HyriseSQL: A SQL Interface for Hyrise
A Technical Documentation

Pedro Flemming
Hasso-Plattner-Institut
Potsdam, Germany
pedro.flemming@student.hpi.de

David Schwalb
Hasso-Plattner-Institut
Potsdam, Germany
david.schwalb@hpi.de

***Abstract*—Hyrise is an in-memory database being developed at the Hasso-Plattner-Institute. In this paper we show how we implemented a SQL interface for Hyrise. We built a parser that processes given SQL queries and generates a C++ object representation. Within Hyrise we implemented the transformation of SQL statements to Hyrise internal operators. We also implemented an architecture to prepare queries on the database server. We evaluate the performance of the parsing process and the performance impact of preparing statements. We show that this new interface has significant advantages compared to the existing interfaces of Hyrise.**

March 20, 2015

I. Introduction

Hyrise is an in-memory database being developed at the chair "Enterprise Platform and Integration Concepts" of Professor Hasso Plattner at the Hasso-Plattner-Institute in Potsdam, Germany [1]. So far, Hyrise has only offered a JSON-interface to allow users to access the database. The Structured Query Language (SQL) is a commonly used language for managing data within a relational database management system.

In this paper we will describe how we have extended Hyrise to support SQL as a query language. In Section II we will look at common open source databases and how they implement their SQL interface and what technologies we could use to implement ours. In Section III we will describe how we parse a given SQL query string and can use the parsing results. Section IV describes how we integrated the SQL parser into Hyrise and transform the query into internal operators. Finally in Section V we will evaluate the parsing performance and compare it to the existing JSON interface. The SQL parser, that we have developed in the scope of this paper, can be found at [2].

II. Related Work

SQL is a common query language for relational database management systems (RDBMS). There are several open source database that implement SQL interfaces. In this section we will look at three of the most popular open source RDBMS: MySQL, PostgreSQL and SQLite. Additionally we will take a brief look at already available SQL parsers. To understand how SQL is usually parsed, we will first describe the common way a parser is built. Parsing and evaluating a text based on a language usually consists of two steps: Lexing and Parsing.

A. Lexing

The first step is called lexing. The lexer takes an input string and generates a sequence of tokens that represent the string. To create such a lexer there are the commonly used tools lex and its successor flex [3]. These tools take a token definition and create a lexer that will be used to tokenize the input. A token definition, in this case, is a list of regular expressions mapped to tokens that will be matched sequentially across the input and therefore create a list of tokens. Some systems use a handwritten lexer instead of creating it with these existing tools. This is a matter of customized functionality for the specific use case.

B. Parsing

The second step is to use these tokens and validate them against a grammar. This step is called parsing. A grammar defines what sequences of tokens form valid sentences in our language. In the same way as with lexing there are tools that take a grammar definition and create a parser based on that grammar. The most popular tools are yacc and its successor GNU bison [3]. They take a BNF-grammar (Backus-Naur-Form) as input and generate a parser that evaluates strings or sequences of tokens based on that grammar. An alternative to bison/yacc is lemon, which is a slightly different parser generator used in SQLite [4]. Whereas a grammar definition of yacc is compatible with bison, a grammar for lemon is not compatible with these other generators. In contrast to bison, lemon will always create a reentrant and thread-safe parser and introduces non-terminal destructors, which aim to avoid memory leaks. Since lemon is considerably newer than bison, there are more applications that rely on the well-tested bison (see II-C). In bison we have to take explicit steps to create a parser that is thread-safe.

These common tools for creating lexers and parsers can be linked together to create a full parser. THe parser takes an input string, generates a sequence of tokens and evaluates them against a grammar. In the parser generators we can add actions to grammar rules, which allow us to execute code and thus build objects that represent the parse tree. This parse tree can, in the case for SQL, be used to map the SQL to internal operators.

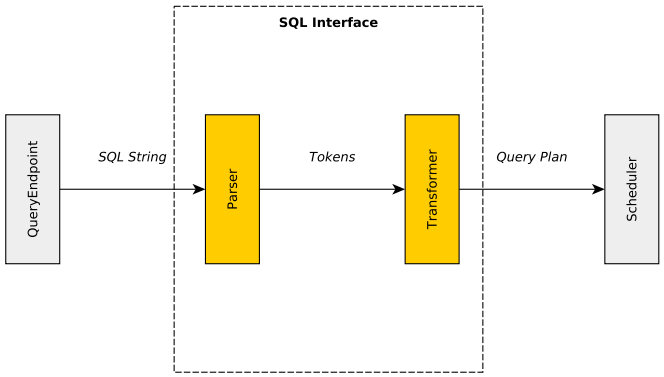

Fig. 1. Parser and Transformation Process

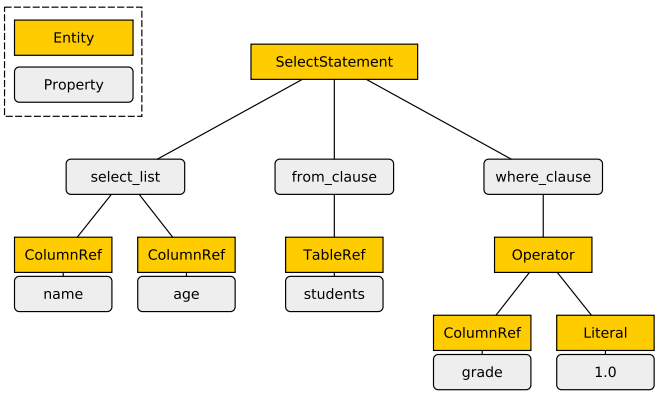

Fig. 2. Simplified parse tree for the query: SELECT name, age FROM students WHERE grade = 1.0

C. Applications

MySQL and PostgreSQL use the bison/yacc parser generator to create their SQL parsers[1] [2]. On the other hand SQLite uses a different parser generator called lemon [4][5].

Parser generators are also popular outside of the SQL context. For example, the PHP parser (ZendParser) uses bison to generate its parser. Before version 3.4 the GCC parser for C++ was based on yacc. In 3.4 they switched to a hand-written parser for more customized functionality [6].

III. Parsing SQL

Because of the popularity of bison and the existing usages in MySQL and PostgreSQL (Section II-C) we decided to implement the parser using flex and bison. The process of executing a given SQL statement is shown in Figure 1

A. SQL Grammar

For the parser generator to be able to create a valid SQL parser we first need to define a valid SQL context-free grammar in Backus-Naur-Form [3]. There are full grammars available for the ISO SQL standard[3]. As discussed in Section II there are open grammars available from popular RDMS (e.g. MySQL, PostgreSQL). It is important for a grammar to be unambiguous. That means that for every possible input there is only a single parse tree. Otherwise the parsing can yield unexpected results.

For Hyrise we also want to make some Hyrise-specific extensions to the standard SQL grammar. For example: Hyrise supports creating tables from 'tbl'-files with its 'TableLoad' operator. We extended the standard SQL grammar with the construct:

```
CREATE TABLE <table_name>
FROM TBL FILE <file_path>;
```

Listing 1. SQL

Since Hyrise is a research database and is also intended for profiling of specific operators, we also intend to support setting explicit transformation-parameters. These parameters will have an impact on the transformation process as they can force the use of specific operators. They can also be used to set operator configuration parameters explicitly, which otherwise would have been automatically estimated by the *SQLStatementTransformer*.

Also creating prepared statements is a SQL feature that has not been uniformly defined in the existing databases. In PostgreSQL 9.4 it is only possible to prepare single statements [7]. Whereas in MySQL 5.1 they chose a string based definition of a preparable statement [8]. We decided to support preparing multiple statements at once. We discuss our implementation in detail in Section IV-E.

B. Storing the Parse Tree

After creating a valid SQL grammar we need to create a object structure that holds the semantic information of the parsed query. With parser generators such as bison, yacc or lemon this can be achieved by defining actions to grammar rules. Actions in this sense are snippets of code, which are executed when the associated terminal or non-terminal is reduced. Through these actions it is possible to create objects that represent the information of the associated rules. A simplified example for our parse tree representation can be seen in Figure 2.

Our representation of the parse tree consists of a class *SQLStatement* and inheriting classes for each statement type (e.g. *SelectStatement*). These statement classes contain all information that is related to a single instance of a SQL statement. Data sources are represented by the class *Table*. A data source for a SELECT statement can be one of a few different constructs. In our implementation it can be either a single reference to a database table, a list of tables (which are combined by cross product), a join of two other data sources or another select statement. These possibilities are grouped in the *Table* class.

The last class we are using to store the parse tree is the *Expr* class. An instance of this class represent an expression

[1]The MySQL grammar can be found at sql/sql_yacc.y.
[2]The PostgreSQL grammar can be found at src/backend/parser/gram.y.
[3]SQL Grammars at http://savage.net.au/SQL/

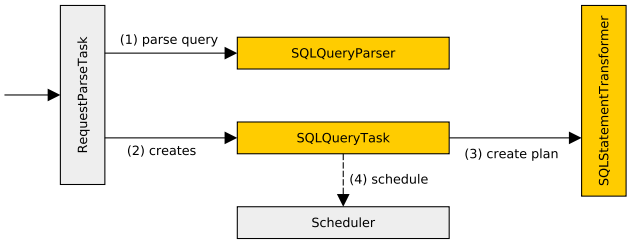

Fig. 3. Overview of the Hyrise dataflow during SQL execution

within a SQL statement. We are grouping Expressions into four different types:

- Literals (String, Integer, Float)
- Identifiers (Column Name, Star: '*', Placeholder: '?')
- Function Calls (SUM, AVG, COUNT, ...)
- Operators

Additionally operators can be grouped into binary and unary operators. In our implementation we are not supporting tertiary operators as of now. The operands of operator expressions and the parameters of function calls are also expressions and can have any expression types. The operators and function calls hold references to their sub-expressions.

IV. Integration into Hyrise

In this section we will present the integration of the SQL parser into Hyrise[4]. We will discuss the architecture of the SQL module and describe the function of the used classes. A overview of that architecture can be seen in Figure 3.

A. Architecture

To integrate a SQL endpoint in Hyrise we extended the already implemented JSON query interface. After receiving a SQL query the endpoint first triggers the parsing of the query. This step will generate an object-representation of the query as described in Section III. We encapsulated this task within the class *SQLQueryParser*.

A SQL query can contain multiple statements. For each statement we create a *SQLQueryTask*. This list of tasks will be executed sequentially. Each SQLQueryTask has the responsibility to transform its assigned statement and have the query plan executed.

Within each SQLQueryTask we have to create a query plan based on the request. This transformation step is handled by the class *SQLStatementTransformer*. Within Hyrise a query plan is represented by a graph of *PlanOperations*. Whatever output is intended to be returned to the user will be forwarded to a instance of a *ResponseTask* that has been created for the current request.

[4]All SQL specific code can be found in the Hyrise repository in the path lib/access/sql/

B. SQLQueryTask

We divided the transformation and execution process of a query into separate tasks, one for each SQL statement, so that a statement is fully executed before the next one starts with the transformation. In an earlier iteration we were transforming all statements at once before executing the entire query plan. This was not sufficient because the transformation of a statement might depend on the execution result of a previous statement.

Example: Consider a query consisting of two statements, where the first statement creates the target table and the second statement inserts data into the table. The transformation of the INSERT statement depends on meta information about the table, such as number of columns or data types. But since the table does not exist before the CREATE statement is executed there is no meta information available.

In this example it would technically be possible to keep track of the meta information by using the information about the previous statement when transforming the second statement. But when we start to consider the many other operations that potentially change the query plan (e.g. query optimization based on data properties) it becomes more favorable to do the transformation only after the previous statement has been executed. Using the sequential transformation process and encapsulating the code associated with the execution of a SQL statement into a SQLQueryTask reduces the overall complexity of the SQL execution architecture.

After the statement has been transformed into a query plan by the *SQLStatementTransformer* the query task ensures that its subsequent task will only be executed after the generated plan has been fully executed. The query plan is then passed to the scheduler and executed.

C. SQLStatementTransformer

In this section we will we discuss how we implemented the transformation of a SQL statement into Hyrise operators. The main entry-point is the class *SQLStatementTransformer*. It will take a *SQLStatement* and return a query plan. The concrete transformation task is delegated to one of four statement type specific transformers. Each specific transformer handles one of the following statement types:

- Data definition (e.g. CREATE, DROP, ALTER)
- Data manipulation (e.g. INSERT, UPDATE, DELETE)
- Data access (e.g. SELECT)
- Prepared Statements (e.g. PREPARE, EXECUTE)

Each of these transformers contains the functionality to transform its statement-types into a query plan. Transforming data definition and data manipulation statements has relatively small complexity. These statements usually require no more than a maximum of three operators to be created. In the case of an INSERT statement we first need to select the table that is the target of the insert. After that we have to create an *InsertScan* operator and finally commit the transaction. The implementation of prepared statements is discussed in detail in Section IV-E.

The most complex transformation process is required for SELECT statements. A SELECT statement consists of several sub clauses. These clauses in their order of operations are: [9]

1) FROM clause
2) WHERE clause
3) GROUP BY clause
4) HAVING clause
5) SELECT clause
6) ORDER BY clause

The FROM clause can even have its own sub clauses as it can contain a JOIN clause or even another SELECT statement. For each of these clauses there are specific operators which have to be used. This results in significantly larger query plans than with other statement types. The following statement, for example, will yield a query plan with 9 operators:

```
SELECT employee_name, employee_id
FROM companies JOIN employees
ON company_id = employee_company_id
WHERE company_id = 4 ORDER BY employee_id;
```

Listing 2. SQL

For each step within the transformation we have to keep track of the resulting table schema. By doing this we can ensure that the query is semantically valid and the execution can complete without errors. By catching possible errors during the transformation we can return an error message to the user with context to the SQL statement rather than returning errors that refer to the context of the operator. This is important feedback for the user for subsequently fixing the query.

D. Transaction Management

Hyrise implements a multi-version concurrency control with lock-free commits [10]. We decided to treat each statement as a separate transaction. After a statement has executed successfully and has committed its changes, the subsequent SQLQueryTask will initialize a new transaction. This means that, if a later statement fails (during transformation or execution), all changes from previously executed statements are still persisted within the database and not rolled back. Also if a statement fails, no subsequent queries are executed and an error is returned to the user.

Another option to handle the transaction management is to treat a full SQL query as a single transaction. This would mean that if any statement within a query fails, all previously successfully executed statements would have to be rolled back. We intend to offer the user a way to decide which of these options to use. As of now only the first option is implemented in Hyrise.

E. Prepared Statement Implementation

An important feature for our SQL implementation is the ability to prepare statements. Prepared statements allow the reuse of SQL queries by parameterizing them and storing them on the database server. This is an usability advantage because complex SQL queries can be defined and called more conveniently by an assigned alias. An example of a preparation and execution of a query is given in Listing 3. In our implementation we wanted to support not only the preparation of single statements but also the preparation of queries consisting of multiple statements. An example of a preparation of multiple statements is given in Listing 4.

```
PREPARE students_by_grade:
SELECT * FROM students WHERE grade = ?;

EXECUTE students_by_grade(2.0);
```

Listing 3. Simple prepared statement example

```
PREPARE students_by_grade2 {
        INSERT INTO accesses VALUES('students');
        SELECT * FROM students WHERE grade = ?;
}
```

Listing 4. Preparing multiple statements

For Hyrise we decided to store the result of the parsing process. We still have to perform the transformation of the parse tree on every execution. This decision is based on the lower implementation complexity and the fact that it is unclear whether preparing the query plan instead would yield significant benefits.

To achieve the parametrization for our representation of the parse tree, we need to keep track of the placeholder occurrences (character '?'). At parsing time we maintain a list of all placeholder expression and assign the parameter order based on the character position of the expression. This is to ensure that placeholders that appear sooner in the string will have a lower parameter index. When executing the prepared statement, we can replace the placeholder expressions with the expression that was given as a parameter and continue with the transformation. With this approach we have to parse the prepared string only once. We still have to do the transformation. This is a performance benefit on all executions. There is still the overhead of parsing and transforming the EXECUTE statement to be considered. The performance impact will be evaluated in detail in Section V-C.

F. Alternative Implementations for Prepared Statements

We also discussed three other possible processes to store prepared statements. The least complex way is to store the string of the query that is going to be prepared. When executing the prepared statement, we first need to replace all occurrences of '?' with the parameters and then start the usual SQL execution process. This is simple to realize but it offers no benefit to the performance. Rather it decreases performance because it introduces the overhead of parsing and transforming the PREPARE and EXECUTE statements ahead of time.

A low-level approach would involve preparing the actual query plan, which is created in the transformation process. The benefit of this approach would be that we do not have to repeat the transformation step when executing a prepared statement. Instead we would only have to apply the parameters to the prepared query plan. To implement this we would have to be able to pass the parameters to the operators after a first creation. We would need to keep track of the target operator and what exact operator setting needs to be changed. This

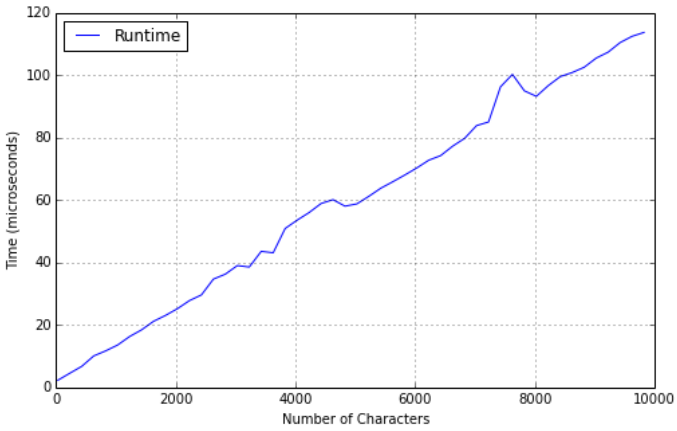

Fig. 4. Number of Character

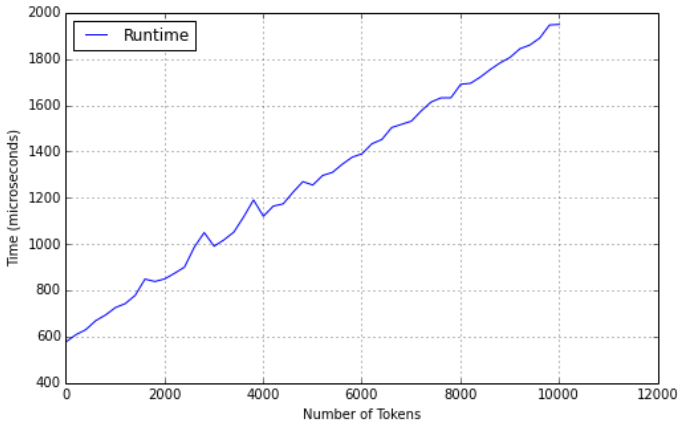

Fig. 5. Number of Tokens

introduces a high-level of complexity into the preparation and execution process as every operator might need customized routines. Also for the current operator design the data types of parameters need to be known ahead. This could be resolved by typing the parameters explicitly when creating the prepared statement. When preparing the query plan we might also limit the query optimizations we can perform. E.g. Query optimization can be done on meta-information such as the number of rows of a subquery. But if we only set a range parameter dynamically and do not know it at transformation time, it is not possible to do those size-based optimizations.

In Microsofts in-memory database Hekaton a more sophisticated approach has been implemented. In Hekaton SQL statements and stored procedures are converted into native code. The Hekaton engine generates C-code from the already optimized query plan. This code is then compiled into a library and loaded into the SQL Server address space. [11][12]

V. Evaluation

In this section we evaluate the performance of the implemented SQL interface. We separately evaluate the performance of the parser and the transformer. We also take a look at the performance impact of preparing statements compared to executing not-prepared statements. The evaluation was performed on a 2x 1.60GHz Intel i5 CPU.

A. Parse Performance

First we evaluate the parse performance. We measured the impact of changing the two major input parameters. These are the number of characters and the number of tokens within a single SQL string.

In Figure 4 you can see the impact of changing the number of characters. While we changed the number of characters, the number of tokens remained constant. We can see that the relation between the number of characters and the expected parsing time is linear. Doubling the number of characters in a SQL string will also double the time required to process it.

In our second experiment we changed the number of tokens within an SQL string while the total number of characters stayed constant at 50000. The results can be seen in Figure 5.

The benchmarks show a linear correlation between the runtime and both the number of characters and the number of tokens. For the number of characters this can be explained by the fact that reading a character requires a constant amount of time. For number of tokens this linear behavior can be explained with the constant time it takes to create a token and the constant time it takes to create an 'Expression' object.

From our measurements we can assume a linear runtime complexity of

$$O(n_c + n_t)$$

with n_c being the number of characters and n_t being the number of tokens in the string. The exact expected runtime can be estimated with

$$time(n_c, n_t) = c_c \cdot n_c + c_t \cdot n_t + c_0$$

where c_c and c_t are coefficients indicating the time impact of adding a character/token and c_0 being the required time to initialize and tear down the parsing process. In our setup the values of the coefficients are

$$c_c = 0.01137 \ \frac{microseconds}{character}$$
$$c_t = 0.13714 \ \frac{microseconds}{token}.$$
$$c_0 = 7.84 \ microseconds$$

These values show that adding a token increases runtime about 12x as much as adding a character to a string.

B. Comparison with JSON interface

The only previously existing interface for users to the database was a JSON interface. In this JSON interface the user would explicitly define the database operators, their parameters and how they interact with each other to create the result. This was a explicit and cumbersome way to define the exact operations, whereas SQL introduces a descriptive

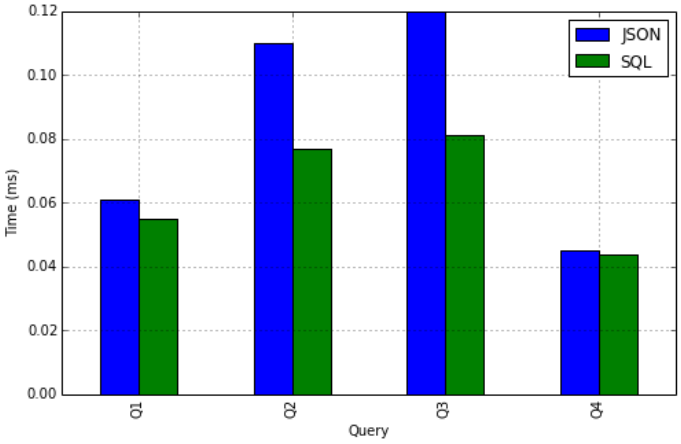

Fig. 6. SQL vs JSON interface

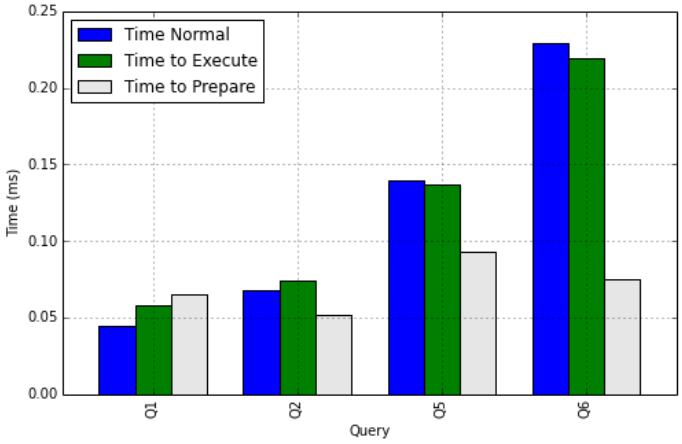

Fig. 7. Prepared Statement Performance

way to define the expected result. When executing SQL, the transformer generates a query plan in the same way that users can define them as JSON.

In contrast to the SQL interface that we presented in this paper, the JSON interface needs no complex transformation as the operators and their configuration are already explicitly defined by the user. On the other hand are such JSON queries significantly longer than equivalent SQL queries. This means that processing JSON strings is also expensive.

In Figure 6 we compare the time that is needed to generate the full query plan for equivalent SQL and JSON queries. We can see that parsing and transforming SQL queries is less expensive than for JSON queries. Parsing long JSON strings is costly and the benefit of using short and concise SQL strings increases with the complexity of queries. The queries that we used are listed in Section V-D.

C. Prepared Statement Performance

In this section we will evaluate the performance impact of preparing a query compared to executing the query unprepared. We executed several different queries and measured their execution times. These queries differ in string length, number of tokens and number of statements.

In Figure 7 we can see that the perfomance impact is highly dependent on the type of query that is executed. The queries increase in total length and number of tokens from left to right. We see that preparing a short query has a peformance disadvantage because the overhead of executing the EXECUTE statement is larger than the benefit of preparing the statement. With longer SQL queries the benefit increases while the overhead remains the same. For longer queries we have a performance benefit for preparing them compared to executing them without preparation.

D. Queries

```
SELECT name, city FROM students WHERE grade <= 2.0;
```

Listing 5. Q1

```
SELECT employee_name, company_name FROM companies
    JOIN employees ON company_id =
    employee_company_id WHERE company_id = 2 ORDER
    BY employee_name;
```

Listing 6. Q2: Select all employees and their companies

```
SELECT name, city, grade FROM (SELECT * FROM
    students WHERE city = 'Potsdam') t1 WHERE grade
    <= 1.5 OR grade >= 3.5;
```

Listing 7. Q3

```
INSERT INTO students VALUES ('Max', 0, 'Musterhausen
    ', 0.0);
```

Listing 8. Q4: Add a new student

```
UPDATE companies SET company_id = 6 WHERE company_id
    = 2;
UPDATE employees SET employee_company_id = 6 WHERE
    employee_company_id = 2;
SELECT employee_name, company_name FROM companies
    JOIN employees ON company_id =
    employee_company_id WHERE company_id = 6 ORDER
    BY employee_name;
```

Listing 9. Q5: Change the ID of a company

```
-- This statement is executed 10 times
INSERT INTO students VALUES ('Max', 0, 'Musterhausen
    ', 0.0);
```

Listing 10. Q6: Batch-insert of 10 newstudents

VI. Conclusion

In this paper we presented our implementation of an SQL interface for the in-memory database Hyrise. This interface offers a descriptive way for users to query Hyrise. It also has a lower pre-processing overhead than the existing JSON-interface, making it the best choice available for usage in performance critical situations.

We also discussed in detail the possibilities for creating prepared statements and presented our solution. We evaluated the performance impact of preparing statements. We showed that it highly depends on the type of query that is being executed whether the preparation will be beneficial or detrimental to the overall performance.

REFERENCES

[1] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden, "Hyrise - a main memory hybrid storage engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[2] P. Flemming, "Sql parser on github.com," https://github.com/hyrise/sql-parser, 2014.

[3] J. Levine, *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[4] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[5] M. Owens, "Embedding an sql database with sqlite," *Linux Journal*, vol. 2003, no. 110, p. 2, 2003.

[6] https://gcc.gnu.org/gcc-3.4/changes.html, 2004.

[7] T. P. G. D. Group, "Postgresql: Documentation: 9.4: Prepare," http://www.postgresql.org/docs/9.4/static/sql-prepare.html, 2015.

[8] O. Corporation, "Mysql :: Mysql 5.1 reference manual :: 13.5 sql syntax for prepared statements," http://dev.mysql.com/doc/refman/5.1/en/sql-syntax-prepared-statements.html, 2015.

[9] B. Nadel, "Sql query order of operations," http://www.bennadel.com/blog/70-sql-query-order-of-operations.htm, 2006.

[10] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner, "Efficient transaction processing for hyrise in mixed workload environments," in *In Memory Data Management and Analysis*, ser. Lecture Notes in Computer Science, A. Jagatheesan, J. Levandoski, T. Neumann, and A. Pavlo, Eds. Springer International Publishing, 2015, pp. 112–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13960-9_9

[11] C. Freedman, E. Ismert, and P.-Å. Larson, "Compilation in the microsoft sql server hekaton engine." *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.

[12] T. Neumann and V. Leis, "Compiling database queries into machine code." *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.