# Database Feature

This article introduces the database feature that allows incoming and outgoing connections to a common relational database system.

The "Database Feature", which is not part of the default Odysseus, must be installed before. See also How to install new features

**2017-11-22: Important: The database feature does not contain the cql parts anymore. If you want to use this, you will ne to install the "database.cql" feature after the "database" feature!**

There are two possibilities how to use a database connection. For the one hand, you can create a persistent connection and reuse this connection in other queries. On the other hand, you can directly create a connection for each query. First, we show how to create and drop persistent connections and show how to use them in PQL and StreamSQL(CQL). Afterwards, we show how to directly setup a connection using PQL.



# Important Notices

**For Oracle**: There was a bug (the Oracle driver said: "Locale not found") that was fixed in Version 1.0.0.14252. However,notice that this bug cannot be fixed by simply updating the Odysseus Studio application. So, you may have to install a fresh and new version of Odysseus Studio to fix the "Locale-Bug".

# Open and drop persistent database connections

This part describes how create and drop a persistent database connection. Such a connection is stored and can be reused by using an unique name so that you do not have to set up all parameters each time you want to access a database. There are two possibilites to create such a persistent database connection.

## Open a connection using preconfigured drivers

**2017-11-22: Important: The database feature does not contain the cql parts anymore. If you want to use this, you will ne to install the "database.cql" feature after the "database" feature!**

There are different drivers for different DBMS (see the list of supported databases) that can be used with a more confortable syntax. For example, the following query creates a connection with the name con1 to a MySQL server on localhost (this is an implicit setting of the driver) with a database called test:

```
#PARSER CQL
#TRANSCFG Standard
#QUERY
CREATE DATABASE CONNECTION con1 AS mysql TO test
```

To use another driver, you may use one of the following:

```
CREATE DATABASE CONNECTION con1b AS oracle TO test
CREATE DATABASE CONNECTION con1c AS postgresql TO test
```

**Note for oracle**: the last token ("test" in this example) is the SID.

You may also define the host and/or the port you want to connect to:

```
CREATE DATABASE CONNECTION con2 AS mysql TO test AT localhost : 3306
```

If no user nor password is used, the driver assumes the user "root" with no password. Since most databases need some credentials, this can be added as follows:

```
CREATE DATABASE CONNECTION con3 AS mysql TO test AT localhost : 3306 WITH USER dbuser PASSWORD dbpassword
```

Of course, it is also possible to omit the host and port:

```
CREATE DATABASE CONNECTION con4 AS mysql TO test WITH USER dbuser PASSWORD dbpassword
```

Notice, that the connection is not established until it used for the first time. Therefore, the correctness of the properties (database name, user, pass etc.) is not checked when the query is translated and installed. We call this a "lazy connection check". To force a check during the installation of the query, you have to disable the lazy connection check by using the NO_LAZY_CONNECTION_CHECK flag as the last parameter:

```
CREATE DATABASE CONNECTION con5 AS mysql TO test AT localhost : 3306 WITH USER dbuser PASSWORD dbpassword
NO_LAZY_CONNECTION_CHECK
```

To sum up, the last query means: "Open a connection to a MySQL server at localhost that is listen on port 3306 and save the connection under con5. Use the user dbuser and the password dbpassword and immediately check the connection property at compile time"

## Open a connection using JDBC

To allow additional drivers that may not have the common properties like host or user or password, there is alternativly a JDBC based interface. This also allows DBMS specific settings via the JDBC string/url.

For a MySQL (equal to the previous section), the following query creates a connection called jdbc1 to a local MySql server.

```
CREATE DATABASE CONNECTION jdbc1 JDBC jdbc:mysql://localhost:3306/test
```

Remember that the JDBC string may have some database specific contents. If the user is not part of the JDBC string, you may also add them directly to query:

```
CREATE DATABASE CONNECTION jdbc2 JDBC jdbc:mysql://localhost:3306/test WITH USER dbuser PASSWORD dbpassword
```

Furthermore, the lazy connection check (see previous section) may also be deactivated as follows:

```
CREATE DATABASE CONNECTION jdbc3 JDBC jdbc:mysql://localhost:3306/test WITH USER dbuser PASSWORD dbpassword
NO_LAZY_CONNECTION_CHECK
```

## Drop a connection

To remove a persistent query, you simply have to use the following command where conname corresponds to your connection name

```
DROP DATABASE CONNECTION conname
```

## Using a connection in PQL

### Reading from a database

If you want to read from a database, you may use the DATABASESOURCE operator. For an existing connection called "con1" we can read from the table called "main":

```
datastream = DATABASESOURCE({connection='con1', table='main'})
```

This query tries to fetch the schema from the table main. If the schema in a MySQL has, for example, the attributes "id, value" with the datatypes "int, varchar", this is mapped to an output schema "id, value" with the datatypes "integer, string" in Odysseus. If a database has a special data type that is not known by JDBC (see java.sql.Types), it is mapped to the Odysseus datatype called "Object". Since the fetching of the schema from the database opens a connection, you can explicitely define the schema alternatively by using the optional "attributes" paramter here:

```
datastream = DATABASESOURCE({connection='con1', table='main', attributes=[['id', 'INTEGER'],['val', 'STRING']]})
```

Since PQL allows to create views and names for reusing connections (see The Odysseus Procedural Query Language (PQL) Framework), you may use this syntax, for example to create a dedicated source entry.

```
datastream := DATABASESOURCE({connection='con1', table='main'})
```

Then you can use this in other queries, like in the following example:

```
example = PROJECT({attributes=['val']}, datastream)
```

Remember the special behevior, if you reuse an existing operator. If there is already an operator that reads from the database, a second operator would join this reading - and not beginning from the start of the table.

Since the reading from the database is done as fast as possible, you possibly want to slow down this. An optional paramter indicates the number of milliseconds that the operator should wait between each tuple that is read. For example, for reading each second (=1000 milliseconds):

```
datastream := DATABASESOURCE({connection='con1', table='main', waiteach=1000})
```

### Writing to a database

You can write a stream into a database by using the DATABASESINK operator. So, if you have for example a stream or an operator that is named "datastream", the DATABASESINK can be used as follows:

```
result = DATABASESINK({connection='con1', table='main'}, datastream)
```

This query opens connection con1 and writes the resulting data stream from the view/source datastream to the table main. This is, of course, simple PQL, so you may also use other operators instead of "datastream". If the table "main" does not exists, it is created before. Furthermore, notice, that the schema of the table main and the schema of the incoming "datastream" must be union compatible.

Since each tuple is just appended to an existing table, you can also drop the table when the query is started:

```
result = DATABASESINK({connection='con1', table='main', drop='true'}, datastream)
```

This makes it possible to create a new table with the according schema. Alternatively, you may just truncate (remove all entries but keep the table and its schema) the table:

```
result = DATABASESINK({connection='con1', table='main', truncate='true'}, datastream)
```

## Using a connection in StreamSQL (CQL)

### Reading from a database

To read from a database with StreamSQL, you first have to create a stream, which is similar to other CREATE-STREAM queries:

```
CREATE STREAM datastream(id INTEGER, value STRING) DATABASE con1 TABLE main
```

This uses connection con1 to open a connection to table main and reads its data. The given schema (in this case id integer and value string) must be compatible with table main. This means that the table main should have the attributes id and value, which should have suitable datatypes.

Like in PQL, reading from a database is done as fast as possible. So, if you want to slow down this, you can use the "each" parameter so that the reading waits a certain time between two reads. So if you want to read a tuple each second, you can use the following syntax:

```
CREATE STREAM datastream(id INTEGER, value STRING) DATABASE con1 TABLE main EACH 1 SECOND
```

The time specification (in this example 1 SECOND) is equal to the specification of time windows, so you may also use other time intervall like "200 MILLISECONDS", "10 SECONDS", "5 MINUTES" or even "1 WEEK"

After creating the stream you can access the stream as usual, e.g.:

```
SELECT * FROM datastream WHERE...
```

### Writing to a database

For writing into a database, you have to create a sink (according to create a source via "create stream")

```
CREATE SINK dbsink AS DATABASE con1 TABLE example
```

This query uses connection con1 to create a sink that writes data into the table example and is called dbsink. If the table example does not exist, it will be created. However, if there is already a table called "example", you can drop the table, so that a new table is created each time:

```
CREATE SINK dbsink AS DATABASE con1 TABLE example AND DROP
```

Alternatively, you may just truncate (remove all entries but keep the table and its schema) the table:

```
CREATE SINK dbsink AS DATABASE con1 TABLE example AND TRUNCATE
```

Now, you can use the stream-to statement to write the outgoing stream of an usual select statement into the sink:

```
STREAM TO dbsink SELECT * FROM samplestream
```

Remember the union compatibility, because the outgoing schema of "SELECT * FROM samplestream" that is written into "dbsink" must be union-compatible with the table of "dbsink", which is the schema of example in this case.

# Direct Database Connections

With PQL you can also run direct connections without using persistent connections as described above. Like in the previous section, you have two possibilities to create connections: preconfigured drivers and JDBC strings. These possibilities are equal for DATABASESOURCE and DATABASESINK.

## Direct reading with preconfigured driver

If you want to define a preconfigured dbms (see also the above section), you can use existing drivers (at the moment "mysql", "postgresql" and "oracle"). In addition to the type, you have at least to define the database where you want connect to. Furthermore, you can or have to use all other operators of DATABASESOURCE (see above), but you don't need the "connection" parameter for the connetion name. So, if you want to read the table "main" from a MySQL database called "test", this looks like the following query:

```
datastream = DATABASESOURCE({table='main', type='mysql', db='test'})
```

Since the host is implicitely localhost or the user is root, you may - according to the create-database-connection statement - also define additional parameters, e.g. the user, the password, the host or the port:

```
datastream = DATABASESOURCE({table='main', type='mysql', db='test', user='dbuser', password='dbpassword',
host='localhost', port=3306})
```

See also the "Reading from a database" for PQL above for the attributes parameter, which is also available, because this query would also fetch the schema from the database - which invokes a connection during the translation of the query.

## Direct reading with JDBC

You can also use a JDBC string instead of a preconfigured driver, for example, if you want to use special properties:

```
datastream = DATABASESOURCE({table='main', jdbc='jdbc:mysql://localhost:3306/test'})
```

While using the jdbc parameter, you can also use the user and password parameter for setting up the credentials:

```
datastream = DATABASESOURCE({table='main', jdbc='jdbc:mysql://localhost:3306/test', user='dbuser',
password='dbpassword'})
```

### Direct writing with preconfigured driver

The DATABASESINK works very similar to the DATABASESOURCE. So, you also have to use at least the parameters "type" and "db" instead of the "connection" parameter for the connection name. So, you can directly write the output of "otherops" into the table "main".

```
datastream = DATABASESINK({table='main', type='mysql', db='test'}, otherops)
```

Of cource, you can also use the drop or truncate parameter of DATABASESINK:

```
datastream = DATABASESINK({table='main', drop='true', type='mysql', db='test'}, otherops)
```

Furthermore, according to the source, there are also additional parameters for user, password, host and port:

```
datastream = DATABASESINK({table='main', type='mysql', db='test', user='dbuser', password='dbpassword',
host='localhost', port=3306}, otherops)
```

### Direct wrtining with JDBC

You can also use a JDBC string instead of a preconfigured driver, for example, if you want to use special properties. For wrting the stream otherops into the table "main":

```
datastream = DATABASESINK({table='main', jdbc='jdbc:mysql://localhost:3306/test'}, otherops)
```

While using the jdbc parameter, you can also use the user and password parameter for setting up the credentials:

```
datastream = DATABASESINK({table='main', jdbc='jdbc:mysql://localhost:3306/test', user='dbuser',
password='dbpassword'}, otherops)
```

# Datatype Mapping

This part introduces the concept of datatype mapping between the relational database system and Odysseus. Since you have to define a union-compatible schema between your database system and Odysseus (e.g. if you want to read a table without fetching the schema), the different definitions of datatypes may be confusing so that I give a short introduction.

Odysseus has different Datatypes (see Data Types for details) like Integer or String while e.g. a MySQL uses int and varchar for the same types. Furthermore, there is the JDBC driver between the database, which also uses other datatypes. So, Odysseus has a mapping that maps all (default) datatypes (called SDFDatatype) to appropriate datatypes from JDBC (see java.sql.Types). The following table shows how Odysseus maps a database data type to a Odysseus datatype:

| JDBC Types | SDFDatatypes |
|---|---|
| Array | Object |
| BigInt | Long |
| Binary | Long |
| Bit | Boolean |
| Blob | Object |

| | |
|---|---|
| Boolean | Boolean |
| Char | String |
| Clob | Object |
| Datalink | Object |
| Date | Date |
| Decimal | Integer |
| Distinct | Object |
| Double | Double |
| Float | Float |
| Integer | Integer |
| Java Object | Object |
| Long NVarChar | String |
| Long VarBinary | Long |
| Long VarChar | String |
| NChar | String |
| NClob | Object |
| Null | Object |
| Numeric | Double |
| NVarChar | String |
| Other | Object |
| Real | Float |
| Ref | Object |
| RowId | Object |
| SmallInt | Integer |
| SQLXML | String |
| Struct | Object |
| Time | Timestamp |
| Timestamp | Timestamp |
| TinyInt | Integer |
| VarBinary | Long |
| VarChar | String |

you may see, for example, that a TINYINT of JDBC is mapped to an integer in Odysseus. Therefore, if the DATABASESOURCE reads a table with datatypes BIGINT and VARCHAR, therse datatypes are mapped to LONG an STRING in Odysseus. Since this mapping is used when reading from the database, there is also a direction how Odysseus transforms its datatypes to JDBC-datatypes for writing data:

| SDFDatatypes | JDBC Types |
|---|---|
| Boolean | Boolean |
| Byte | Binary |
| Date | Date |
| Double | Double |
| End Timestamp | BigInt |

| | |
|---|---|
| Float | Float |
| Integer | Integer |
| Long | BigInt |
| Object | Object |
| Point in Time | BigInt |
| String | VarChar |
| Timestamp | BigInt |
| Start Timestamp | BigInt |

For example, if the DATABASESINK operator creates a new table to insert data, it uses, e.g. Types.BIGINT for a LONG or any timestamp.

Although most of the usual datatypes are comprehensible mapped, you could look up how the specific JDBC-Driver maps the DBMS specific datatypes to a JDBC datatype. For Oracle for example, you may look here: http://docs.oracle.com/javase/1.5.0/docs/guide/jdbc/getstart/table8.7.html

# Supported databases

The following databases are supported:

- MySQL
- PostgreSQL
- Derby
- ~~HSQL~~ (currently not supported)
- Oracle