An Effective Scalable SQL Engine for NoSQL Databases

Ricardo Vilaça, Francisco Cruz, José Pereira, Rui Oliveira

An effective scalable SQL engine for NoSQL databases

Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui Oliveira

HASLab - High-Assurance Software Laboratory
INESC TEC and Universidade do Minho
Braga, Portugal
{rmvilaca,fmcruz,jop,rco}@di.uminho.pt

Abstract. NoSQL databases were initially devised to support a few concrete extreme scale applications. Since the specificity and scale of the target systems justified the investment of manually crafting application code their limited query and indexing capabilities were not a major impediment. However, with a considerable number of mature alternatives now available there is an increasing willingness to use NoSQL databases in a wider and more diverse spectrum of applications and, to most of them, hand-crafted query code is not an enticing trade-off.
In this paper we address this shortcoming of current NoSQL databases with an effective approach for executing SQL queries while preserving their scalability and schema flexibility. We show how a full-fledged SQL engine can be integrated atop of HBase leading to an ANSI SQL compliant database. Under a standard TPC-C workload our prototype scales linearly with the number of nodes in the system and outperforms a NoSQL TPC-C implementation optimized for HBase.

Keywords: SQL; NoSQL; Cloud Computing; Middleware

1 Introduction

With cloud-based databases as part of platform-as-a-service offerings, such as Google's BigTable [7], Amazon's DynamoDB [10], and Yahoo!'s PNUTS [8], HBase [12] and Cassandra [19], NoSQL databases become attractive for a larger and more diverse set of applications.

However, their lack of SQL support represents a major hurdle for a wider adoption. This arises at different levels due to the prevalence of SQL as the standard, widely mastered and efficient query language for databases. Most web scale applications are purposely kept SQL-based for their core data management [23]. Any application at least two, three years old is directly or indirectly (e.g. on top of an object-relational mapping) based on an SQL interface and its migration is usually not straightforward. A large number of tools and middleware coupled to SQL have been developed and matured over the years and are currently at the basis of most application development frameworks.

It is therefore not surprising that SQL compliance has been one of the most requested additions to the Google App Engine platform [15].

This state of affairs has sparked a number of proposals for middleware that exposes higher-level query interfaces on top of the barebones key-value primitives of NoSQL databases. Many of these aim at approximating the traditional SQL abstraction, ranging from shallow SQL-like syntax for simple key-value queries (e.g. CQL for Cassandra, Phoenix [24], PIQL [4]) to the translation of analytical queries into map-reduce jobs [1–3, 20]. However, due to the complexity of SQL existing solutions are limited to a subset of the language, thus not allowing to leverage exiting SQL applications and tools.

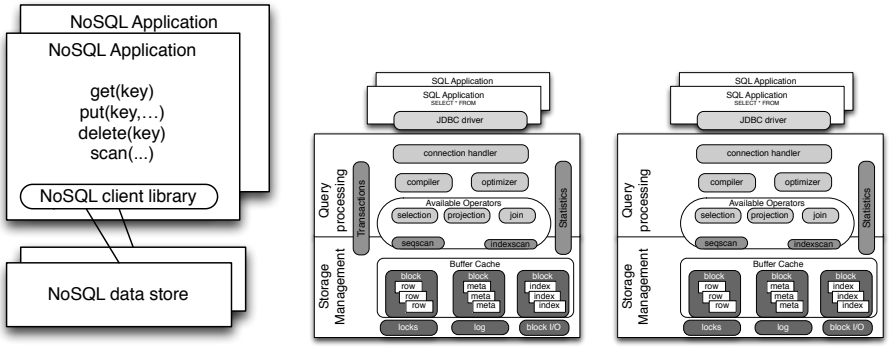
In this paper, we present a *distributed query engine* (DQE) for running SQL queries on top of a NoSQL database, while preserving its scalability and schema flexibility. The DQE allows to combine the expressiveness and performance of a Relational Database Management System (RDBMS) with the scalability and schema flexibility of a NoSQL database. DQE is a transaction-less database [17], preserving the isolation semantics provided by the underlying NoSQL database.

Enabling scalable SQL processing atop of a NoSQL database poses several architectural challenges to the query engine [27, 26]. On one hand, traditional RDBMS architectures include several legacy components such as on-disk data structures, log-based recovery, and buffer management, that were developed years ago but are not suited to modern hardware. Those components impose an huge overhead to transaction processing [17] limiting its scalability. On the other, large scale NoSQL databases have a simple data model, using a simple key-value store or at most variants of the entity-attribute-value (EAV) model [22] which strongly limit the expressiveness of data representation. Therefore, a first challenge consists in addressing the impedance mismatches [21] while supporting SQL queries. Moreover, it implies mapping relational tables and indexes to the NoSQL database tables in such way that the processing capabilities of the database are fully exploited. The other major challenge regards the basic key-value store interface of NoSQL databases which only allows applications to insert, query, and remove individual tuples or, at most, issue range queries based on the primary key of the tuple. These range queries allow for fast iteration over ranges of rows and also allow to limit the number and what columns are returned. However, NoSQL databases do not support partial key scans, but SQL index scans must perform equality and range queries on all or part of the fields of the index.

Contributions. This paper makes the following three contributions. First, we propose an architecture that allows to combine the expressiveness and performance of RDBMS with the scalability of a NoSQL database. The resulting query processing component is stateless regarding application data and can thus be seamlessly replicated and embedded in the client application. Then, we show how the basic key-value operations and data models can be mapped to scan operators within a traditional (SQL enabled) RDBMS. And finally, we describe a complete implementation of DQE with full SQL support, using Apache Derby's [11] query engine and HBase as the NoSQL database.

Roadmap. The remainder of the paper is structured as follows. Section 2 introduces the proposed architecture of the DQE. Section 3 describes how it is implemented using Apache Derby components and HBase. Section 4 presents the experimental evaluation. Section 5 compares our approach to other proposals for query processing on a NoSQL database, and Section 6 concludes the paper.

2 Architecture



(a) NoSQL architecture. (b) Centralized SQL RDBMS. (c) Distributed query engine (DQE).

Fig. 1. Data management architectures.

The proposed architecture is shown in Figure 1(c), in the context of a scalable NoSQL and a traditional RDBMS. A major motivation for a NoSQL database is scalability. As depicted in Figure 1(a), a typical NoSQL database builds on a distributed setting with multiple nodes of commodity hardware. By adding more nodes to the system we can increase both the overall performance and capacity of the system and also its resilience through data replication. By allowing clients to directly contact multiple fragments and replicas, the system can scale also in terms of clients connected. To make this possible, NoSQL databases provide a simple data model as well as basic querying and searching capabilities, that allow applications to insert, query, and remove individual items or, at most, issue range queries based on the primary key of the item [28].

In sharp contrast, a RDBMS is organized as tables (also called relations). We seldom are concerned with the storage structure but instead express queries in a high-level language, invariably SQL. SQL allows applications to realize complex operations and processing capabilities, such as filtering, joining, grouping, ordering and counting.

Our current proposal builds on rewriting the internal architecture of a typical RDBMS, by reusing some of its components and adding new components atop of

a NoSQL database. To understand how components can be reused we examine the internals of a RDBMS roughly splitting it in a *query processor* block and a *storage manager* block (Figure 1(b)).

The query processor is responsible for offering an SQL based API to applications, and for translating the application queries, through compilation and execution, to the underlying storage manager.

The architecture proposed in Figure 1(c) reuses a number of components from the SQL query processor (shown in light gray). In detail, these are: the JDBC driver and client connection handler; the compiler and the optimizer, and a set of generic relational operator implementations. These components can be shielded from changes, as they depend only on components that are reimplemented (shown in medium gray) providing the same interfaces as those that, in the RDBMS, embody the centralized storage functionality (shown in dark gray) and that is removed from our architecture. The components to be reimplemented are the following:

- A mapping from the relational model to the data model of a NoSQL database. This includes: atomic data types and their representation, representation of rows and tables, and representation of indexes.
- A mapping from the relational schema to that of the NoSQL database, which allows data to be interpreted as relational tables (see Section 3);
- Implementation of sequential and index scan operators. This includes: matching the interface and data representation of the database and leveraging the indexing and filtering capabilities in the NoSQL database to minimize data network traffic;

The proposed architecture has the key advantage of being stateless regarding application data. Data manipulation language (DML) statements (SELECT, INSERT, UPDATE and DELETE) can be executed without any coordination among different DQE instances. As a result, the system should retain the scale-out capabilities of the supporting NoSQL database.

In addition, this architecture also offers the possibility to take advantage of the flexible schema exposed by the underlying NoSQL database. That is, each application applies its own view of the schema over the NoSQL database.

3 Implementation

The current DQE prototype was built by reusing Apache Derby components and uses HBase as the NoSQL database. In the following, we start with an overview of both systems.

HBase is a key-value based distributed data storage system based on Bigtable [7]. In HBase, data is stored in the form of HBase tables (HTable) that are multi-dimensional sorted maps. The index of the map is the row's key, column's name, and a timestamp. Columns are grouped into column families. Column families must be created before data can be stored under any column key in that family.

Data is maintained in lexicographic order by row key. Finally, each column can have multiple versions of the same data indexed by their timestamp.

A read or write operation is performed on a row using the row-key and one or more column-keys. Update operations on a single row are atomic, i.e. concurrent writes on a single row are serialized. Any update performed is immediately visible to any subsequent reads. HBase exports a non-blocking key-value interface on the data: put, get, delete, and scan operations.

HBase closely matches the scale-out properties assumed for NoSQL databases. HTables are horizontally partitioned in regions that are assigned to RegionServers nodes. In turn, each region is stored as an appendable file in the distributed file system, Hadoop File System (HDFS) [25] based on GFS [13]. By default, HBase uniformly distributes data among all available nodes in the systems.

Apache Derby is an open source relational database implemented entirely in Java. Derby has a small footprint, about 2.6 megabytes for the base engine and an embedded JDBC driver. In addition, it is easy to install, deploy and use.

Besides providing a complete implementation of SQL and JDBC, Derby has the advantage of already providing an embedded mode, which eases its use as a middleware layer.

The store layer of Derby is split into two main areas, access and raw. The access layer presents a conglomerate (table or index)/row based interface to the SQL layer. It handles table scans, index scans, index lookups, indexing, sorting, locking policies, transactions, isolation levels. The access layer sits on top of the raw store, which provides the raw storage of rows in pages in files, transaction logging, transaction management. Following the architecture proposed in the previous chapter, the raw store layer was removed in our prototype and some components of the access layer were replaced.

3.1 Prototype components

The system is composed of the following layers: (i) query engine, (ii) storage and (iii) file system. Applications issue SQL requests to any query engine node. The query engine node communicates with storage nodes, executes queries and returns the results to applications. We use Derby components to implement the query engine. We reuse its query processing sub-system, both the compiler and the optimizer components. Two new operators for index and sequential data scans have been added to the set of Derby's generic relational operators. These operators leverage HBase's indexing and filtering capabilities to minimize the amount of data that needs to be fetched. The SQL advanced operators such as JOIN and aggregations are not supported by HBase and are implemented at the query engine. The query engine translates the user queries into some appropriate put, get, delete, and scan operations to be invoked on HBase. Each HBase region is stored as an appendable file in the distributed file system. The distributed file system could be implemented using any scalable file system that supports append-only files such as HDFS.

(a) Relational Table (b) Primary Key HTable (c) Unique Index HTable (d) Non-unique Index HTable

Fig. 2. Data Model Mapping

3.2 Relational-tuple store mapping

Relational tables and secondary indexes are mapped to the HBase's data model. We have adopted a simple mapping from a relational table to a HTable. There is a one-to-one mapping where the HBase row's key is the relational primary key (simple or compound) and all relational columns are mapped into a single column family. Since relational columns are not multi-valued, each relational column is mapped to a HTable column. The schema of relational tables is rigid, i.e., every row in the same table must have the same set of columns. However, the value for some relational columns can be NULL and thus an HTable column for a given row exists only if its original relational column for that row is not NULL.

A secondary index of the relational model is mapped into an additional HTable. The additional table is necessary so that data is ordered by the indexed attributes. For each indexed attribute a HTable row is added and its row's key is the indexed attribute. For unique indexes the row has a single column with its value being the key of the matching indexed row in the primary key table. For non-unique indexes there is one column per matching indexed row with the name of the column being the matching row's key. Figure 2(a) depicts a relational table example. The column Number is the primary key and the table has two additional indexes: one unique index on attribute Telephone and a non-unique index on column Address. Therefore, the mapping will have three HTables: base data — Figure 2(b), unique index on column Telephone — Figure 2(c), and non-unique index on column Address — Figure 2(d).

Due to the simple mapping from the relational data model to HBase, the user can take advantage of the flexible schema exposed by the underlying NoSQL database and directly use its data in HBase for simple queries or complex map-reduce jobs.

3.3 Reducing data transfer

In order to reduce network traffic between the query engine and HBase, the implementation of sequential and index scan operators takes advantage of the indexing and filtering capabilities of HBase.

For index scans data is maintained ordered by one or more columns. This allows to restrict the desired rows for a given scan by optionally specifying the

start and the stop keys. In a relational table each column is typed (e.g., char, date, integer, decimal, varchar) and data is ordered according to the natural order of the indexed column data type. However, row keys in HBase are plain byte arrays and neither Derby or HBase byte encoding preserve the data type's natural order. In order to build and store indexes in HBase maintaining the data type's order we need to map row keys into plain bytes in such a way that when HBase compares them the order of the data type is preserved. This mapping has been implemented for integer, decimal, char, varchar and date types. As indexes may be composite, besides each specific data type encoding, we also needed to define a way to encode multiple indexed columns in the same byte array. We do so by simply concatenating them from left to right, according to the order they are defined in the index using a pre-defined separator.

In HBase the start and stop keys of a scan must always refer to all the columns defined in the index. However, when using compound indexes the DQE may generate scans using subsets of the index columns. Indeed, an index scan can use equality conditions on any prefix of the indexed columns (from left to right) and at most one range condition on the rightmost queried column. In order to map these partial scans, the default start and stop keys in HBase are not used but instead the scan expression is run through HBase's BinaryPrefixComparator filter.

The aforementioned mechanisms reduce the traffic between the query engine and HBase by only retrieving the rows that match the range of the index scan. However, a scan can also select non-indexed columns. A naive implementation of this selection would fetch all rows from the index scan and test the relevant columns row by row. In detail, doing so on top of HBase would require a full table scan, which means fetching all the table rows from the different regions and possible different RegionServers. The full table would therefore be brought to the query engine instance and only then discard those rows not selected by the filter. To mitigate this performance overhead, particularly for low selective queries, that this approach may incur, the whole selection is pushed down into HBase. This is done by using the SingleColumnValueFilter filter to test a single column and, to combine them respecting the conjunctive normal form, using the FilterList filter. The latter represents an ordered list of filters that are evaluated with a specified boolean operator FilterList.Operator.MUST PASS ALL (AND) or FilterList.Operator.MUST PASS ONE (OR).

3.4 Metadata

Derby must also store information about the in-memory representation of tables and index, conglomerates, in a persistent manner.

For this we use a special HBase table, ConglomerateInfo, with information for all available conglomerates. ConglomerateInfo has a single column family, MetaInfo, and data for each conglomerate is stored in a row whose key is the conglomerate's identifier. Three columns are used to save information: value, a byte array with the encoding of all information; name, the name of the index or

table to which this conglomerate matches; and size, to store the estimate of the current size (number of rows) of the conglomerate.

The information stored in ConglomerateInfo is mainly modified by Data Definition Language (DDL) statements. However, the size attribute is modified by any update DML statement (INSERT, UPDATE or DELETE) and therefore this attribute may change frequently. Thus, we used a simple mechanism to update the value in a distributed and efficient manner. The size of the table/index is used only for the query optimizer to decide the best query plan and therefore don't need to be the most recent value. In most RDBMS, this value is maintained probabilistically manner and thus we update its value in each query engine instance asynchronously.

Each query engine instance maintains the last estimate it has for the global size (shared by all instances), updates it accordingly to the local changes (when some update DML statement occurs), and periodically updates the value stored in ConglomerateInfo (shared by all instances) and refresh its local estimate. For, this it maintains the delta of the size after the last update to ConglomerateInfo, and in the next update it increments or decrements the size stored in ConglomerateInfo with the delta value, using a special HBase method for this purpose (incrementColumnValue). Then, it resets the delta value for the next time window.

4 Evaluation

The evaluation of our prototype addresses two performance aspects: the overhead imposed by the DQE in terms of added latency, and the system scale-out in terms of the achieved throughput by increasing the number of nodes.

4.1 Overhead

We measure the increased latency of the system resulting from using the DQE instead of a standard HBase client.

Test workload We used a workload typical of NoSQL databases, Yahoo! Cloud Serving Benchmark [9]. YCSB was designed to benchmark the performance of NoSQL databases under different workloads. It has client implementations for several NoSQL databases and a JDBC client for RDBMS. We have used the HBase and JDBC clients without any modifications.

Experimental setting The machines used for the experiments had a 2.4 GHz Dual-Core AMD Opteron(tm) Processor, with 4GB of RAM and a local SATA hard-disk. The machines were interconnected by a switched Gigabit local area network.

For the experiments 2 machines were used: one to run the workload generator, using an embedded connection to the modified Derby; and another running

Table 1. Overhead results (ms)

Workload	1 thread, 100 tps		50 threads, 100 tps	
Operation	HBase	DQE	HBase	DQE
Insert	0.58	0.93	1.04	1.98
Update	0.51	1.3	2.66	3.1
Read	0.53	0.79	1.63	1.7
Scan	1.43	2.9	4.64	6.1

HBase. HBase was run in standalone mode, meaning that the HBase master and HBase RegionServer were collocated in the same machine using the local filesystem.

The YCSB database was populated with 100,000 rows (185MB) and the workload consisted of 1,000,000 operations. The proportion for the different types of operations was 60% reads, 20% updates and 20% scans. The operations were distributed uniformly over the database rows. The size of the scan operator was also a uniform random number between 1 and 10. We measured two runs where each client had 1 and 50 threads, respectively. In both, the target throughput was 100 operations per second.

Results The average latency (in milliseconds) for the YCSB workload is shown in Table 1, for the standard HBase client and DQE.

The results for the insert operations correspond to the loading of the database while other operations are due to the mix of operations generated by the workload itself.

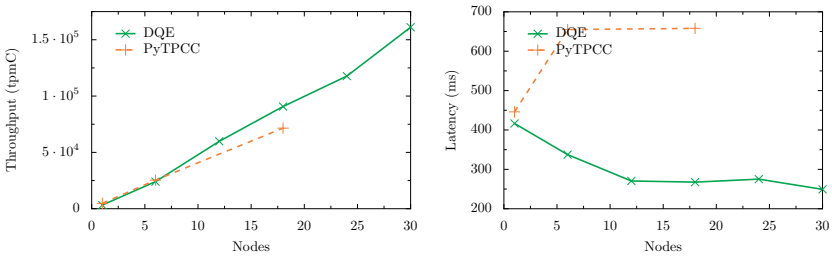
The results show that for all types of operations the query engine can be embedded in the application with an overhead totally in line with the base figures. The additional latency is due to the SQL processing and required marshalling/unmarshaling. Moreover, the overhead of DQE decreases with the increasing number of concurrent clients (threads).

4.2 Scale-out

We measured the increased throughput of the system when varying the number RegionServer nodes from 1 to 30.

Test Workload For the evaluation of the scale-out we used the load of an industry standard on-line transaction processing SQL benchmark, TPC-C.[1] It mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. The warehouses are hotspots of the system and the benchmark defines 10 client per warehouse.

[1] Since the current system does not include a transaction manager, all transactional contexts of the benchmark are simply discarded.



(a) Throughput (b) Latency

Fig. 3. TPC-C results

TPC-C specifies five transactions: NewOrder with 44% of the occurrences; Payment with 44%; OrderStatus with 4%; Delivery with 4%; and StockLevel with 4%. The NewOrder, Payment and Delivery are update transactions while the others are read-only. The traffic is a mixture of 8% read-only and 92% update transactions and therefore is a write intensive workload.

We have used both an existing SQL implementation,[2] without modifications, to drive the DQE, and an existing NoSQL implementation optimized for HBase.[3] Briefly, in the NoSQL implementation, TPC-C columns are grouped into column families, named differently for optimization, and the data storage layout has been optimized.

Experimental Setting We ran all the experiments on a cluster of 42 machines with 3.10GHz GHz Dual-Core i3-2100 CPU, with 4GB of RAM and a local SATA hard-disk. The machines were interconnected by a switched Gigabit local area network.

The TPC-C workload has run from a varying number of machines. For our proposal, we varied the number of client machines from 1 to 10, each running 150 client threads. Each client machine as also running a DQE instance as a middleware layer.

One machine was used to run the HDFS namenode, HBase Master and Zookeper [18]. The remaining machines were RegionServers, each configured with a heap of 3GB, and also running a HDFS DataNode instance.

The TPC-C database was populated according to the number of RegionServers, ranging from 5 warehouses, for a single RegionServer, to 150 warehouses, for 30 RegionServers. All TPC-C tables, were partitioned and distributed so there were 5 warehouses per RegionServer each handling a total of 50 clients. With 150 warehouses, the size of the database was about 75GB.

[2] BenchmarkSQL - `http://sourceforge.net/projects/benchmarksql/`
[3] `https://github.com/apavlo/py-tpcc/wiki/HBase-Driver`

Results The system throughput for a varying configuration of 1, 6, 12, 18, 24, and 30 RegionServers is depicted in Figure 3(a). The results show that DQE presents linear scalability. This is mainly due to the scale independence of the query processing layer and the scalability of the NoSQL database layer.

Furthermore, while with the DQE the system has a slightly lower throughput up tp 6 RegionServers its scalability is better than that of HBase under the NoSQL TPC-C load generator. This can be mainly attributed to the optimizations achieved by Derby's query engine that take advantage of relational operators, filtering and secondary indexes, while manual optimizations and denormalization still incur on greater complexity resulting in a greater overhead and worse scalability.

In this specific case, the query engine can greatly restrict the amount of data retrieved from HBase by, as previously explained, also taking advantage of HBase filters to select non-indexed columns. As a matter of fact, the network traffic when using the DQE is much lower than using the NoSQL TPC-C implementation. In fact, this prevented us from getting results for PyTPCC, the implementation optimized for HBase, with more than 18 RegionServers due to the saturation of the network. The latency figures in Figure 3(b) reflect the problem very clearly.

5 Related Work

Existing NoSQL databases rely on simplified and heterogenous query interfaces, that constitute a barrier on their adoption. Projects like BigQuery [1], Hive [2] or Tenzing [20] try to mitigate this constraint by providing an interface based on SQL over a MapReduce framework and a key/value store, but are mainly intended for data warehousing and analytical purposes.

BigQuery is a Google web service, built on top of BigTable. As query language, it uses a SQL dialect, that is a variation of the standard. It only offers a subset of the standard operators like selection, projection, aggregation and ordering. Joins and other more complex operators are not supported. In addition, data is immutable once uploaded to BigTable. Hive is built on top of Hadoop, a project that encompasses HDFS and the MapReduce framework. Hive also defines a simple SQL-like query language to query data but it offers more complex operators such as equi-joins, which are converted into MapReduce jobs, and unions. Likewise, Tenzing relies on a MapReduce framework to provide a SQL query execution engine, offering a mostly complete SQL implementation. The Hadapt commercial system[4] (previously HadoopDB [3]) is also an analytical driven database, but it takes a slightly different approach by providing a hybrid system. Like Hive, it uses an SQL interface over the MapReduce framework from Hadoop, but replaces the HDFS layer with a cluster of single-node relational databases.

Like Hadapt, the CloudDB [16] project is a hybrid system. However, it supports both OLAP and OLTP by providing three types of data storage systems:

[4] `http://www.hadapt.com/`

a relational database, a NoSQL database and a database oriented for OLAP. Data is stored in the database, according to the guarantees of data consistency required by the user.

Similarly to our approach, PIQL [4] and Megastore [5] propose an architecture with higher-level processing functionality via a database library.

In PIQL, the application issues queries in a new declarative language that is based on a subset of SQL, but extended with new statements and some new operators to always achieve a predictable performance independently of the database size (i.e. scale-independence). However, there are several restrictions on the supported operations. For instance, a *table scan* is not scale-independent and has to be appended with a *limit* statement to bound the results. While this is done to achieve predictable performance it is a major impediment to run legacy applications.

MegaStore is built on top of BigTable and implements some of the features of RDBMS, such as secondary indexing. Nonetheless, join operations must be implemented at the application side. Therefore, applications must be written specifically for MegaStore using its data model and queries are restricted to scans and lookups.

The use of a library-centric component to offer higher-level processing functionality in our prototype is similar to the architecture of PIQL, Megastore, as well as that by Brantner et al. [6]. However, our architecture allows to combine the expressiveness and performance of RDBMS, taking advantage of its query optimizer and full SQL support. This allows our proposal to run existing SQL applications and tools while retaining the scale-out capabilities of the NoSQL database.

On a different perspective, other approaches make use of object mapping tools that allow to bypass the database lower level interfaces. By using [14], the user has at her disposal the generic object interfaces like JPA and JDO that allow her to use NoSQL databases in an almost transparent way, leveraging the knowledge already existent in the area. These solutions have also the advantage of aiding the migration of existent solutions based on object to relation mappers allowing the mix of different types of databases under the same code base.

6 Conclusions and Future Work

We presented a distributed query engine allowing to execute ANSI compliant SQL queries on top of a NoSQL database. The query engine allows to combine the expressiveness and performance of a Relational Database Management System with the scalability and schema flexibility of a NoSQL database.

The developed prototype offers a standard JDBC client interface and can be embedded in the client application as a middleware layer. It is stateless with respect to application data making its replication straightforward. For the execution of data management language statements it does not require any kind of coordination which prevents any undesirable impact on the scalability of the

underlying NoSQL database. The feasibility of the approach is demonstrated by the performance results obtained with the YCSB and TPC-C benchmarks.

Moreover, the comparison with a NoSQL TPC-C implementation optimized for HBase shows that the presented prototype, through the use of the full-fledged query engine from Apache Derby achieves impressive performance.

Acknowledgment

This work is part-funded by: ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020; and European Union Seventh Framework Programme (FP7) under grant agreement n° 257993 (CumuloNimbo).

References

1. BigQuery: Google. http://code.google.com/apis/bigquery/ (2011)
2. Hive: Hive. http://hive.apache.org/ (2011)
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. 2, 922–933 (August 2009)
4. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: PIQL: success-tolerant query processing in the cloud. Proc. VLDB Endow. 5(3), 181–192 (Nov 2011)
5. Baker, J., Bondç, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., L´eon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: CIDR (2011)
6. Brantner, M., Florescu, D., Graf, D., Kossmann, D., Kraska, T.: Building a database on S3. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 251–264. SIGMOD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1376616.1376645
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI'06 (2006)
8. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. (2008)
9. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC'10 (2010)
10. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP'07 (2007)
11. Foundation, A.S.: Apache derby. http://db.apache.org/derby/ (2013)
12. George, L.: HBase: The Definitive Guide. O'Reilly Media (2011)
13. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Operating Systems Review 37(5), 29–43 (2003)

14. Gomes, P., Pereira, J., Oliveira, R.: An object mapping for the Cassandra distributed database. In: Inforum (2011)
15. Google: Cloud SQL: pick the plan that fits your app. http://googleappengine.blogspot.pt/2012/05/cloud-sql-pick-plan-that-fits-your-app.html (May 2012)
16. Hacigümüs, H., Tatemura, J., Hsiung, W.P., Moon, H.J., Po, O., Sawires, A., Chi, Y., Jafarpour, H.: CloudDB: One Size Fits All Revived. In: Proceedings of the 2010 6th World Congress on Services (2010)
17. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 981–992. SIGMOD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1376616.1376713
18. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. pp. 11–11. USENIXATC'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1855840.1855851
19. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. In: LADIS'09 (2009)
20. Lin, L., Lychagina, V., Wong, M.: Tenzing: A SQL implementation on the MapReduce framework. Proceedings of the VLDB Endowment 4(12), 1318–1327 (2011)
21. Meijer, E., Bierman, G.: A co-relational model of data for large shared data banks. ACM Queue 9(3), 30:30–30:48 (Mar 2011), http://doi.acm.org/10.1145/1952746.1961297
22. Nadkarni, P., Brandt, C.: Data Extraction and Ad Hoc Query of an Entity-Attribute-Value Database. Journal of the American Medical Informatics Association 5(6), 511–527 (1998)
23. Rys, M.: Scalable SQL. ACM Queue: Tomorrow's Computing Today 9(4), 30 (Apr 2011)
24. SalesForce.com: Phoenix: A SQL layer over HBase. https://github.com/forcedotcom/phoenix (May 2013)
25. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10. MSST '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/MSST.2010.5496972
26. Stonebraker, M., Cattell, R.: 10 rules for scalable performance in 'simple operation' datastores. Commun. ACM 54(6), 72–80 (Jun 2011), http://doi.acm.org/10.1145/1953122.1953144
27. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proceedings of the 33rd international conference on Very large data bases. pp. 1150–1160. VLDB '07, VLDB Endowment (2007), http://dl.acm.org/citation.cfm?id=1325851.1325981
28. Vilaça, R., Cruz, F., Oliveira, R.: On the expressiveness and trade-offs of large scale tuple stores. In: On the Move to Meaningful Internet Systems, OTM 2010, Lecture Notes in Computer Science, vol. 6427. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-16949-6_5