Embeddable types are generated, synthetic data types that wrap one or more database columns in a generated org.jooq.EmbeddableRecord, as if the column group were an actual UDT.

This is particularly useful for embedded keys, to help with type safe comparison between PRIMARY KEY / FOREIGN KEY columns, specifically when the keys are composite keys.

Embeddable types are a very complex feature that has been added to jOOQ 3.14. A list of known issues for embeddable types is available here: #10527.

6.18.1. Configuration

Embeddable types can be specified in the code generator configuration as follows:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>

      <!-- For each type, one embeddable entry is required -->
      <embeddables>
        <embeddable>

          <!-- The optional catalog of the embeddable type (the catalog of the first matched table if left empty) -->
          <catalog/>

          <!-- The optional schema of the embeddable type (the schema of the first matched table if left empty) -->
          <schema>PUBLIC</schema>

          <!-- The name of the embeddable type -->
          <name>AUDIT</name>

          <!-- An optional, defining comment of an embeddable -->
          <comment>An audit record containing common audit fields.</comment>

          <!-- The name of the reference to the embeddable type. Defaults to <name/> if left blank -->
          <referencingName>AUDIT_REFERENCE</referencingName>

          <!-- An optional, referencing comment of an embeddable. Defaults to <comment/> if left blank -->
          <referencingComment>An audit record containing common audit fields.</referencingComment>

          <!-- A regular expression matching qualified or unqualified table names to which to apply this embeddable specification
               If left blank, this will apply to all tables -->
          <tables>.*</tables>

          <!-- A list of fields to match to an embeddable's attributes. Each field must match exactly one column in each matched
 table.
               A mandatory regular expression matches field names, whereas an optional name can be provided to define the embeddable
               attribute name. If no name is provided, then the first matched field's name will be taken -->
          <fields>
            <field><expression>CREATED_AT</expression></field>
            <field><expression>CREATED_BY</expression></field>
            <field><name>MODIFIED_AT</name><expression>MODIFIED_AT|CHANGED_AT</expression></field>
            <field><name>MODIFIED_BY</name><expression>MODIFIED_BY|CHANGED_BY</expression></field>
          </fields>
        </embeddable>

        <!-- A more minimal configuration might look like this -->
        <embeddable>
          <name>MONETARY_AMOUNT</name>
          <fields>
            <field><expression>AMOUNT</expression></field>
            <field><expression>CURRENCY</expression></field>
          </fields>
        </embeddable>
      </embeddables>
    </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()

      // For each type, one embeddable entry is required
      .withEmbeddables(
        new EmbeddableDefinitionType()

          // The optional catalog of the embeddable type (the catalog of the first matched table if left empty)
          .withCatalog("")

          // The optional schema of the embeddable type (the schema of the first matched table if left empty)
          .withSchema("PUBLIC")

          // The name of the embeddable type
          .withName("AUDIT")

          // An optional, defining comment of an embeddable
          .withComment("An audit record containing common audit fields.")

          // The name of the reference to the embeddable type. Defaults to <name/> if left blank
          .withReferencingName("AUDIT_REFERENCE")

          // An optional, referencing comment of an embeddable. Defaults to <comment/> if left blank
          .withReferencingComment("An audit record containing common audit fields.")

          // A regular expression matching qualified or unqualified table names to which to apply this embeddable specification
          // If left blank, this will apply to all tables
          .withTables(".*")

          // A list of fields to match to an embeddable's attributes. Each field must match exactly one column in each matched table.
          // A mandatory regular expression matches field names, whereas an optional name can be provided to define the embeddable
          // attribute name. If no name is provided, then the first matched field's name will be taken
          .withFields(
            new EmbeddableField()
              .withExpression("CREATED_AT"),
            new EmbeddableField()
              .withExpression("CREATED_BY"),
            new EmbeddableField()
              .withName("MODIFIED_AT")
              .withExpression("MODIFIED_AT|CHANGED_AT"),
            new EmbeddableField()
              .withName("MODIFIED_BY")
              .withExpression("MODIFIED_BY|CHANGED_BY")
          ),

        // A more minimal configuration might look like this
        new EmbeddableDefinitionType()
          .withName("MONETARY_AMOUNT")
          .withFields(
            new EmbeddableField()
              .withExpression("AMOUNT"),
            new EmbeddableField()
              .withExpression("CURRENCY")
          )
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {

      // For each type, one embeddable entry is required
      embeddables {
        embeddable {

          // The optional catalog of the embeddable type (the catalog of the first matched table if left empty)
          catalog {}

          // The optional schema of the embeddable type (the schema of the first matched table if left empty)
          schema = 'PUBLIC'

          // The name of the embeddable type
          name = 'AUDIT'

          // An optional, defining comment of an embeddable
          comment = 'An audit record containing common audit fields.'

          // The name of the reference to the embeddable type. Defaults to <name/> if left blank
          referencingName = 'AUDIT_REFERENCE'

          // An optional, referencing comment of an embeddable. Defaults to <comment/> if left blank
          referencingComment = 'An audit record containing common audit fields.'

          // A regular expression matching qualified or unqualified table names to which to apply this embeddable specification
          // If left blank, this will apply to all tables
          tables = '.*'

          // A list of fields to match to an embeddable's attributes. Each field must match exactly one column in each matched table.
          // A mandatory regular expression matches field names, whereas an optional name can be provided to define the embeddable
          // attribute name. If no name is provided, then the first matched field's name will be taken
          fields {
            field {
              expression = 'CREATED_AT'
            }
            field {
              expression = 'CREATED_BY'
            }
            field {
              name = 'MODIFIED_AT'
              expression = 'MODIFIED_AT|CHANGED_AT'
            }
            field {
              name = 'MODIFIED_BY'
              expression = 'MODIFIED_BY|CHANGED_BY'
            }
          }
        }

        // A more minimal configuration might look like this
        embeddable {
          name = 'MONETARY_AMOUNT'
          fields {
            field {
              expression = 'AMOUNT'
            }
            field {
              expression = 'CURRENCY'
            }
          }
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

The previous configuration, if matched correctly against relevant tables, produces the following org.jooq.EmbeddableRecord implementations (abbreviated):

```
public class AuditRecord extends EmbeddableRecordImpl<AuditRecord> {
    public AuditRecord(Timestamp createdAt, Timestamp modifiedAt, String createdBy, String modifiedBy) { /* ... */ }

    // Getters, setters
}
```

```
public class MonetaryAmountRecord extends EmbeddableRecordImpl<MonetaryAmountRecord> {
    public MonetaryAmountRecord(BigDecimal amount, String currency) { /* ... */ }

    // Getters, setters
}
```

Assuming we have a TRANSACTIONS table like this:

```
CREATE TABLE transactions (
  id BIGINT NOT NULL PRIMARY KEY,
  created_at TIMESTAMP NOT NULL,
  modified_at TIMESTAMP,
  created_by VARCHAR(100) NOT NULL,
  modified_by VARCHAR(100) NOT NULL,

  -- Other columns here
  amount DECIMAL(18, 2) NOT NULL,
  currency VARCHAR(10) NOT NULL
);
```

With the previous embeddable type definitions, we would get a table that looks like this:

```
public class Transactions extends TableImpl<TransactionsRecord> {
    // Field initialisations omitted for simplicity

    // The AUDIT embeddable and its physical fields that it represents
    public TableField<TransactionsRecord, Integer> ID;
    public TableField<TransactionsRecord, Timestamp> CREATED_AT;
    public TableField<TransactionsRecord, Timestamp> MODIFIED_AT;
    public TableField<TransactionsRecord, String> CREATED_BY;
    public TableField<TransactionsRecord, String> MODIFIED_BY;
    public TableField<TransactionsRecord, AuditRecord> AUDIT_REFERENCE;

    // The MONETARY_AMOUNT embeddable and its physical fields that it represents
    public TableField<TransactionsRecord, BigDecimal> AMOUNT;
    public TableField<DRecord, String> CURRENCY;
    public TableField<DRecord, MonetaryAmountRecord> MONETARY_AMOUNT;
```

Notice how the embeddable type is just an auxiliary column, which, by default, does not replace its underlying columns. In order to replace the underlying columns, use the <replaceFields/> flag

You can now work with the underlying columns, as always, or use the embeddable types instead:

```
create.insertInto(TRANSACTIONS)
      .columns(TRANSACTIONS.ID, TRANSACTIONS.AUDIT_REFERENCE, TRANSACTIONS.MONETARY_AMOUNT)
      .values(
            1,
            new AuditRecord(Timestamp.valueOf("2000-01-01 00:00:00"), null, "user", null),
            new MonetaryAmountRecord(new BigDecimal("20.00"), "EUR"))
      .execute();
```

These columns can be used in all types of statements, including e.g. SELECT, INSERT, UPDATE, DELETE

6.18.2. Overlapping embeddable types

A previous section explained how embeddable types work in general. In some cases, there's a risk of overlapping embeddable types, which can mainly happen when using embedded keys, but also in some other cases.

For example, when defining an embeddable for each one of these UNIQUE constraints:

```
CREATE TABLE order_item (
  order_id BIGINT NOT NULL REFERENCES order
  product_id BIGINT NOT NULL REFERENCES product,
  item_no INT NOT NULL,
  quantity INT NOT NULL,

  -- Each order_item has a unique-per-order item_no, which acts as a sequential number
  CONSTRAINT pk_order_item PRIMARY KEY (order_id, item_no),

  -- Each product can only have one order_item
  CONSTRAINT uk_order_item UNIQUE (order_id, product_id)
);
```

The two UNIQUE constraints overlap. If they are represented by an org.jooq.EmbeddableRecord, each (e.g. because of using the embedded keys feature), then both of the embeddable types will reference the ORDER_ID column. jOOQ will make sure that the ORDER_ID column is not generated twice in SQL statements, where this is forbidden, e.g. in INSERT statements:

```sql
INSERT INTO order_item (
  order_id,
  product_id,
  item_no,
  quantity
)
VALUES (
  12,
  15,
  1,
  10
);
```

```java
create.insertInto(ORDER_ITEM)
      .columns(
        ORDER_ITEM.PK_ORDER_ITEM,
        ORDER_ITEM.UK_ORDER_ITEM,
        ORDER_ITEM.QUANTITY
      )
      .values(
        new PkOrderItemRecord(12L, 15L),
        new UkOrderItemRecord(15L, 1),
        1
      )
      .execute();
```

Despite the value 15L having been provided twice, it is produced in the generated SQL query only once (the second copy of the value is ignored).

6.18.3. Field replacement

A previous section explained how embeddable types work in general. In most cases, in the presence of an embeddable type, the original fields are no longer interesting in most queries, so you will want to replace them by the embeddable.

Assuming we have again a TRANSACTIONS table like this:

```sql
CREATE TABLE transactions (
  id BIGINT NOT NULL PRIMARY KEY,
  create_date TIMESTAMP NOT NULL,
  modified_date TIMESTAMP,
  created_by VARCHAR(100) NOT NULL,
  modified_by VARCHAR(100) NOT NULL,

  -- Other columns here
  amount DECIMAL(18, 2) NOT NULL,
  currency VARCHAR(10) NOT NULL
);
```

With the previous embeddable type definitions, and a <replacesFields/> flag like this:

XML (standalone and maven)

```xml
<configuration>
  <generator>
    <database>
      <embeddables>
        <embeddable>
          <name>MONETARY_AMOUNT</name>
          <fields>
            <field><expression>AMOUNT</expression></field>
            <field><expression>CURRENCY</expression></field>
          </fields>
          <replacesFields>true</replacesFields>
        </embeddable>
      </embeddables>
    </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()
      .withEmbeddables(
        new EmbeddableDefinitionType()
          .withName("MONETARY_AMOUNT")
          .withFields(
            new EmbeddableField()
              .withExpression("AMOUNT"),
            new EmbeddableField()
              .withExpression("CURRENCY")
          )
          .withReplacesFields(true)
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      embeddables {
        embeddable {
          name = 'MONETARY_AMOUNT'
          fields {
            field {
              expression = 'AMOUNT'
            }
            field {
              expression = 'CURRENCY'
            }
          }
          replacesFields = true
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

... we would get a table that looks like this:

```
public class Transactions extends TableImpl<TransactionsRecord> {
    // Field initialisations omitted for simplicity

    // The AUDIT embeddable and its physical fields that it represents, but the fields are not accessible
    private TableField<TransactionsRecord, Integer> ID;
    private TableField<TransactionsRecord, Timestamp> CREATED_AT;
    private TableField<TransactionsRecord, Timestamp> MODIFIED_AT;
    private TableField<TransactionsRecord, String> CREATED_BY;
    private TableField<TransactionsRecord, String> MODIFIED_BY;
    public TableField<TransactionsRecord, AuditRecord> AUDIT_REFERENCE;

    // The MONETARY_AMOUNT embeddable and its physical fields that it represents , but the fields are not accessible
    private TableField<TransactionsRecord, BigDecimal> AMOUNT;
    private TableField<DRecord, String> CURRENCY;
    public TableField<DRecord, MonetaryAmountRecord> MONETARY_AMOUNT;
```

Not only are the fields no longer accessible (which means we cannot use them directly, nor do they get in the way with auto completion, etc.), but we also don't get the fields anymore when using SELECT * queries.

This flag is turned on when using embedded keys or embedded domains, where the original field reference is undesired.

6.18.4. Embedded keys

A very useful application of embeddable types are PRIMARY KEYS, UNIQUE constraints, and FOREIGN KEYS. There is are only few good use-case of joining two tables by two columns that are not the FOREIGN

KEY and its referenced PRIMARY / UNIQUE key. If two such columns are chosen, this is mostly because of a typo, or even because of misunderstanding the underlying schema.

You can turn on the feature like this:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>

      <!-- Use regular expressions to match the keys that should be replaced by embeddables. -->
      <embeddablePrimaryKeys>.*</embeddablePrimaryKeys>
      <embeddableUniqueKeys>.*</embeddableUniqueKeys>
    </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 regular expressions to match the keys that should be replaced by embeddables.
      .withEmbeddablePrimaryKeys(".*")
      .withEmbeddableUniqueKeys(".*")
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {

      // Use regular expressions to match the keys that should be replaced by embeddables.
      embeddablePrimaryKeys = '.*'
      embeddableUniqueKeys = '.*'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

This will automatically produce an embeddable type configuration for each PRIMARY KEY and/or UNIQUE key, as well as for each FOREIGN KEY referencing the PRIMARY KEY or UNIQUE key. The generated configuration could be written manually, but the configuration generation algorithm is not trivial when keys overlap, or when tables reference a "remote" primary key transitively, through a relationship table.

Applying this to our sample database, along with either well-designed constraint names (generated embeddables use constraint names), or a programmatic generator strategy, or a configurative matcher strategy, we might be getting org.jooq.EmbeddableRecord types like these:

```
// Both primary and foreign key produce the respective primary key record
Result<Record5<PkBookRecord, String, PkAuthorRecord, String, String>> result =
create.select(
            BOOK.PK_BOOK,
            BOOK.TITLE,
            BOOK.FK_BOOK_AUTHOR,
            AUTHOR.FIRST_NAME,
            AUTHOR.LAST_NAME)
      .from(BOOK)
      .join(AUTHOR)
      // This join compiles
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      // This wrong join wouldn't compile
      // .on(BOOK.LANGUAGE_ID.eq(AUTHOR.ID))
      .fetch();

for (Record5<PkBookRecord, String, PkAuthorRecord, String, String> record : result) {
    System.out.println("ID        : " + record.value1().getId());
    System.out.println("TITLE     : " + record.value2());
    System.out.println("AUTHOR_ID : " + record.value3().getId());
    System.out.println("FIRST_NAME: " + record.value4());
    System.out.println("LAST_NAME : " + record.value5());
}
```

Notice how:

- Each primary key produces an embeddable record type.
- Both primary key and foreign key columns reference the primary key record type.
- This means that only matching primary / foreign key columns can be compared in joins. It is not sufficient for them to be both of type java.lang.Integer

Composite keys

This feature is even more powerful when keys are composite! You no longer have to list each pair of join columns in your join predicates, just join by primary / foreign key embeddable type. Advantages are:

- You'll never forget a column again when joining by composite keys
- You'll never forget to refactor all your queries, in case you add / remove a column from a key. Just re-generate the jOOQ code, and the queries are automatically updated

The second bullet can also be achieved using implicit joins, or the synthetic ON KEY clause.

6.18.5. Embedded domains

A very useful application of embeddable types are DOMAIN types. A DOMAIN type is a combination of:

- A semantic type name
- A data type (or other domain type)
- A default value
- A CHECK constraint

The combination name/type is enough to describe a semantic type like EMAIL across your schema, much better than e.g. VARCHAR(10). With embedded domains, you can generate a Java type for each domain type, and have that automatically attached to all your columns.

You can turn on the feature like this:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>

      <!-- Use regular expressions to match the domains that should be replaced by embeddables. -->
      <embeddableDomains>.*</embeddableDomains>
    </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 regular expressions to match the domains that should be replaced by embeddables.
      .withEmbeddableDomains(".*")
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {

      // Use regular expressions to match the domains that should be replaced by embeddables.
      embeddableDomains = '.*'
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
As always, when regular expressions are used, they are regular expressions with default flags.

This will automatically produce an embeddable type configuration for each DOMAIN. For example, this schema:

```
CREATE DOMAIN email AS varchar(100);
CREATE DOMAIN year AS int;

CREATE TABLE user (
  id bigint NOT NULL PRIMARY KEY,
  name varchar(100) NOT NULL,
  email email NOT NULL,
  created year NOT NULL
);
```

The above would generate the following set of classes (simplified):

```
public class EmailRecord extends EmbeddableRecordImpl<EmailRecord> {
    public void setValue(String value) { ... }
    public String getValue() { ... }
    public EmailRecord() { ... }
    public EmailRecord(String value) { ... }
}

public class YearRecord extends EmbeddableRecordImpl<YearRecord> {
    public void setValue(Integer value) { ... }
    public Integer getValue() { ... }
    public YearRecord() { ... }
    public YearRecord(Integer value) { ... }
}
```

These classes are now referenced by embedded fields in the User table and UserRecord record (simplified):

```
public class User extends TableImpl<UserRecord> {
    public final TableField<UserRecord, Integer> ID;
    public final TableField<UserRecord, String> NAME;
    public final TableField<UserRecord, EmailRecord> EMAIL;
    public final TableField<UserRecord, YearRecord> CREATED;
}
```

With these generated fields, you can create semantically type safe queries:

```
create.insertInto(USER)
      .columns(USER.ID, USER.NAME, USER.EMAIL, USER.CREATED)
      .values(1, "domain_user", new EmailRecord("domain@user.com"), new YearRecord(2020))
      .execute();
```

6.19. 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.

6.20. 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

6.21. 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.

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.

6.22. 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)

```
<dependency>
    <!-- Use org.jooq                for the Open Source Edition
             org.jooq.pro            for commercial editions with Java 17 support,
             org.jooq.pro-java-11    for commercial editions with Java 11 support,
             org.jooq.pro-java-8     for commercial editions with Java 8 support,
             org.jooq.trial          for the free trial edition with Java 17 support,
             org.jooq.trial-java-11  for the free trial edition with Java 11 support,
             org.jooq.trial-java-8   for the free trial edition with Java 8 support

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-meta-extensions-hibernate</artifactId>
    <version>3.17.8</version>
</dependency>
```

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      properties {
        property {
          key = 'hibernate.physical_naming_strategy'
          value = 'org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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.

6.23. 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-3.16.0.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
<?xml version="1.0"?>
<information_schema xmlns="http://www.jooq.org/xsd/jooq-meta-3.16.0.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
<?xml version="1.0"?>
<information_schema xmlns="http://www.jooq.org/xsd/jooq-meta-3.16.0.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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle 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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.
This XML configuration can now be checked in and versioned, and modified independently from your live database schema.

6.24. 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)

```xml
<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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.

Additional properties

Additional properties include:

- logExecutedQueries: Whether queries that are executed by the DDLDatabase should be logged for debugging purposes and auditing purposes.
- logExecutionResults: Whether results that are obtained after executing queries by the DDLDatabase should be logged for debugging and auditing purposes.

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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  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 andgradle code generation for more details.

Ad-hoc SQL

You can provide additional SQL that is prepended to your scripts to initialise the DDLDatabase, in case it can interpret it. This can also be used ***instead of*** providing scripts for quick code generation testing use-cases:

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.ddl.DDLDatabase</name>
      <properties>
        <!-- Add additional SQL that is interpreted *before* any scripts -->
        <property>
          <key>sql</key>
          <value>create table t (i int primary key);</value>
        </property>

        <!-- The usual scripts -->
        <property>
          <key>scripts</key>
          <value>src/main/resources/database.sql</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(

        // Add additional SQL that is interpreted *before* any scripts
        new Property()
          .withKey("sql")
          .withValue("create table t (i int primary key);"),

        // The usual scripts
        new Property()
          .withKey("scripts")
          .withValue("src/main/resources/database.sql")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.

Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      name = 'org.jooq.meta.extensions.ddl.DDLDatabase'
      properties {

        // Add additional SQL that is interpreted *before* any scripts
        property {
          key = 'sql'
          value = 'create table t (i int primary key);'
        }

        // The usual scripts
        property {
          key = 'scripts'
          value = 'src/main/resources/database.sql'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle 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:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta-extensions</artifactId>
  <version>3.17.8</version>
</dependency>
```

6.25. LiquibaseDatabase: Code generation from Liquibase XML, YAML, JSON files

If you are using Liquibase, you will have defined your database as a set of Liquibase change sets, in XML, YAML, or JSON. That database definition is complete and self contained, and can easily be used as a source of meta information by the jOOQ code generator.

For example, the following database.xml script could be used (please refer to the liquibase documentation for YAML or JSON formats):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                      http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.8.xsd">
    <changeSet author="authorName" id="changelog-1.0">
        <createTable tableName="MY_TABLE">
            <column name="MY_COLUMN" type="TEXT">
                <constraints nullable="true" primaryKey="false" unique="false" />
            </column>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

In order to use the above as a source of jOOQ's code generator, you will simply need to set up your code generation configuration as follows:

XML (standalone and maven)

```xml
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.liquibase.LiquibaseDatabase</name>
      <properties>

        <!-- Specify the classpath location of your XML, YAML, or JSON script. -->
        <property>
          <key>scripts</key>
          <value>/database.xml</value>
        </property>

        <!-- Whether you want to include liquibase tables in generated output

             - false (default)
             - true: includes DATABASECHANGELOG and DATABASECHANGELOGLOCK tables -->
        <property>
          <key>includeLiquibaseTables</key>
          <value>false</value>
        </property>

        <!-- Properties prefixed "database." will be passed on to the liquibase.database.Database class
             if a matching setter is found -->
        <property>
          <key>database.liquibaseSchemaName</key>
          <value>lb</value>
        </property>

        <!-- The property "changeLogParameters.contexts" will be passed on to the
             liquibase.database.Database.update() call (jOOQ 3.13.2+).
             See https://www.liquibase.org/documentation/contexts.html -->
        <property>
          <key>changeLogParameters.contexts</key>
          <value>!test</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.liquibase.LiquibaseDatabase")
      .withProperties(

        // Specify the classpath location of your XML, YAML, or JSON script.
        new Property()
          .withKey("scripts")
          .withValue("/database.xml"),

        // Whether you want to include liquibase tables in generated output
        //
        // - false (default)
        // - true: includes DATABASECHANGELOG and DATABASECHANGELOGLOCK tables
        new Property()
          .withKey("includeLiquibaseTables")
          .withValue(false),

        // Properties prefixed "database." will be passed on to the liquibase.database.Database class
        // if a matching setter is found
        new Property()
          .withKey("database.liquibaseSchemaName")
          .withValue("lb"),

        // The property "changeLogParameters.contexts" will be passed on to the
        // liquibase.database.Database.update() call (jOOQ 3.13.2+).
        // See https://www.liquibase.org/documentation/contexts.html
        new Property()
          .withKey("changeLogParameters.contexts")
          .withValue("!test")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      name = 'org.jooq.meta.extensions.liquibase.LiquibaseDatabase'
      properties {

        // Specify the classpath location of your XML, YAML, or JSON script.
        property {
          key = 'scripts'
          value = '/database.xml'
        }

        // Whether you want to include liquibase tables in generated output
        //
        // - false (default)
        // - true: includes DATABASECHANGELOG and DATABASECHANGELOGLOCK tables
        property {
          key = 'includeLiquibaseTables'
          value = false
        }

        // Properties prefixed "database." will be passed on to the liquibase.database.Database class
        // if a matching setter is found
        property {
          key = 'database.liquibaseSchemaName'
          value = 'lb'
        }

        // The property "changeLogParameters.contexts" will be passed on to the
        // liquibase.database.Database.update() call (jOOQ 3.13.2+).
        // See https://www.liquibase.org/documentation/contexts.html
        property {
          key = 'changeLogParameters.contexts'
          value = '!test'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.
If you prefer not to work with the Liquibase ClassLoaderResourceAccessor and want to use the FileSystemResourceAccessor instead, you will need to provide a rootPath, starting from jOOQ 3.16, which depends on Liquibase 4 (or later).

XML (standalone and maven)

```
<configuration>
  <generator>
    <database>
      <name>org.jooq.meta.extensions.liquibase.LiquibaseDatabase</name>
      <properties>

        <!-- Specify the root path, e.g. a path in your Maven directory layout -->
        <property>
          <key>rootPath</key>
          <value>${basedir}/src/main/resources</value>
        </property>

        <!-- Specify the relative path location of your XML, YAML, or JSON script. -->
        <property>
          <key>scripts</key>
          <value>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.extensions.liquibase.LiquibaseDatabase")
      .withProperties(

        // Specify the root path, e.g. a path in your Maven directory layout
        new Property()
          .withKey("rootPath")
          .withValue("${basedir}/src/main/resources"),

        // Specify the relative path location of your XML, YAML, or JSON script.
        new Property()
          .withKey("scripts")
          .withValue("database.xml")
      )
    )
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      name = 'org.jooq.meta.extensions.liquibase.LiquibaseDatabase'
      properties {

        // Specify the root path, e.g. a path in your Maven directory layout
        property {
          key = 'rootPath'
          value = '${basedir}/src/main/resources'
        }

        // Specify the relative path location of your XML, YAML, or JSON script.
        property {
          key = 'scripts'
          value = 'database.xml'
        }
      }
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

Dependencies

Note that the org.jooq.meta.extensions.liquibase.LiquibaseDatabase class is located in an external dependency, which needs to be placed on the classpath of the jOOQ code generator. E.g. using Maven:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta-extensions-liquibase</artifactId>
  <version>3.17.8</version>
</dependency>
```

Additional dependencies may be required by liquibase, e.g. when using YAML. Please refer to the liquibase documentation about additional dependencies.

6.26. 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-3.16.0.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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    name = 'org.jooq.codegen.XMLGenerator'
  }
}
```

See the configuration XSD andgradle 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.

6.27. KotlinGenerator

jOOQ can generate Kotlin code instead of Java code, which allows for leveraging a few kotlin language features also in generated code.

In order to use the KotlinGenerator, simply place the following class reference into your code generation configuration:

XML (standalone and maven)

```
<configuration>
  <generator>
    <name>org.jooq.codegen.KotlinGenerator</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.KotlinGenerator")
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    name = 'org.jooq.codegen.KotlinGenerator'
  }
}
```

See the configuration XSD andgradle code generation for more details.
Most of the code generation configuration remains the same for as long as it's independent of the generation language. But there are a few kotlin specific configuration flags, which are documented below:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true. -->
      <implicitJoinPathsAsKotlinProperties>true</implicitJoinPathsAsKotlinProperties>

      <!--  Workaround for Kotlin generating setX() setters instead of setIsX() in byte code for mutable properties called
            <code>isX</code>. Default is true. -->
      <kotlinSetterJvmNameAnnotationsOnIsPrefix>true</kotlinSetterJvmNameAnnotationsOnIsPrefix>

      <!-- Generate POJOs as data classes, when using the KotlinGenerator. Default is true. -->
      <pojosAsKotlinDataClasses>true</pojosAsKotlinDataClasses>
    </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()

      // Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true.
      .withImplicitJoinPathsAsKotlinProperties(true)

      // Workaround for Kotlin generating setX() setters instead of setIsX() in byte code for mutable properties called
      // <code>isX</code>. Default is true.
      .withKotlinSetterJvmNameAnnotationsOnIsPrefix(true)

      // Generate POJOs as data classes, when using the KotlinGenerator. Default is true.
      .withPojosAsKotlinDataClasses(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Tell the KotlinGenerator to generate properties in addition to methods for these paths. Default is true.
      implicitJoinPathsAsKotlinProperties = true

      // Workaround for Kotlin generating setX() setters instead of setIsX() in byte code for mutable properties called
      // <code>isX</code>. Default is true.
      kotlinSetterJvmNameAnnotationsOnIsPrefix = true

      // Generate POJOs as data classes, when using the KotlinGenerator. Default is true.
      pojosAsKotlinDataClasses = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.28. ScalaGenerator

jOOQ can generate Scala code instead of Java code, which allows for leveraging a few Scala language features also in generated code.

In order to use the ScalaGenerator, simply place the following class reference into your code generation configuration:

XML (standalone and maven)

```
<configuration>
  <generator>
    <name>org.jooq.codegen.ScalaGenerator</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.ScalaGenerator")
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    name = 'org.jooq.codegen.ScalaGenerator'
  }
}
```

See the configuration XSD andgradle code generation for more details.
Most of the code generation configuration remains the same for as long as it's independent of the generation language. But there are a few Scala specific configuration flags, which are documented below:
XML (standalone and maven)

```
<configuration>
  <generator>
    <generate>

      <!-- Generate POJOs as case classes, when using the ScalaGenerator. Default is true. -->
      <pojosAsScalaCaseClasses>true</pojosAsScalaCaseClasses>
    </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 POJOs as case classes, when using the ScalaGenerator. Default is true.
      .withPojosAsScalaCaseClasses(true)
  )
```

See the configuration XSD andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // Generate POJOs as case classes, when using the ScalaGenerator. Default is true.
      pojosAsScalaCaseClasses = true
    }
  }
}
```

See the configuration XSD andgradle code generation for more details.

6.29. 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 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.17.8</version>

  <executions>
    <execution>
      <id>jooq-codegen</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>

  <configuration>
    ...
  </configuration>
</plugin>
```

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 with Java 17 support,
               org.jooq.pro-java-11    for commercial editions with Java 11 support,
               org.jooq.pro-java-8     for commercial editions with Java 8 support,
               org.jooq.trial          for the free trial edition with Java 17 support,
               org.jooq.trial-java-11  for the free trial edition with Java 11 support,
               org.jooq.trial-java-8   for the free trial edition with Java 8 support

           Note: Only the Open Source Edition is hosted on Maven Central.
                 Import the others manually from your distribution -->
      <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>
```

Please refer to the Maven documentation for more details.

6.30. 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-3.17.8.jar"/>
      <pathelement location="/path/to/jooq-meta-3.17.8.jar"/>
      <pathelement location="/path/to/jooq-codegen-3.17.8.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 17 support,
               org.jooq.pro-java-11    for commercial editions with Java 11 support,
               org.jooq.pro-java-8     for commercial editions with Java 8 support,
               org.jooq.trial          for the free trial edition with Java 17 support,
               org.jooq.trial-java-11  for the free trial edition with Java 11 support,
               org.jooq.trial-java-8   for the free trial edition with Java 8 support

           Note: Only the Open Source Edition is hosted on Maven Central.
                 Import the others manually from your distribution -->
      <groupId>org.jooq</groupId>
      <artifactId>jooq-codegen</artifactId>
      <version>3.17.8</version>
    </dependency>
    <!-- Add JDBC drivers and other required artifacts -->
  </dependencies>
</plugin>
```

6.31. Running the code generator with Gradle

Run generation with the Gradle plugin

We 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

Alternatively, programmatic configuration can be used

If you don't want to use the above third party plugin, 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-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support */
    compile 'org.jooq:jooq:3.17.8'

    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:3.17.8'
        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'))))
```

6.32. 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
- -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.jdbc.autoCommit (boolean): Whether the JDBC connection should be put in autocommit mode
- -Djooq.codegen.jdbc.initScript (string): A script to run after creating the JDBC connection, and before running the code generator
- -Djooq.codegen.jdbc.initSeparator (string): The separator used to separate statements in the initScript, defaulting to ";"
- -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
- -Djooq.codegen.target.packageName (string): The output package name for generated code
- -Djooq.codegen.target.directory (string): The output directory for generated code
- -Djooq.codegen.target.encoding (string): The output encoding for generated code
- -Djooq.codegen.target.locale (string): The output locale for generated code

In case of conflict between the above default value and a more concrete, local configuration, the latter prevails and the default is overridden.

7. Tools

These chapters hold some information about tools to be used with jOOQ

7.1. API validation using the Checker Framework or Error Prone

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:

```
<dependency>
  <!-- Use org.jooq                for the Open Source Edition
           org.jooq.pro            for commercial editions with Java 17 support,
           org.jooq.pro-java-11    for commercial editions with Java 11 support,
           org.jooq.pro-java-8     for commercial editions with Java 8 support,
           org.jooq.trial          for the free trial edition with Java 17 support,
           org.jooq.trial-java-11  for the free trial edition with Java 11 support,
           org.jooq.trial-java-8   for the free trial edition with Java 8 support

           Note: Only the Open Source Edition is hosted on Maven Central.
                 Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-checker</artifactId>
  <version>3.17.8</version>
</dependency>
```

... 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>

      <!-- Optionally, provide a default allowed dialect -->
      <arg>-J-Dorg.jooq.checker.dialects.allow=H2</arg>

      <!-- And/or a default required dialect -->
      <arg>-J-Dorg.jooq.checker.dialects.require=H2</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.

7.2. jOOQ Refaster

jOOQ Refaster is no longer supported or shipped with jOOQ 3.15+. You may still use jOOQ Refaster 3.13 and 3.14 with new versions of jOOQ, at your own risk.

7.3. jOOQ Console

The jOOQ Console is no longer supported or shipped with jOOQ 3.2+. You may still use the jOOQ 3.1 Console with new versions of jOOQ, at your own risk.

8. 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.

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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.

8.5. Embeddable

JPA knows the jakarta.persistence.Embeddable, which has an similar representation in jOOQ, also called embeddables. See also the manual's section on how to attach embeddables to generated code.

8.6. 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.

8.7. 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.

8.8. 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)

8.9. @OneToOne or @ManyToOne

In JPA, you may have a @ManyToOne mapping like this:

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    @Column
    private String title;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "language_id")
    private Language language;

    // [...]
}
```

In JPA, you get to specify whether such associations should be fetched eagerly or lazily. In jOOQ, this is irrelevant as you will always express your intent explicitly via a query. Nothing is ever done automatically.

From a mapping perspective, it can be convenient to nest a parent object in the child object (irrespective of whether this is a @OneToOne or @ManyToOne relationship). In jOOQ, such a mapping is always ad-hoc, on a per query basis, so it does not need to reflect your actual database model. For example, you can easily "nest" a Title in a Book. With jOOQ, nesting is done directly in SQL using ORDBMS features (native or emulated).

So, instead of having entity classes, you might have DTOs like these (note, we might not need to project the ID):

```
public record Language(String code, String description) {}
public record Book(String title, Language language) {}
```

The DTOs may be reusable or ad-hoc, per query, it's entirely up to you. And you map your query data into the above records as follows:

```
create.select(
           BOOK.TITLE,

           // Nested record here:
           row(

               // Implicit join here
               BOOK.language().CD,
               BOOK.language().DESCRIPTION

           // Ad-hoc converter here:
           ).mapping(Language::new))
      .from(BOOK)

      // Ad-hoc converter here
      .fetch(Records.mapping(Book::new));
```

The above mapping is completely compile time type safe.

Some of the jOOQ features used in this section are:

- ad-hoc conversion
- nested records
- nested table expressions (this can be convenient, but it is far more likely you project only what's strictly needed)
- implicit joins (these are optional to the task of nesting, but greatly simplify it)

8.10. @OneToMany or @ManyToMany

In JPA, you may have a @OneToMany mapping like this:

```
@Entity
public class Author {

    @OneToMany(mappedBy = "author")
    private Set<Book> items;

    // [...]
}
```

In JPA, you get to specify whether such associations should be fetched eagerly or lazily. In jOOQ, this is irrelevant as you will always express your intent explicitly via a query. Nothing is ever done automatically.

From a mapping perspective, it can be convenient to nest a set (or list, whatever) of child objects in the parent object (irrespective of whether this is a @OneToMany or @ManyToMany relationship). In jOOQ, such a mapping is always ad-hoc, on a per query basis, so it does not need to reflect your actual database model. For example, you can easily "nest" a set of Language values in a BookStore, as we'll see later. With jOOQ, nesting is done directly in SQL using ORDBMS features (native or emulated).

So, instead of having entity classes, you might have DTOs like these (note, we might not need to project the ID):

```
public record Book(String title) {}
public record Author(String firstName, String lastName, List<Book> books) {}
```

The DTOs may be reusable or ad-hoc, per query, it's entirely up to you. And you map your query data into the above records as follows:

```
create.select(
           AUTHOR.FIRST_NAME,
           AUTHOR.LAST_NAME,

           // Nested collection here:
           multiset(
               select(BOOK.TITLE)
               .from(BOOK)
               .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))

           // Ad-hoc converter here:
           ).convertFrom(r -> r.map(Records.mapping(Book::new))))
      .from(AUTHOR)

      // Ad-hoc converter here
      .fetch(Records.mapping(Author::new));
```

The above mapping is completely compile time type safe.

Some of the jOOQ features used in this section are:

- ad-hoc conversion
- MULTISET value constructor

9. Reference

These chapters hold some general jOOQ reference information

9.1. 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
- Ingres
- MariaDB
- Microsoft Access
- MySQL
- Oracle
- PostgreSQL
- Redshift
- SQL Server
- SQLite
- 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.

9.2. 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.

9.2.1. 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.

Hence, jOOQ currently doesn't explicitly support JDBC BLOB and CLOB data types. If you use any of these data types in your database, jOOQ will map them to byte[] and String instead. In simple cases (small data), this simplification is sufficient. In more sophisticated cases, you may have to bypass jOOQ, in order to deal with these data types and their respective resources.

9.2.2. BOOLEAN data type

The SQL standard and some databases 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.

9.2.3. 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

9.2.4. 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

9.2.5. 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.

9.2.6. XML data types

jOOQ supports a text representation of the standard SQL XML data type in the form of org.jooq.XML.

9.2.7. Spatial data types

jOOQ supports non-JDBC but standard SQL spatial data types, which come in two forms:

- org.jooq.Geometry
- org.jooq.Geography

9.2.8. 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.

9.2.9. 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 these SQL dialects:

- H2
- 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.

9.2.10. 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 -->
      <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
      .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 andprogrammatic code generation for more details.
Gradle

```
myConfigurationName(sourceSets.main) {
  generator {
    database {

      // Use this flag to force DATE columns to be of type TIMESTAMP
      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 andgradle code generation for more details.
For more information, please refer to the manual's section about custom data type bindings and forced types.

9.2.11. Domains

A DOMAIN is a specialisation of another data type, adding any of the following additional restrictions (depending on the database dialect):

- A DEFAULT value
- A NOT NULL constraint
- A COLLATION
- A set of CHECK constraints

9.3. 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.

```
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.

9.4. 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 https://jmock.org. These tests verify that various org.jooq.QueryPart implementations render correct SQL and bind variables correctly.

Integration tests

This is the most important part of the jOOQ test suites. Some 1500 queries are currently run against a standard integration test database. Both the test database and the queries are translated into every one of the 14 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 14 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.

9.5. Security

This section talks about a few common security related issues in jOOQ, which developers should be aware of.

9.5.1. 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.

9.5.2. 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.

9.5.3. 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.

9.5.4. 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.

9.6. 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.

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:

o 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
o 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.

9.7. 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.

9.7.1. 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).

9.7.2. 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.

9.7.3. 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.

9.7.4. 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.:

9.7.5. 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?

This blog post discusses the topic of the cost of useless surrogate keys in relationship tables, from a performance perspective.

9.7.6. 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?
);
```

9.7.7. 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.

(!) Our pattern transformation feature can auto detect bad queries for you, e.g. ***COUNT(*) > 0 style queries*** *or* ***COUNT(expr) > 0 style queries****.*

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...

9.7.8. 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, or use MULTISET or MULTISET_AGG to nest the collection directly in SQL, 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.

9.7.9. 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.

9.7.10. 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.

9.7.11. 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.

9.7.12. 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;
```

9.8. 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.

9.9. 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 jakarta.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 RecordHandler or RecordMapper