Speaking in Tongues: SQL Access to NoSQL Systems

Julian Rith
FAU Erlangen-Nürnberg, INF6
Martensstr. 3
91058 Erlangen, Germany
Julian.Rith@fau.de

Philipp S. Lehmayr
FAU Erlangen-Nürnberg, INF6
Martensstr. 3
91058 Erlangen, Germany

Klaus Meyer-Wegener
FAU Erlangen-Nürnberg, INF6
Martensstr. 3
91058 Erlangen, Germany
Klaus.Meyer-Wegener@fau.de

ABSTRACT

Non-relational data stores, which are usually called NoSQL systems, have become an important class of data management systems. They often outperform the relational systems. Yet there is no common way to interface with NoSQL systems. The Structured Query Language (SQL) has already proven to be useful to provide a uniform query language for all the relational systems. We identify a subset of SQL for access to NoSQL systems. Our extensible middleware translates SQL queries to the query languages of the connected NoSQL systems. The migration between these systems is thereby greatly simplified as well as the comparison of the supported NoSQL systems.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*relational databases, query processing*

General Terms

Languages, Measurement, Performance

Keywords

NoSQL, Mapping, SQL

1. INTRODUCTION

Relational database management systems (RDBMS) are dominant in the domain of enterprise applications. During the last decade, data management systems have gained a broader area of application, in particular in web applications. The requirements of these applications differ from those of enterprise applications. While the guarantee of consistency is essential for the latter, it is considered less important for the former. Instead, availability, performance, and scalability are of much more interest. *Non-relational data stores* (NoSQL systems) which in some respect resemble record-oriented database systems were developed to meet these requirements. The goals are achieved by simplifications of the data models and by a reduced set of access operations, often abbreviated as "CRUD", which stands for create, read, update, and delete (at record level). There are no set-oriented operations, that is, no joins. Typically, application code must be written to compensate for this.

RDBMS have the Structured Query Language (SQL) as a common query language [9], which is an accepted standard and widely known by developers and users. However, there is no common query language for NoSQL systems. This has re-introduced the Babylonian confusion that had been overcome by the definition of SQL.

The idea presented in this paper is to retain the benefits of SQL in the context of NoSQL systems. For that purpose, we have developed a *middleware* that parses SQL queries and translates them to the query languages of selected NoSQL systems. Of course, due to their limited functionality, only a subset of SQL can be offered.

By providing a common interface to different data stores it is easier to compare them, and to migrate applications between them.

http://dx.doi.org/10.1145/2554850.2555099

2. RELATED WORK

Attempts to provide a uniform way to access NoSQL systems via XQuery [14], a new query language called Unstructured Query Language (UnQL) [2], or a Java interface [3] have already been made.

As an alternative approach, SQL has been adapted to access single NoSQL systems [4, 11, 1]. UnityJDBC, for instance it currently only maps to MongoDB (in addition to relational systems). Our middleware instead allows to map SQL to more than one NoSQL system and thus eases comparison. Furthermore, our system is extensible by including more adapters.

Mapping SQL to different query languages has also been an aspect of federated databases [13]. It was called command transformation back then. Since the query languages of NoSQL systems are very different, we must create completely new instances of that transformation.

3. ARCHITECTURE

A basic taxonomy for the NoSQL systems that can be the target of our query mapping is given in [5]. The three classes that are identified in addition to the RDBMSs are key-value stores, document stores, and extensible-record stores. Their functionality has roughly been described as CRUD, but for the design of the mapping, some more detail needs to be considered.

First, we separate two groups of SQL queries: those that are executable on all (or at least many) NoSQL systems, and those that are only executable on a few systems (maybe just one). They must be distinguished to give the users the option to favor either portability or functionality.

In the first group of queries, we can have only the equivalent of CRUD in SQL. That is, an `INSERT` statement (the "create") is possible, but must not have a subselect clause, and a key must explicitly be set. A `SELECT` statement (the "read") must not include joins, the `WHERE` clause may only use the keys, and the `SELECT` clause may only contain the values. Grouping and aggregation cannot be used. Similarly, the `WHERE` clause of an `UPDATE` statement may only use the key, and the `SET` clause can only modify the value. Finally, the `WHERE` clause of a `DELETE` statement must use the key. That can be considered the core of SQL for the execution on NoSQL systems.

The second group of queries begins with the `CREATE TABLE` statement as the most important statement of the data definition language (DDL). It cannot be used with key-value stores because their structure is too simple. For the other NoSQL systems however, a mapping can be defined, if the statement does not include integrity constraints like the definition of foreign keys and the `CHECK` clause. The `CREATE INDEX` statement can be mapped to a couple of systems (e.g. Cassandra, MongoDB, HBase) and is sometimes even required if selection predicates on certain columns are to be used.

Also in the second group we have search based on non-key attributes, and content-based wild-card searches (in all document stores) using the `LIKE` operator in the `WHERE` clause. Aggregation can only be done on a few selected systems (e.g., MongoDB). Overall, a huge variety can be found here. We have decided not to restrict the SQL syntax too much, but to offer all the clauses and to return an error message if a mapping to the particular NoSQL system in use cannot be done.

Another decision was not to reimplement operations missing in the selected systems in the middleware. Likewise, some limitations of NoSQL systems that go beyond the query language – most prominent: no support for transactions – are not compensated by the middleware. Also, triggers are not covered. The cost of the reimplementation is considered to be too high at the moment. It would counteract the performance benefits of NoSQL systems, which would make them lose their reason to exist.

Many NoSQL systems are supplied with a MapReduce framework [8], which allows to execute more complex queries [12]. Chattopadhyay et al. [6] proved that in fact most of SQL can be mapped to MapReduce functions. However, the performance benefits of NoSQL systems are again lost when this is done. Therefore, we decided not to use MapReduce to implement SQL.

The architecture of the middleware is shown in Figure 1. SQL statements are directly sent to the middleware. The parser creates the intermediate representation as a standard parse tree. The connectors take the parse tree and map it to the specific interface of their NoSQL system, if possible. Otherwise, an error message is returned. The connection to the NoSQL systems is also managed by the connectors and reported to the application.

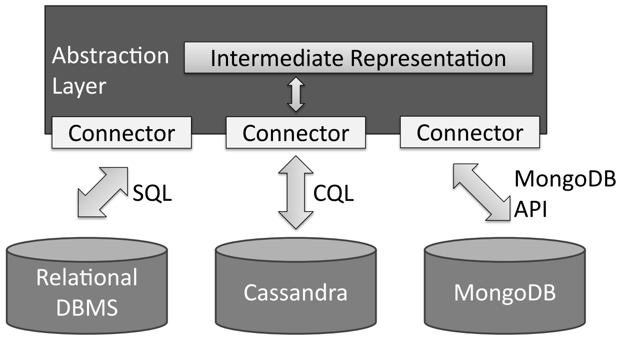

Figure 1: Architecture of the middleware

4. IMPLEMENTATION

We have started with two systems, an extensible-record store (Cassandra) and a document store (MongoDB). Cassandra has been selected because its query language CQL is very similar to SQL. MongoDB has profound query capabilities that allow the translation of more complex queries. Both systems can easily provide support for the first group of queries and offer many options for extensions with respect to the second group. Key-value stores have been omitted due to their simple data model and their lack of support for complex queries. Their mapping would have been much simpler. In addition to these two systems we have also incorporated a common open source RDBMS to compare the performance of our middleware. In this case, the SQL query is obviously neither parsed nor mapped, but simply passed to the underlying system.

The middleware is a class library that is developed in C# on the basis of the .NET Framework 4.0. To provide extensibility, the middleware relies on the Managed Extensibility Framework (MEF). The SQL parser uses the "Another Tool for Language Recognition" (ANTLR) parser generator [10]. The SQL grammar is based on the grammar of the Macroscope portable SQL ADO.NET provider[1].

The system now can execute SQL statements like:

```
CREATE TABLE Persons (
  PersonID INT PRIMARY KEY,
  LastName VARCHAR(255),
  FirstName VARCHAR(255),
  City VARCHAR(255)
);

INSERT INTO Persons
VALUES (1, 'Doe', 'John', 'Houston');

SELECT * FROM Persons
WHERE PersonID = 1;
```

5. EVALUATION

To evaluate the performance of the middleware, we have used the Yahoo! Cloud Serving Benchmark (YCSB) [7] which seems to be the only benchmark available for NoSQL systems. We have reimplemented it in C# to integrate it into our infrastructure. The workload F of YCSB (read-modify-write) appeared to be the best choice to measure the overall

[1]URL: `http://macroscope.sourceforge.net/`

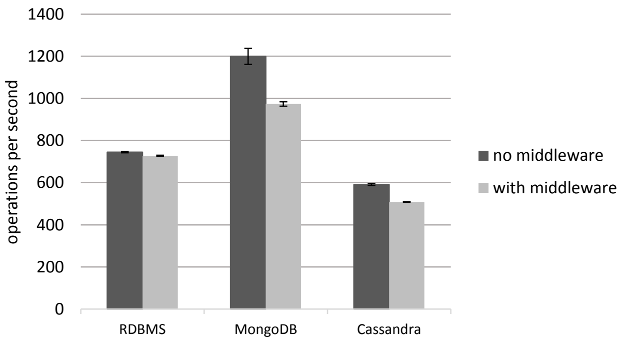

Figure 2: Operations per second without and with the middleware

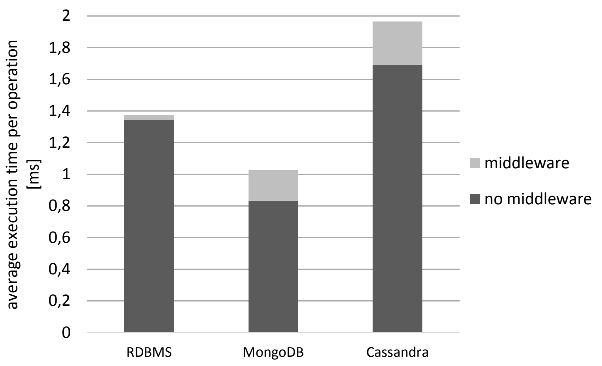

Figure 3: Average execution time per operation with fraction of the middleware

performance. The other workloads emphasize only one kind of operation. Each run has executed 70,000 operations (that is, operation pairs of read and write) and has been repeated 20 times.

The measurements have been performed on an Intel Core 2 Quad Q9450 processor with 8 GB RAM running openSUSE 11.1. The disk system offered a data rate of 75 MB/s.

The series of measurements ran the benchmark on the three systems without the middleware and with it. The resulting throughput values are shown in Fig. 2.

As expected, the middleware has hardly any effect in case of the RDBMS, because it only passes the SQL statement. In case of the two NoSQL systems, the anticipated reduction of throughput is in the order of 13-18%, which seems acceptable with respect to the benefits. The loss is a bit higher in the case of MongoDB, because the number of operations is higher and therefore the mapping is done more often.

In absolute time, the parsing and translation on average takes 0.2 ms for MongoDB, and 0.3 ms in case of Cassandra, see figure 3. The difference is not significant.

6. CONCLUSIONS AND OUTLOOK

With our prototype we have proven that it is feasible to build an extensible middleware that translates a subset of SQL to the query languages of NoSQL systems and still preserve the performance benefits that NoSQL systems offer. This provides a common interface for the large variety of these systems and thus eases application development. It also eases migration between different data stores.

NoSQL systems are under continuous development. Additional query functionality can be made available by enhancing the connectors. Some systems are about to implement even some transaction support. For instance, Cassandra and Redis now offer mechanisms to serialize operations and to run them in an isolated fashion. Once a significant coverage can be found here, the middleware should be extended to offer that functionality, too.

The authors thank the anonymous reviewers for their valuable hints.

7. REFERENCES

[1] UnityJDBC. `http://www.unityjdbc.com/`.

[2] UnQL. `http://unql.sqlite.org/`.

[3] P. Atzeni, F. Bugiotti, and L. Rossi. SOS (save our systems): a uniform programming interface for non-relational systems. In *Proc. 15th Int. Conf. on Extending Database Technology*, EDBT'12, pages 582–585, New York, NY, USA, 2012. ACM.

[4] A. Calil and R. dos Santos Mello. SimpleSQL: a relational layer for SimpleDB. In *Proc. 16th East European Conf. on Advances in Databases and Information Systems*, ADBIS'12, pages 99–110, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2010.

[6] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing an SQL implementation on the MapReduce framework. In *Proc. of VLDB*, pages 1318–1327, 2011.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symp. on Operating Systems Design & Implementation*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] J. Melton. SQL language summary. *ACM Computing Surveys*, 28(1):141–143, 1996.

[10] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.

[11] J. Roijackers. Bridging SQL and NoSQL. Master's thesis, Eindhoven University of Technology, 2012.

[12] N. Ruflin, H. Burkhart, and S. Rizzotti. Social-data storage systems. In *Databases and Social Networks*, DBSocial'11, pages 7–12, New York, NY, USA, 2011. ACM.

[13] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, Sept. 1990.

[14] H. Valer, C. Sauer, and T. Härder. XQuery processing over NoSQL stores. In *Proceedings of the 25th GI-Workshop "Grundlagen von Datenbanken 2013", Ilmenau, Germany, May 28 - 31, 2013 2013*, 2012.