A Simple Approach for Executing SQL on a NoSQL Datastore

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 datastores have been initially introduced to support a few concrete extreme scale applications. Limited query and indexing capabilities were therefore not a major impediment, as the specificity and scale of the target application justified the investment in manually crafting application code.

With a number of alternatives now available and mature, there is an increasing willingness to use NoSQL datastores in a wider and more diverse spectrum of applications, where hand-crafted query code is a much less enticing trade-off.

We address this challenge with a simple approach for running SQL queries on top of a NoSQL datastore while preserving the underlying scalability, flexible schema and transaction-less semantics. We demonstrate our approach with a running prototype atop HBase. Our evaluation, conducted using YCSB shows the minimal overhead compared to the direct usage of HBase and the evaluation using an unmodified SQL implementation of a standard relational database workload, TPC-C, shows that the proposal presents linear scalability. Moreover, the comparison with a TPC-C implementation optimized for HBase, shows that complex SQL applications can be easily run and even achieve better results.

Keywords: NoSQL; Cloud Computing; Dependability; Middleware

1 Introduction

With cloud-based datastores in Platform-as-a-Service offerings, such as Google’s BigTable [7], Amazon’s DynamoDB [10], and Yahoo!’s PNUTS [8] and open source packages, such as HBase and Cassandra [17], NoSQL becomes attractive for a wider spectrum of applications. However, most Web-scale applications applications (such as Facebook, MySpace, and Twitter) still remain SQL-based for their core data management [21]. In fact, one of the most requested additions to the Google App Engine platform has been a SQL database [13].

Without full SQL support, it is hard to provide a smooth migration of legacy applications, which makes it a hurdle to the adoption of NoSQL datastores by a wider potential market. Moreover, a high number of tools coupled to SQL have been developed over the years. Consequently, having full SQL support makes all
these tools immediately available to developers of Web-scale applications over NoSQL datastores.

As a result, given the prevalence of SQL as a query language for databases, Web-scale applications would highly benefit from an efficient SQL query engine running on top of scalable NoSQL datastores. Such query engine should also be able to scale with the database. This is only possible without the coordination overhead, introduced by transactional semantics.

This has sparked a number of proposals for middleware that exposes higher-level query interfaces on top of the barebones key-value primitives. 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 [22], PIQL [4]) to translation of analytical queries into map-reduce jobs [2, 1, 18, 3]. However, due to the complexity of SQL, existing solutions are limited to a subset of SQL, thus not allowing to leverage exiting SQL applications and tools.

In this paper, we present an architecture of a distributed query engine (DQE) for running SQL queries on top of a NoSQL datastore, while preserving the underlying scalability; flexible schema and transaction-less semantics. The DQE allows to combine the power of performance of Relational Database Management Systems (RDBMS) by taking advantage of its query optimizer and full SQL support, with the effectiveness and scalability of a NoSQL datastore.

Support for scalable SQL atop an elastic datastore poses several architectural challenges to the query engine, in fact, to the same extent as any scalable query engine [25, 24]. On the 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 [15] limiting its scalability.

On the other hand, large scale NoSQL datastores have a simple data model, using a simple key-value store or at most variants of the entity-attribute-value (EAV) model [20]. Therefore, one of the challenges consists of addressing the impedance mismatches [19] while supporting SQL queries. Moreover, it implies mapping relational tables and indexes to the datastore tables in such way that the processing capabilities of the datastore are exploited at its most.

Another challenge is concerned with NoSQL datastores only offering a simple key-value store interface, which allows applications to insert, query, and remove individual tuples or at most range queries based on the primary key of the tuple. The range queries allow for fast iteration over ranges of rows and also allow to limit the number and what columns are returned. However, NoSQL datastores do not support partial key scans, but index scans in RDBMS must perform equals and range queries on all or a subset of the fields of the index.

Contributions. In this context, we make the following contributions:
First, we propose an architecture that allows to combine the power and performance of RDBMS with the scalability of a NoSQL datastore. The resulting query processing component is mostly stateless, thus can be seamlessly replicated and even embedded in the application. Second, we show that the simple
key-value operations and data models can be matched to scan operators within a traditional RDBMS. Third, we describe a complete implementation of DQE with full SQL support, using Derby and HBase as the NoSQL datastore. The DQE provides a standard JDBC client interface, which can be plugged in existing applications and middleware (e.g. object-relation mappers). The prototype uses as much as possible the indexing and filtering capabilities of HBase, namely filters and coprocessors. Fourth, we validate our design, showing that it has minimal overhead, presents linear scalability, and it can even achieve better results than complex HBase applications.

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

2 Architecture

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 datastore is scalability. As depicted in Figure 1(a), a typical NoSQL datastore builds on a distributed setting with multiple nodes of commodity hardware. Just by adding more nodes into the system (i.e. scaling out), one can not only increase the overall performance and capacity of the system, but also its resilience, and thus availability, by means of 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, they provide a simple data model as well as primitive querying and searching capabilities, that allows applications to insert, query, and remove individual items or at most range queries based on the primary key of the item [26].

In sharp contrast, a RDBMS is organized as tables (called relations) and developers are not concerned with the storage structure, but instead express queries in a high-level language: SQL. SQL allows applications to realize complex operations and processing capabilities, such as filtering, joining, grouping, ordering and counting.

Our proposal builds on a rewrite of the internal architecture of RDBMS, by reusing some existing components, by adding new components atop scalable NoSQL datastores as well as removing several components that are not needed on modern hardware which would limit scalability. Towards understanding how components can be reused in our proposal, we examine the internals of traditional RDBMS architecture, dividing it roughly in a query processor and a storage manager functions, Figure 1(b).

The query processor is responsible for offering a relational SQL based API for applications, and to translate the application queries, comprising two main stages: compilation and execution of the query.

The architecture proposed (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 re-implemented (shown in medium gray) providing the same interfaces as those that, in the RDBMS, embody the centralized storage functionality (shown in dark gray), which are removed from our architecture. In detail the components that must be re-implemented are the following:

- A mapping from the relational model to the data model of a NoSQL datastore. This includes: atomic data types and their representation; representation of rows and tables; representation of indexes.
- A mapping from the relational schema to that of the datastore, 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 datastore; taking advantage of indexing and filtering capabilities in the datastore, to minimize data network traffic;
- A stub of the transaction management.

The proposed architecture has the key advantage of being stateless regarding data. In fact, data manipulation language (DML) statements can be executed without coordination among different client application instances. Likewise, the proposed architecture also offers the possibility to take advantage of the flexible schema exposed by the underlying datastore. That is, each application applies its own view of the schema over the NoSQL datastore. The proposed architecture
should, therefore, retain the seamless scale-out of the NoSQL datastore and application.

3 Implementation

The current prototype, DQE, is built by reusing Apache Derby components and HBase as the NoSQL datastore.

HBase Overview: 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, as HTables are horizontally partitioned in regions. In turn, regions are assigned to RegionServers, and each region is stored as an appendable file in the distributed file system, Hadoop File System (HDFS) [23] based on GFS [11].

Derby Overview: Apache Derby is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0.

Derby has a small footprint, about 2.6 megabytes for the base engine and an embedded JDBC driver. In addition 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 matches the desired ability to function 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 was removed in our prototype and some components of the access layer were replaced.
### 3.1 Prototype architecture

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 mostly reuse the query processing sub-system of Derby. The storage management sub-system is, however, replaced to be able to operate on HBase. For query processing Apache Derby’s compiler and optimizer were reused. Two new operators for index and sequential data scans have been added to the set of Apache 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.

### 3.2 Relational-tuple store mapping

Relational tables and secondary indexes are mapped to the HBase’s data model. We adopted a simple mapping from a relational table to an 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 an 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 attribute.

---

**Fig. 2. Data Model Mapping**
row with the name of the column being the matching row’s key. Figure 2(a) depicts an example relational table. 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).

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 possibly 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 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 in a probability manner and thus we update its value in each query engine instance in an asynchronous manner.

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 the proposed system was made regarding two different aspects. Firstly, we measured its overhead in relation to standard HBase. Secondly, we evaluated the scale-out properties of our proposal.

4.1 Overhead

In this section we measure the overhead of our proposal, in terms of latency, compared to the standard HBase client, using a workload typical of NoSQL datastores, Yahoo! Cloud Serving Benchmark, YCSB [9].
Table 1. Overhead results (ms)

<table>
<thead>
<tr>
<th>Workload</th>
<th>1 thread, 100 tps</th>
<th>50 threads, 100 tps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>HBase</td>
<td>DQE</td>
</tr>
<tr>
<td>Insert</td>
<td>0.58</td>
<td>0.93</td>
</tr>
<tr>
<td>Update</td>
<td>0.51</td>
<td>1.3</td>
</tr>
<tr>
<td>Read</td>
<td>0.53</td>
<td>0.79</td>
</tr>
<tr>
<td>Scan</td>
<td>1.43</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Test Workload YCSB was designed to benchmark the performance of NoSQL datastores under different workloads. It has client implementations for several NoSQL datastores and a JDBC client for RDBMS. We have used the HBase and JDBC clients without further modifications.

Experimental Setting The machines used for those experiments have 2.4 GHz Dual-Core AMD Opteron(tm) Processor, with 4GB memory and a local SATA disk. The machines are interconnected by a switched Gigabit local area network.

For these experiments, 2 machines were used for the evaluation of our proposal: one to run the workload generator using an embedded connection to the modified Derby; and other one running 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 consists of 1,000,000 operations. The proportion for the different types of operations was read=0.6, update=0.2 and scan=0.2. The operations are distributed uniformly over database rows. The size of scan operator was also an uniform random number between 1 and 10. Each client has 1 or 50 threads and a target throughput of 100 operations per second.

Results The overhead in terms of average latency (in milliseconds) for the YCSB workload is shown in Table 1. We compare the overhead of our proposal, DQE, against the standard HBase client.

The results for the insert operation are from the load of the database while others are from the mixture of operation generated by the workload itself.

The results show that for all types of operations the query engine can be embedded in the application with a minimal overhead when compared to the direct usage of HBase. The additional overhead is due to the SQL processing and additional marshaling/unmarshalling. Moreover, the overhead of DQE decreases with the increasing number of concurrent clients (threads).

4.2 Scale-out

We evaluated the scalability of our proposal in terms of achieving increased throughput by scaling-out the system from a cluster with a single RegionServer to 30.
**Test Workload** For the evaluation of the scale-out we used the load of a industry standard on-line transaction processing SQL benchmark, TPC-C. It mimics a whole-sale 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.

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\(^1\), without modifications, to evaluate our proposal, and an existing TPC-C implementation optimized for HBase\(^2\). Briefly, in the HBase implementation TPC-C columns are grouped into column families, named differently for optimization, and data storage layout has been optimized.

**Experimental Setting** We ran the experiments on a cluster of 42 machines with 3.10GHz GHz Quad-Core i3-2100 CPU, with 4GB memory and a local SATA disk. The machines are interconnected by a switched Gigabit local area network.

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

One machine is used to run the HDFS namenode, HBase Master and Zookeeper [16]. The remaining machines are RegionServers, each configured with a heap of 3 GB, 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 is about 75 GB.

**Results** The throughput under the scaling-out of the system with 1, 6, 12, 18, 24, and 30 RegionServers is depicted in Figure 3(a). The results show that our proposal presents linear scalability. This is mainly due to the scale independence of the query processing layer and the NoSQL datastore layer. Moreover, as previously shown the query processing layer has a low overhead and has the advantage of being stateless.

Furthermore, while our proposal, using an existing SQL implementation, has a slightly lower throughput for 1 and 6 RegionServers than the implementation specifically developed and optimized for HBase, it scales better.

\(^1\) BenchmarkSQL - [http://sourceforge.net/projects/benchmarksql/](http://sourceforge.net/projects/benchmarksql/)

\(^2\) [https://github.com/apavlo/py-tpcc/wiki/HBase-Driver](https://github.com/apavlo/py-tpcc/wiki/HBase-Driver)
This can be mainly attributed to the optimizations achieved by the distributed query engine that takes advantage of relational operators, filtering and secondary indexes, while manual optimizations and de-normalization still incur on greater complexity resulting in a greater overhead and affecting the desired scalability.

In this specific case the distributed query engine can greatly restrict the amount of data retrieved from HBase by also taking advantage of HBase Filters to select non-indexed columns as previously explained. As a matter of fact, the network traffic when using our proposal is much lower than using the TPC-C implementation for HBase. In fact, this prevents us from from getting results for this implementation with more than 18 RegionServers because network was saturated. The result for latency, in Figure 3(b), confirms these statements.

5 Related Work

Existent NoSQL datastores rely on a simplified and heterogenous query interfaces, which constitutes a barrier on their adoption. Projects like BigQuery [1], Hive [2] or Tenzing [18] 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 thus 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 the HDFS and the MapReduce framework. On the one hand, 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\(^3\) (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 a 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 \([14]\) project is a hybrid system. However, it supports both OLAP and OLTP by providing three types of data storage systems: a relational database, a NoSQL datastore 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 the SQL query language, 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* as is not scale-independent 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 on 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 by Brantner et al. \([6]\). However, our architecture allows to combine the power and performance of RDBMS, taking advantage of its query optimizer and full SQL support, with the scalability of a NoSQL datastore. This allows our proposal to run existing SQL applications and tools while retaining the seamless scale-out of the NoSQL datastore and application.

On a different perspective, other approaches make use of object mapping tools that allow to bypass the database lower level interfaces. By using \([12]\), the user has at her disposal the generic object interfaces like JPA and JDO that allows her to use NoSQL datastores 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 datastores under the same code base.

### 6 Conclusions and Future Work

The proposed approach provides a SQL query engine for a NoSQL datastore, including the standard JDBC client interface. At the same time, it provides the

\(^3\) [http://www.hadapt.com/](http://www.hadapt.com/)
ability to use the transaction-less isolation level and flexible schema exposed by
the underlying datastore. The result is a query engine that can be embedded in
the application as a middleware layer, without the need for central components or
distributed coordination, and thus does not impact the ability to scale-out. The
feasibility of the approach is demonstrated by the performance results obtained
with YCSB and TPC-C.

Moreover, the comparison with a TPC-C implementation optimized for HBase
shows that by simply using the distributed query engine, SQL applications can
be easily run and even achieve better results.

Acknowledgment

This work is part-funded by: ERDF - European Regional Development Fund
through the COMPETE Programme (operational programme for competitive-
ess) 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 Frame-
work Programme (FP7) under grant agreement n° 257993 (CumuloNimbo).

References

3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: 
   HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for 
4. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: 
   PIQIL: success-tolerant query processing in the cloud. Proc. VLDB Endow. 5(3), 
   181–192 (Nov 2011)
   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, 
   Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for 
   Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!’s hosted data 
   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 avail-
    able key-value store. In: SOSP’07 (2007)