Design of a Portable SQL Wire Protocol

by

Stefan Burnicki

Master Thesis

22nd September 2015

Area of Studies	Computer Science
Matriculation Number	3863082
Principal Supervisor	Prof. Dr. Torsten Grust
Co-Supervisor	Prof. Dr. Klaus Ostermann

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Ort, Datum	Unterschrift

Abstract

Today the common way to access relational database management systems (DBMSs) from applications is to use either the Open Database Connectivity (ODBC) application programming interface (API) or the Java Database Connectivity (JDBC) API. Both have been designed to be database neutral, so they can be used for different DBMSs in the same way. Since also the common database language SQL is standardized, the whole client code could ideally be portable.

However, in contrast to web servers or email servers, the wire protocol used by the different DBMSs is not standardized, only the API and the language are. Instead, ODBC and JDBC require specific drivers to be installed on the client which implement the vendor specific wire protocol for communication with the DBMS.

The intention of this thesis is to develop a portable SQL wire protocol which allows a client to be completely database neutral and use the same code to access different DBMSs. Therefore, a new protocol called SQP is designed in this thesis which bases on modern technologies like JSON, MessagePack, and WebSocket. The focus of the protocol is to be easily implementable and usable, while supporting the most important database features like query execution, transaction management, and cursors. A major challenge is the encoding of the actual data since all DBMSs provide slightly different sets of supported data types.

To show that SQP works as intended it is implemented as a Java client API and an SQL proxy server. The idea of the proxy server is to delegate database operations to any DBMS through exchangeable backends. Therefore a client should be able to access different databases with the same code, just by using different backends. To show this, two backend implementations are provided: One for the popular open source database PostgreSQL, and one the proprietary closed-source database Transbase.

With simple experiments it's shown how the implementations can be used for both database independent and database specific operations. To demonstrate that SQP is also easily usable in other programming languages, a simple JavaScript example will be introduced before a conclusion about the thesis is drawn.

Zusammenfassung

Heutzutage werden meist die Programmierschnittstellen (APIs) Open Database Connectivity (ODBC) und Java Database Connectivity (JDBC) verwendet, um auf Datenbankmanagementsysteme (DBMSs) zuzugreifen. Beide APIs sind standardisiert und daher datenbankneutral, sodass man verschiedene DBMSs auf die gleiche Art und Weise ansprechen kann. Da die Datenbanksprache SQL ebenfalls standardisiert ist, könnte der gesamte Client-Programmcode portabel und nicht datenbankspezifisch sein. Allerdings ist im Gegensatz zu Webservern oder EMail-Servern das eigentliche Protokoll zur Kommunikation mit DBMSs nicht standardisiert. Deshalb müssen auf jedem Client spezielle Treiber für ODBC und JDBC installiert werden, welche die datenbankspezifischen Protokolle implementieren.

Im Rahmen dieser Masterarbeit wird deshalb des neue Protokoll SQP konzipiert, welches portabel sein soll, sodass Clients komplett datenbankneutral sein können. Dabei soll SQP möglichst einfach implementierbar und benutzbar sein und trotzdem alle wichtigen Datenbank-Features wie die Ausführung von Abfragen, Steuerung von Transaktionen oder "Cursors" unterstützen. Dazu setzt SQP auf aktuelle Technologien wie JSON, MessagePack und WebSocket. Eine wesentliche Herausforderung ist dabei die Kodierung der eigentlichen Daten in Standard-Typen, da alle DBMSs etwas unterschiedliche Datentypen unterstützen.

Um zu zeigen, dass SQP auch wirklich funktioniert, wurde es in Form einer Java Client API und eines SQL Proxy Servers implementiert. Die Idee hinter dem Proxy Server ist es, Datenbankoperationen an die DBMSs durch austauschbare "Backends" weiterzuleiten. Folglich sollte ein Client auf verschiedene Datenbanken zugreifen können, indem einfach die Backends des Servers ausgetauscht werden. Dazu wurden zwei Backends implementiert: Eins für die beliebte Open Source-Datenbank PostgreSQL und eins für die proprietäre Datenbank Transbase.

An einfachen Beispielen wird gezeigt, wie die Implementierung sowohl für datenbankunabhängige als auch für datenbankspezifische Operationen verwendet werden kann. Außerdem wird ein JavaScript-Beispiel vorgestellt, um zu demonstrieren, dass man SQP auch einfach aus anderen Programmiersprachen heraus benutzen kann. Zuletzt wird ein Fazit aus der gesamten Arbeit gezogen.

Thanks to
Dr. Markus Rothmeyer being the industrial advisor of this thesis,
Dr. Christian Roth and Transaction GmbH for
the insights into Transbase and related advices,
and **Thomas Uhl** for having the original idea for this thesis.

Contents

Acronyms

API application programming interface

BCD Binary Coded Decimal

BLOB Binary Large Object

CLI Call Level Interface

CLOB Character Large Object

DBMS database management system

DCL Data Control Language

DDL Data Definition Language

DML Data Manipulation Language

DRDA Distributed Relational Database Architecture

JD Julian Date

JDBC Java Database Connectivity

JSON JavaScript Object Notation

JSR Java Specification Request

JVM Java Virtual Machine

LOB Large Object

ODBC Open Database Connectivity

RPC Remote Procedure Call

SEQUEL Structured English Query Language

TLS Transport Layer Security

TLV Type-Length-Value

UTC Coordinated Universal Time

XML Extensible Markup Language

1 | Introduction

Web servers support HTTP and HTTPS, so any web browser is able to access content on any web server. File servers implement FTP, because there are dozens of matured clients that people use for sending and receiving files. Similarly, mail servers have interfaces for SMTP, POP3, and IMAP, because that's what all mail clients use for e-mail exchange. By using common protocols, a wide range of servers and clients can be developed independently without bothering about each other. This is different for database management systems (DBMSs), which are commonly simply referred to as "databases".

While all common relational databases understand SQL, the standard language for database queries, there is no standard protocol which can be used to communicate this query and retrieve the results. Instead, and in contrast to all of the examples mentioned above, only the application programming interface (API) to access databases has been standardized.

Prominent implementations of this standard, called Call Level Interface (CLI), are Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). Also Microsoft developed a similar API called ADO.NET. Today, these APIs are still the common choice for most clients to access relational databases, while the actual protocol used for sending the data over the wire highly differs between the separate database products.

1.1. Motivation

On May 29, 1995 there was a "SQL reunion" with people who worked in the early field of relational databases. A transcript of this reunion was published [McJ97]. One of the attendees was Jim Gray, a developer of IBM System R, the first relational DBMS. During the discussion of ODBC he said:

In fact, ODBC has no [format and protocol]; it's a procedure call, and then what happens underneath is a mystery, magic. In fact, what happens underneath is a driver from one or another vendor. This is a terrible situation unless there is only one kind of client, and only one version of each server, because then you just get the particular thing; otherwise you end up with an N-squared problem. ([McJ97, p. 109])

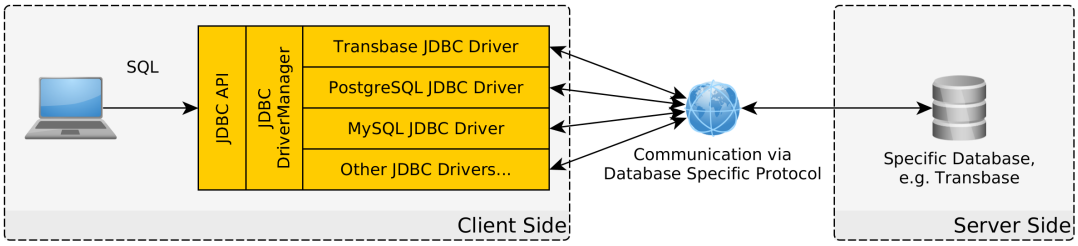

Figure 1.1.: The current situation with JDBC in order to access a database.

Figure 1.1 describes this situation for JDBC: For each existing database, there exists one driver that was specifically implemented for Java and the vendor's database. Now there are at least ODBC and ADO.NET to cover programming languages like C, C++, and C#, all of which also need database specific drivers.

So for an optimal database support, there would be the need of one driver per language and database, which is exactly what Jim Gray feared in 1995. Of course, given today's variety of programming languages, vendors cannot handle this. So another approach is to provide language specific wrapper implementations for ODBC. This way for example a python application can talk to a database through ODBC.

However, either approach increases the complexity of the client which eventually not only affects the vendor, but also the application developer, who has to understand and deal with this stack, and probably even the client who needs to install all necessary libraries and drivers.

The whole problem arises from the fact that only the language and the API have been standardized, but not the protocol which is actually used for the communication. This way, the client always needs to have this database specific implementation in one or another form. There are existing approaches concerning this matter, like Distributed Relational Database Architecture (DRDA) which was already developed by IBM in 1988 to 1994 but couldn't gain broad acceptance. Some of these existing approaches are briefly discussed in section 6.1.

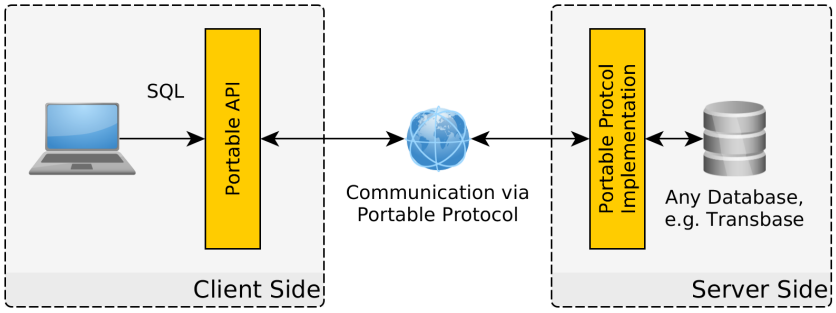

Figure 1.2.: The ideal situation is a standard protocol for communication with relational databases.

A new approach for this problem is therefore proposed in this thesis: A wire protocol is designed and implemented, which is portable and can be used for the actual communication between the client and a database. So similarly to web servers or mail servers, the clients should work independently from the relational database that is in use on the other end.

The situation with such a protocol and implementations in both client and server is shown in Figure 1.2. The servers and clients would just need to implement an interface for this protocol and wouldn't need to care about the details of the communication partner. And if this protocol was very easy to use, a complex client API wouldn't be necessary at all for simple communication.

```
query = "INSERT INTO weather (city, temp_lo, temp_hi, date) " +
        "VALUES ('Stuttgart', 13, 28, '2015-08-21')";
websocket = new WebSocket("ws://localhost:8080/");
websocket.onopen = function(e) {
    websocket.send('H' + JSON.stringify({database : "testdb"}));
    websocket.send('S' + JSON.stringify({query: query}));
    websocket.close();
};
```

Listing 1.1: A minimal working example of pure JavaScript inserting data into a database using the new protocol.

The protocol and its implementation proposed in this thesis are able to achieve this. Listing 1.1 shows a very minimal, yet complete and working example of how pure JavaScript directly uses the protocol to insert data into a database. Without this protocol, this wouldn't be possible with JavaScript at all, especially not without any client library, and probably not with such a few lines of code. Also, the type of database is not

mentioned in the code. In fact, for standard operations like this the database operating in the background is completely exchangeable, without any difference for the client.

1.2. Scope and Objectives

There are two main objectives of this thesis: The design of a protocol for unified database communication, and its implementation in the form of an SQL proxy server and a modern API.

Software is popular if many people tend to use it. And in this case, these people are software developers that use the protocol and the implementation to access a database. Therefore it should

- be easy to implement
- be easy to use
- support all common database operations
- work independently from a specific database
- allow database specific operations
- be usable for real-life scenarios
- be easily extensible

The new protocol is called SQP which can be interpreted as “Simple Query Protocol” or “Structured Query Protocol”, a reference to “Structured Query Language” which is the old meaning of SQL, before it was stated to be a self-contained name.

To assure that SQP works with different databases, it should get implemented in a way that it can be used with at least two common databases. For this thesis the open source database PostgreSQL and the closed source database Transbase are used. It would be ideal to directly implement the protocol in these databases, however adding a protocol to two databases, that were never meant to support more than one protocol can quickly get a very elaborate task.

So instead, the protocol gets implemented in form of a “SQL proxy server” as depicted in Figure 1.3. The idea is to have a server that implements SQP and delegates the actual database operations to any relational database through a “backend”. In this case, backends

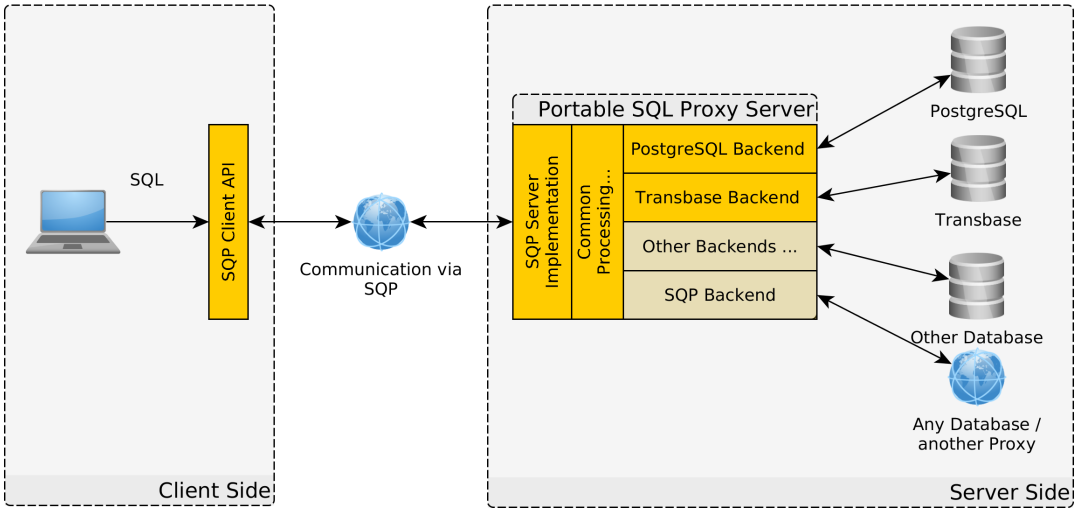

Figure 1.3.: The database communication with a proxy server that understands SQP and different databases as possible backends.

for PostgreSQL and Transbase are implemented to show that the database is exchangeable while the client code stays the same.

The backend can be compared to the JDBC drivers. However, one important difference is that the backend only needs to be developed once for all clients, not per language as it's the case with existing technology. The SQL proxy server, which will also be referred to as "proxy" in this thesis, can get very useful in environments to abstract the application from the actual database. It could also provide additional functionality, as suggested in 6.3. So by implementing the proxy, the usability of SQP can be shown without modifying the databases themselves, while creating useful software.

As the scope of this thesis is limited, it's very important to focus on the essentials to achieve these objectives. This means that protocol basics like authentication, encryption, and compression will be considered in design of SQP, but not implemented. These are all complex matters by themselves that are not directly related to the execution of database operations.

While the execution of SQL queries is obviously an operation that has to be supported by a database protocol, this is different for other operations. The reason is that many operations can be expressed on both the language level via SQL and the protocol level. This gets clear when the implementation of two database protocols are compared as in section 2.4.

As a consequence the support for database independent operations is strictly oriented towards what common client applications do with the database. These are for example operations around modifying and receiving data.

Other operations, like data model administration, user management, or data manipulation via procedures, are highly dependent on a database's feature set and SQL dialect. So support for these operations won't be part of the protocol, but remain to the capabilities of the database's language level, which is left untouched by the protocol.

1.3. Structure of the Thesis

First this thesis will start with the introduction of some necessary background information in chapter 2. This includes a short description of important database related information in general, the DBMSs PostgreSQL and Transbase, and some protocol basics.

The design of the SQP protocol is the essence of chapter 3. Therefore this chapter includes more detailed requirements for the protocol, the protocol's basics, message flow, encoding and supported data types.

After this, the implementation of the protocol basics, the proxy server with database specific backends, and the client API is described and discussed in chapter 4.

In chapter 5 the support for important database operations is evaluated by discussing some experiments. It also provides an example that compares the JavaScript and Java implementation of a simple SQP application.

Last but not least, related work is shortly discussed in chapter 6, before a conclusion is drawn for this thesis and an outlook on the future use of this work is given.

2 | Background

Although thinking outside the box is always a good thing when developing new creative approaches, it's necessary to get to know the basic matter in order not to make well-known mistakes or forget essentials. So before starting with anything else, the existing approaches of communication between client and database are examined in detail. Then the databases PostgreSQL and Transbase are introduced and compared, to see how both the feature set and protocol can differ for two databases. Lastly, basics of protocol design, data serialization and related contemporary technologies are introduced.

2.1. Communication with Relational Databases

Databases are a complex topic with many access points and research fields, so it should be clearly emphasized what's important for this project: the communication with relational databases.

Relational databases are the oldest and most common database category. There are also other categories like object-oriented, or (semi-)structured databases, but these may work completely differently. In relational databases data is stored in *relations*, which are often referred to as tables. A relation consists of *attributes*, which are seen as the columns of the table, and *tuples*, which conform to rows. A tuple contains data for each existing attribute [SE07, p. 68f].

Databases offer lots of important features when it comes to performance, data transformation, and data consistency. However, actual database features are barely touched by this thesis. Instead, the focus is on communication: In order to use the database an application needs to connect to it, express its needs, and retrieve the results.

2.1.1. The Database Language SQL

SQL is the de facto standard language to talk to DBMSs. It has its origins in the early 1970s, when its predecessor Structured English Query Language (SEQUEL) was developed by IBM [KKH08, p. 1]. In 1986 ANSI published the first standard for SQL, which was then adopted by ISO. Multiple versions of the standard have been published since, all incorporating more functionality and commands [KKH08, p.10].

The SQL standard from 1992 [Int92], also called SQL-92, was an important milestone and many databases attempted to implement it. However, SQL-92 was already pretty voluminous, so databases only conformed to the "Entry" level, which is the lowest of the three levels of conformance defined by the standard. Since 1999, the standard was separated into multiple "packages", where only one package contains the core functionality while the others are optional. This made it easier for databases to conform to defined parts of the standard [Gro14, p. 2018].

The commands in the SQL language can be split into three subsets:

Data Manipulation Language (DML) includes the insertion, deletion, and update of data, as well as retrieving data from the database.

Data Definition Language (DDL) contains actions for creating, modifying, and deleting tables, as well as managing data constraints.

Data Control Language (DCL) is mostly about access control, database monitoring, and user privileges. [SE07, p. 114]

While DDL and DCL are important for database administrators, they are of minor importance for many client applications. Applications often work with fixed data models and specific users, and care about the data only.

Although the SQL standard exists, different dialects emerged that vary from vendor to vendor, as functionality often needed to be added to DBMSs before they were part of the standard [KKH08, p.14f]. Especially DDL, DCL, but also elaborate DML commands are often dialect-prone. This is different for the basic DML tasks implemented in many applications: Commands like `INSERT`, `UPDATE`, or `SELECT` with `JOIN` and `GROUP BY` clauses are already part of the "Entry" level of SQL-92, thus implemented by many databases. This is important, as it allows applications to use SQL commands that work with different databases.

2.1.2. SQL Data Types

The SQL standard also contains a number of standard data types. However, all DBMSs have their own set of types that are really supported [DS10, p. 4ff]. Most databases support the types defined in SQL-92. These are namely: `CHARACTER`, `CHARACTER VARYING`, `BIT`, `BIT VARYING`, `NUMERIC`, `DECIMAL`, `INTEGER`, `SMALLINT`, `FLOAT`, `REAL`, `DOUBLE PRECISION`, `DATE`, `TIME`, `TIMESTAMP`, and `INTERVAL` [Int92, p. 19]. In practice there are even some more data types that became common for databases. The meaning of these types and the impact of having both database specific and standard data types is discussed in section 3.4.

2.1.3. The SQL Call Level Interface

To send SQL queries and related data to a database, a protocol and an API are required, which most database vendors develop on their own. To unify the database access, a standard API concept called Call Level Interface (CLI) was developed and published in 1995 [Int95]. It's part of the official SQL standard since.

The CLI contains more concepts than executing SQL statements and retrieving the results. So in order to design a protocol that supports all common database operations, it's worth taking a closer look at what else is part of the CLI.

ODBC and JDBC

The popular implementations of the CLI are ODBC (C, C++, Fortran) and JDBC (Java, JVM languages in general) which are today the standard for accessing databases. Both work very similar: A *DriverManager* is used to load a specific database *Driver* which implements the vendor-specific access to the DBMS. For JDBC, there are four common types of database drivers:

Type 1 drivers use a bridging technology to an existing ODBC driver. While this saves effort for vendors with existing ODBC drivers, this approach requires a separate installation of the ODBC driver on each target system and might be less efficient than the other driver types.

Type 2 drivers wrap the native database API which is often written in C or C++. Also this approach requires the installation of additional software on the client and might be slower since wrapping an API often requires data conversion.

Type 3 drivers communicate with a middleware application which could be database independent. So the actual native calls to a database are executed on the server side by the middleware.

Type 4 drivers implement the native database protocol directly, for example by using TCP/IP sockets.

Type 4 drivers are best for direct database access, as they avoid the wrapping effort and the need of additionally installed software [Ree00, p. 30f].

SQP is not a contradiction to a CLI implementation, but could be implemented as a driver instead. In JDBC this would probably be a type 4 driver, because it's implementing a database protocol. However, it can be also seen as some kind of type 3 driver with respect to the SQL proxy server, which could also be seen as some type of middleware.

Cursor Representation

Cursors are a fundamental concept of SQL that is also represented in the CLI. A cursor is a named pointer to a row of a result table of a query. It can be used to receive this row, but also to update or delete it [Com95, p. 38]. Cursors are important for example if the application iterates over a huge result set. Without cursors, the application would need to receive and buffer all data rows at once.

The representation of a cursor is called *result set* in the CLI. It can be used to iterate over all results while it implicitly fetches new rows from the database in the background. If the cursor is defined to be *scrollable*, the result set can also do other cursor movements, as positioning the cursor on a specific row or fetching the previous one. By default the cursor is forward-only so the result set can only fetch next rows [Com95, p 50].

Query Execution

SQL commands can be either executed directly or in two phases. In the two phase model they are first prepared and then executed. This model should be used whenever a the statement gets executed more then once, because then the database only needs to compile and optimize the query once after the preparation step [Com95, p. 44].

```sql
INSERT INTO employees (firstname, lastname, picture, birthdays)
VALUES (?,?,?,?)
```

Listing 2.1: An SQL command with placeholders for dynamic parameters.

The commands might need data which can either be included as a literal in the command itself, or by using dynamic parameters. Listing 2.1 shows a command that uses question marks as placeholders for the dynamic parameters. The data then needs to be bound to the different fields via CLI functions, before the query is executed [Com95, p. 46f].

Binding data separately has a number of advantages. Obviously, data might not be always efficiently representable as a literal. So if the *picture* attribute of the previous example represents binary data, it would be necessary to use a text encoding like base64 or hexadecimal to send it.

Another benefit is security with untrusted data: If the input came from a user with mean intents, a naive application could take the input "'); DROP TABLE employees; - -" and use it as a date literal in a query string. This could result in a query like Listing 2.2 which would delete the whole *employees* table[1]. This attack is also known as "SQL injection". Using dynamic parameters prevents this case as it strictly separates the input data from the query itself. In this case, the execution would simply fail because the input data is not valid.

```sql
INSERT INTO employees  (firstname, lastname, picture, birthdays)
VALUES ('John', 'Doe', NULL, ''); DROP TABLE employees; --')
```

Listing 2.2: An SQL command after using untrusted input as a literal value for the date field.

Although the standard states that dynamic parameters can be used in both execution models, it's only implemented for the two-phase execution in JDBC.

Transaction Management

Transactions are used to treat several queries as a logical unit that can be reverted or accepted atomically. SQL defines the commands `START TRANSACTION` to start a transaction and `COMMIT` and `ROLLBACK` to eventually complete or abort it.

[1] if multiple commands per query execution are allowed

However, CLI forbids the use of these commands and introduces a call-level transaction management. The defined behavior is that transactions are implicitly started whenever a query is executed and there is no active transaction. It can be finished with an explicit function that either aborts or completes the transaction. Thereby a transaction is always local to a connection or always global to all connections to a database [Com95, p. 63]. In JDBC's normal transaction management, all transactions are local to one connection, i.e. queries being executed over a second connection are handled in a separate transaction.

Metadata Retrieval and Introspection

The SQL-92 standard defines a schema called "INFORMATION_SCHEMA" that should be existent in all databases and include tables and views with metadata about the database, e.g. the existing tables, privileges, constraints [Int92, p. 75]. However, CLI abstracts the access to this information by using a couple of designated functions [Com95, p. 54]. This may have practical reasons, since some databases don't implement the standard completely and don't provide this information in form of the standard view. With separate functions, a database driver can use database specific queries to retrieve this information.

In addition, functions for introspection are defined by the CLI. These can be used to get information about the CLI implementation and the DBMS server, as for example information about the supported data types [Com95, p. 57].

2.2. PostgreSQL

In 1986, Michael Stonebraker started a project called "POSTGRES" at the University of California in Berkeley, as a successor to the DBMS Ingres. A SQL interpreter was added to POSTGRES in 1994 and it was eventually renamed to PostgreSQL in 1996, although it's still often simply referred to as "Postgres" [Gro14, p. lxiiff].

Today, PostgreSQL is a free and open-source object-relational DBMS that supports a wide range of features and conforms to the most recent SQL standard from 2011 [Gro14, p. 2018]. It is very well documented with respect to its internals and its usage which makes it popular for not only science, but also companies and organizations.

There are a couple of aspects in PostgreSQL which are interesting for this thesis and are therefore described in the next sections.

2.2.1. Extensibility

PostgreSQL's feature set exceeds the SQL standard in various ways. For example `money`, `point`, and `macaddr` are data types that can be used in PostgreSQL databases [Gro14, p. 107f]. Furthermore, the user can extend PostgreSQL's capabilities by adding new functions, operators, index methods, custom data types, procedural languages, and other features [Gro14, p. lxii]. The PostGIS project[1] is an example for utilizing this extensibility to add support for geographic objects and functions to PostgreSQL.

Extensibility is an interesting property with respect to this thesis, because it makes clear that a protocol like SQP can not simply be restricted to the SQL standard, but must be flexible. While this is not very interesting for functions which are simply used inside the untouched SQL commands, it especially affects the data types as discussed in section 3.4.

2.2.2. The PostgreSQL Wire Protocol

The protocol for communication between client and server is implemented as a native API [Gro14, p. 652], an ODBC driver[2], and a JDBC driver[3]. The message flow of the protocol consists basically of two phases: The start-up and the main phase. During the first phase, the connection is established with optional encryption, the client gets authenticated and parameters of client and server are exchanged [Gro14, p 1859f].

In the main phase the client can either send queries or switch to the copy mode. The latter one is for copying whole tables and is of minor importance here. Many operations, like metadata request or functions calls, are realized in PostgreSQL via SQL commands and don't require separate protocol messages.

Simple Queries

There are two query modes in the protocol: One for "simple queries" and one for "extended queries". The message flow for a simple `SELECT` query is depicted in Figure 2.1.

First, a message containing one or multiple SQL commands is sent to the server. If the query was about selecting data, the server will send a description of upcoming data rows.

[1] `http://www.postgis.net`
[2] `https://odbc.postgresql.org`
[3] `https://jdbc.postgresql.org`

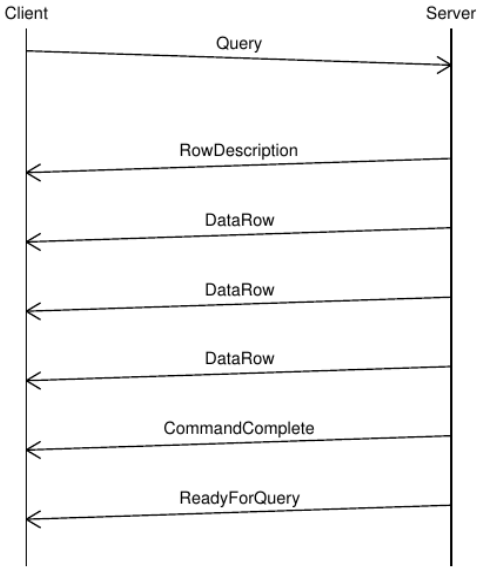

Figure 2.1.: The message flow for simple queries in the PostgreSQL wire protocol. Taken from [Urb14, p.29].

Then multiple data rows with the actual data follow, before the server signals it's finished by sending a "CommandComplete" message. Afterwards the server notifies the client that it's ready to receive new queries.

Extended Queries

Extended queries execution consists of three different phases:

1. A message with an SQL query is sent to the server to get parsed. If the query contains placeholders then the type of these fields should be specified in the message. After this the client can optionally request a description of the required parameters.
2. The parameters are sent to the server which creates a "portal" on the server that can be named. The client can optionally request a description of the rows that would result from this portal.
3. The client executes the portal and the server responds with several data rows. The client can define a maximum number of rows to receive. If the result contains more rows then the portal needs to be re-executed to receive the next rows. When the query ran to completion, a CommandComplete message is sent by the server.

In contrast to the simple query execution, the extended execution doesn't allow multiple commands per message. This makes it less vulnerable to SQL injections as described in

subsection 2.1.3. In fact, the whole concept of query execution is very similar to what the CLI defines. Also the portals are closely related to forward-only cursors. It's even possible to refer to portals as cursors with SQL commands. However, scrollable cursors are supported at the language level only in PostgreSQL.

2.2.3. Data Format and Large Objects

PostgreSQL supports two data formats to send the actual data rows: binary and text. With simple queries, the server responds always in text format, while extended queries can define the data format per parameter and resulting column [Urb14, p. 28ff]. The binary representations of the data types are however not documented and can only be deduced from the native API [Urb14, p. 23]. A look into the JDBC source code reveals that this is a weakness of PostgreSQL, since only a few data types are actually sent in binary format. Most types are sent and received in the less efficient text representation.

The size of normal data types is limited to 1 GB. In addition there are up to 2 GB big Binary Large Objects (BLOBs) which are handled separately by a "LargeObject API". This API is similar to the UNIX file-system interface as it provides functions like *open*, *read*, *write*, *lseek*, etc. BLOBs can then only get referenced by their ID in the actual data tables. This is faster than saving binary data inside the tables, but therefore automatic actions like data deletions by triggers aren't possible. Instead, only the ID would be deleted from the table [Gro14, p. 793ff].

2.3. Transbase

Although the DBMS Transbase also has its origins in a scientific research project, the database developed quite differently from PostgreSQL. Initiated at the Technische Universität München, it is now a closed-source commercial product developed by Transaction GmbH since 1987 [Gmb15a].

Transbase comes in different variants and focuses on efficiency, low resource usage, and scalability. A special feature is its multi-dimensional index mechanism called "Hypercube" [Gmb15b] [MRP+01]. In contrast to PostgreSQL, Transbase has a smaller feature set and doesn't provide as many ways to extend it. This also means that its number of supported data types is fixed.

Like every DBMS, Transbase as well has its specialties as for example the generic `DATETIME` data type, which can be *ranged* by qualifiers to span different fields. For example `DATETIME[MM:HH]` would only save months, days, and hours, but no year, minutes, seconds, or fractions. The standard `DATE`, `TIME`, and `TIMESTAMP` data types are just aliases for ranged `DATETIME` types in Transbase [Gmb10, p. 182ff].

Otherwise Transbase supports SQL-92 "Intermediate" level and has own extensions, some of which are also part of later SQL standards [Gmb10, p. 11]. However, some common commands like `START TRANSACTION`, `COMMIT`, `ROLLBACK`, or `DECLARE CURSOR` are not supported. For applications developers the absence of these commands probably doesn't matter, since the operations are mostly used via the CLI anyway.

2.3.1. The Transbase Wire Protocol

The Transbase protocol for client/server communication is implemented as a native C API and a JDBC driver. Other interfaces, like the ODBC driver and a PHP implementation, use the native API as their basis [Gmb15b, p. 3]. The protocol itself is undocumented as it's proprietary and only used internally. However, for the interest of this thesis, Transaction GmbH provided the JDBC source code and gave permission to deduce information about the protocol from it.

Also the Transbase protocol starts with setting up the connection (with optional encryption), authentication of the client, and some parameter exchange. After this, the client can send different request messages to the server. Each request has a very custom binary format and expects one specific response message.

Transaction Management

Some features like transactions and cursors are not supported by Transbase at the language level, but only at the protocol level. Therefore there are much more existing request messages than in PostgreSQL's protocol.

Also, the transaction management is much more verbose in Transbase: While in PostgreSQL each SQL command is an own transaction, if not stated otherwise via SQL, all commands must be explicitly associated to a transaction in the Transbase protocol. So there is a number of different requests for creating, beginning, finishing, and closing transactions which need to be send along with the actual queries.

Query Execution

Like PostgreSQL, Transbase has two execution models for simple, one-time queries, and for extended queries with dynamic variables. The latter one is called "stored queries", since the query is first stored on the server and then executed with parameters.

A significant difference is that for each query mode there are a number of different request types which depend on the actual *content* of the SQL command. For example DDL commands are executed with a different request compared to DML statements with updates, or DML statements that select data. This is probably due to the fact that there are different responses to these statements and the protocol needs to know which response to expect.

Stored "update" queries with dynamic parameters can be executed with a whole batch of parameter sets, and not just one. This way, Transbase executes the stored query multiple times with all parameter sets and treats all executions as one atomic operation. This is likely to be very efficient if many tuples in the database need to be inserted or updated.

If the query is used to select data rows, the protocol behaves very similar to the CLI: An explicit cursor is opened on the server side, which can also be scrollable in contrast to PostgreSQL. Then fetch messages can be sent with regard to the cursor's ID to fetch either the next rows, or rows from other positions, if the cursor is defined scrollable.

2.3.2. Data Format and Large Objects

Each data type in Transbase has one fixed representation, which is mostly binary. Alternative data formats are not supported. Interestingly, floating point values ("FLOAT" and "DOUBLE") are sent as text in Transbase, probably to avoid problems with different binary floating point representations in heterogeneous setups.

All data types have a size limit of about 4000 bytes, which slightly varies between the different types and depending on the database's defined page size [Gmb10, p. 14]. So for all bigger data, either BLOBs or Character Large Objects (CLOBs) have to be used. Both can take objects that are up to about 2 GB in size, while a CLOB is optimized for text in order to benefit from features like fulltext search. Similarly to PostgreSQL, BLOBs and CLOBs are sent with separate messages and therefore separate API calls [Gmb10, p. 213]. However, in other terms they behave like normal data types in Transbase, thus they can be used with triggers and constraints.

2.4. Comparison of Transbase and PostgreSQL

PostgreSQL and Transbase are very different DBMSs. The comparison of both only relates to the aspects that are mentioned in the previous sections and should summarize the relevant aspects for this thesis.

First of all, both DBMS support at least the SQL-92 standard, but have their own dialect and extensions, which also means that they support different data types. Both wire protocols have binary messages that are used in a request-response pattern. However, PostgreSQL's protocol is more flexible, since the response type to a request isn't statically defined.

Similar to the concepts in the CLI, the query execution mechanism differentiate between one-time and multiple-time queries with dynamic data bind (see subsection 2.1.3). Results are fetched in a cursor-like manner, even if PostgreSQL doesn't support all cursor features at the protocol level.

In fact, the PostgreSQL wire protocol focuses more on the actual communication while most operations are executed at the language level via SQL. This is very different from Transbase: Some features aren't even supported at the language level, so there are many different message types to implement them at the protocol level.

Both databases have support for large objects and send them separated from "normal" data types. However, their field of application is quite different: PostgreSQL treats them as completely separate objects, but allows the normal types to be quite big. Transbase in contrast has lower limits for normal data size, but integrates the large objects better.

Specialties exist in both protocols: On the one hand, PostgreSQL has support for different data formats, which is a very flexible approach. On the other hand, a good idea in the Transbase protocol is to be able to send whole parameter batches which saves communication round-trips in bulk operations.

All in all both databases have very different approaches to solve similar problems. This makes them suitable to test SQP, since it needs to provide a way that enables both databases to work without altering their internal behavior.

2.5. Protocol Encoding

The protocols discussed before are obviously application layer protocols that don't care about transport and network. For communication in a network like the Internet, the protocols are layered on top of TCP/IP which cares about routing, data integrity, and correctness on the wire. Naturally, also SQP is an application layer protocol. So the main focus of the protocol is on how the information is represented.

2.5.1. Data Serialization and Encoding

When two parties communicate they want to exchange information. Typically, this is done by sending *messages* which contain the encoded information. There are four objectives that need to be kept in mind when choosing the encoding:

Efficiency: The message should be as compact as possible.

Delimiting: The receiver must be able to recognize the beginning and end of a message.

Data transparency: The message should be able to include arbitrary (encoded) bits.

Ease of decoding: It should be easy to regain the information from the message.

These objectives are of course ideals and may stay in conflict which each other. However, it's important to consider them when choosing the protocol encoding [Sha08, p. 241f].

Information is mostly structured and consists of different fields with values. Therefore an important part of the encoding of structured data is the *serialization* of its structure. So while encoding is more general about the representation on the wire, serialization is about how the different fields are represented. In practice they are not always clearly separable, because one can condition the other.

There are three common approaches to serialize structured information:

Simple binary: Fixed groups of bits are used to represent the values of the different fields.

Type-Length-Value (TLV): All fields are stored as a triple with the type of the value, its length, and the value itself.

Matched tag: The value of each field is enclosed by an identifying start tag and a matching end tag. [Sha08, p. 241f]

In practice, protocols often use a mixture of these approaches to benefit from their advantages. For example both the Transbase protocol and the PostgreSQL protocol are a mixture of the first two approaches: Messages start with some fixed binary group that identifies the message type and includes type-specific information. After this, there is often a variable part with a variable-sized value in the TLV format.

2.5.2. Standard Data Formats

An alternative to defining own rules for data representation are existing data formats that define standards for serialization and encoding of structured information. In literature the standardized ASN.1 language is often used to describe the structure of protocol data in an encoding-independent way. The ASN.1 description and a *rule set* can then be applied to actually encode the data. Beside rule sets for binary formats (BER, CER, DER), there is also a rule set for data encoding using the popular Extensible Markup Language (XML) [Sha08, p. 246ff].

Unfortunately there are not many modern frameworks or existing application protocols that use ASN.1, its encoding rules, or the binary data formats. Only the data format XML is supported in many programming languages and used by many application protocols. However, also XML loses popularity for its overhead and unreadability. More and more, it's replaced by new data formats for which also no (standardized) ASN.1 rule sets exist.

2.5.3. The JavaScript Object Notation

A very popular data format is JavaScript Object Notation (JSON), because it's simple and readable while having less overhead than XML. Other than the name suggests, it only has its origins in JavaScript, but is actually programming language independent. It's specified by the two standards RFC 7159 [Bra14] and ECMA-404 [jso13] and being used in many web APIs, as for example the APIs of Twitter and Facebook.

JSON supports four different value types, which are all encoded as UTF-8 text: *object*, *array*, *number*, *string*. In addition, the values `false`, `true`, and `null` are allowed. A *string* value is text in double quotes with some escaped special characters. *Numbers* are simply decimal ASCII representation with support for scientific notation [jso13, p. 2f].

More interesting are the types *object* and *array*. An *array* is an ordered list containing zero or more values. *Objects* are zero or more name/value pairs, whose order doesn't

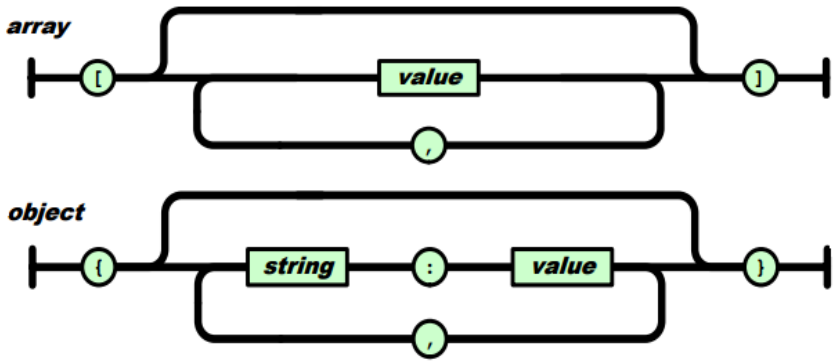

Figure 2.2.: The syntax of JSON *array* and *object* types. An array starts an ends with a square bracket and contains a comma-separated value list. An object is enclosed by braces and contains a comma-separated list of name/value pairs. Taken from [jso13, p.2f].

matter. Their syntax is depicted in Figure 2.2 [jso13, p.2f]. Listing 2.3 gives an example for a JSON object that describes a person.

As the listing shows, JSON syntax is very easy to understand and decode, while it's also delimiting, as it uses the *matched tag* approach. However, all data is represented as UTF-8 text, but text is not as efficient as binary data.

```
{
    "name" : "John Doe",
    "height" : 1.82,
    "languages" : ["english", "german"],
    "married": false
}
```

Listing 2.3: An exemplary JSON object describing a person.

2.5.4. Popular Binary Data Formats and MessagePack

There are a number of popular binary data formats for data transfer, as for example Avro[1], Protobuf[2], or Thrift[3]. These formats are mostly used in the context of Remote Procedure Call (RPC). They work schema-based which means that a data description,

[1] https://avro.apache.org
[2] https://developers.google.com/protocol-buffers
[3] http://thrift.apache.org

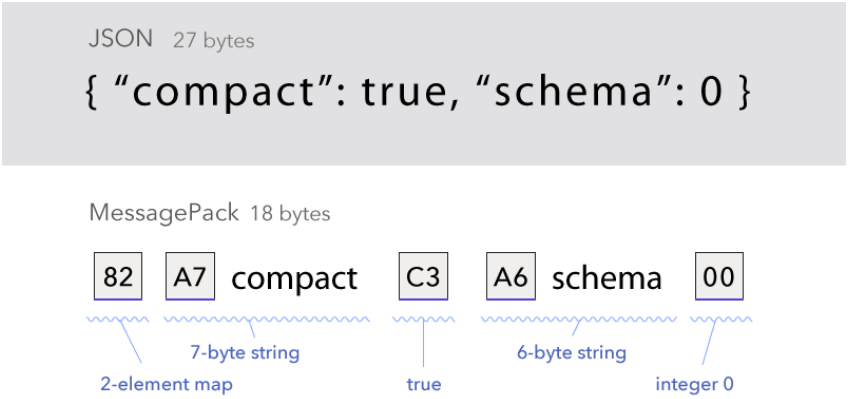

Figure 2.3.: A comparison of exemplary data encoded with JSON and MessagePack. Each character represents one byte. Boxes also represent one byte, while their content is the byte value in hexadecimal. Taken from `http://msgpack.org/`

the schema, has to be defined. Both client and server need this schema in order to use the type-specific encoding.

The advantage of these formats is that typed objects are supported and the resulting encoded format is extremely efficient in both time and space. This is due to the fact that the data structure is known and doesn't have to be included in the encoded data. The downside of schema-based approaches is that they are less flexible than self-describing formats and that all parties that want to use them have to know the schema.

JSON in contrast is self-describing, as it is a tag-matching approach that includes the field names of the object type. To maintain the flexibility and easiness of JSON, but be more efficient at the same time, a number of data formats emerged that are *JSON-compatible*, meaning that they use the same logical structure as JSON. Examples for these formats are BSON[1], Smile[2], or MessagePack[3].

MessagePack, also called “MsgPack”, is probably one of the most widespread JSON-compatible binary data formats. There are more than 50 different implementations for different programming languages and frameworks[4]. An example of how an object encoded by MessagePack compares to JSON is illustrated in Figure 2.3.

The idea behind MessagePack is basically to use a very compact TLV format: Every JSON value is prefixed by an identifier that defines the type and length of the value. In

1 `http://bsonspec.org`
2 `http://wiki.fasterxml.com/SmileFormat`
3 `http://msgpack.org/`
4 as listed on `http://msgpack.org/`

the example it's shown that the byte `0x82` stands for a map (equivalent of an object) with two elements. What's special is that the identifier is also variable in length and depends on the bits of the identifier itself. For example if the most significant bit is `0`, the other bits represent a signed integer. Therefore values between `0` and `127` can be represented in one byte, like the last value in the example.

Benchmark results show that MessagePack is both efficient in encoding time and space and can even compete with schema based binary formats. Results for a benchmark of serialization in the Java Virtual Machine (JVM) can be found in appendix A.1.

2.5.5. JSON Schema

```
{
    "title": "Simple Person",
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        },
        "height": {
            "type": "number",
            "minimum": 0,
            "description": "Height in metres"
        },
        "languages": {
            "type": "array",
            "items": { "type": "string" }
        },
        "married": {
            "type": "boolean"
        }
    },
    "required": ["name", "height", "languages"]
}
```

Listing 2.4: An exemplary JSON schema for the object of Listing 2.3

Although a schema is not required for JSON, it is sometimes required to describe the data structure, for example for automatic data validation or introspection. Therefore "JSON Schema" was developed and exists as a draft for a standard [GZC13].

Listing 2.4 is an example for such a schema that describes an object as given in Listing 2.3. The syntax of the schema is thereby JSON itself, which makes it easy to understand and process. As shown in line 10, the schema can also define restrictions for specific types and contain semantic descriptions as in line 11.

JSON Schema has much more features than shown in the example, but a detailed description would go beyond the scope of this thesis and can be found in the standard draft [GZC13] or on the official website[1].

2.6. The WebSocket Protocol

When it comes to modern web APIs and new protocols, HTTP is always a popular choice, because it's widespread and easy to use. However, HTTP is stateless and the connections are only short-lived and unidirectional. As this doesn't suffice for some applications, workarounds like session IDs and polling have been used to overcome these limitations. Nevertheless, the underlying communication is still based on multiple short-lived connections causing high overhead and the need for connection management and mapping [FM11, p. 4].

As a simpler solution, the WebSocket protocol has been developed and standardized [FM11]. It enables an ongoing two-way communication between a client and a server on top of TCP. Originally being thought of as a protocol for web applications running in a browser, it turned out to be very useful for a much wider range of applications.

2.6.1. The WebSocket Handshake

The WebSocket protocol is compatible with HTTP. This means the first message is actually a HTTP message that includes an upgrade request as shown in Listing 2.5. The server then answers with an upgrade confirmation that switches the protocol effectively to WebSocket.

A great advantage of this handshake is that it allows the protocol to establish a connection on the same port as HTTP (port 80) by just using a different URL. So for all systems with firewalls it isn't necessary to unblock a new port just to let WebSocket clients reach the server.

[1] `http://json-schema.org/`

```
GET /url/on/server HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: sqp
Sec-WebSocket-Version: 13
```

Listing 2.5: A HTTP-compatible Handshake from a WebSocket client.

A noteworthy field in the handshake is the `Sec-WebSocket-Protocol` field. This field can be used to specify the actual application protocols the client is able to understand. The server can then choose one of these protocols for communication or decline the connection [FM11, p. 5ff].

WebSocket also supports encryption via Transport Layer Security (TLS). Other useful extensions like protocol compression and connection multiplexing are currently in development [Gri13, p. 297].

2.6.2. WebSocket Messages

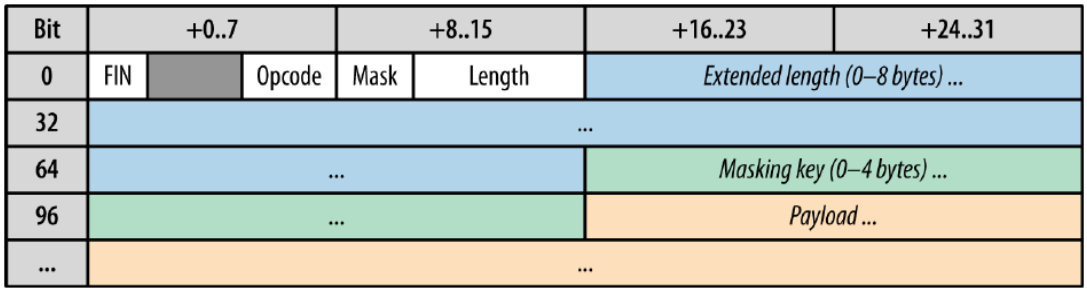

Figure 2.4.: A WebSocket frame in box notation with 2 to 14 bytes overhead and the actual payload. Taken from [Gri13, p. 295].

After the handshake the client and server can communicate bidirectionally by sending messages. In fact, WebSocket only defines a framing format that splits each application message into one or more frames. Thereby each frame has minimal overhead as shown in Figure 2.4.

In the first byte the opcode is defined which identifies the frame type. Supported are the control frames “close”, “ping”, and “pong”, and message frames “text”, “binary”, and “continuation”. The latter ones are used to send messages in WebSocket:

Binary Messages consist of one “binary” and zero or more “continuation” frames. The payload can be arbitrary binary data.

Text Messages consist of one “text” and zero or more “continuation” frames. The payload must be valid UTF-8 encoded text.

The last frame of each message, which might also be its first frame, always has the “fin” bit set to 1, so the receiver knows that the message is finished after the frame. This chaining mechanism allows messages of arbitrary length in WebSocket [Gri13, p. 295f]. More details about WebSocket are of minor importance for this thesis and can be found in [Gri13] and [FM11].

3 | Protocol Design

The new database protocol SQP is implemented as a subprotocol of WebSocket. It brings standardized connection establishment and a standardized format for arbitrary long messages, so these features don't have to be implemented as a custom solution. Instead, existing frameworks can be used, so both the design and implementation of SQP can focus on the message contents.

Even if it turned out that WebSocket doesn't gain acceptance in the long run then SQP could simply be layered on top of another message based protocol, since its core functionality isn't bound to WebSocket in any way.

Beside this, SQP will also benefit from other useful WebSocket features, like:

- Possible encryption of the communication via TLS.
- Upcoming extensions and improvements of WebSocket, like compression and connection multiplexing.
- Usage on port 80 alongside with HTTP.

In all further sections, the implementation of SQP as a WebSocket subprotocol will be assumed. So when referring to "messages" the actual content of the WebSocket messages is meant, without regard to the WebSocket frames and their headers.

3.1. SQP Messages

The first idea was to encode messages as JSON objects, since it's widely used, supported in many programming languages, and easy to understand. However, this approach has two major drawbacks.

The first drawback is that even small status messages would need to be an object with a field for the message type. This would result in a fixed overhead of at least 5 bytes, assuming that the field is called "type". Also, the whole message would need to be parsed first, before the type of the message could be identified, since the fields of JSON objects are unordered.

To avoid this, the message consists of two parts: An one byte ASCII character as the message identifier, followed by an *optional* JSON payload. Listing 3.1 shows an example for such a message. This approach is elegant, because it separates the message type from its content. This allows on the one hand early recognition of message types, such that unexpected messages can be rejected directly. On the other hand, simple messages can be as small as one byte, since the payload is optional.

Using an ASCII character as the message ID limits the possible number of messages with different printable IDs to 95. If the protocol needed to support more message types, which is very questionable, one could also define an "extension" ID indicating that there is a second byte to identify the message type. This way the number of possible printable IDs would nearly double.

```
1 H {
2     "database": "testDatabase"
3 }
```

Listing 3.1: A formatted example of an SQP **Hello** message consisting of an ASCII character as identifier and a JSON payload. When being sent over the wire, the redundant spaces would be left out.

The second drawback is that JSON defines that the data is always encoded as UTF-8 text. However, especially database applications are all about sending and receiving data. So using JSON as the data format would be inefficient for some applications. This would contradict the objective of being easily usable in real-life applications (see section 1.2).

The idea to solve this is to not restrict SQP to one data format, but allow two different formats: JSON and MessagePack. What makes this approach powerful, is that MessagePack's structure is completely JSON-compatible. Consequently, the data format is not really important for message handling, as long as the message structure is described on an abstract level.

So the client can decide whether to use JSON or MessagePack: While JSON is easier to use and human readable, MessagePack needs an extra library but is extremely efficient.

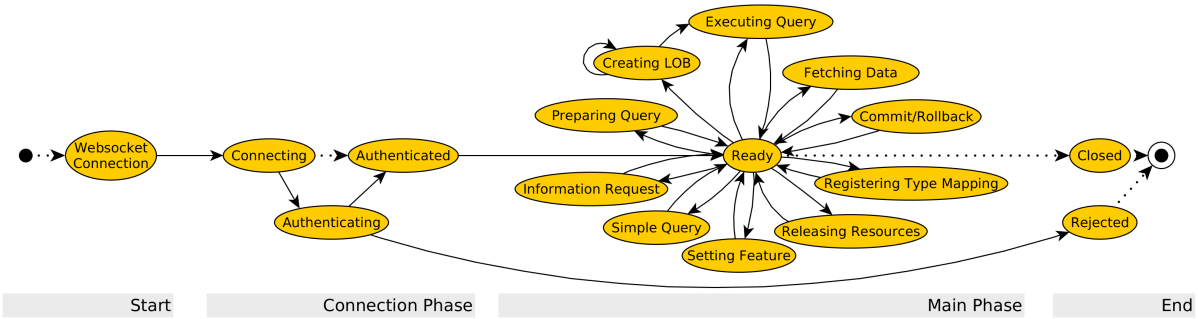

Figure 3.1.: A state machine showing the different states of the protocol. Arrows with dotted lines are transitions that can happen without SQP messages being sent. In all other transitions an SQP message is sent from the client to the server or vice versa.

Remarkably, the protocol does not need extra semantics to specify the payload type, since WebSocket defines two message formats: binary and text messages. So when a client sends a text message then the server assumes to receive JSON payload and answers in the same format. Respectively, the client can send binary messages with a MessagePack payload and will get a binary response.

A complete overview over the messages supported by SQP with their name, ID, and rough payload description is given by Table 3.1. While their use is described more detailed in the following sections, examples for these messages can be found in appendix A.2.

3.2. Protocol States and Message Flow

The communication with SQP can be described with protocol states which are illustrated by the state machine in Figure 3.1. Transitions from one state to another mostly happen when the server receives or sends an SQP message, which is referred to as the *message flow* of the protocol.

The communication starts with a client establishing a WebSocket connection. To start the actual SQP protocol, the client needs to send a **Hello** message, containing at least the database to connect to. The server can then challenge the client to authenticate with a specific authentication method which the client needs to respond to. If the server rejects

ID	Message Name	Content of Payload
H	Hello	Database name
A	AuthenticationResponse	Credential
S	SimpleQuery	SQL query, cursor parameters*
P	PrepareQuery	SQL query, statement ID*
X	ExecuteQuery	Query parameters*, statement ID*, cursor parameters*
F	FetchData	Cursor ID*, fetch parameters*
L	Release	Cursor IDs*, Statement IDs*
I	InformationRequest	Subject, detail*
T	SetFeature	Feature values
M	TypeMapping	Name, schema, keywords*
K	Commit	
R	Rollback	
G	LobRequest	Object ID, offset*, size*
!	Error	Error type, error message
r	Ready	
a	AuthenticationRequest	Credential type
p	PrepareComplete	
c	CursorDescription	Cursor ID, column metadata, scrollable flag
#	RowData	Row data
x	ExecuteComplete	Number of affected rows
e	EndOfData	"more" flag to indicate if more data is available
l	ReleaseComplete	
i	InformationResponse	Response type, value
t	SetFeatureComplete	
k	TransactionFinished	
m	TypeMappingRegistered	Native type
o	LobReceived	
*	LobAnnouncement	Object ID

Table 3.1.: List of supported SQP messages. The three sections stand for the allowed sender of these messages: client, server, or both. If the "Content of Payload" field is empty then the message has no payload. Content marked with an asterisk is optional.

the authentication then it sends an **Error** and closes the whole WebSocket connection. Otherwise the server switches into the main phase by sending a **Ready** message.

In the main phase the client can send request messages to initiate different operations which are discussed in detail in the following sections. Once an operation is finished, the server changes back into the "ready" state where it waits for new operations. If an operation failed, the server sends an **Error** message and either closes the connection, or sends a **Ready** message in case it was able to recover from the error, so the client can proceed its work.

At this point it's worth to mention that all messages sent by the client have a capital letter as an ID, while most response messages have small letters as IDs. A small letter thereby indicates that the server switches back to ready state. If the ID is a special character instead, like the **Error** message ID or the **RowData** ID, it means other messages will follow and the server is not yet ready for new queries. As letters and special characters can be distinguished by using corresponding bit masks, this can simplify the implementation of server response handling. Often, but not always, request-response pairs share the same letter in a different case.

3.3. Supported Database Operations

As exposed in section 2.4, database operations can either be communicated at the language level or at the protocol level. So if a new database protocol is to be implemented without changing the database internals then there are basically two ways to go:

1. The protocol oriented database needs an SQL parser that finds database operations in the language and invokes the internal protocol operation.
2. The language oriented database generates SQL queries for certain protocol operations and executes them.

Obviously, the second approach is both simpler and more efficient, since the creation of specific queries is always faster and less error-prone than parsing and evaluating SQL queries. Consequently, SQP supports the important operations at the protocol level, like the Transbase protocol. However, it tries to keep all operations as simple and flexible as possible, similar to PostgreSQL. So in the end, it takes the best out of the two worlds. In the following sections the most important database operations are discussed at example use cases.

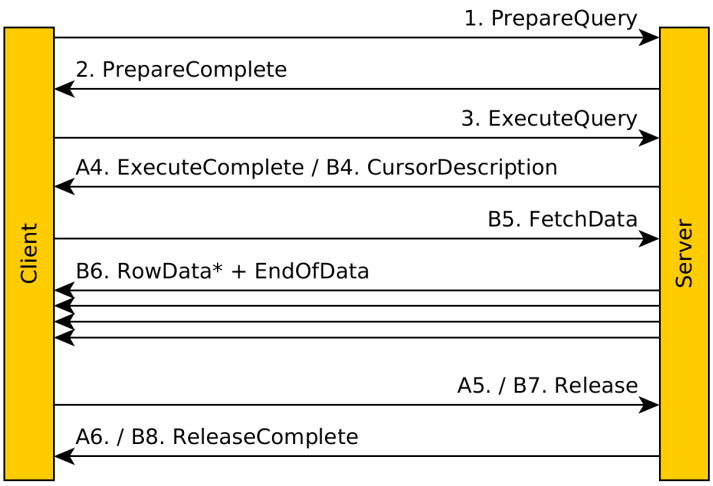

Figure 3.2.: The message flow for executing an SQL query with SQP in multiple steps. Depending on the query, the server answers with a different message in step 4, so the client may also fetch data as part of step B5 and B6.

3.3.1. Extended Query Execution

The most important use case is the execution of queries. Like the other protocols and the CLI, SQP supports two different query execution models. The message flow for the "extended query execution" is illustrated in Figure 3.2.

First the client sends a **PrepareQuery** to the server (1) which contains the SQL statements with possible placeholders for dynamic parameters. Optionally, the client can specify a string as the statement ID, so it can later refer to it. If no ID is provided, a default ID is generated and used by the server. The server confirms the operation by responding with a **PrepareComplete** (2) and switches to "ready" state.

The client can then execute the statement by sending an **ExecuteQuery** message (3) including the ID of the statement to be executed and a *batch* of parameters. A batch means that there might be multiple value sets for the placeholders, so the statement is executed multiple times. If the statement is about querying data then the batch must only contain one parameter set. The client might then also define a cursor ID and specify the "scrollable" flag that defines whether the cursor should be scrollable or not.

So in case of querying data, the server opens a cursor and sends a **CursorDescription** back (B4) which contains information about the result rows and the cursor's ID. Alternatively, the server confirms the execution with an **ExecuteComplete** message (A4)

including the number of affected rows for update, insert, or delete statements. Afterwards, the server enters the "ready" state again.

After the cursor has been opened, the client can fetch data from it by sending a **FetchData** message (B5) which includes the cursor's ID, the maximum number of rows to fetch, and the fetch direction, if the cursor is scrollable. Depending on the result the server sends zero or more **RowData** messages with the actual tuples. Then the server sends an **EndOfData** message (B6) which includes a "more" flag that specifies whether the cursor has more data to fetch. Afterwards the server switches to the "ready" state.

At the end, the client can release the created statements and cursors by sending a **Release** message with the statement and cursor IDs to close (A5 / B7). The server will confirm the close with a **ReleaseComplete** (A6 / B8).

Both statement IDs and cursor IDs are completely optional to be defined. If the client leaves them out, default IDs will be assumed. Whenever a statement or cursor is opened with the ID of an existing one, it will be released first. So if the client doesn't intend to use multiple queries at the same time, it doesn't need to bother about the IDs.

As the server switches back to ready state after each operation, the order of these operations actually doesn't really matter, as long as they refer to existing resources. So a client could for example first prepare two queries, and then execute them.

It can be seen that the SQP cursor support resembles the SQL cursor features much more than for example the PostgreSQL protocol. However, "updatable" cursors are not yet supported. Support for them could be simply added to SQP in the future by specifying a new flag in the **ExecuteQuery** message and introducing **UpdatePositioned** and **DeletePositioned** messages.

3.3.2. Simple Query Execution

The **SimpleQuery** message can be seen as a shortcut of an extended query execution. It simplifies query execution by sending just one message which contains the SQL statement to execute and optionally a cursor ID, the scrollable flag, and the maximum number of rows to fetch. Since no parameters can be included in the message, the SQL statement shouldn't contain any placeholders.

Depending on the query, the server either simply answers with an **ExecuteComplete**, or with a series of **CursorDescription**, zero or more **RowData** messages and an **EndOfData** message. So the query behaves similar to a sequence of Prepare/Execute/Fetch, just without creating the statement for subsequent reuse. The cursor however is existent and can be used for further **FetchData** operations, if needed.

3.3.3. Using Large Objects

SQP has support for CLOBs and BLOBs, as discussed in subsection 3.4.2. Both are handled the same way and simply referred to as Large Objects (LOBs). Just like in the other database protocols, LOBs are sent separately, before they can be referenced by their ID in an **ExecuteQuery** message. Sending LOBs breaks the normal message flow to reduce encoding overhead: Instead of sending one message and waiting for responses, the client sends a **LobAnnouncement** with the LOB ID and then the *unencoded* LOB in a separate message, without any ID or payload structure. After the server has received the announcement and the LOB, it sends a **LobReceived** message as confirmation.

Retrieving LOBs is handled separately as well. Like in the execution, the LOBs are only referenced by their ID in **RowData** messages. The client is then able to fetch it by sending a **LobRequest** message, which needs to contain the LOB ID and optionally a specific offset and length in order to fetch only a defined chunk. The server responds to such a request with a **LobAnnouncement** followed by the requested *unencoded* LOB chunk in a separate message.

3.3.4. Explicit Transaction Management

By default, all SQL query executions are treated as separate transactions which is called "auto-commit" mode. So the client doesn't need to bother about transaction management for simple intentions. However, as the possibility to manually control transactions that span over multiple queries is a fundamental feature of databases, this is also possible with SQP.

To set server parameters, SQP defines a **SetFeature** message that can be sent to the server. In this case, auto-commit can be turned off by sending a **SetFeature** message with the field "autoCommit" set to `false`. The server will confirm this operation by sending back a **SetFeatureComplete** message.

At this point, a new transaction is implicitly started with the next query being executed on the server. This transaction can then be finished by either sending a **CommitTransaction** or **RollbackTransaction** message to the server, which then either confirms or reverts the changes of the queries included in the transaction. On success, the server responds with a **TransactionFinished** message or sends an *Error* if the operation failed. In the latter case, the server automatically executes a rollback to get into a consistent state. A new query after a finished transaction implicitly starts a new transaction.

There are two other special cases: If a transaction is not committed, but the client re-enables auto-commit then the active transaction will be committed. If there is an uncommitted transaction and the client closes the connection, the active transaction will be rolled back.

This behavior only allows one transaction at a time per server connection. While this can be seen as a drawback towards the Transbase protocol, most users will probably not see this as a limitation since this restriction also holds for most CLI implementations that are used to access the databases.

What's not yet implemented in SQP is the ability to set different levels of isolation. This can be necessary to define how concurrent access to data affected by transactions is handled by the server. By default, the transaction mode is "serializable", meaning that the output of concurrent operations is the same as in a serial execution, which may lead to wait times [Int92, p. 68]. However, this feature could simply be added in the future by using the **SetFeature** message.

3.3.5. Requesting Information

For requesting information and allowing some introspection, SQP defines an **InformationRequest** message that can be sent to the server with a specified "subject" and an optional "detail".

An example would be a message with the subject `TypeSchema` and the detail `Date` to get a description of SQP's `Date` data type as described in subsection 3.4.4. The server responds with an **InformationResponse** with the "responseType" set to `Schema` and the "value" containing a JSON Schema that defines the requested type.

3.3.6. Registering a Type Mapping

Yet another feature of SQP is the possibility to register a *type mapping* which is described in detail in subsection 4.2.4. This can be done by sending a **TypeMapping** message with a given JSON schema, a name, and keywords. On success the server answers with a **TypeMappingRegistered** message.

3.4. Data Types

When a client uses a database, it will obviously insert, update, or fetch data from it. All these operations require transporting the actual tuple data via the protocol. When sending this data, the receiver needs to get the encoded value and know the type of data it has received in order to be able to properly decode and process it.

This is not a problem for native database protocols, since both the client and server implementation know the DBMS' type system. So for each data type an ID and a value encoding is defined which is known to all protocol implementations. Even when the type system is extensible, as in PostgreSQL, the ID and encoding of the type are known to both parties as soon as the client sends the definition to the server.

In a protocol that should work across various databases, things are different: The client implementation can't rely on any type system of the DBMS, since all databases have their specialties and support a different set of data types, as pointed out in section 2.4.

To solve this issue, SQP has three features:

1. It defines a set of standard data types that is based on the types defined in SQL-92.
2. It provides the possibility to send DBMS specific data types that exceed the SQL standard.
3. It allows the client to register a type mapping, which maps a client-defined data type to one that is supported by the DBMS.

So database independent applications can rely on the standard types or use a type mapping, while clients that know the DBMS on the server side can also use its native types.

Byte number:	0..3				4..7				8..11			
JSON (ASCII):	[	2	0	1	5	,	1	2	,	2	4	]
MsgPack (hex):	93	cd	07	df	0c	18						

Figure 3.3.: The date “24th December 2015” in SQP format encoded with both JSON and MessagePack. While the JSON encoding is 12 bytes long, MessagePack only needs 6.

3.4.1. Data Type Encoding

Binary data encodings are very efficient so they are a natural choice for database protocols. However, decoding them can be very elaborate, and encoded data is mostly difficult for humans to read or understand. Text representations on the other hand are flexible and easy to understand, but inefficient in size and speed.

Getting the best out of both worlds was probably the motivation for PostgreSQL to have both a binary and text format for all data types[1]. However, this approach has a downside, as PostgreSQL's JDBC driver shows: Only some data types are implemented in the binary format, while many are always represented as text. This is probably due to the fact that the binary formats are hard to understand and undocumented, so the developers decided to implement the easier format.

To learn from this problem, a different approach is chosen for SQP. The encoding of data types isn't defined directly, but only as a JSON-compatible type. Like the SQP messages, the data types can then be encoded as text via JSON or in an efficient binary format via MessagePack. Either way, the data encoding is portable, and processing is kept extremely simple, as existing implementations care about the decoding. Figure 3.3 shows an example of a date being much more efficiently encoded with MessagePack than with JSON.

To be really efficient, non-primitive types should not be represented as objects, but as arrays, like in the example. This saves a lot of space by not including the field names for each value, but it also means that the values are not self-describing. Therefore type information is sent separately in the protocol, so the value encoding can focus on data and structure. For example, when data is fetched from the server the type information is only included in the cursor description, while the actual row data values can stay compact.

[1] In fact, servers internally need to be able to understand text representation of each data format anyway, so they can be used as a literal inside an SQL statement.

Other than PostgreSQL, SQP doesn't support a per-column differentiation of the data format, since it doesn't make sense in the context of SQP: If the client supports MessagePack it will use it for the whole message and all data, since it's more efficient and the actual processing of data doesn't change. Otherwise, if it doesn't support MessagePack for messages then it also won't support it for data values.

3.4.2. SQP Standard Data Types

So that the client can work independently from a specific database implementation, a number of standard data types is defined in SQP which are listed in Table 3.2.

What seems to be dubious at first is that many data types share the same JSON type. However, this is okay since they don't need to be self-describing and they don't really differ when being transferred. Their only difference is a semantic one when being decoded: For example a value of type `TinyInt` can safely be assumed to be in the range of -128 to 127, even if it's encoded as a JSON number, which could represent any numbers.

Some of the types can take the additional parameters *precision (p)* and *scale (s)*, whose meaning depends on the specific data type. In contrast to the data type, these parameters aren't necessary to decode the data since the JSON representation is always sufficiently meaningful.

The mapping to JSON types is mostly straightforward:

`Boolean` is simply mapped as a JSON `boolean`.

`TinyInt`, `SmallInt`, `Integer`, and `BigInt` are mapped as `number`s.

`Real` and `Double` are also `number`s. While they can have arbitrary precision in JSON, MessagePack will be encoded in a IEEE 754[1] compatible way, so there is no loss of precision.

`Char`, `VarChar`, and `XML` are all simply `string`s. Since JSON `string`s are delimited there is no problem decoding them without their *precision* parameter. Only for `VarChar` the information of maximum allowed characters gets lost. However, this is mostly irrelevant for applications, as the database checks this restriction.

[1] IEEE 754 is the standard defining the representation of floating point numbers in computers

Name	Description	JSON Format
Boolean	Truth value	`boolean`
TinyInt	1 byte signed integer number	`number`
SmallInt	2 byte signed integer number	`number`
Integer	4 byte signed integer number	`number`
BigInt	8 byte signed integer number	`number`
Real	4 byte IEEE754 floating point number	`number`
Double	8 byte IEEE754 floating point number	`number`
Char(p)	Fixed-length string with p characters	`string`
VarChar(p)	Variable-length string with max. p characters	`string`
XML	Xml Data	`string`
Binary(p)	Fixed-length byte vector with p bytes	*base64 string / raw*
VarBinary(p)	Variable-length byte vector with max. p bytes	*base64 string / raw*
CLOB	Reference to a Character Large Object	`string`
BLOB	Reference to a Binary Large Object	`string`
Null	SQL NULL value	`null`
Decimal(p,s)	Exact, fixed-point number with $(p - s)$ digits before and s digits after the decimal point	`string`
Date	Calendar date with year, month, day	`[number, number, number]`
Time(p)	Wall-clock time with a resolution of 10^{-p} seconds, where currently $p = 9$, and optional time-zone offset in seconds	`[[number, number, number, number], number?]`
TimeStamp(p)	Calendar date and time	`[Date, Time(p)]`
Interval(p)	Signed time interval with a resolution of 10^{-p} seconds, where currently $p = 9$	`[boolean, number, number, number, number, number, number]`
Custom	A DBMS specific or custom type	*depending on type*

Table 3.2.: Name, description, and JSON-compatible format of the data types that are natively supported in SQP. Values with question mark are optional. Grouped by logical function.

Byte number:	0..3				4..7				8..11			
String (ASCII):	3	1	4	.	1	5	9	2	6	5	3	5
BCD (hex):	0c	08	23	41	5c	0f	0e	03				

Figure 3.4.: The number "314.15926535" encoded both as string and based on the Binary Coded Decimal code. The BCD code consists of the precision and scale parameters and bytes that encode two decimal digits, beginning at the least significant digits.

`Binary` and `VarBinary` are special since they represent binary data which has no related primitive JSON type. While MessagePack actually supports `raw binary` data, it needs to be converted to a base64 `string` in JSON. Fortunately most frameworks care about this special treatment automatically, since encoding binary data is a common issue.

`CLOB` and `BLOB` are simply `string` IDs as their content is transferred separately (see subsection 3.3.3)

`Null` is not really a type, but represents a `NULL` value, which means that a value has been left out. In order to not define all types in a way that supports a missing value explicitly, a common approach is to use the special `NULL` value. On the wire this value is simply a JSON `null`. More on this is described in subsection 4.1.3.

The remaining types in SQP are more elaborate and discussed separately.

3.4.3. The Decimal Data Type

The data type `Decimal(p,s)` is used to represent arbitrary long, exact decimal values with $(p - s)$ digits before and s digits after the decimal point. So in contrast to the IEEE 754 numbers, this data type is popular to represent exact money or measurement values. Furthermore, it's also often used to overcome the limits of the normal integer types.

Although the JSON specification allows the `number` type to have arbitrary length and precision, this does not hold for binary alternatives like MessagePack. A representation of such a type would need a non-standardized inefficient format. So MessagePack is limited to either 8 byte signed integers or 8 byte IEEE 754 numbers, which don't suffice in this case. Therefore `Decimal` needs another representation.

A popular approach for encoding arbitrary decimal numbers is called Binary Coded Decimal (BCD). As the name already suggests, it's the decimal representation of a number encoded with binary numbers. Figure 3.4 shows how this would look like for a `Decimal`. A similar approach could also be used in a JSON-compatible format by creating an array where each value represents a set of decimal digits, as in `[[3, 14], [15, 92, 65, 35]]`.

Nevertheless, there is a problem with this approach. If the client sends or receives a `Decimal` it has to save it in order to process it. However, the common programming languages don't have a standardized structure which is used for such a type. Consequently, initializing these structures with binary data is mostly unsupported or depends strongly on the programming language's implementation of structures. The only initialization method that most programming languages support, is by parsing a string. So instead of defining a BCD-based `Decimal` representation which is hard to understand and use, they are simply represented as strings in SQP.

The disadvantage of using strings is obviously the space it takes to encode a `Decimal`. However, by allowing the scientific notation (the "e"-notation) of numbers, the string size can also be rather short for very high or low values. Given that also the official PostgreSQL JDBC driver uses strings to send decimals, the practical relevance of the encoding size seems to be limited.

If it turned out that many applications really need a very compact way of representing decimals, this could also be implemented as a negotiable feature in the protocol: The client could set a "BCDDecimals" feature to `true`, so the server would then use a BCD-based representation, like the one mentioned before.

3.4.4. Temporal Data Types

Temporal data types are typically no primitive types, which makes their encoding very interesting. SQP supports four temporal data types: `Date`, `Time`, `Timestamp`, and `Interval`. As the support should not be limited to a specific database, there shouldn't be narrow restrictions in regard to their possible range and resolution. Therefore common representations like UNIX timestamps (seconds since 1st January 1970) are not eligible for a simple solution here.

Date

To support a broad range of date values, the Julian Date (JD) format is theoretically a good choice, as it can be represented as a 4 byte integer and supports dates between -4713 BC and 5874897 AD, which is a considerably broad range.

Unfortunately, most common programming languages don't provide functions or data structure initialization with JD values. Although the algorithms used to convert a JD to a Gregorian calendar date are fairly easy, they are still required to process the data, which contradicts the for "ease of use" requirements.

The chosen way is therefore to represent a date simply as a triple of *year*, *month*, and *day*. To support dates BC, the *year* field can be set to a negative number. So this format has no real restrictions in range which makes it suitable for SQP. At the same time it's easy to understand and process, as many time structures of common programming languages support initialization by these values. Even the efficiency is okay, as in MessagePack this triple would only have 6 bytes (see Figure 3.3), which is only 2 bytes more than a JD has, but considerably less than using a string representation (12 bytes).

Time

The same basic approach is used for `Time`, so a triple with *hour*, *minute*, *second* can be used as a basis. Because of the small value ranges of the three fields, the whole triple only has 4 bytes when being encoded by MessagePack. However, this triple is not sufficient, as today's time values need to support a higher resolution than seconds and an optional timezone.

The `Time` value usually takes a *precision* parameter p which defines the resolution of the time to be 10^{-p}. Allowing arbitrary p values would lead to the same difficulties as in `Decimal`. Fortunately, the value of p is limited in practice: Most modern applications and databases support time resolution up to nanoseconds, which corresponds to $p = 9$.

So to keep the `Time` format simple, it just allows a fourth field which takes the nanoseconds value, so it uses $p = 9$ by default. If this resolution doesn't suffice in the future, yet another field could be added which takes the value of the parameter p. By setting it to a value greater than 9, even higher time resolutions would be simply possible. For now, the nanosecond resolution completely suffices.

Timezones are a difficult topic, since their names and abbreviations are not standardized and don't uniquely identify the time offset from Coordinated Universal Time (UTC). The time offsets of the different time zones are influenced by political decisions, have been changed over time, and depend on whether daylight saving time is being observed, and when it starts or ends [Gro14, p. 134f]. Fortunately, the SQL-92 standard defines that a timezone in SQL only refers to its *offset* and can be defined in hours and minutes.

Therefore, the SQP `Time` type also just supports an *optional* offset field. However, instead of using two fields for *hour* and *minute*, it only uses one field for the total offset in seconds. This reduces the size of the data structure and allows higher offset resolutions that could get useful in the future.

So a final example for a time value with offset is given in Listing 3.2. Surprisingly, the encoding size for this value in MessagePack is 13 bytes, which is just 1 byte more than what the binary format of PostgreSQL needs [Gro14, p. 128].

```
1 [[13,47,33,250000000],7200]
```

Listing 3.2: A JSON formatted **Time** value representing the time "13:47:33.25" with a UTC offset of +02:00.

Timestamp

`Timestamp`s represent a time at a specific date, therefore they bring the same challenges as the `Date` and `Time` types. The straight-forward result to this is to just use a combination of both values, as shown in Listing 3.3. This makes the `Timestamp` powerful, easy to understand, and easy to decode.

```
1 [[2015,9,21],[[13,47,33,250000000], 7200]]
```

Listing 3.3: A JSON formatted `Timestamp` value representing the time "13:47:33.25" with a UTC offset of +2 hours at the 21st September 2015.

The size of this `Timestamp` ranges between 13 bytes and 20 bytes in MessagePack encoding, depending on the concrete values. While the upper limit is considerably high, it still excels an ISO string representation (30 bytes), which is also used in the PostgreSQL JDBC driver.

Temporal Intervals

`Intervals` follow the same approach as the other temporal data types and are therefore specified as an 8-tuple. The first field is a `boolean` representing the sign of the interval: `true` stands for a positive interval and `false` for a negative one. The other fields specify the *years*, *months*, *days*, *hours*, *minutes*, *seconds*, and *nanoseconds*.

The SQL-92 standard states that there are two types of intervals in particular: year-month intervals and day-time intervals. This difference is used because days cannot simply be converted to months or vice versa [Int92, p. 30]. The SQP format can be used to transfer both by just setting the corresponding fields and leaving all other fields set to `0`. This approach still leaves room for improvement, since some fields are actually wasted, depending on the interval type. So if this type should be optimized in future releases, one could think about differentiating between the two possible interval types.

On the other side, support for intervals doesn't seem to be standardized at all: The JDBC specification completely leaves out support for intervals, while ODBC supports all kinds of intervals separately (e.g. hour-minute intervals). So the proposed approach for `Interval` is used because it supports all cases and is easy in use and implementation on both client and server.

3.4.5. Custom Data Types

As the protocol needs to be extensible to allow DBMS-specific features, it also allows to use data types which are not standardized. These types are called "custom" types. To differentiate them from standard types it's recommended to give them a meaningful name. For example, `pg_point` would refer to PostgreSQL's *point* type. The encoding of this type can then be defined by the server implementation. So `pg_point` is for example defined as a tuple of two numbers.

The client can use SQP's introspection features to get information about the data type. It would do so by sending an **InformationRequest** with subject `TypeSchema` and detail `pg_point` to the server, which would respond with an **InformationResponse** containing the JSON schema shown in Listing 3.4.

However, this also works the other way round and is called *type mapping*. The client can send a JSON schema and a custom type name to the server via a **TypeMapping** message.

```
{
    "type": "array",
    "title": "PostgreSQL Point",
    "items": [
        {
            "id": "x",
            "type": "number",
            "description": "X Coordinate"
        },
        {
            "id": "y",
            "type": "number",
            "description": "Y Coordinate"
        }
    ],
    "minItems": 2,
    "additionalItems": false
}
```

Listing 3.4: The JSON schema for PostgreSQL's *point* type.

The server then tries to find a data type which is compatible to the defined schema and remembers the custom name for it. An example is provided in subsection 5.2.2.

When the client wants to use a custom data type for binding an SQL statement's dynamic parameter, the **ExecuteQuery** message needs to have the corresponding parameter type set to `Custom` and add another field with the custom name. The custom name is not set as the parameter type because this field is restricted to values that refer to SQP standard types. This ensures that custom and standard types are not mixed up and can be handled separately, as described in subsection 4.2.5.

When a client fetches data from the server it needs to convert all values to one of the SQP standard types. This can mostly be done by creating a string representation of the value and sending it as a `VarChar`. To receive custom types the client needs to explicitly tell the server what it's able to understand. In the given example it does so by sending a **SetFeature** message with the field *allowNativeTypes* being set to `pg_point`. Then the server knows that a conversion to a standard type is not necessary for this type and can use the custom encoding instead.

3.4.6. Comparison of SQP Data Types and the SQL Standard

Most of the SQP standard data types are based on the popular SQL-92 standard. However this doesn't hold for all. The types `NUMERIC(P,S)`, `INTEGER(P)`, and `FLOAT(P)` are not defined in SQP, since they don't have a real practical relevance: Most databases define `NUMERIC(P,S)` and `DECIMAL(P,S)` as synonyms, so they have the same implementation. The generic `INTEGER(P)` and `FLOAT(P)` just map to concrete integer or floating point types, based on the parameter P. These concrete types are directly supported by SQP, so there is no need to implement the generic version and worry about the parameter and its restrictions.

Even though not part of the SQL-92 standard, SQP supports the types `Boolean`, `XML`, `CLOB`, and `BLOB`, as they are supported in many common databases.

On the other side, SQP does not support collection types as `ARRAY` or `MULTISET`, although they are part of one of the more recent SQL standards. This is because the support for collections differs highly between the databases: Some databases, like Transbase, don't support collections at all and some restrict the collection types to be arrays of a another type, as PostgreSQL. Even other databases have no restrictions at all and simply call for example a row a `MULTISET` of values. So there doesn't seem to exist a practical standard when it comes to the support of collection types, and their purpose in a relational database is questionable in general. As a consequence they are not part of the SQP standard types, but can be implemented as custom types.

3.5. Summary

SQP aims to be simple in understanding, implementation and use, while being flexible and allowing universal database access. Using WebSocket, JSON, and MessagePack as basic technologies allows the SQP design to focus on the important things: database operations and data transport.

Common challenges of protocol basics, like encryption and message definition, are solved by WebSocket and therefore available in many existing implementations. New features that are going to be introduced in WebSocket will also be available to SQP, so it can benefit from them without changing the SQP implementation itself.

The popular JSON data format allows SQP messages and data types to be structured, understandable, and easily processable. As JSON encoding is text based, the efficient and binary MessagePack data format can be used as an alternative. Since it's JSON-compatible, the client doesn't really get in touch with it's binary encoding, but can still think about it as JSON-structured data. For both frameworks there are many existing implementations, which significantly decreases the work of implementing SQP.

To transport the actual data from the database, SQP defines a standard set of useful data types that are based on the SQL-92 standard, but with respect to the practical needs. By separating the type information from the value, the encoding of these types with MessagePack can be very efficient. The efficiency may not completely compete with optimized and custom binary formats, but is still a lot better than using only text, which is a good trade-off for keeping the protocol and data simple to process.

SQP is not limited to standard SQL, but also provides ways to support DBMS specialties at the protocol level. Examples for this are information requests, feature settings, type mappings and custom data types.

4 | Implementation

To show that SQP actually works as designed, it has been implemented as a client library and an SQL proxy server with two different backends, as already described in the objectives in section 1.2. The implementation consists of different modules which also have external dependencies, as depicted in Figure 4.1.

Core provides implementations for SQP messages and data types while using *Jackson* as a decoding framework with support for JSON and MessagePack.

Client is an SQP client API based on the *Tyrus* Java WebSocket implementation.

Proxy is the SQP server implementation based on the *Vert.x* platform and a *JSON schema validator*.

Backend defines the interfaces to be implemented for concrete databases in order to be usable with the proxy server.

PostgreSQL Backend implements a PostgreSQL backend by using the core protocol implementation from the PostgreSQL JDBC driver.

Transbase Backend is a backend implementation for Transbase which uses *TBX*, the sublayer of the Transbase JDBC driver.

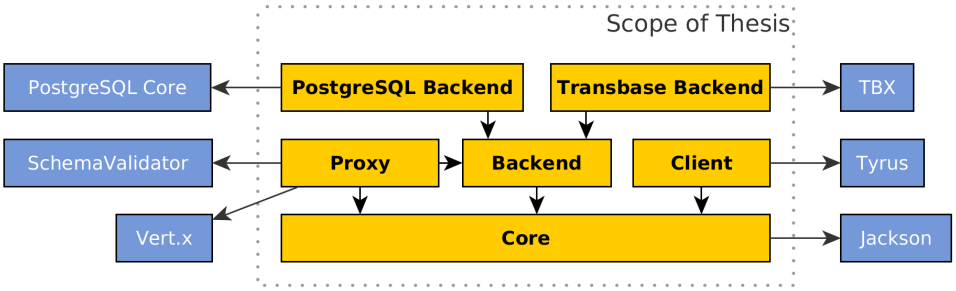

Figure 4.1.: Overview of the different modules of the implementation and their internal and external dependencies.

All modules have been implemented in Java 8 with respect to modern and up-to-date approaches, such as asynchronous execution. They will be discussed in detail in this chapter.

4.1. The Core Module

Client and server implementation differ in their functionality, but they both eventually need to send and receive messages with data. The *core* module therefore implements structures for SQP messages, SQP data types, and the encoding functionality. It's the only place where data encoding/decoding is implemented. Consequently, all other modules can work at an abstract level, focus on their main functionality, and don't need to worry about the actual representation of SQP data on the wire.

4.1.1. Encoding and Decoding

The core package provides two interfaces called `MessageEncoder` and `MessageDecoder`. For encoding, it's only necessary to pass the SQP message, the data format (`Text` or `Binary`) and an `OutputStream` to the `MessageEncoder`. It then writes the encoded message in either JSON or MessagePack to the output stream. Similarly, the `MessageDecoder` only needs the data format and an `InputStream` and returns the decoded SQP message.

Both interfaces are implemented using the `ObjectMapper` of the popular Jackson framework[1] which is able to perform the following tasks:

- Decoding JSON data to a number of compatible generic Java structures, for example `Map<T>` for JSON objects, `List<T>` for JSON arrays, and Java primitives. A JSON object in a compatible Java structure will be referred to as a "JSON-format object".
- Encoding a JSON-format object as JSON.
- Mapping a JSON-format object to custom Java objects, if somehow possible. Jackson is able to use getters, setters, and constructors to do this.
- Directly encode a custom object to JSON, respectively decode JSON data as a custom object.

[1] `https://github.com/FasterXML/jackson`

Today, Jackson supports additional data formats via extensions, while preserving the same `ObjectMapper` interface. The MessagePack Java implementation includes such an extension for Jackson[1]. So using MessagePack instead of JSON only requires initializing a different `ObjectMapper`, while the actual usage remains unchanged.

4.1.2. Message Classes

For each defined message type there is a data class that is derived from the abstract `SqpMessage` class. The data classes only contain members with getters and setters for all corresponding message fields. A central `MessageType` enumeration is used to save which messages are implemented and to associate the correct ASCII character IDs with the concrete message classes. This is possible because in Java enumerations are a set of fixed *objects*, rather than just integers, as in other languages.

When a message is to be decoded, the `MessageDecoder` reads the first byte as an ASCII character and gets the corresponding type by looking it up in `MessageType`. The `ObjectMapper` is then used to decode the payload and initialize an object of the concrete `SqpMessage` descendant. Similarly, the `MessageEncoder` just writes the message's associated ASCII ID before using the `ObjectMapper` to encode the message object in the desired data format as the payload.

4.1.3. Data Types

For the defined SQP data types this works very similar: For all types there is a class derived from `SqpValue`, while an `SqpTypeCode` enumeration associates type names with the concrete subclass and a JSON schema.

One advantage of having the type code separated from the value class is the implementation of `SqpNull`, which represents the SQL `NULL` value. In contrast to the other value classes, it doesn't have a specified type code, but just uses the type code of any other type which it's representing. Therefore the other value classes don't need to regard this special value.

Figure 4.2 shows the `SqpValue` hierarchy. Most of the classes themselves are very simple and either wrap a related Java primitive or contain multiple fields corresponding to the more complex SQP types. The `SqpValue` class defines a number of conversion methods to Java types as `asString()`, `asInteger()`, `asLocalDate()`, etc. If the SQP types can

[1] `https://github.com/msgpack/msgpack-java`

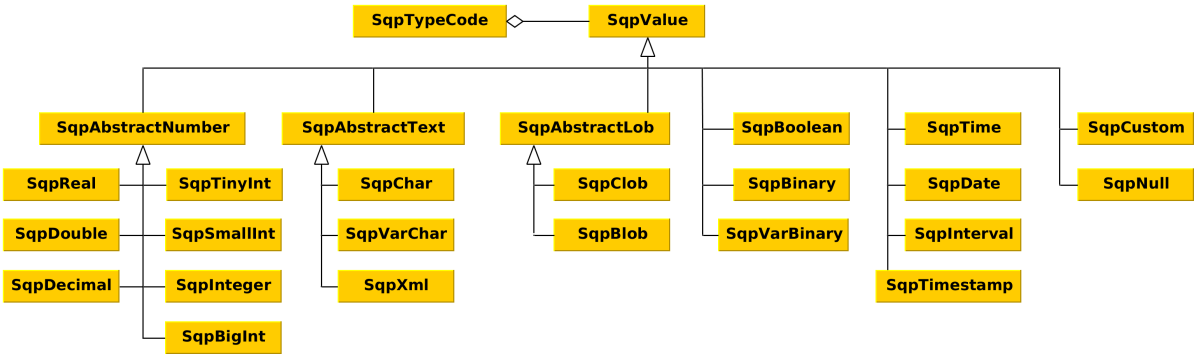

Figure 4.2.: Hierarchy of the classes representing the SQP types.

be converted to these values then the subclasses can override these methods. In addition to the concrete conversion methods, the generic `as<T>()` method exists, which utilizes the `ObjectMapper` functionality of mapping the object to any other compatible class `T`. These methods make `SqpValue` objects very convenient to use, as shown in chapter 5.

Encoding and decoding data types needs to be done explicitly and requires an additional step in comparison to `SqpMessage` objects. This is because the `SqpValue` objects would be encoded as JSON objects by the `ObjectMapper`, but are actually desired to be encoded as tuples. As a consequence, `SqpValue` defines the abstract method `getJsonFormatValue` and the static method `createFromJsonFormat`. The first one is overridden by all concrete value classes in order to return a JSON-format object which represents the value as the defined JSON type, so the `ObjectMapper` can take this object to generate the correctly encoded data. The second method is the counterpart: It takes a `SqpTypeCode` and a JSON-format object and initializes the corresponding `SqpValue` subtype by calling a special factory method that the subtype needs to provide.

4.2. The Proxy Server

The proxy server implements the server side of SQP, handles common tasks, such as connection management, parameter decoding and information requests, and delegates database operations to a specified backend. Thereby it completely decouples the actual protocol implementation from the backend in use.

4.2.1. Architecture

The proxy server is implemented based on the modern polyglot Vert.x[1] framework which also provides a WebSocket server implementation. What is really interesting about Vert.x is that it works *event driven* and *non blocking* which also applies to the overall proxy architecture.

A Vert.x application typically consists of "verticles" that are notified by Vert.x whenever a certain event occurs. In this case, the verticle is the WebSocket server which registers a callback at Vert.x that is called for all new established connections. The verticle then executes all necessary work without blocking the thread. This means for example that the server does not send a message over the network and then has to wait for a response to it. Instead, it registers a callback at Vert.x which gets called whenever a new network message for this connection has been received.

Thereby Vert.x runs an event loop which waits for events, looks for registered callbacks and eventually calls the callback in the event loop's thread. This means that callbacks are only invoked one at a time, while assured that it's always in the same thread context. This way the verticle doesn't have to deal with concurrency issues, while still utilizing the core to capacity, since it's never blocking it, but handles other events in the meantime. As this holds for one core, a Vert.x application scales by launching as many event loops and instances of verticles as there are cores in the machine. Events thus get evenly distributed over the event loops and therefore over the active verticles.

There are of course also operations which are blocking, for example accessing the file system or network communication. For these basic tasks Vert.x provides non-blocking interfaces which work callback-based and can be used in the event driven manner. Other blocking operations, such as huge computations, or old blocking APIs, have to be executed in a so called "worker" thread. This is a regular thread that runs concurrently to the event loops and is therefore allowed to block execution when waiting for a result. For a seamless integration into the non-blocking operation, Vert.x provides a simple interface to delegate function calls to a worker thread and get notified via a callback when it's finished. Other interesting approaches about Vert.x and non-blocking programming can be found in the appendix A.3.

Figure 4.3 shows the overall architecture of the proxy server implementation in form of a class diagram. In the middle there is the processing layer of the proxy which has no

[1] `http://vertx.io/`

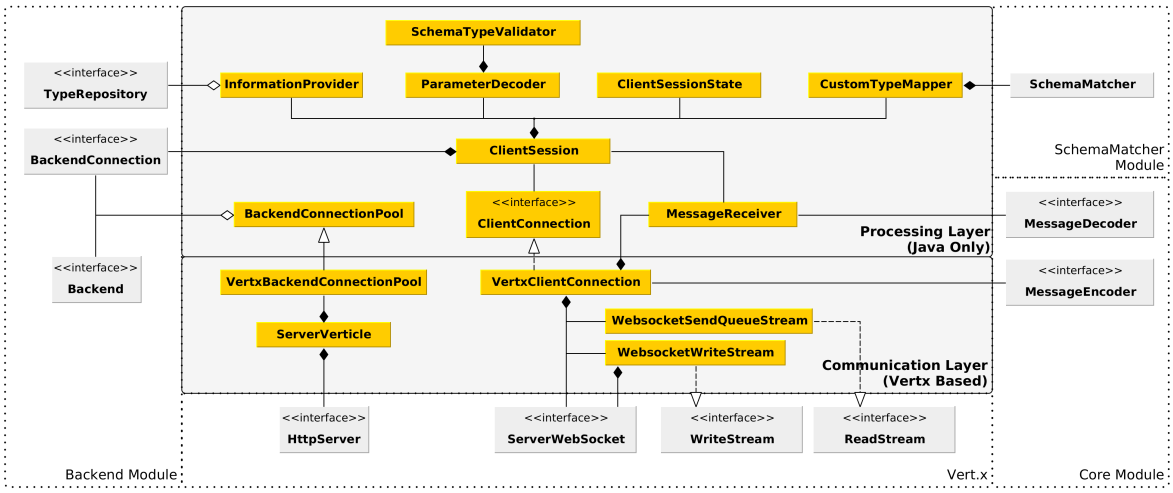

Figure 4.3.: Important classes in the proxy server implementation and their dependencies on other modules.

external direct dependencies. The idea of the processing layer is to actually implement the state machine depicted in Figure 3.1. As the figure suggests, the `ClientSession` is one central element of the processing layer which is responsible to handle exactly one client.

In contrast to the processing layer, the communication layer is implemented using Vert.x. This layer manages the network communication and notifies the processing layer of new events, like incoming WebSocket frames, in a framework-neutral way. The processing layer can also use the functionality of the communication layer through neutral interfaces that ensure that the actual communication layer implementation can be replaced by another one.

The other implementation modules (SchemaMatcher, Backend, Core) are only partially included in this figure in order to illustrate the actual dependencies.

4.2.2. Workflow of the Proxy Server

As described, the whole workflow of the server is event based. To understand this concept better, the reactions on different events that can occur during its lifespan are discussed in the following sections. After finishing all event reactions, the server returns control to the event loop and waits for new events.

Server Initialization

The first step is the actual initialization of the `ServerVerticle`, the main class which implements the SQP server as a verticle. It loads a configuration file which includes server settings, the class name of the backend to use, and a backend specific part. A `VertxBackendConnectionPool` is then created which attempts to load the backend class and initializes it by passing the backend specific settings.

After the connection pool and the backend have been set up, the verticle creates an HTTP server with WebSocket support. Before the server starts to listen on a configured port and path, a callback for new WebSocket connections is registered with the server. The initialization of the verticle is then finished and it returns to the event loop.

Client Connection Establishment

When a WebSocket client successfully connects to the running WebSocket server, Vert.x invokes the registered callback of the verticle. The callback gets a new `ServerWebSocket` instance as an argument which can be used to communicate with the new client.

The verticle then instantiates a `VertxClientConnection`, the main class of the communication layer which serves as a glue between the processing layer and Vert.x' `ServerWebSocket`. Furthermore, the actual `ClientSession` and a corresponding `MessageReceiver` object are created. Lastly, the `VertxClientConnection` registers some callbacks with the `ServerWebSocket` in order to be notified about incoming WebSocket frames.

Client Input Handling

An incoming frame can trigger a whole series of actions. To get a better overview, this process is depicted in Figure 4.4. All starts with a client sending a new WebSocket frame to the server (1). When Vert.x receives the frame, it invokes the `VertxClientConnection` (2) which passes the frame's content to the `MessageReceiver` (3). The `MessageReceiver` buffers all incoming frames (4) until the incoming frame is the final frame of the message. It then utilizes the `MessageDecoder` from the core package (5) in order to get the decoded `SqpMessage` from the buffer (6).

This message gets passed to the `ClientSession` (7) which puts it into a message queue (8). The `ClientSession` then checks in which state it currently is and if it is able to process the queued message (9). The states correspond to those discussed before in section 3.2. For

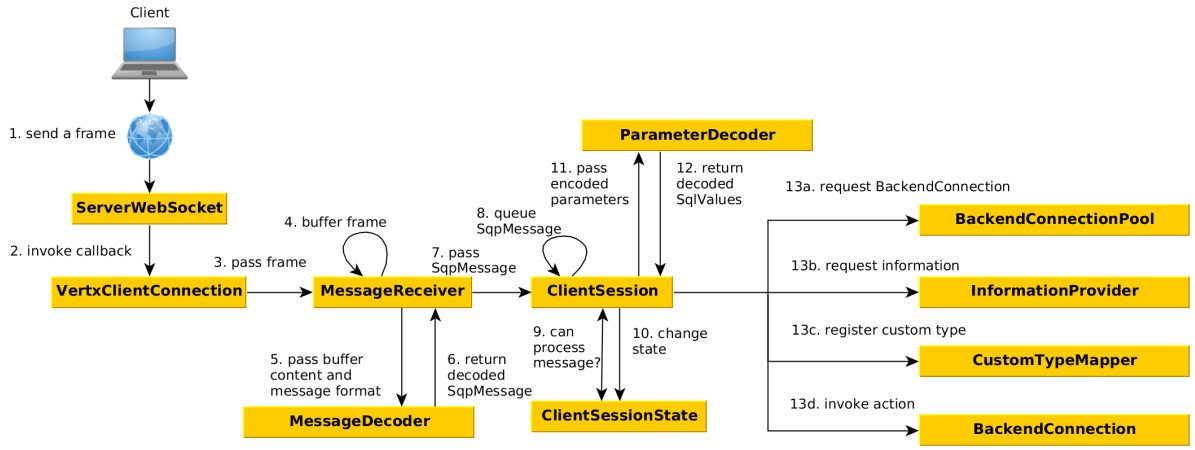

Figure 4.4.: The different steps that might be executed when a WebSocket frame is received.

example, if the `ClientSession` still waits for data of a previous **FetchData** message, it is in state "Fetching Data" and not able to process an incoming **PrepareQuery** message. If however the `ClientSession` is in state "Ready", it's not busy and can start the processing of the message by transitioning into a new state "Preparing Query" (10).

The incoming message might contain SQP data as dynamic parameters for a query which needs to be decoded explicitly, as already mentioned in subsection 4.1.3. If this is necessary, the `ClientSession` passes the encoded data and some extra information on to the `ParameterDecoder` (11) which decodes the values and returns the decoded `SqpValue` objects (12). The functionality of the ParameterDecoder is explained in detail in subsection 4.2.5.

Then, depending on the message type, the `ClientSession` invokes one of the four different actions:

HelloMessage: A new client started communication, so the server requests a new connection to the backend at the `BackendConnectionPool` (13a).

InformationRequest: The `InformationProvider` is used to handle this request (13b) as described in subsection 4.2.3.

TypeMapping: The `CustomTypeMapper` is invoked in order to handle this request (13c) (see subsection 4.2.4).

Others: The desired action is invoked at the backend (13d) as described in section 4.3.

Either way, the `ClientSession` only invokes the action and passes a callback to get notified about the result, so the action itself can be executed asynchronously if it's a blocking action. Therefore message processing ends at this point and the event loop can continue its work.

Sending Responses

When one of the invoked actions finishes, it invokes the callback it received from the `ClientSession`. The `ClientSession` then constructs a response message object and passes it to the `ClientConnection` interface, which is implemented by `VertxClientConnection`.

This then encodes the message in the format of the original request that lead to this response. The encoded message is appended to the `WebSocketSendQueueStream` which is an implementation of Vert.x asynchronous streaming mechanism. Basically it queues outgoing data and, whenever possible, passes it to the `WebSocketWriteStream` which sends it as WebSocket frames through the `ServerWebSocket` to the client.

Closing the Connection

The client can simply close the WebSocket connection to the server when it's finished with its work. Vert.x notifies the `VertxClientConnection` about this, which passes the information through the `ClientSession` to the `BackendConnectionPool`. This eventually closes the associated `BackendConnection`, before the `ClientSession` switches to the "Dead" state, in which it's unusable.

4.2.3. Providing Information

The InformationProvider cares about answering **InformationRequest** messages that have a *subject* and an optional *detail*. Currently implemented subjects are:

SupportedNativeTypes to find out which kind of native data types are supported by the backend.

DBMSName to find out the type of database in use.

MaxPrecision to find out the maximum supported precision of a data type.

MaxScale to find out the maximum supported scale of a data type.

TypeSchema to get the JSON schema for a specific data type.

The last three subjects refer to a specific data type, so the name of the data type must be set as the *detail*.

When handling such a request, the server first checks if it's able to respond to it directly. If this is not possible, or the subject is unknown, it delegates the request to the backend. Therefore it would be even possible to implement custom information requests in a backend. If the backend isn't able to respond to the request either, the proxy sends a response message to the client with the *response type* set to `Unknown`.

The type related requests are special since they can refer to SQP data types or the custom types of the backend, which are also called "native" types. Although native types are backend related information, the server is able to answer these requests always directly by querying the backend's `TypeRepository` which provides information about the native types (see subsection 4.3.1).

4.2.4. Custom Type Mapping

Type mapping allows for a client application to define a custom data type that is transparently mapped by the server to one of the native types supported by the backend. For example a client could define a `year_month` type which is a tuple of two integers. A PostgreSQL backend supports the native `point` type which is a tuple of two floating point numbers. Therefore the custom `year_month` type is compatible to the native `point` type, so a mapping can be created.

Type mappings can be used as a compromise for client applications: The code using the custom data type might not work with *all* existing databases, but with some of them. Thus the same client code can be used with all compatible databases, although the specific implementation of the data type might differ. Without this possibility, the client application code would always need to include the native data type names and wouldn't be portable. An example for such an application is given in subsection 5.2.2.

The **TypeMapping** message, which is used to register such a mapping, contains three important fields: The name of the custom mapping, the schema to be mapped, and an optional list of keywords. The list of keywords is used to give the server hints which native data types should be preferred if there are multiple candidates. This field is useful since the native types that should be used for the mapping in different databases are probably

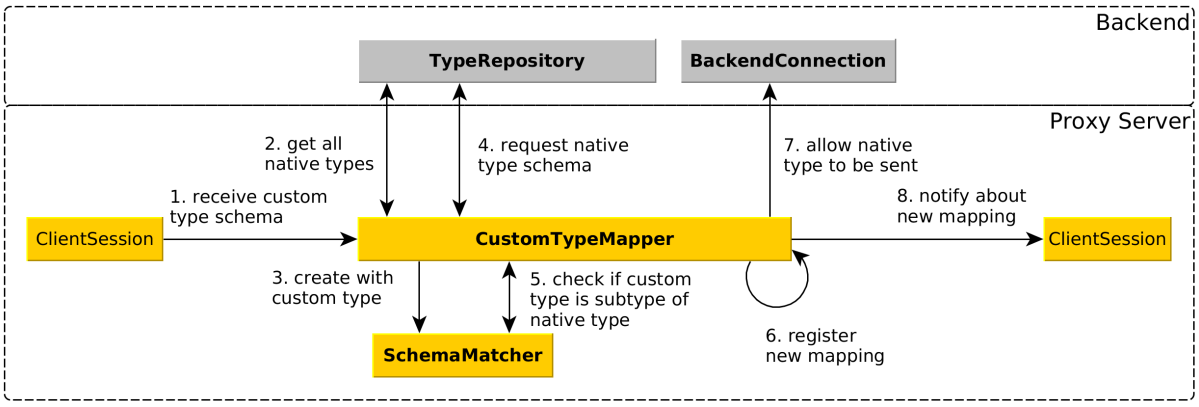

Figure 4.5.: The different steps that are executed when a custom data type is mapped to a native data type.

known at time of development. So instead of leaving the type mapping completely to chance, the application developer can already express his preferences for the mapping.

The process of finding a type mapping is illustrated in Figure 4.5. When `CustomTypeMapper` receives a custom schema to map (1), it first retrieves supported native types from the backend's `TypeRepository`. Then it creates a `SchemaMatcher` for the custom schema. The SchemaMatcher is actually a small module that works independently from the server. It was developed to determine if one JSON schema is compatible to another, because it can hold the same information. More on the implementation of the `SchemaMatcher` can be found in appendix A.4. So for each native type, the server gets the corresponding JSON schema from the backend (4) and checks if it's compatible to the custom schema (5). If they are compatible, the mapping between the custom type and the native types are saved (6). Also the backend is notified that data of the mapped native type is allowed to be sent to the client in the native format (7). Lastly, the `CustomTypeMapper` notifies the `ClientSession` about the new mapping, which sends a respond message to the client. If no mapping was found, an error is sent instead.

The actual type mapping is completely transparent to the backend: When incoming data is of a custom type, it's mapped to the native type before the backend can process it. On the other side, the backend is allowed by the `CustomTypeMapper` to send data of the mapped type in its native format (see step 7). This works because the `SchemaMatcher` assured that the native format is compatible to the custom format, so the client won't notice a difference between them. Nevertheless, the **TypeMappingRegistered** message that is sent back to the client also contains the mapped native type name, hence the client knows what to expect from the server.

There is one remaining issue in this context: Assuming that the client registered a data type `position` which gets mapped to a backend's native type called `coordinate`, the backend is allowed to send data of type `coordinate` in its native format. However, there may be more attribute values of type `coordinate` that the client doesn't consider to be of type `position`, but these attribute values get sent in native format anyway.

However, this scenario does not have practical relevance: type mapping should be used whenever the client has to stay compatible with multiple databases. If the data structure contains two attribute values of type "coordinate", where only one is expected to be a "position", the client is working on a database-specific dataset and the partial compatibility via custom types doesn't make sense at all.

4.2.5. Decoding Parameters

As already mentioned in subsection 4.1.3, parameters cannot be directly decoded to `SqpValue` objects by the `MessageDecoder` since they are encoded as primitives or tuples which are not self-describing. Instead, the `MessageDecoder` will only decode parameters to generic JSON-format objects.

The client typically sends data when it executes a prepared query with dynamic parameters via the **ExecuteQuery** message. This message has three important fields related to the parameters: *parameterTypes*, *parameters*, and *customTypes*. The first one only defines the SQP standard types of the parameters, while the second includes the actual JSON-format objects. For most data types the conversion to `SqpValue` is straightforward by using `createFromJsonFormat` which was introduced in subsection 4.1.3.

However, for `Custom` parameters this is different. In this case, the *customTypes* field needs to be defined and include the corresponding custom type names for all `Custom` parameters. Thus the names may refer to registered type mappings or to native types as defined in the backend.

If the type name refers to a registered type mapping, the server changes the custom type name to the mapped native type name. In addition, the input has to be validated. For native types, value validation in the proxy server is not required, since the DBMS does this before actually executing the query. However, the custom type registered by the client is only *compatible*, and not equal to the native type. This means that it might be more restrictive. Example: the client registered the `position` type to be a tuple of two integers in the range between 0 and 10. This type was mapped by the server to PostgreSQL's

`point` type which is a tuple of two arbitrary floating point numbers. As `point` allows values that are invalid for `position`, the input has to be validated, because the DBMS does not know these restrictions.

Validation can be done by the "Json Schema Validator"[1]. It uses the JSON schema sent by the client when registering the type mapping and checks if the JSON-format object validates it. If it doesn't, the server sends an **Error** message to the client and aborts message processing. Otherwise it applies the type mapping by treating the parameter as if it was of the native type it's mapped to.

So if the parameter is of a native type or has been mapped to one, the proxy constructs a `SqpCustom` object from the parameter which simply wraps the JSON-format value. In addition, the native type name is saved in that instance, so the backend can receive a self-describing `SqpCustom` object and process it accordingly.

In the end, the `ParameterDecoder` returns a batch of self-describing `SqpValue` objects which can be passed to the backend for processing. This way the backend doesn't need to care about the different fields in the message or the type mappings.

4.3. Backends for the Proxy Server

Backends are the glue between the proxy server and the actual DBMS by implementing the vendor specific database protocol. The backend module specifies some simple interfaces that need to be implemented, so that the server can use them. The architecture of the backend implementation itself depends only very marginally on it.

4.3.1. The Backend Module

Figure 4.6 shows the three interfaces that each backend needs to implement and the four different cases in which they are used by the server.

The `Backend` gets initialized when the server starts. It thereby can do all operations that need to be executed only once, such as the `TypeRepository` initialization (case A). Afterwards, the `Backend` is only used by the `BackendConnectionPool` to create new `BackendConnection` objects whenever the `ClientSession` requests a new connection (case B).

[1] `https://github.com/fge/json-schema-validator`

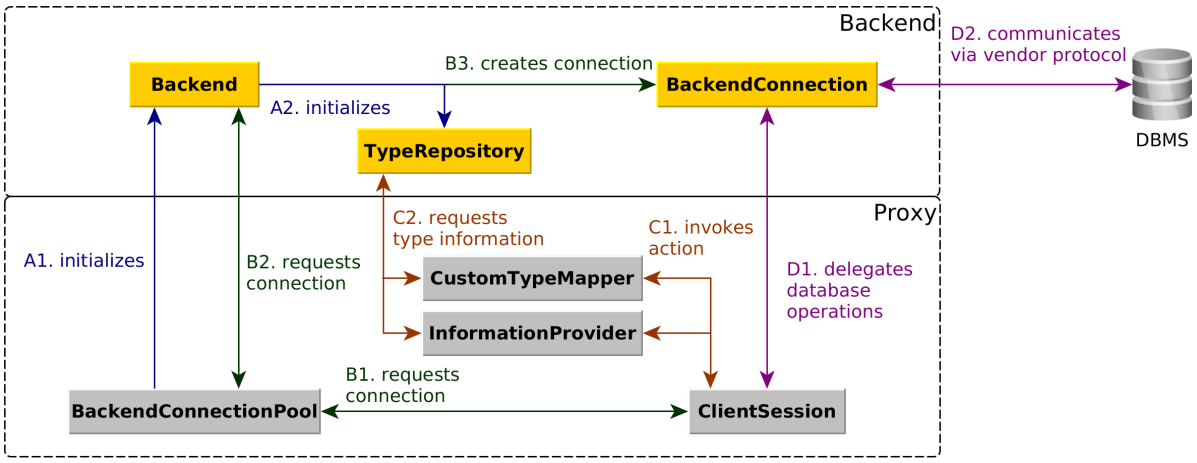

Figure 4.6.: Different cases (A to D) in which backends are used by the proxy server.

As discussed in section 4.2, the `CustomTypeMapper` and `InformationProvider` use the backend's `TypeRepository` (case C). The implementation of the `TypeRepository` is optional and only required if the backend supports native data types. In this case it needs to provide two functions: the first one returns a list of supported native data types and the second one needs to return a JSON schema for each of them.

The most important interface is the `BackendConnection` which is used to execute operations on a specific DBMS by implementing the vendor specific communication with it (case D). Each instance is used to handle the communication for exactly one client. The instance is never invoked concurrently which avoids a lot of trouble in its implementations. The `BackendConnection` defines 12 functions with the following tasks:

- Connecting to the database.
- Closing the database connection.
- Executing a simple query.
- Preparing a query with dynamic parameters.
- Executing a prepared query with given `SqpValues` as parameters.
- Fetching data from a cursor.
- Releasing specified prepared statements and cursors.
- Setting features to a specific value.
- Committing the currently running transaction (commit).

- Aborting the currently running transaction (rollback).
- Getting information about a given *subject* and optional *detail*.
- Retrieving a specified chunk of a LOB.

The event driven approach of the server is also continued in the definition of the functions: Each function must be non-blocking and has to provide a callback function to be invoked whenever errors occur or results are available. Complete definitions of the introduced interfaces and more information on the implementation of the callbacks can be found in appendix A.5.

The communication between the `BackendConnection` and the DBMS is a typical example for blocking operations, because the `BackendConnection` has to actively wait for results from the external DBMS. So to integrate these blocking operations into the non-blocking API, the backend has basically two choices: it can use an event driven network interface or an `AsyncExecutor` object. The `AsyncExecutor` is an abstraction of the Vert.x interface to execute functions in a worker thread and to get notified about the result. It's passed to the backend on initialization, so it can be made accessible to the `BackendConnection`s. Therefore it's very easy for the backend to execute blocking operations whenever necessary.

4.3.2. The PostgreSQL Backend

The first major backend implementation is the PostgreSQL backend. As the scope of the thesis is limited, this backend hasn't been written from scratch, but uses a "core" submodule of the existing JDBC PostgreSQL driver. This saves a lot of work, since this package contains the official Java implementation of the protocol. However, the downside is that it's a blocking API, so most operations have to be executed using the `AsyncExecutor`, and not by using an event driven network interface. Many operations, like preparing and executing queries, are just a matter of calling the correct PostgreSQL API functions with modified arguments. Therefore only the more interesting aspects are discussed in detail in the following.

Data Type Conversion

When data is sent to or fetched from the server, it needs to be converted from standard SQP types to PostgreSQL types or vice versa. For most types there is a simple mapping,

as many of them are the same and only need a different name and format. However, PostgreSQL's type system supports many more types than SQP and is also extensible, so not all types can be mapped in a simple way. Therefore two different approaches are used:

1. The PostgreSQL data type is mapped to a compatible SQP data type, even if their semantics differ somewhat. For example, PostgreSQL's `money` type is mapped to a SQP `Decimal`, since it is able to represent the same value without loss in precision.
2. Unknown or unmappable types are converted to `VarChar`. As all types can be directly fetched from the database in text format, it's simply possible to use these text representations as `VarChar` values, even if the backend doesn't know the type. In this way the client is at least able to receive the data at all.

The alternative is to send or receive these special data types in a "native" format. For example the data type `point` is supported as the native type called `pg_point`. If the client allows the server to use the native format for `pg_point`, a point is sent as a tuple of two 8 byte floating point numbers. Otherwise, it's sent as a `VarChar` formatted as `(x.xxx, y.yyy)`.

For communication with the DBMS nearly all of PostgreSQL's data types are encoded as text in the current backend implementation. Implementing them in binary format is tedious, even for those that are sent in binary format in the JDBC driver, because the binary formats are undocumented and the algorithms used in the JDBC driver are not part of its "core" submodule. This means that there is still room for performance improvements in the future.

While the mapping is clearly defined when data is fetched from the DBMS, it's ambiguous when the client sends data which is not in native format. If for example the client sends a `Decimal` value, it might refer to PostgreSQL's `decimal` or `money` type. Fortunately, PostgreSQL is very flexible when receiving textual representation of data, so it can automatically determine the correct data type and cast the input if necessary. If binary data formats shall be supported by the backend, it has to be checked if PostgreSQL is equally flexible with binary data. If not, the backend should either stick to the text format for ambiguous type mappings or request information about the expected data type from the DBMS.

Support for Parameter Batches

SQP allows to execute a prepared query with placeholders multiple times by sending a batch of parameters. However, the PostgreSQL protocol doesn't support this. So instead, the prepared query is explicitly executed multiple times via the protocol, each time with a different set of parameters from the batch.

Scrollable Cursors

As described in subsection 2.2.2, the PostgreSQL protocol does not support scrollable cursors at the protocol level. So when fetching data, only the next tuples can be fetched from the database, not previous tuples. In SQP, scrollable cursors are supported by setting a flag in the **SimpleQuery** or **ExecuteQuery** messages, so the backend needs to close this gap.

Currently this is done by caching the fetched data if the opened cursor has been defined to be scrollable. Therefore the backend responds to requests for previous tuples by accessing the cache instead of fetching them from the database. This approach can of course become very memory consuming for large data sets.

A more elegant approach would be to implement scrollable cursors implicitly at the language level. To do this, the backend needs to send commands like `DECLARE CURSOR`, `FETCH`, or `MOVE` to the DBMS. So if a `FETCH` statement with the direction set to `PRIOR` is executed, the protocol fetches the previous rows. This approach requires more network roundtrips between the backend and the `DBMS`, since it needs to execute more statements. It is therefore necessary to investigate the performance impact and to review this solution in comparison to the caching solution.

Protocol Operations at the Language Level

Other operations that are not supported by PostgreSQL at the protocol level have been implemented by using statements at the language level. For example, commit and rollback of a transaction have been implemented by sending a `COMMIT` or `ROLLBACK` statement as simple queries to the database. Although the PostgreSQL API supports auto-commit, it also implements it at the language level: When the first statement is executed without auto-commit, the PostgreSQL API automatically executes a simple `BEGIN` query before, starting

the transaction. This shows that the whole transaction management is implemented at the language level in PostgreSQL.

4.3.3. The Transbase Backend

For the same reasons as for the PostgreSQL backend also the backend for Transbase uses a sublayer of the official JDBC driver which is called "TBX". Since it allows to implement many operations in a straightforward way, only special aspects or those that completely differ from the PostgreSQL backend are described.

Transbase Data Types

Since Transbase only has a fixed set of supported data types, the mapping to SQP types is much easier than in the PostgreSQL backend. There is no need to define a standard mapping to `VarChar` because there are no unknown data types. Some ambiguities exist in the Transbase backend as well. Transbase for example supports multiple types to represent binary data. Fortunately, it is very flexible when it comes to data type compatibility: Transbase defines several data type categories and all types are compatible with a defined hierarchy (see [Gmb10, p. 15f]). When dealing with ambiguous mappings, the backend only needs to make sure to use the lowest type of the hierarchy which can represent the data correctly. Transbase can then implicitly upcast it if required.

A specialty of Transbase is the ranged `DATETIME` data type, as introduced in section 2.3. Its concrete flavor is usually dynamically defined by two range qualifiers that restrict the temporal fields to any subrange between *years* and *milliseconds*. However, supporting this type as a native type is not straightforward, since SQP's type system doesn't support dynamic native data types.

To resolve this issue, the backend treats each `DATETIME` flavor as a different native type with the name `TB_DATETIME[XX:YY]`, where `XX` and `YY` are range qualifiers. Each data type is represented by a tuple of integers, where the tuple size and value limits depend on the concrete flavor. For example the `TB_DATETIME[DD:MI]` data type is automatically recognized to be a triplet of integers which represents the *day* (between 1 and 31), *hour* (between 0 and 23), and *minute* (between 0 and 59). As there are 28 different combinations of possible `DATETIME` flavors, they haven't been implemented manually. Instead, the backend parses the native type name and interprets the range qualifiers.

The mapping to static types is also preserved by Transbase's `TypeRepository`: When initialized, the `TypeRepository` generates the name of all possible `DATETIME` flavors and for each flavor it automatically generates a matching JSON schema with the correct limits for the individual fields. Since the `TypeRepository` is only initialized once at the backend initialization, also the generation of the type names and schemas is only done once. This implementation is a good example of how dynamic data types can be mapped to the SQP model for native data types.

Native Support for Parameter Batches

The Transbase protocol directly supports the execution of queries with a batch of parameters. So instead of explicitly executing the query multiple times, the parameter batch is converted to Transbase types and sent to the DBMS for execution. This saves network roundtrips and is more efficient since the DBMS can optimize this type of batch execution.

Native Scrollable Cursors

While the PostgreSQL backend requires to find an own solution to support scrollable cursors, it was easy to implement this for Transbase. Since the Transbase protocol itself supports scrollable cursors, they can be transparently created on the DBMS without any additional effort. When fetching data, the backend can pass parameters to define the fetch direction and a possible offset to access the requested data.

Explicit Transaction Management

One fundamental difference to the PostgreSQL API is that in Transbase all queries need to be associated to a transaction object when being executed. To implement support for auto-commit mode, the backend has to create a transaction object explicitly before the query is executed. Commit and close of the transaction object are performed afterwards. Thereby all steps are implemented by the Transbase protocol with specific requests to the DBMS. They are not allowed at the language level as in PostgreSQL.

Manipulating the SQL Query

Although explicitly excluded from the scope of this thesis, the SQL queries are slightly modified by the Transbase backend. However, this work was just adapted from the Transbase JDBC driver and is only used to change the names of some standard SQL functions to database specific function names. This shows the potential of the backends to support standardized SQL syntax, even if it's not part of the DBMS' SQL dialect.

4.3.4. Summary

Both the PostgreSQL and Transbase backends use sublayer APIs of their JDBC driver which saves work for re-implementing the database protocols. Many of the operations specified by the `BackendConnection` interface can be implemented with simple conversions and API calls. Unfortunately, both APIs are blocking APIs, so they need to be executed by worker threads in order to be integrated in the event driven approach of the proxy server.

Nevertheless, both backends have some specialties that require individual approaches to implement them. They provide good examples of how protocol actions can be solved at the language level, how missing features can be implemented by the backend, or how dynamic types can be supported by the static type extension system of SQP. Furthermore, they show how extensive tasks like the explicit transaction management can be simplified by the SQP protocol.

On the other side, some difficulties like ambiguous data type mappings have been revealed. Thanks to the flexibility of the DBMSs, they didn't turn out to be fatal for the backend implementations.

4.4. The SQP Client API

The common approach for a new database protocol would be to implement it as a JDBC driver, since JDBC is the official CLI implementation for Java. However, to exploit the opportunities of this thesis, the client module has been developed as an independent API instead. Therefore it is not restricted by the specifications of JDBC, but can to use new approaches based on up-to-date technologies in order to present an alternative API to communicate with databases.

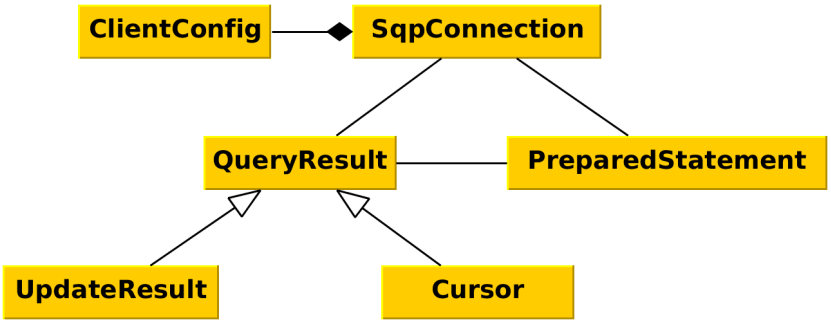

Figure 4.7.: The different interfaces that form the Java SQP client API.

The extent of JDBC constantly increased over the last years and it would be too time consuming to completely implement it within the scope of this thesis. For example more metadata requests are defined by JDBC than implemented so far as SQP information requests.

4.4.1. Architecture

The basis for network communication is a WebSocket implementation. In May 2013 the Java Specification Request (JSR) 356 was released which defines a Java API for WebSocket and became part of the Java Enterprise Edition 7 standard[1]. As clients typically use Java Standard Edition (SE), there are also JSR 356 compatible WebSocket implementations for Java SE, as for example Tyrus[2] or parts of Jetty[3]. Since JSR 356 may become a standard for an upcoming Java SDK, it makes sense to use such an implementation. Tyrus is used in this implementation for basic WebSocket support, since it is more compact than including Jetty.

Figure 4.7 shows the interfaces that actually define the client API. Only the interfaces `SqpConnection`, `Cursor`, and `PreparedStatement` are important for the client to interact with the server. Optionally, the `ClientConfig` interface can be used to set connection-wide settings, as for example the data format, which is MessagePack by default. The `QueryResult` interface is just a "tagging interface" that is used to cohere possible outcomes of a query which can be either a `Cursor` or an `UpdateResult`.

When a query is sent to a DBMS, it may take some time to get the results because of both the network communication and the query execution itself. If the workload of the

[1] `https://jcp.org/en/jsr/detail?id=356`
[2] `https://tyrus.java.net/`
[3] `http://www.eclipse.org/jetty/`

network is high, or the query is computational expensive, the occurring delay will be clearly noticeable for the client application. Common APIs like JDBC block the executing thread while waiting for the query result, or abort because of a timeout. If the query isn't executed in a separate thread, the whole client application is wasting valuable CPU time. As an alternative approach the client has been implemented as an asynchronous non-blocking API which allows to either dynamically react on available results, or to block execution and wait for the results until they are needed.

However, this approach is not event driven in the same way as the server implementation since there is no platform in use which runs an event loop and makes sure event handlers are invoked in the correct thread context. Therefore using callbacks in this architecture is a risky approach, since they might be invoked from different threads or even concurrently. It will also quickly get very hard to keep track of the running main thread, when only callbacks are used to invoke other actions. This would require the client application to be designed very carefully with regard to these issues, which is not a reasonable approach.

An alternate way is to implement the client API with structures which are called "Futures" or "Promises". A Future is an object that represents a result that will be available at a later time, but isn't available yet. Java supports Futures since version 1.5, but the associated features were very limited, so they weren't very popular. This changed with Java 8: A new class called `CompletableFuture` was added that acts like a monad. A monad is a construct from functional programming languages that allows to represent a whole context of computations which can be modified from the outside [She14]. Using them allows the client to easily construct asynchronous actions while not losing track of operational contexts. This approach is illustrated in the following in more detail.

4.4.2. Different Ways of Using the API

Although the client API executes all operations asynchronously, it makes sure that they are executed in the correct order on the server (see appendix A.6 for details). However, asynchronous execution means that nearly all operations return a `CompletableFuture` instead of the expected result. This allows the client to use the API in three different ways:

Fire-and-Forget: The easiest way is to invoke the database operations and don't care about the results, as shown in Listing 4.1. Since they are executed asynchronously, the function would not throw an exception if the execution failed. While this way is

the easiest and fastest, it doesn't allow sensible error handling. Database operations are executed even if previous operations failed.

```
1 conn.execute("DELETE FROM users WHERE name='Foo'");
2 conn.execute("INSERT INTO users (name, age) VALUES ('Foo', 43)");
```

Listing 4.1: Queries being asynchronously executed in the fire-and-forget way without checking for their success.

Conventional: The client can simply wait for the result of a `CompletableFuture` by calling its `join` method. This method then either returns the awaited result or throws an exception if the database operation failed, so further operations won't be executed. Thus by simply appending ".join()" to operations, as shown in Listing 4.1, the API behaves like a conventional, blocking API.

```
1 String query = "DELETE FROM users WHERE name='Foo'";
2 UpdateResult result = conn.executeUpdate(query).join();
3 System.out.println("Removed " + result.getAffectedRows() + " rows.");
4 conn.execute("INSERT INTO users (name, age) values ('Foo', 43)").join();
```

Listing 4.2: Queries being executed in a blocking way that returns the awaited results or throws an exception, like conventional APIs.

Event driven: The `CompletableFuture` class provides over 50 monadic methods that allow dynamic reaction on the result or failure of an execution, as shown in Listing 4.3. The code defined in lines 3 to 5 is only executed if the query invoked in line 1 is successful. Since the operation in line 5 also returns a `CompletableFuture`, the next action in line 7 can simply be chained. It's also possible to execute the event reaction in another thread by calling `thenAcceptAsync`, as shown in line 8 and 9. Finally in line 10, error handling is appended to the concatenation. If one of the chained actions failed, the others will not be executed, but the error handler will be invoked instead.

The event driven way is what makes the API especially interesting and different from other APIs: While it exploits the advantages of the asynchronous execution, it allows sensible error handling and the modeling of dependencies between different database operations in addition.

The client can be implemented using the methods which best meets the requirements of the application. While simple applications may favor one of the simpler ways, a more

```
conn.executeUpdate("DELETE FROM users WHERE name='Foo'")
    .thenCompose(result -> {
        System.out.println("Removed "+result.getAffectedRows()+" rows.");
        String stmt = "INSERT INTO users (name, age) values ('Foo', 43)";
        return conn.execute(stmt);
    }).thenCompose(result ->
        conn.executeSelect("SELECT * FROM users WHERE name='Foo'")
    ).thenAcceptAsync(cursor ->
        processAllResults(cursor, true)
    ).exceptionally(this::handleError);
```

Listing 4.3: Queries are executed in the event driven way by using the monadic methods of `CompletableFuture` in order to dynamically react on asynchronous results and errors.

complex application can use the event driven way. Thereby the client can mix the different approaches seamlessly, which makes the approach very powerful for a number of different applications.

4.4.3. Connecting to a Server

```
try(SqpConnection connection = SqpConnection.create()) {
  connection.connect("localhost", 8080, "/", "proxytest");
  ...
} catch (SqpException e) {
  // catch any problems of the operations with the connection
} catch (IOException e) {
  // error if closing the connection fails
}
```

Listing 4.4: Example of how to create a connection from the client to a server.

To make the ideas of the client API comprehensible, they are illustrated by some simple programming examples. Listing 4.4 shows how the connection to the server can be established. The first line shows a *try-with-resources* construct that makes sure the connection is closed at the end, regardless of errors. The second line is used to actually connect to the database "proxytest" on server "localhost", which is accessed on port "8080" and path "/". This call is asynchronous, so there is no guarantee for the existence of an actual connection after the function call completed.

Lines 4 to 8 deal with error handling: A `SqpException` might occur when the connect or the following actions fail and the `IOException` will be thrown if closing the connection fails.

4.4.4. Query Execution

The following examples show code that can be executed with an established connection, so it would be integrated in line 3 of the previous example.

Simple Queries

```
1 CompletableFuture<updateResult> future = connection.executeUpdate(
2   "DELETE FROM sales WHERE revenue < 1.0"
3 );
4 future.thenAccept(updateResult ->
5     log(Level.INFO, "Sanity check removed " +
6         updateResult.getAffectedRows() + " revenues < 1.00.")
7 );
```

Listing 4.5: A simple SQL query execution with dynamic reaction on the result.

Listing 4.5 shows how a simple, text-only query can be executed. There are three functions that can be used to invoke a simple query execution: `execute`, `executeUpdate`, and `executeSelect`. They merely differ in the result type that is expected by the query: The first expects a generic `QueryResult`, whereas the others expect an `UpdateResult` or a `Cursor`, respectively.

The actual return value of the function is a `CompletableFuture`, as shown in line 1. It can be used to set up a follow-up action, e.g. an action to log the result. This code (line 5 and 6) is then executed as soon as the result is present, in the context of the thread that reports the result.

Prepared Queries

Listing 4.6 shows an example of a prepared query whose dynamic parameters are explicitly bound by the API. The first operation already prepares the query on the server and creates a `PreparedStatement` object to work with. This operation is asynchronous, so

```
1  PreparedStatement stmt = connection.prepare(
2    "SELECT SUM(s.revenue) as ‘revenue‘, e.name as ‘name‘" +
3    "FROM employees e, sales s" +
4    "WHERE employees.id = s.eid AND location = ? AND year > ? " +
5    "GROUP BY e.id"
6  );
7  CompletableFuture<Cursor> revenueFuture =
8    stmt.bind(0, "Germany").bind(1, 2000).executeSelect();
9
10 timeExpensiveOperation();
```

Listing 4.6: A prepared query with parameter binding.

the preparation of the statement is not necessarily finished on the server when working with the object. Line 8 shows how the `PreparedStatement` can be used in a *fluent* way to bind parameters. *Fluent* means in this context that the `bind` operations always return the object itself, so they can be chained as in the example.

The `bind` method is overloaded a number of times for different parameter types, so it can be used to conveniently bind different Java types. The actual conversion from the Java type to an `SqpValue` is performed by the `PreparedStatement`. In addition, a generic bind method is supported that can bind any `SqpValue`. If binding fails, for example because the input data was invalid, the bind methods throw an `SqpException`. This is another example of what kind of errors can be caught at the end of the first example (Listing 4.4).

Further features are the binding of custom data types and sending a whole batch of parameters. To bind custom types, a `bind` method is provided which simply takes the parameter index, the custom type name, and the JSON-format value. To send batches, the method `addBatch` can be executed to save the bound parameters. All following calls to `bind` methods then refer to a new parameter set. Both features are shown in the examples of section 5.2.

The statement is executed on line 8, where `executeSelect` is used to signal that the query should result in a `Cursor`. As the query contains computationally intensive aggregations and grouping functionality, it could take some time to be computed for large tables. Thanks to the asynchronous API, the program doesn't need to wait for the results, but can execute some time expensive operations in the meantime, which would have to be done later anyway.

4.4.5. Receiving Data

```
1  try (Cursor cursor = future.join()) {
2    System.out.println("German employee revenues since year 2000");
3    while(cursor.nextRow()) {
4      System.out.println(
5        cursor.at("name").asString() + " made a total of " +
6        cursor.at("revenue").asBigDecimal()
7      );
8    }
9  } catch (CompletionException e) {
10   // deal with the error while execution
11 } catch (SqpException e) {
12   // problems with getting data from the cursor (type conversion, etc.)
13 }
```

Listing 4.7: A cursor retrieved from CompletableFuture is used to iterate over the results of a query.

In Listing 4.7 the API is used in the conventional way, where the `join` method is executed to wait for the results of the `revenueFuture` from the previous example. So if the execution of the query results in errors, `join` will throw a `CompletionException` that can be caught and handled. Otherwise, it will return the awaited `Cursor` object.

Note that a *try-with-resources* construct is used because the resulting cursor is a *closeable* resource that releases the cursor on the server. Although this would be done automatically when the connection closes, it's better to do this manually, so it's not forgotten when the code gets more complex and the connection is used for further operations.

Lines 3 to 8 show how the cursor can be used as an iterator to process all result rows. Thereby the cursor is always positioned on a specific row, so it is able to access the different columns via their index or name, as in this example. The result of the `at` function is always an `SqpValue` which can be used to convert the data to Java types, as already discussed in subsection 4.1.3. Beside this, the `Cursor` also provides a `getColumnMetadata` function to get the name, SQP type, native type, precision, and scale of the different columns.

Although fetching data is an explicit protocol operation, it's done implicitly by the `Cursor`: whenever it runs out of data, it sends a **FetchData** message to the server to get new data, if available. Due to the nature of the iterator, this fetch operation works *blocking*.

So at this point there is room for improvement. An alternative approach to implement this would be to register a callback that is invoked whenever new data is available for the cursor. However, this would mix the worlds of Futures and callbacks.

If cursor operations like fetching data, finding a column, or data conversion fail, they throw an `SqpException`. This will caught and dealt with in line 11 and 12 in the example.

What hasn't been mentioned before is that the `execute` and `executeSelect` functions of both `SqpConnection` and `PreparedStatement` can use a *scrollable* flag to define whether the cursor to be opened shall be scrollable. If a `Cursor` is scrollable, the `previousRow` method can be used to scroll the cursor in the other direction. Positioning the `Cursor` by specifying an "offset" is not yet implemented in the client API, but might be in the future.

4.4.6. Large Object Management

The separate handling of LOBs in SQP is hidden by the client API. In order to send LOBs to the server, `PreparedStatement` provides the methods `bind(int, InputStream)` and `bind(int, ReadStream)` to create and reference a BLOB or a CLOB, respectively. When these methods are invoked, the `PreparedStatement` will immediately start to upload the streams to the server in the background by using the send-queue. So when the actual execution of the statement is triggered, also the **ExecuteQuery** message is queued and it is therefore assured that the server receives the query after the uploads.

However, this also means that the execution is invoked even if an upload failed. This is not a problem in practice since the execution will fail early in the server, as it notices that a LOB is missing. Still, this not very elegant and might be improved in the future by introducing dependencies between messages in the send-queue. To give the client the possibility to find out which error caused the failure of an execution, the `PreparedStatement` combines all `CompletableFuture`s of the uploads and the execution itself to a new one that represents all the operations in their correct order. This is then returned to the client, so it can use the object to exactly find out which operation caused the error.

Fetching a LOB from the server is also hidden by the API. When a returned value is a LOB, the `Cursor` associates a special `LobStream` with the corresponding `SqpClob` or `SqpBlob` object. The client is then able to call the `getInputStream` or `getReader` methods to access the `LobStream` and can then read the data. Thereby the stream automatically

sends `LobRequest` messages to the server to get new LOB chunks. The chunk size can be defined in the `ClientConfig`, as it might be crucial for the performance.

Since Java's stream concept is designed to be blocking, the underlying `LobRequest` messages of the `LobStream` are executed in a blocking manner as well. At this point it would be worth thinking about an asynchronous alternative concept to get the data whenever available, instead of waiting for it.

4.4.7. Other Operations

The client API also provides other methods which are designed very similar to the protocol operations. For example `SqpConnection` has methods for getting information, setting features, creating type mappings, committing, or aborting transactions. All of these actions are not more than an API abstraction of the protocol, since the protocol is already designed to be easy to use for these simple operations. Some of these operations are used in the examples in section 5.2.

4.4.8. Summary

The presented client API is intended to use a modern approach and to be very easy to use. It allows the client to execute concurrent code, without dealing with tedious and error-prone constructs. Nevertheless it provides enough flexibility to be used in different ways that fit best for the application. as for example by waiting for results, or dynamically reacting to them.

Although the API currently has a smaller feature set than JDBC, it contains approaches that are very different from JDBC but perfectly exploit the advantages of the underlying protocol, which justifies the implementation as an independent API.

Nevertheless, it wouldn't be hard to implement this protocol as a JDBC driver. As both the proposed API and JDBC are based on the ideas of SQL CLI, the JDBC driver for SQP could actually use much of the current implementation. It would just need to implement some wrapper objects and always wait for the actual results in order to be a blocking API. Furthermore, the project would probably have to be extended by some settable features and information requests. This however shouldn't be a complicated matter since the general protocol architecture covers all necessary operations.

5 | Experimental Evaluation

The proposed implementation can be used to show that SQP works as intended. This is done by an automated test framework and some experiments which are introduced and discussed in the following sections. Doing so, the speed of the experiments is not measured, since it would only distract from the real goals of the thesis which are of a functional nature.

This doesn't mean that SQP is slow, but as mentioned in the previous sections, the client library, the proxy server, and the backend implementations all leave room for optimization. Measuring the speed without the intent and effort to optimize before, would not lead to representable results. Therefore the experiments strictly focus on the functional aspects of SQP.

5.1. Integration Tests

During the development of the implementation a test framework was developed which runs over 200 tests to assure that different parts of the implementation work as desired. While some tests are only unit tests which verify that single components work correctly, there are also many integration tests which test the whole implementation stack from the client to the database backend.

All previously discussed aspects of SQP are inspected by the integration tests: simple queries, prepared queries, cursors, information requests, feature settings, transaction management, backend specific data types, custom data types via type mappings, LOB support, and the usage of both data formats JSON and MessagePack.

For each aspect, multiple test cases exist which also check the behavior in case of errors. All integration tests are executed in two "flavors": with the PostgreSQL backend and with the Transbase backend. This assures that both backends implement identical behavior, so

SQP can be used independently from the database. Since the test cases are short and self-describing, but not really exciting, they are introduced in appendix A.7 only.

One idea in the early implementation stages was to develop a JDBC backend that could be used with arbitrary databases. This backend would eventually be slower than "native" database backends and wouldn't support database specific features, but should at least be suitable as a simple "bridge" to easily access different databases through the proxy server.

However, a first prototype quickly revealed that this doesn't work as intended and some tests failed. Especially the transaction management could not be implemented in a way that worked with both JDBC drivers exactly in the same way. This shows that there are differences between both drivers, despite the fact that they were developed to fulfill the same specifications.

Thanks to the test framework this problem can be avoided with SQP: If a new backend is implemented for the proxy server, the existing tests can be run with a new flavor in order to show that the defined scenarios also work with the new backend.

5.2. Example Use Cases

Instead of discussing all test cases, two elaborate experiments have been designed and are discussed in order to show that SQP is suitable for both database specific use as well as database independent use.

5.2.1. Database Specific Experiment

Listing 5.1 shows the main function of a simple application which saves "waypoints". Each waypoint consists of an auto-generated *id*, a *position*, and a specific *timestamp*. The example has explicitly been designed to work with PostgreSQL. Its complete code can be found in appendix A.8.1.

To verify that the correct database is used, the name of the DBMS is requested in the first line. Then the client code executes two DDL statements (lines 3 to 9) to drop the existing table and create a new one. While the first query is database neutral, the second is PostgreSQL specific since it uses the `serial` and `point` data types. Line 9 defines that the success of the table creation should be reported at the standard output.

```
connection.getInformation(String.class, InformationSubject.DBMSName)
          .thenAccept(name -> System.out.println("DBMS: " + name));
connection.execute("DROP TABLE IF EXISTS waypoints");
connection.execute("CREATE TABLE waypoints " +
    "(" +
    "  \"id\" serial," +
    "  \"timestamp\" timestamp with time zone NOT NULL," +
    "  \"position\" point NOT NULL" +
    ")").thenRun(() -> System.out.println("Created table."));

OffsetDateTime now = OffsetDateTime.now();
OffsetDateTime[] timestamps = {
    now.minusHours(1).withNano(123456789), now.withNano(987654321)};
double[][] positions = {{1.5, -3.2}, {1.2, -2.8}};
System.out.println("Data to insert: ");
for (int i = 0; i < timestamps.length; i++) {
    printWaypoint(i + 1, positions[i], timestamps[i]);
}

String insertStmt = "INSERT INTO waypoints " +
                    "(timestamp, position) VALUES (?, ?)";
try (PreparedStatement stmt = connection.prepare(insertStmt)) {
    stmt.bind(0, timestamps[0]).bind(1, "pg_point", positions[0]).addBatch()
        .bind(0, timestamps[1]).bind(1, "pg_point", positions[1]);
    stmt.executeUpdate().thenAccept(
        res -> System.out.println(
            "Inserted " + res.getAffectedRows() + " rows."));
}

String selectStmt = "SELECT * FROM waypoints ORDER BY id";
connection.allowReceiveNativeTypes("pg_point");
try (Cursor cursor = connection.executeSelect(selectStmt).join()) {
    System.out.println("Actual data: ");
    while (cursor.nextRow()) {
        printWaypoint(
            cursor.at("id").asInt(),
            cursor.at("position").as(double[].class),
            cursor.at("timestamp").asOffsetDateTime()
        );
    }
}
```

Listing 5.1: Example that uses PostgreSQL specific features.

In lines 11 to 18 some artificial input data is defined and printed, so it can be compared to the data fetched from the database. Lines 20 to 28 use a prepared statement to insert the data into the created table. The *position* values are bound as a native `point` data type, which is referred to by the proxy server as `pg_point`. Also, the parameters for both input tuples are *batched* and the statement is only executed once.

In line 31, the client allows the server to send `pg_point` in its native format, so it doesn't get mapped to a standard SQP type. The data is then fetched in the following lines and printed. Line 37 shows how the `SqpValue` object returned by the `at` function can easily be used to convert the data to an arbitrary, compatible Java structure, as an array of doubles.

```
1 Data to insert:
2 Waypoint 1: (x=1.5,y=-3.2) at 2015-09-12T11:23:14.123456789+02:00
3 Waypoint 2: (x=1.2,y=-2.8) at 2015-09-12T12:23:14.987654321+02:00
4 DBMS: PostgreSQL
5 Created table.
6 Inserted 2 rows.
7 Actual data:
8 Waypoint 1: (x=1.5,y=-3.2) at 2015-09-12T11:23:14.123456+02:00
9 Waypoint 2: (x=1.2,y=-2.8) at 2015-09-12T12:23:14.987654+02:00
```

Listing 5.2: Output of the database specific experiment with the code of Listing 5.1.

Listing 5.2 shows the output of this experiment. The output shows, that the experiment works, even with the database specific DDL statements and data types in use, as it is able to execute all defined operations.

The sequence in the output is significant: although the retrieval of the DBMS name and the execution of the DDL statements are invoked as a first step in the application, their result is printed after the artificial data is written. This is a consequence of the asynchronous nature of the client. The order of the output of the different database operations corresponds to their order of invocation. This verifies, that the approach described in section 4.4 works as intended.

Another interesting observation can be made in lines 8 and 9: the resolution of the timestamps in the waypoints is lower than in the actual input data. This is because PostgreSQL only supports microsecond resolution [Gro14, p. 118f], so the value is cropped when being inserted into the database.

5.2.2. Database Independent Experiment

The second experiment has been designed to be portable and work independently from a concrete database. This experiment includes the usage of a type mapping, so it is not compatible with all databases, but it is a good example of how this feature can be used. The application stores "friends" into a table by inserting a *name*, a *height* and a *birthday* which contains a month and a day value.

In contrast to the previous experiment, it is required that the table already exists in the database. The required table is called "friends" and has the columns *name*, *height*, and *birthday*. The type of the first column needs to be a variable length character value, the second a 4 byte floating point integer and the third a type for saving a month-day pair. In PostgreSQL, *birthday* is implemented as a `point`, whereas the Transbase model uses a `DATETIME[MO:DD]`. To access the last field the client tries to register a type mapping for a custom type `monthDay` which is defined by the JSON schema in Listing 5.3.

```
1 {
2     "type": "array",
3     "minItems": 2,
4     "additionalItems": false,
5     "items": [
6         {"type": "integer", "minimum": 1, "maximum": 12},
7         {"type": "integer", "minimum": 1, "maximum": 31}
8     ]
9 }
```

Listing 5.3: JSON schema of the `monthDay` type used in Listing 5.4.

While the purpose of using a custom type to save birthdays without the year is questionable, it gives an idea of how type mappings can be used in general. If the target databases would all support spatial data types, a similar approach could be used to write portable applications that access spatial data.

The main function of the experiment is shown in Listing 5.4. Its structure is very similar to the previous experiment and its complete code can be found in appendix A.8.2. In the first lines the table is cleared and the DBMS name is retrieved.

Then the type mapping is registered at the server, where `MONTH_DAY_SCHEMA` contains the schema as defined in Listing 5.3. When registering the type mapping, the keywords "point" and "[mo:dd]" are used (line 5), so the server finds the correct data types for

```
connection.execute("DELETE FROM friends");
connection.getInformation(String.class, InformationSubject.DBMSName)
        .thenAccept(name -> System.out.println("DBMS: " + name));
connection.registerTypeMapping("monthDay", MONTH_DAY_SCHEMA,
                               "point", "[mo:dd]")
        .thenAccept(orig -> System.out.println("Mapping to " + orig));

String[] names = { "John Doe", "Mary Jane" };
int[][] birthdays = { {2, 29}, {12, 31} };
float[] heights = { 1.84f, 1.59f };
System.out.println("Data to insert: ");
for (int i = 0; i < names.length; i++) {
    printPerson(names[i], birthdays[i], heights[i]);
}

String insertStmt = "INSERT INTO friends " +
                    "(name, birthday, height) VALUES (?, ?, ?)";
try (PreparedStatement stmt = connection.prepare(insertStmt)) {
    for (int i = 0; i < names.length; i++) {
        stmt.bind(0, names[i]).bind(2, heights[i])
            .bind(1, "monthDay", birthdays[i]);
        if (i != names.length -1) stmt.addBatch();
    }
    stmt.executeUpdate().thenAccept(res -> System.out.println(
        "Inserted " + res.getAffectedRows() + " friends."));
}

String selectStmt = "SELECT * FROM friends";
try (Cursor cursor = connection.executeSelect(selectStmt).join()) {
    System.out.println("Actual data: ");
    while (cursor.nextRow()) {
        printPerson(
            cursor.at("name").asString(),
            cursor.at("birthday").as(int[].class),
            cursor.at("height").asFloat()
        );
    }
}
```

Listing 5.4: Experiment with a custom data type which works with both PostgreSQL and Transbase.

both backends and doesn't accidentally choose a different compatible type to create the mapping. As shown in line 6, the native type which is used by the server for the mapping is printed, so it can be reviewed in the output.

In lines 7 to 14, artificial input data is defined and printed for comparison to the data fetched from the database. The birthdays are already defined as tuples of integers containing the month and day values. Then the input data are inserted into the "friends" table by using a prepared statement with batched parameters.

In lines 28 to 38 the data is fetched from the database and printed. In this experiment the data conversion to a custom data type is shown in line 34.

The experiment is executed twice, once with a PostgreSQL database and secondly with a Transbase database. The output of the two cases is provided in Listing 5.5 and 5.6, respectively.

```
1 Data to insert:
2 John Doe is 1.84m tall and has birthday on Feb, 29
3 Mary Jane is 1.59m tall and has birthday on Dec, 31
4 DBMS: PostgreSQL
5 Mapping to pg_point
6 Inserted 2 friends.
7 Actual data:
8 John Doe is 1.84m tall and has birthday on Feb, 29
9 Mary Jane is 1.59m tall and has birthday on Dec, 31
```

Listing 5.5: Output of the database independent experiment with the code as defined in Listing 5.4 using a PostgreSQL database.

```
1 Data to insert:
2 John Doe is 1.84m tall and has birthday on Feb, 29
3 Mary Jane is 1.59m tall and has birthday on Dec, 31
4 DBMS: Transbase
5 Mapping to tb_DATETIME[MO:DD]
6 Inserted 2 friends.
7 Actual data:
8 John Doe is 1.84m tall and has birthday on Feb, 29
9 Mary Jane is 1.59m tall and has birthday on Dec, 31
```

Listing 5.6: Output of the database independent experiment with the code from Listing 5.4 and using a Transbase database.

As shown in the output, the experiment works with both databases. The defined input data and the data fetched from the database is the same in both executions, as the lines 6 to 9 show. Line 4 shows, however, that two different database types are used.

This can also be seen in line 7, where the native type used for the type mapping is shown: while for PostgreSQL the `monthDay` type gets mapped to a `pg_point`, it is mapped to `TB_DATETIME[MO:DD]` for Transbase.

Because of the type mapping this experiment is rather complicated. Without using a type mapping it would work with much more DBMSs than PostgreSQL and Transbase. Nevertheless, it perfectly shows how code can be made portable in order to work with different databases.

5.3. Implementation in JavaScript

So far only examples in Java were provided, since the client API has been implemented in Java. However, because of the simple design of SQP, it is also easily usable by other languages. JavaScript is a good choice to show an alternative implementation, since it comes with built-in support for JSON and WebSocket.

There is no real client API for SQP in JavaScript. However, less than 30 lines of code are needed to easily establish an SQP connection, send messages and react on the results as shown in Listing 5.7.

The `send` function can be used to send an SQP message and register a response handler. The message is constructed by converting the *content*, which should be a JavaScript object, to a JSON string and appending it to the *id*. The *handler* is then appended to an existing list of handlers.

The `messageHandler` function is used to respond to incoming WebSocket messages (see line 26). If it detects an error message, it prints the message to the console. Otherwise, it checks if there are registered response handlers. The handlers must be JavaScript objects that contain callbacks for each message type they should react to. So if a handler has a callback defined for a specific message ID, it gets invoked (see line 17). Furthermore, handlers must set the *end* member to an ID to define when they finished handling response message. If the message id matches this value, the response handler is removed (see line 18).

```
databaseName = "exampleDB";
mainmethod = main;
handlers = [];
function send(id, content, handler) {
   if (handler) handlers.push(handler);
   websocket.send(id + JSON.stringify(content))
}
function messageHandler(evt) {
    var msg = evt.data;
    var id = msg[0];
    var payload = msg.length < 2 ? null : JSON.parse(msg.substring(1));
    if (id == '!') {
        console.log("Error " + payload.errorType + ": " + payload.message);
        return;
    }
    if (handlers.length < 1) return;
    if (id in handlers[0]) handlers[0][id](payload);
    if (handlers[0].end == id) handlers.shift();
}
function start() {
    send('H', {database : databaseName}, { end: 'r', r: mainmethod });
}

websocket = new WebSocket("ws://localhost:8080/");
websocket.onopen = start;
websocket.onmessage = messageHandler;
```

Listing 5.7: Utility functions to start a SQP session, send messages and for reacting on results.

Last but not least, the `start` function sends the **Hello** message to the server to connect to a database and invokes the main function as soon as the server responds with a **Ready** message. The `start` function is invoked as soon as the WebSocket connection is open (see lines 24 and 25).

Listing 5.8 shows the main function of an experiment that has been designed to be functionally equivalent to the previous experiment, as discussed in subsection 5.2.2. The full code, with the functions introduced before, the JSON schema and a print function can be found in appendix A.8.3.

The code resembles the main function of the Java implementation: the table is emptied, the DBMS name is retrieved and the type mapping is registered. Then the test data

```
send('S', {query: "DELETE FROM friends"});
send('I', {subject: 'DBMSName'}, {
    end : 'i',
    i: function(msg) { console.log("DBMS: " + msg.value)}
});
send('M', {name: "monthDay", schema: MONTH_DAY_SCHEMA,
           keywords: ["point", "[mo:dd]"]},{
    end: 'm',
    m: function (msg) { console.log("Mapping to " + msg.native) }
});

var names = [ "John Doe", "Mary Jane" ];
var birthdays = [[2, 29], [12, 31]];
var heights = [1.84, 1.59];
console.log("Data to insert: ");
for (var i = 0; i < names.length; i++) {
    printPerson(names[i], birthdays[i], heights[i]);
}

var insertStmt = "INSERT INTO friends " +
                    "(name, birthday, height) VALUES (?, ?, ?)";
send('P', {query : insertStmt});
send('X', {
        parameterTypes: ['VarChar', 'Custom', 'Real'],
        customTypes: ["monthDay"],
        parameters: [
            [names[0], birthdays[0], heights[0]],
            [names[1], birthdays[1], heights[1]],
        ]},{
    end: 'x',
    x: function(msg) {
        console.log("Inserted " + msg.affectedRows + " friends.")
}});

var handler = { end: 'e' };
handler['c'] = function(msg) { console.log("Actual data: ")};
handler['#'] = function(msg) {
    printPerson(msg.data[0], msg.data[1], msg.data[2])
};
var selectStmt = "SELECT name, birthday, height FROM friends";
send('S', { query : selectStmt }, handler);
```

Listing 5.8: The main function of a JavaScript implementation which is equal to Listing 5.4.

is defined and printed. It's then inserted into the table via a prepared statement and batched parameters, while using the custom format of the `monthDay` type. At last the data is fetched from the table by using a simple query which doesn't require separate data retrieval.

Obviously the commands are primarily message oriented, so the developer needs to know how SQP messages work. Therefore it is advisable for the future to abstract the usage from the protocol internals with more methods. On the other hand, this example shows how simple it is to access the database, even if there is no real client API implementation. Similar to the Java client API the execution of the different database operations don't affect each other, as they are sent without checking for the success of the previous operation.

When the experiment is executed, for example by integrating it into a HTML web page or pasting it into a browser's JavaScript console, it prints *exactly* the same output as the Java implementation (see Listing 5.5, respectively in Listing 5.6). Since JavaScript works event based even the sequence of the output is the same.

This experiment shows that SQP can be used with programming languages other than Java, if there is support for WebSocket and JSON (or MessagePack). The effort it takes to implement a client API is low, depending on the level of abstraction that it should provide.

5.4. Summary

The Java implementation of SQP is verified to work with an automated test framework and some experiments. The test framework is not only useful for the purpose of this thesis, but is also a good method to assure during further development that backend implementations for the proxy server behave in the same way for different database types.

The experiments show, that SQP is useful for both database specific and database independent applications. Furthermore the experiments are good examples to demonstrate how the client API can be utilized for concurrent execution with dynamic reaction on results.

The JavaScript implementation shows that no complex client API is needed to execute database operations. Instead, database operations can be executed with very few lines of code if there is support for WebSocket and JSON or MessagePack. Since these technologies are very popular, implementations exist for most programming languages.

6 | Conclusion

The complexity of using SQL CLI as the standard for accessing databases was known for a long time. Nevertheless, ODBC, JDBC, and ADO.NET became the standard solutions for database access. The idea of this thesis is to take a step back and go another way by providing a simple and portable wire protocol for common database operations that avoids the complexity of the existing approaches.

6.1. Related Work

There are a number of projects that have similar intentions or provide similar software at the first sight. However, all of them differ considerably from SQP.

The **Distributed Relational Database Architecture (DRDA)** was designed by IBM from 1988 to 1994, a time period during which they did not join the SQL Access Group which developed the SQL CLI [McJ97, p. 58f]. As the name suggests, DRDA is not just a protocol, but an architecture with a client, an application server, and a DBMS, communicating via different protocols [BLM+01, p. 83ff]. However, it got never really popular and has a very much higher complexity than SQP.

SQL Relay[1] is an open source database connection management package. It also includes a SQL proxy server which is database independent. However, the general focus is on features like connection pooling, drop-in replacement, query routing, and more. Although it also provides a common protocol which is usable to communicate with the proxy server, the protocol is undocumented and has not been designed to be usable for other clients or servers. In contrast to SQP so far it supports some advanced database independent operations like table creation.

[1] `http://sqlrelay.sourceforge.net/`

Also **Data Direct**[1] from Progress[2] is a product which uses a wire protocol to abstract from specific data sources. However, this product is highly proprietary and closed source, so no details are known and it cannot freely be used by clients and servers.

The open source project **webstore**[3] is a web-API that supports multiple "datastore" backends and is not limited to relational databases. It allows to execute arbitrary SQL commands in a database via the web-API. Beside this, it cannot compete with a complete wire protocol as it doesn't support common relational database operations like transaction management and cursors.

6.2. Evaluation of Objectives

After the discussion of the protocol design, the implementation of SQP, and the proposed experiments, the objectives of this thesis defined in section 1.2 can be evaluated:

Easy to implement: By using WebSocket and JSON or MessagePack, the implementation of the protocol is very easy. This was demonstrated in section 5.3 where less than 30 lines of JavaScript are required to access a database.

Easy to use: The messages supported by the protocol are based on very basic operations that are also defined by the SQL CLI. Therefore executing these operations in SQP is only a matter of using the correct messages, as shown in section 5.3. The Java client API shows that this can be reduced to a small number of objects to easily invoke all operations.

Support all common database operations: The integration tests and experiments assure that common database operations like simple queries, prepared queries, cursors, and transaction management actually work as intended. There are elaborate tasks which are not yet supported, but should be implemented in the future as described in section 6.3.

Work independently from a specific database: By abstracting data types and not requiring database specific information, SQP is designed to be completely database neutral. Furthermore, type mappings are supported by SQP to allow the use of non-standard data types in a database neutral way. The SQL proxy server with

[1] `https://www.progress.com/datadirect-connectors`
[2] `https://www.progress.com/`
[3] `https://github.com/okfn/webstore`

exchangeable DBMS-specific backends is very powerful and can be used to demonstrate the fulfillment of the design goals of SQP. Also the experiment discussed in subsection 5.2.2 confirms this.

Allow database specific operations: Since the protocol provides support for native data types, custom information requests, and feature settings, the client code can use all database specific features, unless portability is a strict requirement. This is confirmed by the experiment discussed in subsection 5.2.1.

Usable for real-life scenarios: Although not being the focus of the protocol design, using MessagePack as an alternate data format allows the protocol to be space efficient while maintaining the advantages of JSON. Currently, the implemented proxy server already allows to use SQP for accessing PostgreSQL and Transbase. However, SQP can easily be extended for other databases. With the Java API for SQP, a modern approach was implemented that can be used in different ways to fit the needs of the client application best.

Easily extensible: The flexible concepts of information requests, feature settings, and data types allow SQP to be easily extensible without breaking established functionality. Furthermore, SQP's message design also allows to add completely new messages, without affecting existing implementations.

Summarizing, it has been shown that the design and implementation of SQP meets the defined objectives.

6.3. Outlook

Although the current state of SQP and its implementation is already sufficient to execute common database operations, they also provide much potential to be further extended in the future.

6.3.1. Possible Future Extensions of the Protocol

What has been excluded in the thesis, but should be implemented very soon, is authentication. So far, the server needs to run in a trusted environment, as the current implementation does not support login methods. A very straight-forward approach would be to use the "HTTP Basic authentication" technique, since the underlying WebSocket

protocol is HTTP compatible. As this approach only affects the WebSocket connection, there wouldn't even be a need to change SQP to support this.

Furthermore, some useful but elaborate features could be implemented. These are for example updatable cursors, transaction isolation modes, function calls, requesting details about dynamic parameters of a prepared query, or returning auto-generated values when inserting data. Most of these functions would only require extension of existing features like information requests or feature settings, or implement some new simple messages. Either way, they do not require really new concepts, so their implementation should be fairly easy.

Another idea for extending SQP is to abstract important DDL statements at the protocol level, to have database independent support for creating tables, or managing constraints. These abstractions would benefit from SQP's type system as a way to refer to data types in a database independent manner.

6.3.2. Possible Future Extensions of the SQL Proxy Server

The SQL proxy server has considerable potential for future use cases. Of course, more backends would significantly increase the value of this server, so it could be used with a larger number of database types, for example with MySQL, Oracle, Microsoft SQL Server, or DB2.

Besides this, SQP could also implement some application server features like drop-in replacement of databases, connection pooling, load balancing, or throtteling. Even completely different non-standard functionality of databases like the electronic payment for data could be handled by the server.

Furthermore, it could completely decouple the client authentication from the database authentication, so modern authentication methods like OAuth could be supported, or a more detailed user rights management could be introduced, without affecting the DBMS.

Even SQL analysis and modification could be implemented in the proxy server. Although this is a very challenging task, it would open a whole new range of possibilities.

All in all, it can safely be stated that the protocol SQP is a new approach to easily access different relational databases in a portable manner. It has great potential for future use, which makes the implementations valuable and the thesis successful in its intention.

List of Figures

List of Tables

List of Listings

Bibliography

[BLM+01] Hernando Bedoya, Daniel Lema, Vijay Marwaha, Dave Squires, and Mark Walas. *Advanced Functions and Administration on DB2 Universal Database for iSeries*. Redbooks. International Business Machines Corporation, fourth edition, 2001.

[Bra14] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014.

[Com95] X-Open Company. *Data Management: SQL Call Level Interface (CLI)*. X Open document / C: X Open document. X/Open Company, 1995.

[DS10] M.J. Donahoo and G.D. Speegle. *SQL: Practical Guide for Developers*. The Practical Guides. Elsevier Science, 2010.

[FM11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[Gmb10] Transaction Software GmbH. Transbase SQL Reference Manual, 2010. Version: 6.8.1.40.

[Gmb15a] Transaction Software GmbH. Achievements and Milestones. `https://www.transaction.de/en/about/company-history.html`, 2015. Accessed: 2015-09-01.

[Gmb15b] Transaction Software GmbH. Transbase Product Sheet. `https://www.transaction.de/fileadmin/downloads/pdf/Datenblaetter/Product-Data-Sheet-Transbase-SQL-Database-Management-System-english.pdf`, 2015. Accessed: 2015-09-01.

[Gri13] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, 2013.

[Gro14] The PostgreSQL Global Development Group. *PostgreSQL 9.4.0 Documentation*. `http://www.postgresql.org/docs/manuals/`, 2014.

[GZC13] F. Galiegue, K. Zyp, and G. Court. *JSON Schema: core definitions and terminology*. Internet Engineering Task Force, January 2013. Available at `http://json-schema.org/latest/json-schema-core.html`.

[Int92] International Organization for Standardization. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. 1992.

[Int95] International Organization for Standardization. *ISO/IEC 9075-3:1995: Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI)*. 1995.

[jso13] *The JSON Data Interchange Format*. ECMA International, first edition, October 2013.

[KKH08] K. Kline, D. Kline, and B. Hunt. *SQL in a Nutshell*. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[McJ97] Paul R. McJones. The 1995 sql reunion: People, project, and politics, may 29, 1995. *Digital System Research Center Report*, SRC1997-018, 1997.

[MRP+01] Volker Markl, Frank Ramsak, Roland Pieringer, Robert Fenk, Klaus Elhardt, and Rudolf Bayer. The transbase hypercube RDBMS: multidimensional indexing of relational tables. In *ICDE 2001, Demo Session Abstracts (Informal Proceedings)*, pages 4–6, 2001.

[Ree00] G. Reese. *Database Programming with JDBC and Java*. Java (o'Reilly) Series. O'Reilly, 2000.

[SE07] S. Sumathi and S. Esakkirajan. *Fundamentals of Relational Database Management Systems*. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2007.

[Sha08] Robin Sharp. *Principles of Protocol Design*. Springer Berlin Heidelberg, 2008.

[She14] Oleg Shelajev. Monadic futures in java 8: How to organize your data flow and avoid callback hell. `http://zeroturnaround.com/rebellabs/monadic-futures-in-java8/`, 2014. Online, accessed September 9, 2015.

[Urb14] Jan Urbanski. Postgres on the wire, 2014. PGCon 2014, Ottawa.

Glossary

ANSI

The American National Standards Institute is a non-profit organization for developing and publishing standards for services, products, systems, and more.

ASCII

The American Standard Code for Information Interchange is a very common 7 bit encoding for some basic alphanumeric characters and control characters.

base64

Base64 is an encoding scheme that is often used to represent binary data as an ASCII string.

IBM System R

The first relational database management system. It also included the language SEQUEL, the predecessor of SQL.

ISO

The International Organization for Standardization is composed from multiple national standard organizations and develops and publishes international standards in various fields.

middleware

Middleware is software providing different services to other software while hiding the complexity of the underlying system.

SQL

The database language standardized by ISO and IEC for talking to relational database management systems.

A | Appendix

A.1. JVM Serializers Benchmarks

A benchmark[1] gives an overview over various JVM visualization techniques and an idea about their efficiency. Figure A.1 and Figure A.2 show some recent results of the benchmark. The test has been run on a Windows 8.1 system with a quadcore Intel64 Family 6 Model 60 CPU. The JVM used is "Oracle Corporation 1.7.0_75".

It can be seen that MessagePack (denoted as "msgpack") is one of the best serialization techniques in both time and space. The various JSON techniques are less efficient, especially when it comes to the size of the serialized data.

[1] `https://github.com/eishay/jvm-serializers`

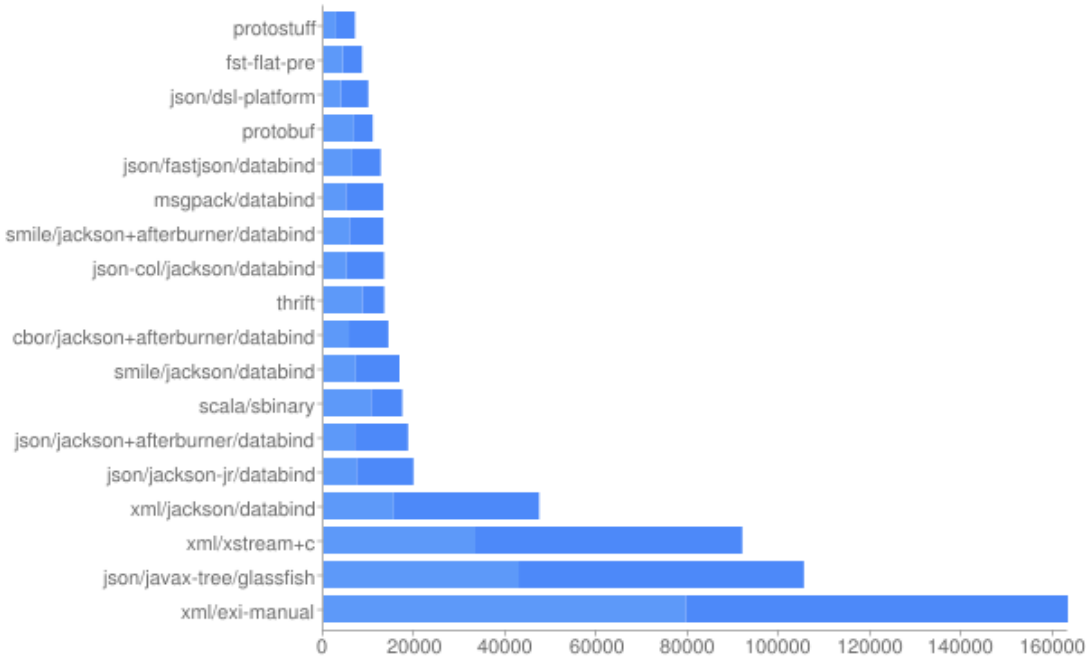

Figure A.1.: Serialization time and deserialization time of various JVM serialization techniques. Taken from `https://github.com/eishay/jvm-serializers/wiki` on 10th September 2015.

Figure A.2.: Size and compressed size in bytes of various JVM serialization techniques. Taken from `https://github.com/eishay/jvm-serializers/wiki` on 10th September 2015.

A.2. SQP Message Examples

The following listings give some examples for important SQP messages. **AuthenticationRequest** and **AuthenticationResponse** have been left out since they haven't been implementeds. Also **Commit**, **Rollback**, **Ready**, **PrepareComplete**, **ReleaseComplete**, **SetFeatureComplete**, **TransactionFinished**, and **LobReceived** have been left out since they have no payload and are therefore trivial (IDs only).

The messages are in JSON format and formatted with newlines and spaces for better readability. On the wire the messages could as well be sent in MessagePack and the formatting would be left out for being more space efficient.

A.2.1. Messages Sent By the Client

```
H {
    "database": "testDatabase"
}
```

Listing A.1: Exemplary **Hello** message which defines the database to connect to.

```
S {
    "query": "SELECT * FROM testTable",
    "cursorId": "SimpleCursor",
    "scrollable" : true,
    "maxFetch": 10
}
```

Listing A.2: Exemplary **SimpleQuery** message which defines the query to execute. If the query creates a cursor, the cursor ID, a flag telling if the cursor should be scrollable, and the maximum number of tuples to be fetched can optionally be defined.

```
P {
    "query": "SELECT * FROM testTable WHERE attribute=?",
    "id": "query1"
}
```

Listing A.3: Exemplary **PrepareQuery** message defining the query to be prepared (might include placeholders) and an optional statement ID which can be used later to refer to this query.

```
X {
    "statementId": "query1",
    "cursorId": "cursor1",
    "scrollable": true,
    "parameterTypes": ["Custom"],
    "customTypes": ["customType"],
    "parameters": [["val1"]]
}
```

Listing A.4: Exemplary **ExecuteQuery** message which defines the statement ID to execute, a cursor ID of the new cursor, a flag to indicate if the cursor should be scrollable, the parameter types, the custom types used, and the parameter batch itself. All fields are optional. Standard IDs are assumed if no specific IDs are given.

```
F {
    "cursorId": "cursor1",
    "position": 0,
    "maxFetch": 10,
    "forward": true
}
```

Listing A.5: Exemplary **FetchData** message which defines the cursor ID to fetch the data from, the position to fetch from, the maximum number of tuples to fetch, and the fetch direction. All fields are optional. The standard cursor ID is assumed if left out. The position and "forward" flag can only be used with scrollable cursors.

```
L {
    "cursors": ["cursor1"],
    "statements": ["query1", "query2"]
}
```

Listing A.6: Exemplary **Release** message which defines lists of cursor IDs and statement IDs to close. Both are optional.

```
I {
    "subject": "TypeSchema",
    "detail": "customType"
}
```

Listing A.7: Exemplary **InformationRequest** message which defines the subject to get information about and an optional detail which is required for some subjects.

```
T {
    "autoCommit": false
}
```

Listing A.8: Exemplary **SetFeature** message which defines the features and values to set as the fields of the payload.

```
M {
    "name": "customType",
    "keywords": ["Text", "deviceId"],
    "schema": {
        "type": "string",
        "pattern": "^/dev/[^/]+(/[^/]+)*$"
    }
}
```

Listing A.9: Exemplary **TypeMapping** message which defines the name of the new type, keywords to prefer some types for mapping, and the JSON schema for the type.

```
G {
    "id": "myLob",
    "offset": 1024,
    "size": 1024
}
```

Listing A.10: Exemplary **LobRequest** message which defines the id of the LOB to get, the offset, and chunk size to get. So this example will fetch the second kilobyte of the LOB.

A.2.2. Messages Sent By the Server

```
! {
    "errorType": "ConnectionFailed",
    "message": "The database 'notExisting' was not found"
}
```

Listing A.11: Exemplary **Error** message which defines an error ID and a message.

```
# {
    "data": ["foo", [23, 2]]
}
```

Listing A.12: Exemplary **RowData** message which represents one data tuple with data of the types sent in the cursor description.

```
x {
    "affectedRows": 13
}
```

Listing A.13: Exemplary **ExecuteComplete** message defining the number of rows affected by the execution.

```
e {
"more": true
}
```

Listing A.14: Exemplary **EndOfData** message sent after **RowData** messages which defines if there is more data to fetch.

```
c {
    "cursorId": "Default",
    "scrollable": true,
    "columns": [
        {
            "name": "firstCol",
            "type": "VarChar",
            "nativeType": "character varying",
            "precision": 80,
            "scale": 0
        },
        {
            "name": "otherCol",
            "type": "Custom",
            "nativeType": "pg_point",
            "precision": 0,
            "scale": 0
        }
    ]
}
```

Listing A.15: Exemplary **CursorDescription** message which defines the ID of the new cursor, if it's scrollable and which attributes (columns) it has.

```
i {
    "responseType": "Text",
    "value" : "PostgreSQL Version 9.2",
}
```

Listing A.16: Exemplary **InformationResponse** message containing the type of the response to an **InformationRequest** and the actual value.

```
m {
    "native": "pg_point"
}
```

Listing A.17: Exemplary **TypeMappingRegistered** message containing the native type which is used for the mapping which was requested via the **TypeMapping** message.

A.2.3. Messages Sent By the Server or Client

```
* {
    "id": "myLob"
}
```

Listing A.18: Exemplary **LobAnnouncement** message to define the ID of the LOB which is sent in an upcoming separate WebSocket message.

A.3. More On Vert.x

Vert.x is a *reactive* framework for the JVM. Reactive in the context means that it is intended to be responsive (fast and consistent response times), resilient (responsive in case of failure), elastic (responsive with different workloads), and message driven (see `http://www.reactivemanifesto.org/`).

A.3.1. Asynchronous Streams and Pumps

An interesting concept of Vert.x are asynchronous streams and so called *Pumps*. Common Java streams usually block in two situations:

1. If there is no data available to read in an `InputStream`
2. If data is written to the target medium in an `OutputStream`

Vert.x solved these problems in two new kind of streams: `ReadStreams` and `WriteStreams`. In contrast to the `InputStream`, a `ReadStream` isn't polled for data (as in the "read" method), but calls a data handler whenever data is available. So there is no situation

where a thread needs to wait for data. If the `ReadStream` produces more data than needed, it can be *paused* and *resumed* at another time, so in between the data handler isn't invoked anymore.

The `WriteStream` has a non-blocking write function. However, as this means that it's possible to feed more data into the stream than it can actually handle, it has a `writeQueueFull` method to check whether it can actually take more data. If the `WriteStream` is full, it can invoke a "drain handler" as soon as it's able to take more data again.

A Vert.x `Pump` is used to connect a `WriteStream` and a `ReadStream`. If the `Pump` is started, it registers the `WriteStream`'s write method at the `ReadStream` as the data handler. When the `WriteStream` is full, the `ReadStream` gets paused until the `WriteStream` calls a drain handler which resumes the `ReadStream`. This concept allows the connection of two streams that are able to asynchronously interact with each other, depending on available data and buffer size.

In the proxy server the `WebsocketSendQueueStream` implements a `ReadStream` and the `WebsocketWriteStream` implements a `WriteStream`. Outgoing data, as encoded messages, are queued in the `WebsocketSendQueueStream` which is connected via a `Pump` to the `WebsocketWriteStream`. When the `WebsocketWriteStream` writes data to the socket, it automatically makes sure to split the data into frames when reasonable.

A.3.2. Non-Blocking Programming

Although non-blocking programming seems a little like "magic" in the first place, it's really just a matter of thinking about the program execution.

One important about non-blocking programming is that functions might return before the actual result is available. This might happen because there was a worker thread started with some blocking operation, or because the verticle must wait for another event, as a network response. For this reason a common approach is to pass callbacks that are invoked when the result is available. So whenever it's unknown whether an action is executed directly or asynchronously, it's important to not think in the normal call-result pattern, but in a one way direction: The task is only started and then the function terminates.

When the called function finishes - with or without interrupting the execution - it needs to invoke the callback function. Not to forget to invoke the callback, and therefore notifying

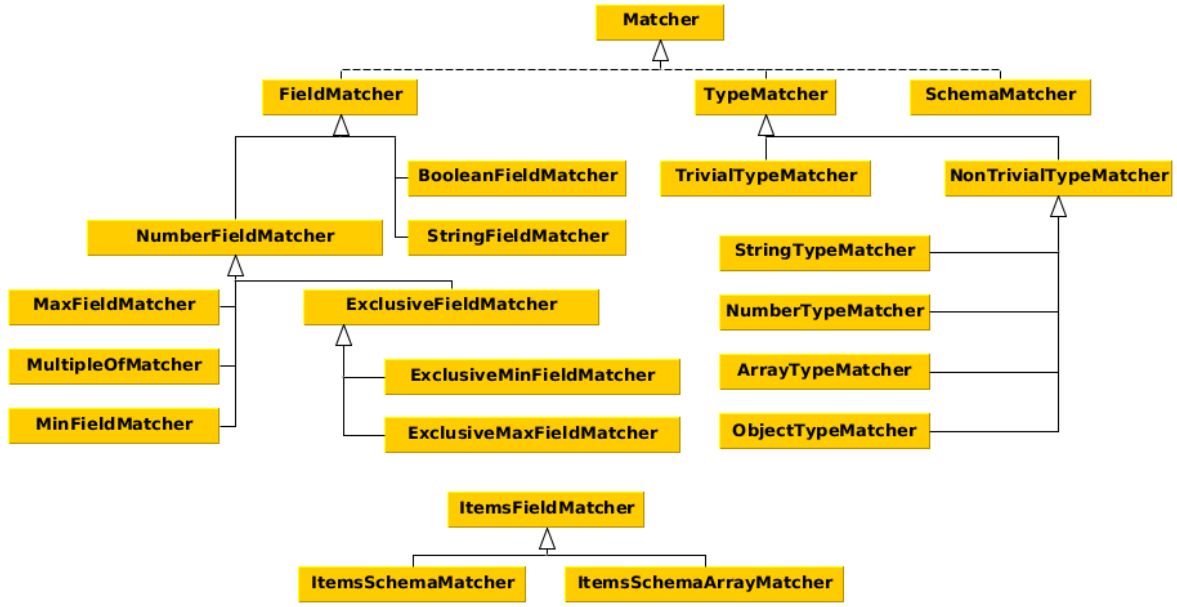

Figure A.3.: An overview over the class hierarchy of the `SchemaMatcher` where each matcher class has a very specific target to match.

the verticle to resume its work is an important task to keep in mind when programming with callbacks.

Another important matter to keep in mind at this point is which thread will execute the callback. Vert.x with its event loop will make sure that a callback is always executed from the event-loop thread, the "native" thread of the verticle. So the developer doesn't need to fear that the verticle will be executed concurrently by different threads, which makes programming easy and avoids the whole error class of concurrency problems.

This is a huge difference to non-blocking systems that work without an event-loop, as for example the client API implementation.

A.4. Schema Matching

A `SchemaMatcher` is an object that is initialized from a JSON schema. This object can then be used to check for compatibility against other schemas, i.e. to check if the represented schema is a subtype of another schema. To do this, the `SchemaMatcher` consists of whole hierarchy of matchers that all care about one single aspect of a JSON schema, as illustrated in Figure A.3. They can be used to check the compatibility of types in general, a specific data type in detail, or a specific fields.

The implemented schema matcher doesn't support all JSON schema features, so for example references to other schemas and advanced relations as `anyOf` aren't supported. However, the current implementation is sufficient for basic tuple-based data types used in SQP.

A.5. Backend Interfaces and Helper Classes

A.5.1. The Backend Interfaces

```
1  public interface Backend {
2      void init(Configuration configuration, AsyncExecutor asyncExecutor)
3           throws ConfigurationException;
4
5      BackendConnection createConnection();
6
7      default TypeRepository getTypeRepository() {
8          return TypeRepository.Empty;
9      }
10 }
```

Listing A.19: The `Backend` interface that each proxy server backend needs to implement.

```
1  public interface TypeRepository {
2      String getSchema(String typename);
3      List<String> getNativeTypes();
4  }
```

Listing A.20: The `TypeRepository` interface that a proxy server backend can implement if it wants to add support for native data types.

```
public interface BackendConnection {
    void connect(String databaseName, Consumer<String> disconnectHandler,
                 SuccessHandler connectionHandler);
    void close();

    void simpleQuery(String sql, String cursorId, boolean scrollable,
                     int maxFetch, ResultHandler<QueryResult> resultHandler);

    void prepare(String sql, String statementId,
                 SuccessHandler successHandler);
    void execute(String statementId, String cursorId,
                 List<List<SqpValue>> parameters, boolean scrollable,
                 ResultHandler<QueryResult> resultHandler);
    void fetch(String cursorId, int position, int numRows, boolean forward,
               ResultHandler<QueryResult> resultHandler);
    void release(Collection<String> statementIds, Collection<String> cursorIds,
                 SuccessHandler successHandler);

    void setFeatures(List<FeatureSetting<?>> featureSettings,
                     SuccessHandler successHandler);
    void commit(SuccessHandler successHandler);
    void rollback(SuccessHandler successHandler);

    void getInformation(InformationSubject subject, String detail,
                        ResultHandler<InformationRequestResult> resultHandler);
    void getLob(String id, long offset, long length,
                ResultHandler<LobStream> resultHandler);
}
```

Listing A.21: The `BackendConnection` interface that a proxy server backend needs to implement for connection specific database operations.

Helper Classes for Communication

Figure A.4 shows the classes that are used for communication with the proxy server. One basic idea about the backends is that they are also required to work non-blocking. Therefore a central class for communication is the `ResultHandler` which is passed to the backend's functions as a callback to the proxy. But it's not just a callback for the results, but also contains a callback for errors. The `ErrorHandler` interface is implemented by the `ClientSession`, so whenever the `ClientSession` passes a callback to the backend, it is

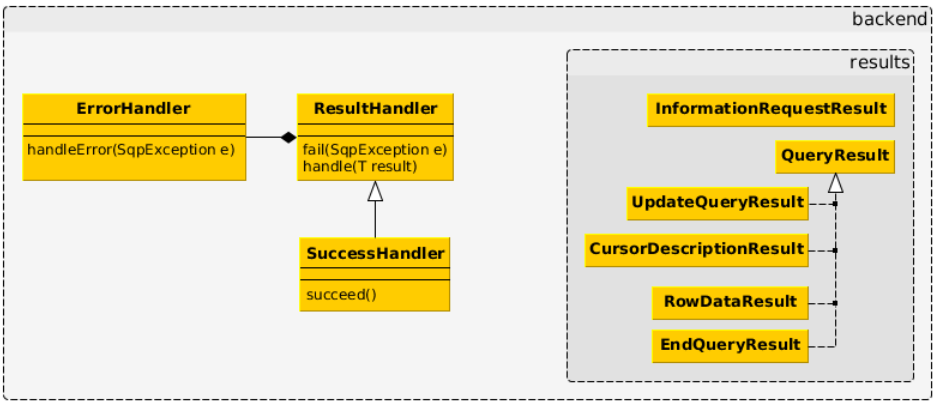

Figure A.4.: Classes of the backend package used for communication with the proxy server.

able to call the `ClientSession`'s error handling method by calling the `ResultHandler`'s `fail` method.

The `handle` of the `ResultHandler` function is used to pass back results. Typical results to be passed back or those of the "results" subpackage. These classes look very similar to the response message classes the proxy server uses to send messages to the client. This is because many response messages are created with the content of the result classes. Nevertheless, they were kept separated, in order to decouple the messages from the backend result, so there is no confusion and changes in SQP don't affect the backend API and vice versa.

The `SuccessHandler` class is just a special `ResultHandler` that doesn't return concrete results, but just "succeeds" if an action is finished.

A.6. Ordered Execution in the Client API

In a blocking API the order of execution of multiple operations is obvious because one operation is always completed before the next one is executed. With asynchronous operations, as in the client API, this is different, which is especially interesting if the result of multiple queries depend on each other. Listing A.22 shows an exampled combined of the discussed snippets of section 4.4. At a closer look, it can be seen that the three operations (connect, modifying revenues, aggregating them) are all semantically depending on each other, although not being connected in the code by using the monadic functions of `CompletableFuture`.

```
try(SqpConnection connection = SqpConnection.create()) {
  connection.connect("localhost", 8080, "/", "proxytest");

  connection.executeUpdate("DELETE FROM sales WHERE revenue < 1.0")
    .thenAccept(updateResult ->
      log(Level.INFO, "Sanity check removed " +
          updateResult.getAffectedRows() + " revenues < 1.00.")
    );

  PreparedStatement stmt = connection.prepare(
    "SELECT count(s.revenue) as `revenue`, e.name as `name`" +
    "FROM employees e, sales s" +
    "WHERE employees.id = s.eid AND location = ? AND year > ? " +
    "GROUP BY e.id");
  CompletableFuture<Cursor> future =
    stmt.bind(0, "Germany").bind(1, 2000).executeSelect();

  // the query is running, however, we can do some other
  timeExpensiveOperation();

  // now we actually need the results of the cursor
  try (Cursor cursor = future.join()) {
    System.out.println("German employee revenues since year 2000");
    while(cursor.nextRow()) {
      System.out.println(
        cursor.at("name").asString() + " made a total of " +
        cursor.at("revenue").asBigDecimal()
      );
    }
  } catch (CompletionException e) {
    // deal with the error while execution
  } catch (SqpException e) {
    // problems with getting data from the cursor (type conversion, etc)
  }
} catch (SqpException e) {
  // problem with parameter binding
} catch (IOException e) {
  // deal with the connection error
}
```

Listing A.22: The combined example of how the non-blocking client API can be used with a simple query, a prepared query and a cursor.

Fortunately there are two mechanisms that make sure that they are still executed in order: The first one is the single "send-queue" which is used internally by the `SqpConnection` to send the messages to the server. So instead of using multiple threads and sending the messages as soon as the action is invoked by the user, the queue assures that a new message is only sent to the server after all previous messages have already been sent. The second mechanism is the stateful implementation of the server: Whenever an operation is currently executing, the server queues all incoming messages instead of executing them concurrently.

An advantage of this implementation is that the time the server needs to proceed a query can already be utilized by the client to send new data. The downside is, however, that if the first query fails, the second query would still be executed, as it's already in the server's message queue. To avoid this, the client would need to explicitly use the features of `CompletableFutures` and build a construct that executes the second query after the first one finished successfully.

So depending on the semantics and importance of the queries, the client can either create a semantic dependency via the `CompletableFuture` or use the time to send more data and risk that the second query is executed even if the first one fails. Either way, it's assured that the second query is executed after the first.

A.7. Integration Tests for Functional Evaluation

To get an idea of how the test framework works, it's shortly introduced and a few simple tests are shown as examples. All of the following tests assume there is an active database connection, all database tables are empty, and error handling isn't necessary. They are used to test both the PostgreSQL and the Transbase backend.

In the project, the TestNG[1] framework is used to set up the environment to meet these assumptions and run the actual test code. The test code uses Hamcrest matchers[2] as a very human-readable way to verify behavior and output. So to assure for example that the connection's `isConnected` method returns **true**, the following expression could be used:

```
assertThat(connection.isConnected(), is(true));
```

[1] `http://testng.org/doc/index.html`
[2] `http://hamcrest.org/`

It is important to mention that even simple integration tests don't explicitly validate all underlying communication. Instead they are checked implicitly. A good example for this is the `Cursor` object. In the tests, the connection is initialized with the option `cursorMaxFetch` set to `1`. So whenever the cursor implicitly fetches new data from the server, it fetches at most one record. While this is highly ineffective in productive environments, it's good for testing as the cursor will always send explicit **FetchData** messages whenever it iterates over data.

Another example is the Cursor's `at` method which accesses the column via its name. To do this, the **CursorDescription** sent by the server must be correct, or otherwise the `Cursor` wouldn't be able to find the columns by name.

The presented tests use a simple table called "weather" with the attributes (columns) listed in Table A.1. This table isn't really interesting, but is sufficient to show that basic database operations work. In fact, using a complex table with many different data types is not a good idea since the complexity will distract from the actual purpose of the tests, which are the execution of the operations itself.

Attribute Name	Attribute Type
city	variable sting of characters, maximal length 80
temp_lo	4 byte integer number
temp_hi	4 byte integer number
prob	4 byte IEEE754 floating point number
date	A temporal date without time

Table A.1.: Attribute names and type of the *weather* table which is used in some tests.

A.7.1. Cursors

To test a simple cursor, two arbitrary tuples are inserted in the database:

```
('TestCity', -12, 30, 0.2, '1015-07-12')
('FooCity', 0, 21, 1.3, '1900-01-03')
```

Then the actual test shown in Listing A.23 is executed. Lines 1 to 3 get the cursor by executing a `SELECT` statement. In line 8 to 11, the *city* field is exemplary checked in detail

for having the correct data type. Lines 14 to 17 check the *date* column in detail and lines 20 to 22 check that the cursor works.

As already stated, this test might be simple, but requires SQP's **SimpleQuery**, **FetchData**, and **CursorDescription** messages to work as intended. Also SQP's type system is and their backend implementation is tested by this.

```
1  Cursor cursor = connection.executeSelect(
2      "SELECT * FROM weather WHERE city LIKE '%City' ORDER BY city DESC"
3  ).join();
4
5  assertThat(cursor.nextRow(), is(true));
6
7  // check city field in detail
8  SqpValue firstField = cursor.at(0);
9  assertThat(firstField, is(instanceOf(SqpVarChar.class)));
10 SqpVarChar strField = (SqpVarChar) firstField;
11 assertThat(strField.asString(), is("TestCity"));
12
13 // check named date field for correct conversion
14 LocalDate date = cursor.at("date").asLocalDate();
15 assertThat(date.getYear(), is(1015));
16 assertThat(date.getMonth(), is(Month.JULY));
17 assertThat(date.getDayOfMonth(), is(12));
18
19 // make sure the next row has different data
20 assertThat(cursor.nextRow(), is(true));
21 assertThat(cursor.at("city").asString(), is("FooCity"));
22 assertThat(cursor.nextRow(), is(false)); // should be at end
```

Listing A.23: Test to check that receiving data from the server via the protocol really works.

A.7.2. Prepared Queries

Listing A.24 shows a test that basically checks if prepared statements work. In lines 8 and 9, the columns are bound in an arbitrary order. The inserted data is then fetched from the database to assure that the data was actually inserted correctly. This tests requires SQP's **PrepareQuery**, **ExecuteQuery**, **FetchData**, and **SimpleQuery** messages and their respective server responses to work as intended.

```
1  PreparedStatement stmt = connection.prepare(
2      "INSERT INTO weather (city, temp_lo, temp_hi, prob, date)" +
3      "VALUES (?,?,?,?,?)"
4  );
5  LocalDate today = LocalDate.now();
6  // bind in some strange order to test that this works
7  double floatVal = 0.12455454;
8  stmt.bind(0, "ElRey").bind(4, today).bind(3, floatVal)
9      .bind(1, -10).bind(2, 0);
10 CompletableFuture<UpdateResult> insertFuture = stmt.executeUpdate();
11 Cursor cursor = connection.executeSelect("SELECT * FROM weather").join();
12 assertThat(insertFuture.get().getAffectedRows(), is(1));
13 assertThat(cursor.nextRow(), is(true));
14 assertThat(cursor.at("city").asString(), is("ElRey"));
15 assertThat(cursor.at("temp_lo").asInt(), is(-10));
16 assertThat(cursor.at("temp_hi").asInt(), is(0));
17 assertThat(cursor.at("prob").asDouble(), is(closeTo(floatVal, 0.0000001)));
18 assertThat(cursor.at("date").asLocalDate(), is(today));
```

Listing A.24: Test to check that prepared statements with explicit data binding work.

A.7.3. Transactions

Listing A.25 is one of the tests that use explicit transaction management. Therefore *autoCommit* is deactivated in line 1, before two queries are executed, of which one is rolled back (line 9). The function called in line 10 makes sure that only one tuple is in the database. This test requires SQP's **SimpleQuery**, **SetFeature**, **Commit**, and **Rollback** messages and their respective server responses to work as intended.

```
1  connection.setAutoCommit(false).join();
2  connection.execute(
3      "INSERT INTO weather (city, temp_lo, temp_hi, prob, date) " +
4      "VALUES ('test1', 0, 50, 0.1, '2015-01-01')");
5  connection.commit();
6  connection.execute(
7      "INSERT INTO weather (city, temp_lo, temp_hi, prob, date) " +
8      "VALUES ('test2', -10, 3, 9.99999, '1111-11-11')");
9  connection.rollback();
10 checkHasData("test1"); // data in table from the first transaction
```

Listing A.25: Test to check that explicit transactions work with rollback and commit.

A.7.4. Information Request

Listing A.26 simply checks if the server is able to return database specific information by using an **InformationRequest**. In line 4 the answer for either database is valid, since the test is executed with both backends.

```
1 String answer = connection.getInformation(String.class,
2 InformationSubject.DBMSName).join();
3 assertThat(answer,
4     is(either(equalTo("Transbase")).or(equalTo("PostgreSQL"))));
```

Listing A.26: Test to check that information requests with backend specific information.

A.8. Full Source Code of Experiments

The following sections simply contain the complete code of the experiments discussed in chapter 5.

A.8.1. Database Specific Example

```
1  package org.sqp.examples;
2
3  import org.sqp.client.Cursor;
4  import org.sqp.client.PreparedStatement;
5  import org.sqp.client.SqpConnection;
6  import org.sqp.core.InformationSubject;
7  import org.sqp.core.exceptions.SqpException;
8
9  import java.io.IOException;
10 import java.time.OffsetDateTime;
11 import java.time.format.DateTimeFormatter;
12
13 public class DatabaseSpecificExample {
14     public static void main(String[] args) {
15         try (SqpConnection connection = SqpConnection.create()) {
```

```
            connection.connect("localhost", 8080, "/", "exampleDB")
                      .thenRunAsync(() -> {
                try {
                    testWaypoints(connection);
                } catch (Exception e) {
                    report(e);
                }
            }).join();
        } catch (Exception e) {
            report(e);
        }
    }
    private static void testWaypoints(SqpConnection connection)
        throws IOException, SqpException {
        connection.getInformation(String.class, InformationSubject.DBMSName)
                  .thenAccept(name -> System.out.println("DBMS: " + name));
        connection.execute("DROP TABLE IF EXISTS waypoints");
        connection.execute("CREATE TABLE waypoints " +
                "(" +
                "  \"id\" serial," +
                "  \"timestamp\" timestamp with time zone NOT NULL," +
                "  \"position\" point NOT NULL" +
                ")").thenRun(() -> System.out.println("Created table."));

        OffsetDateTime now = OffsetDateTime.now();
        OffsetDateTime[] timestamps = {
                now.minusHours(1).withNano(123456789), now.withNano(987654321)};
        double[][] positions = {{1.5, -3.2}, {1.2, -2.8}};
        System.out.println("Data to insert: ");
        for (int i = 0; i < timestamps.length; i++) {
            printWaypoint(i + 1, positions[i], timestamps[i]);
        }

        String insertStmt = "INSERT INTO waypoints " +
                "(timestamp, position) VALUES (?, ?)";
        try (PreparedStatement stmt = connection.prepare(insertStmt)) {
```

```
52              stmt.bind(0, timestamps[0]).bind(1, "pg_point", positions[0])
53                  .addBatch()
54                  .bind(0, timestamps[1]).bind(1, "pg_point", positions[1]);
55              stmt.executeUpdate().thenAccept(
56                  res -> System.out.println(
57                          "Inserted " + res.getAffectedRows() + " rows."));
58          }
59
60          String selectStmt = "SELECT * FROM waypoints ORDER BY id";
61          connection.allowReceiveNativeTypes("pg_point");
62          try (Cursor cursor = connection.executeSelect(selectStmt).join()) {
63              System.out.println("Actual data: ");
64              while (cursor.nextRow()) {
65                  printWaypoint(
66                          cursor.at("id").asInt(),
67                          cursor.at("position").as(double[].class),
68                          cursor.at("timestamp").asOffsetDateTime()
69                  );
70              }
71          }
72      }
73      private static void printWaypoint(int id, double[] pos,
74                                        OffsetDateTime timestamp) {
75          String isoDT = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(timestamp);
76          System.out.println("Waypoint " + id + ": (x=" + pos[0] + "," +
77                  "y=" + pos[1] + ") at " + isoDT);
78      }
79      private static void report(Exception e) {
80          System.out.println("Error occurred: " + e.getMessage());
81      }
82
83  }
```

Listing A.27: The full source code of the database specific experiment discussed in subsection 5.2.1.

A.8.2. Database Independent Example

```
1  package org.sqp.examples;
2
3  import org.sqp.client.Cursor;
4  import org.sqp.client.PreparedStatement;
5  import org.sqp.client.SqpConnection;
6  import org.sqp.core.InformationSubject;
7  import org.sqp.core.exceptions.SqpException;
8
9  import java.io.IOException;
10 import java.time.Month;
11 import java.time.format.TextStyle;
12 import java.util.Locale;
13 /*
14 Tables that need to exist for use:
15 -- for transbase
16 CREATE TABLE friends
17 (
18     name character varying(50),
19     birthday DATETIME[MO:DD],
20     height float
21 )
22 -- for postgres
23 CREATE TABLE friends
24 (
25     name character varying(50),
26     birthday point,
27     height real
28 )
29  */
30 public class DatabaseIndependentExample {
31     public final static String MONTH_DAY_SCHEMA = "{ \n" +
32             "  \"type\": \"array\", \n" +
33             "  \"minItems\": 2, \n" +
34             "  \"additionalItems\": false, \n" +
```

```
            "  \"items\": [\n" +
            "    {\"type\": \"integer\", \"minimum\": 1, \"maximum\": 12}, \n" +
            "    {\"type\": \"integer\", \"minimum\": 1, \"maximum\": 31}\n" +
            "  ]\n" +
            "}";
    public static void main(String[] args) {
        try (SqpConnection connection = SqpConnection.create()) {
            connection.connect("localhost", 8080, "/", "exampleDB");
            testBirthdays(connection);
        } catch (Exception e) {
            report(e);
        }
    }
    private static void testBirthdays(SqpConnection connection)
        throws IOException, SqpException {
        connection.execute("DELETE FROM friends");
        connection.getInformation(String.class, InformationSubject.DBMSName)
                .thenAccept(name -> System.out.println("DBMS: " + name));
        connection.registerTypeMapping("monthDay", MONTH_DAY_SCHEMA,
                                       "point", "[mo:dd]")
                .thenAccept(orig -> System.out.println("Mapping to " + orig));
        String[] names = { "John Doe", "Mary Jane" };
        int[][] birthdays = { {2, 29}, {12, 31} };
        float[] heights = { 1.84f, 1.59f };

        System.out.println("Data to insert: ");
        for (int i = 0; i < names.length; i++) {
            printPerson(names[i], birthdays[i], heights[i]);
        }

        String insertStmt = "INSERT INTO friends " +
                "(name, birthday, height) VALUES (?, ?, ?)";
        try (PreparedStatement stmt = connection.prepare(insertStmt)) {
            for (int i = 0; i < names.length; i++) {
                stmt.bind(0, names[i]).bind(2, heights[i])
                    .bind(1, "monthDay", birthdays[i]);
```

```
                if (i != names.length -1) stmt.addBatch();
            }
            stmt.executeUpdate().thenAccept(res -> System.out.println(
                "Inserted " + res.getAffectedRows() + " friends."));
        }

        String selectStmt = "SELECT * FROM friends";
        try (Cursor cursor = connection.executeSelect(selectStmt).join()) {
            System.out.println("Actual data: ");
            while (cursor.nextRow()) {
                printPerson(
                    cursor.at("name").asString(),
                    cursor.at("birthday").as(int[].class),
                    cursor.at("height").asFloat()
                );
            }
        }
    }
    private static void printPerson(String name, int[] birthday,
                                    float height) {
        String month = Month.of(birthday[0])
                .getDisplayName(TextStyle.SHORT, Locale.getDefault());
        System.out.println(name + " is " + height +
                "m tall and has birthday on " +
                month + ", " + birthday[1]);
    }
    private static void report(Exception e) {
        System.out.println("Error occurred: " + e.getMessage());
    }

}
```

Listing A.28: The full source code of the database independent experiment discussed in subsection 5.2.2.

A.8.3. JavaScript Example for Independent Example

```
months = ["", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
MONTH_DAY_SCHEMA = {
    type: "array",
    minItems: 2,
    additionalItems: false,
    items: [
        {type: "integer", "minimum": 1, "maximum": 12},
        {type: "integer", "minimum": 1, "maximum": 31}
    ]
};
function main() {
    send('S', {query: "DELETE FROM friends"});
    send('I', {subject: 'DBMSName'}, {
        end : 'i',
        i: function(msg) { console.log("DBMS: " + msg.value)}
    });
    send('M', {name: "monthDay", schema: MONTH_DAY_SCHEMA,
               keywords: ["point", "[mo:dd]"]},{
        end: 'm',
        m: function (msg) { console.log("Mapping to " + msg.native) }
    });

    var names = [ "John Doe", "Mary Jane" ];
    var birthdays = [[2, 29], [12, 31]];
    var heights = [1.84, 1.59];
    console.log("Data to insert: ");
    for (var i = 0; i < names.length; i++) {
        printPerson(names[i], birthdays[i], heights[i]);
    }


    var insertStmt = "INSERT INTO friends " +
                     "(name, birthday, height) VALUES (?, ?, ?)";
```

```
35      send('P', {query : insertStmt});
36      send('X', {
37              parameterTypes: ['VarChar', 'Custom', 'Real'],
38              customTypes: ["monthDay"],
39              parameters: [
40                   [names[0], birthdays[0], heights[0]],
41                   [names[1], birthdays[1], heights[1]],
42              ]},{
43          end: 'x',
44          x: function(msg) {
45              console.log("Inserted " + msg.affectedRows + " friends.")
46      }});
47
48      var handler = { end: 'e' };
49      handler['c'] = function(msg) { console.log("Actual data: ")};
50      handler['#'] = function(msg) {
51          printPerson(msg.data[0], msg.data[1], msg.data[2])
52      };
53      var selectStmt = "SELECT name, birthday, height FROM friends";
54      send('S', { query : selectStmt }, handler);
55  }
56  // Other utility
57  function printPerson(name, birthday, height) {
58    console.log(name + " is " + height + "m tall and has birthday on " +
59                months[birthday[0]] + ", " + birthday[1]);
60  }
61  // Functions to handle and send SQP messages
62  databaseName = "exampleDB";
63  mainmethod = main;
64  handlers = [];
65  function messageHandler(evt) {
66      var msg = evt.data;
67      var id = msg[0];
68      var payload = msg.length < 2 ? null : JSON.parse(msg.substring(1));
69      if (id == '!') {
70          console.log("Error " + payload.errorType + ": " + payload.message);
```

```
71          return;
72      }
73      if (handlers.length < 1) {
74         return;
75      }
76      if (id in handlers[0]) {
77          handlers[0][id](payload);
78      }
79      if (handlers[0].end == id) {
80         handlers.shift();
81      }
82  }
83  function send(id, content, handler) {
84     if (handler) handlers.push(handler);
85     websocket.send(id + JSON.stringify(content))
86  }
87  function start() {
88      send('H', {database : databaseName}, {
89          end: 'r',
90          r: mainmethod
91      });
92  }
93  // This code actually invokes the processing
94  websocket = new WebSocket("ws://localhost:8080/");
95  websocket.onopen = start;
96  websocket.onmessage = messageHandler;
```

Listing A.29: The full source code of the database independent experiment implementation in JavaScript as discussed in section 5.3.