# XQuery processing over NoSQL stores

Henrique Valer
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern,
Germany
valer@cs.uni-kl.de

Caetano Sauer
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern,
Germany
csauer@cs.uni-kl.de

Theo Härder
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern,
Germany
hearder@cs.uni-kl.de

## ABSTRACT

Using NoSQL stores as storage layer for the execution of declarative query processing using XQuery provides a high-level interface to process data in an optimized manner. NoSQL refers to a plethora of new stores which essentially trades off well-known ACID properties for higher availability or scalability, using techniques such as eventual consistency, horizontal scalability, efficient replication, and schema-less data models. This work proposes a mapping from the data model of different kinds of NoSQL stores—key/value, columnar, and document-oriented—to the XDM data model, allowing for standardization and querying NoSQL data using higher-level languages, such as XQuery. This work also explores several optimization scenarios to improve performance on top of these stores. Besides, we also add updating semantics to XQuery by introducing simple CRUD-enabling functionalities. Finally, this work analyzes the performance of the system in several scenarios.

## Keywords

NoSQL, Big Data, key/value, XQuery, ACID, CAP

## 1. INTRODUCTION

We have seen a trend towards specialization in database markets in the last few years. There is no more one-size-fits-all approach when comes to storing and dealing with data, and different types of databases are being used to tackle different types of problems. One of these being the *Big Data* topic.

It is not completely clear what Big Data means after all. Lately, it is being characterized by the so-called 3 V's: *volume*—comprising the actual size of data; *velocity*—comprising essentially a time span in which data data must be analyzed; and *variety*—comprising types of data. Big Data applications need to understand how to create solutions in these data dimensions.

RDBMS's have had problems when facing Big Data applications, like in web environments. Two of the main reasons for that are scalability and flexibility. The solution RDBMS's provide is usually twofold: either (i) a horizontally-scalable architecture, which in database terms generally means giving up joins and also complex multi-row transactions; or (ii) by using parallel databases, thus using multiple CPUs and disks in parallel to optimize performance. While the latter increases complexity, the former just gives up operations because they are too hard to implement in distributed environments. Nevertheless, these solutions are neither scalable nor flexible.

NoSQL tackles these problems with a mix of techniques, which involves either weakening ACID properties or allowing more flexible data models. The latter is rather simple: some scenarios—such as web applications—do not conform to a rigid relational schema, cannot be bound to the structures of a RDBMS, and need flexibility. Solutions exist, such as using XML, JSON, pure key/value stores, etc, as data model for the storage layer. Regarding the former, some NoSQL systems relax consistency by using mechanisms such as multi-version concurrency control, thus allowing for eventual-consistent scenarios. Others support atomicity and isolation only when each transaction accesses data within some convenient subset of the database data. Atomic operations would require some distributed commit protocol—like two-phase commit—involving all nodes participating in the transaction, and that would definitely not scale. Note that this has nothing to do with SQL, as the acronym NoSQL suggests. Any RDBMS's that relaxes ACID properties could scale just as well, and keep SQL as querying language.

Nevertheless, when comes to performance, NoSQL systems have shown some interesting improvements. When considering update- and lookup-intensive OLTP workloads—scenarios where NoSQL are most often considered—the work of [13] shows that the total OLTP time is almost evenly distributed among four possible overheads: *logging*, *locking*, *latching*, and *buffer management*. In essence, NoSQL systems improve locking by relaxing atomicity, when compared to RDBMS's.

When considering OLAP scenarios, RDBMS's require rigid schema to perform usual OLAP queries, whereas most NoSQL stores rely on a brute-force processing model called *MapReduce*. It is a linearly-scalable programming model for processing and generating large data sets, and works with any data format or shape. Using MapReduce capabilities, parallelization details, fault-tolerance, and distribution aspects are transparently offered to the user. Nevertheless, it requires implementing queries from scratch and still suffers from the lack of proper tools to enhance its querying capa-

24th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 29.05.2012 - 01.06.2012, Lübbenau, Germany.

bilities. Moreover, when executed atop raw files, the processing is inefficient. NoSQL stores provide this elemental structure, thus one could provide a higher-level query language to take full advantage of it, like Hive [18], Pig [16], and JAQL [6].

These approaches require learning separated query languages, each of which specifically made for the implementation. Besides, some of them require schemas, like Hive and Pig, thus making them quite inflexible. On the other hand, there exists a standard that is flexible enough to handle the offered data flexibility of these different stores, whose compilation steps are directly mappable to distributed operations on MapReduce, and is been standardized for over a decade: XQuery.

## Contribution

Consider employing XQuery for implementing the large class of query-processing tasks, such as aggregating, sorting, filtering, transforming, joining, etc, on top of MapReduce as a first step towards standardization on the realms of NoSQL [17]. A second step is essentially to incorporate NoSQL systems as storage layer of such framework, providing a significant performance boost for MapReduce queries. This storage layer not only leverages the storage efficiency of RDBMS's, but allows for pushdown projections, filters, and predicates evaluation to be done as close to the storage level as possible, drastically reducing the amount of data used on the query processing level.

This is essentially the contribution of this work: allowing for NoSQL stores to be used as storage layer underneath a MapReduce-based XQuery engine, Brackit—a generic XQuery processor, independent of storage layer. We rely on Brackit's MapReduce-mapping facility as a transparent-distributed execution engine, thus providing scalability. Moreover, we exploit the XDM-mapping layer of Brackit, which provides flexibility by using new data models. We created three XDM-mappings, investigating three different NoSQL implementations, encompassing the most used types of NoSQL stores: *key/value*, *column-based*, and *document-based*.

The remainder of this paper is organized as follows. Section 2 introduces the NoSQL models and their characteristics. Section 3 describes the used XQuery engine, Brackit, and the execution environment of XQuery on top of the MapReduce model. Section 4 describes the mappings from various stores to XDM, besides all implemented optimizations. Section 5 exposes the developed experiments and the obtained results. Finally, Section 6 concludes this work.

## 2. NOSQL STORES

This work focuses on three different types of NoSQL stores, namely *key/value*, *columnar*, and *document-oriented*, represented by Riak [14], HBase[11], and MongoDB[8], respectively.

Riak is the simplest model we dealt with: a pure key/-value store. It provides solely read and write operations to uniquely-identified values, referenced by key. It does not provide operations that span across multiple data items and there is no need for relational schema. It uses concepts such as *buckets*, *keys*, and *values*. Data is stored and referenced by bucket/key pairs. Each bucket defines a virtual key space and can be thought of as tables in classical relational databases. Each key references a unique value, and there are no data type definitions: *objects* are the only unit of data storage. Moreover, Riak provides automatic load balancing and data replication. It does not have any relationship between data, even though it tries by adding link between key/value pairs. It provides the most flexibility, by allowing for a per-request scheme on choosing between availability or consistency. Its distributed system has no master node, thus no single point of failure, and in order to solve partial ordering, it uses *Vector Clocks* [15].

HBase enhances Riak's data model by allowing columnar data, where a table in HBase can be seen as a map of maps. More precisely, each key is an arbitrary string that maps to a row of data. A row is a map, where columns act as keys, and values are uninterpreted arrays of bytes. Columns are grouped into column families, and therefore, the full key access specification of a value is through column family concatenated with a column—or using HBase notation: a *qualifier*. Column families make the implementation more complex, but their existence enables fine-grained performance tuning, because (i) each column family's performance options are configured independently, like read and write access, and disk space consumption; and (ii) columns of a column family are stored contiguously in disk. Moreover, operations in HBase are atomic in the row level, thus keeping a consistent view of a given row. Data relations exist from column family to qualifiers, and operations are atomic on a per-row basis. HBase chooses consistency over availability, and much of that reflects on the system architecture. Auto-shardling and automatic replication are also present: shardling is automatically done by dividing data in regions, and replication is achieved by the master-slave pattern.

MongoDB fosters functionality by allowing more RDBMS-like features, such as secondary indexes, range queries, and sorting. The data unit is a *document*, which is an ordered set of keys with associated values. *Keys* are strings, and *values*, for the first time, are not simply objects, or arrays of bytes as in Riak or HBase. In MongoDB, values can be of different data types, such as strings, date, integers, and even embedded documents. MongoDB provides *collections*, which are grouping of documents, and *databases*, which are grouping of collections. Stored documents do not follow any predefined schema. Updates within a single document are transactional. Consistency is also taken over availability in MongoDB, as in HBase, and that also reflects in the system architecture, that follows a master-worker pattern.

Overall, all systems provide scaling-out, replication, and parallel-computation capabilities. What changes is essentially the data-model: Riak seams to be better suited for problems where data is not really relational, like logging. On the other hand, because of the lack of scan capabilities, on situations where data querying is needed, Riak will not perform that well. HBase allows for some relationship between data, besides built-in compression and versioning. It is thus an excellent tool for indexing web pages, which are highly textual (thus benefiting from compression), as well as inter-related and updatable (benefiting from built-in versioning). Finally, MongoDB provides documents as granularity unit, thus fitting well when the scenario involves highly-variable or unpredictable data.

## 3. BRACKIT AND MAPREDUCE

Several different XQuery engines are available as options for querying XML documents. Most of them provide either (i) a lightweight application that can perform queries on documents, or collections of documents, or (ii) an XML database that uses XQuery to query documents. The former lacks any sort of storage facility, while the latter is just not flexible enough, because of the built-in storage layer. Brackit[1] provides intrinsic flexibility, allowing for different storage levels to be "plugged in", without lacking the necessary performance when dealing with XML documents [5]. By dividing the components of the system into different modules, namely *language*, *engine*, and *storage*, it gives us the needed flexibility, thus allowing us to use any store for our storage layer.

## Compilation

The compilation process in Brackit works as follows: the *parser* analyzes the query to validate the syntax and ensure that there are no inconsistencies among parts of the statement. If any syntax errors are detected, the query compiler stops processing and returns the appropriate error message. Throughout this step, a data structure is built, namely an *AST* (Abstract Syntax Tree). Each node of the tree denotes a construct occurring in the source query, and is used through the rest of the compilation process. Simple rewrites, like *constant folding*, and the introduction of let bindings are also done in this step.

The *pipelining* phase transforms FLWOR expressions into *pipelines*—the internal, data-flow-oriented representation of FLWORs, discussed later. Optimizations are done atop pipelines, and the compiler uses global semantics stored in the AST to transform the query into a more-easily-optimized form. For example, the compiler will move predicates if possible, altering the level at which they are applied and potentially improving query performance. This type of operation movement is called *predicate pushdown*, or *filter pushdown*, and we will apply them to our stores later on. More optimizations such as *join recognition*, and *unnesting* are present in Brackit and are discussed in [4]. In the *optimization* phase, optimizations are applied to the AST. The *distribution* phase is specific to distributed scenarios, and is where MapReduce translation takes place. More details about the distribution phase are presented in [17]. At the end of the compilation, the *translator* receives the final AST. It generates a tree of executable physical operators. This compilation process chain is illustrated in Figure 1.

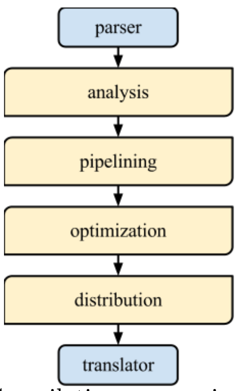

Figure 1: Compilation process in Brackit [5].

[1]Available at http:\\www.brackit.org

## XQuery over MapReduce

Mapping XQuery to the MapReduce model is an alternative to implementing a distributed query processor from scratch, as normally done in parallel databases. This choice relies on the MapReduce middleware for the distribution aspects. *BrackitMR* is one such implementation, and is more deeply discussed in [17]. It achieves a distributed XQuery engine in Brackit by scaling out using MapReduce.

The system hitherto cited processes collections stored in HDFS as text files, and therefore does not control details about encoding and management of low-level files. If the DBMS architecture [12] is considered, it implements solely the topmost layer of it, the set-oriented interface. It executes processes using MapReduce functions, but abstracts this from the final user by compiling XQuery over the MapReduce model.

It represents each query in MapReduce as sequence of *jobs*, where each job processes a section of a FLWOR pipeline. In order to use MapReduce as a query processor, (i) it breaks FLWOR pipelines are into map and reduce functions, and (ii) groups these functions to form a MapReduce job. On (i), it converts the logical-pipeline representation of the FLWOR expression—AST—to a MapReduce-friendly version. MapReduce uses a tree of *splits*, which represents the logical plan of a MapReduce-based query. Each *split* is a non-blocking operator used by MapReduce functions. The structure of splits is rather simple: it contains an AST and pointers to successor and predecessor splits. Because splits are organized in a bottom-up fashion, leaves of the tree are *map* functions, and the root is a *reduce* function—which produces the query output.

On (ii), the system uses the split tree to generate possibly multiple MapReduce job descriptions, which can be executed in a distributed manner. *Jobs* are exactly the ones used on Hadoop MapReduce [20], and therefore we will not go into details here.

## 4. XDM MAPPINGS

This section shows how to leverage NoSQL stores to work as storage layer for XQuery processing. First, we present mappings from NoSQL data models to XDM, adding XDM-node behavior to these data mappings. Afterwards, we discuss possible optimizations regarding data-filtering techniques.

## Riak

Riak's mapping strategy starts by constructing a *key/value tuple* from its low-level storage representation. This is essentially an abstraction and is completely dependent on the storage used by Riak. Second, we represent XDM operations on this key/value tuple. We map data stored within Riak utilizing Riak's linking mechanism. A key/value pair *kv* represents an XDM *element*, and key/value pairs linked to *kv* are addressed as children of *kv*. We map key/value tuples as XDM elements. The name of the element is simply the name of the bucket it belongs to. We create one bucket for the element itself, and one extra bucket for each link departing from the element. Each child element stored in a separated bucket represents a nested element within the key/value tuple. The name of the element is the name of the link between key/values. This does not necessarily decrease data locality: buckets are stored among distributed nodes based on hashed keys, therefore uniformly distributing the

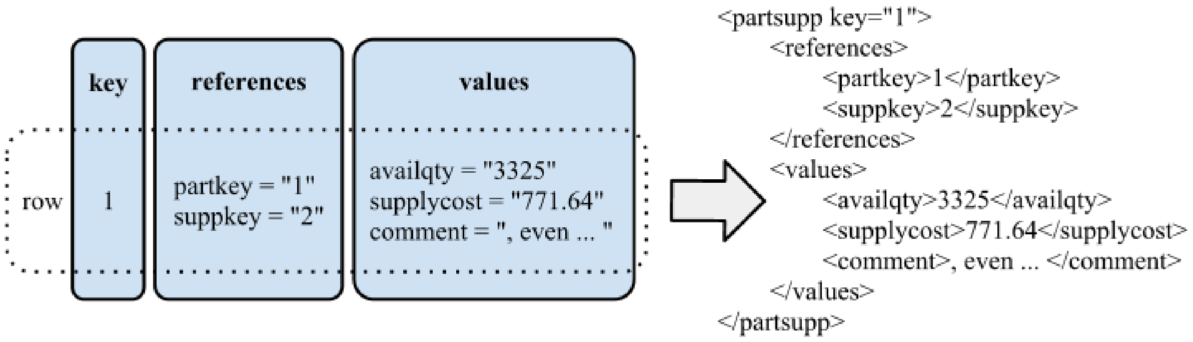

Figure 2: Mapping between an HBase row and an XDM instance.

load on the system. Besides, each element has an attribute *key* which Riak uses to access key/value pairs on the storage level.

It allows access using key/value as granularity, because every single element can be accessed within a single *get* operation. Full reconstruction of an element *el* requires one access for each key/value linked to *el*. Besides, Riak provides atomicity using single key/value pairs as granularity, therefore consistent updates of multiple key/value tuples cannot be guaranteed.

## HBase

HBase's mapping strategy starts by constructing a *columnar tuple* from the HDFS low-level-storage representation. HBase stores column-family data in separated files within HDFS, therefore we can use this to create an efficient mapping. Figure 2 presents this XDM mapping, where we map a table *partsupp* using two column families: *references* and *values*, five qualifiers: *partkey*, *suppkey*, *availqty*, *supplycost*, and *comment*. We map each row within an HBase table to an XDM element. The name of the element is simply the name of the table it belongs to, and we store the key used to access such element within HBase as an attribute in the element. The figure shows two *column families*: *references* and *values*. Each column family represents a child element, whose name is the name of the column family. Accordingly, each qualifier is nested as a child within the column-family element from which it descends.

## MongoDB

MongoDB's mapping strategy is straight-forward. Because it stores JSON-like documents, the mapping consists essentially of a *document field* → *element* mapping. We map each document within a MongoDB collection to an XDM element. The name of the element is the name of the collection it belongs to. We store the *id*—used to access the document within MongoDB—as an attribute on each element. Nested within the collection element, each field of the document represents a child element, whose name is the name of the field itself. Note that MongoDB allows fields to be of type *document*, therefore more complex nested elements can be achieved. Nevertheless, the mapping rules work recursively, just as described above.

## Nodes

We describe XDM mappings using object-oriented notation. Each store implements a *Node* interface that provides node behavior to data. Brackit interacts with the storage using this interface. It provides general rules present in XDM [19], Namespaces [2], and Xquery Update Facility [3] standards, resulting in navigational operations, comparisons, and other functionalities. *RiakRowNode* wraps Riak's *buckets*, *key/-values*, and *links*. *HBaseRowNode* wraps HBase's *tables*, *column families*, *qualifiers*, and *values*. Finally, *MongoRowNode* wraps MongoDB's *collections*, *documents*, *fields*, and *values*.

Overall, each instance of these objects represents one unit of data from the storage level. In order to better grasp the mapping, we describe the HBase abstraction in more details, because it represents the more complx case. Riak's and MongoDB's representation follow the same approach, but without a "second-level node". *Tables* are not represented within the Node interface, because their semantics represent where data is logically stored, and not data itself. Therefore, they are represented using a separated interface, called *Collection*. *Column families* represent a *first-level-access*. *Qualifiers* represent a *second-level-access*. Finally, *values* represent a *value-access*. Besides, first-level-access, second-level-access, and value-access must keep track of current indexes, allowing the node to properly implement XDM operations. Figure 3 depicts the mapping. The upper-most part of the picture shows a node which represents a data row from any of the three different stores. The first layer of nodes—with *level = 1st*—represents the *first-level-access*, explained previously. The semantic of first-level-access differs within different stores: while Riak and MongoDB interpret it as a value wrapper, HBase prefers a column family wrapper. Following, HBase is the only implementation that needs a *second-level-access*, represented by the middle-most node with *level = 2nd*, in this example accessing the wrapper of *regionkey = "1"*. Finally, lower-level nodes with *level = value* access values from the structure.

## Optimizations

We introduce projection and predicate pushdowns optimizations. The only storage that allows for predicate pushdown is MongoDB, while filter pushdown is realized on all of them. These optimizations are fundamental advantages of this work, when compared with processing MapReduce over raw files: we can take "shortcuts" that takes us directly to the bytes we want in the disk.

*Filter and projections pushdown* are an important optimization for minimizing the amount of data scanned and processed by storage levels, as well as reducing the amount

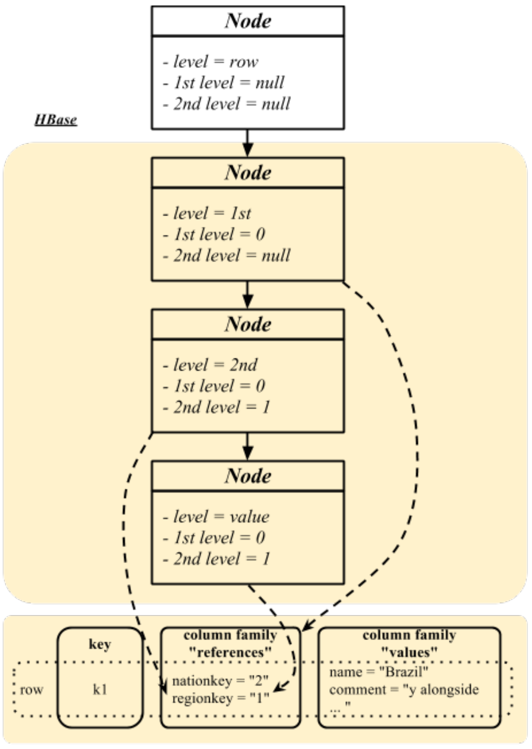

Figure 3: Nodes implementing XDM structure.

of data passed up to the query processor. *Predicate push-down* is yet another optimization technique to minimize the amount of data flowing between storage and processing layers. The whole idea is to process predicates as early in the plan as possible, thus pushing them to the storage layer.

On both cases we traverse the AST, generated in the beginning of the compilation step, looking for specific nodes, and when found we annotate the collection node on the AST with this information. The former looks for path expressions (*PathExpr*) that represents a *child* step from a collection node, or for *descendants* of collection nodes, because in the HBase implementation we have more than one access level within storage. The later looks for general-comparison operators, such as *equal*, *not equal*, *less than*, *greater than*, *less than or equal to*, and *greater than or equal to*. Afterwards, when accessing the collection on the storage level, we use the marked collections nodes to filter data, without further sending it to the query engine.

## NoSQL updates

The used NoSQL stores present different API to persist data. Even though XQuery does not provide data-storing mechanisms on its recommendation, it does provide an extension called *XQuery Update Facility* [3] for that end. It allows to add new nodes, delete or rename existing nodes, and replace existing nodes and their values. XQuery Update Facility adds very natural and efficient persistence-capabilities to XQuery, but it adds lots of complexity as well. Moreover, some of the constructions need document-order, which is simply not possible in the case of Riak. Therefore, simple-semantic functions such as “insert” or “put” seam more attractive, and achieve the goal of persisting or updating data.

The *insert* function stores a value within the underlying store. We provide two possible signatures: with or without a *$key*, therefore allowing for both insertions and updates.

```
db:insert($table as xs:string,
          $key as xs:string,
          $value as node()) as xs:boolean
```

The *delete* function deletes a values from the store. We also provide two possible signatures: with or without *$key*, therefore allowing for deletion of a giveng key, or droping a given table.

```
db:delete($table as xs:string,
          $key as xs:string) as xs:boolean
```

## 5. EXPERIMENTS

The framework we developed in this work is mainly concerned with the feasibility of executing XQuery queries atop NoSQL stores. Therefore, our focus is primarily on the proof of concept. The data used for our tests comes from the *TPC-H* benchmark [1]. The dataset size we used has 1GB, and we essentially scanned the five biggest tables on TPC-H: *part*, *partsupp*, *order*, *lineitem*, and *customer*. The experiments were performed in a single Intel Centrino Duo dual-core CPU with 2.00 GHz, with 4GB RAM, running Ubuntu Linux 10.04 LTS. HBase used is version 0.94.1, Riak is 1.2.1, and MongoDB is 2.2.1. It is not our goal to assess the scalability of these systems, but rather their query-procedure performance. For scalability benchmarks, we refer to [9] and [10].

### 5.1 Results

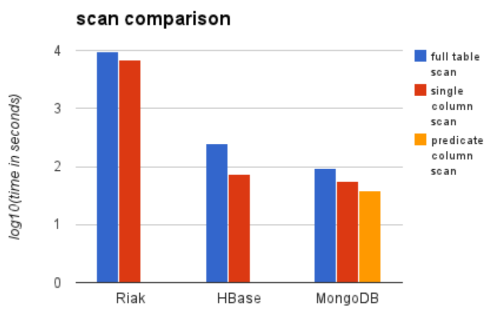

Figure 4: Latency comparison among stores.

Figure 4 shows the gathered latency times of the best schemes of each store, using log-scale. As we can see, all approaches take advantage from the optimization techniques. The blue column of the graph—*full table scan*—shows the latency when scanning all data from TPC-H tables. The red column —*single column scan*—represents the latency when scanning a simple column of each table. Filter pushdown optimizations explain the improvement in performance when compared to the first scan, reducing the amount of data flowing from storage to processing level. The orange column—*predicate column scan*—represents the latency when scanning a single column and where results were filtered by a predicate. We have chosen predicates to cut in half the amount of resulting data when compared with *single column scan*. The querying time was reduced in approximately 30%, not reaching the 50% theoretically-possible-improvement rate, essentially because of processing overhead. Nevertheless, it shows how efficient the technique is.

In scanning scenarios like the ones on this work, MongoDB has shown to be more efficient than the other stores, by always presenting better latency. MongoDB was faster by design: trading of data-storage capacity for data-addressability has proved to be a very efficiency-driven solution, although being a huge limitation. Moreover, MongoDB uses pre-caching techniques. Therefore, at run-time it allows working with data almost solely from main memory, specially in scanning scenarios.

## 6. CONCLUSIONS

We extended a mechanism that executes XQuery to work with different NoSQL stores as storage layer, thus providing a high-level interface to process data in an optimized manner. We have shown that our approach is generic enough to work with different NoSQL implementations.

Whenever querying these systems with MapReduce—taking advantage of its linearly-scalable programming model for processing and generating large-data sets—parallelization details, fault-tolerance, and distribution aspects are hidden from the user. Nevertheless, as a data-processing paradigm, MapReduce represents the past. It is not novel, does not use schemas, and provides a low-level record-at-a-time API: a scenario that represents the 1960's, before modern DBMS's. It requires implementing queries from scratch and still suffers from the lack of proper tools to enhance its querying capabilities. Moreover, when executed atop raw files, the processing is inefficient—because brute force is the only processing option. We solved precisely these two MapReduce problems: XQuery works as the higher-level query language, and NoSQL stores replace raw files, thus increasing performance. Overall, MapReduce emerges as solution for situations where DBMS's are too "hard" to work with, but it should not overlook the lessons of more than 40 years of database technology.

Other approaches cope with similar problems, like *Hive*, and *Scope*. Hive [18] is a framework for data warehousing on top of Hadoop. Nevertheless, it only provides equi-joins, and does not fully support point access, or CRUD operations—inserts into existing tables are not supported due to simplicity in the locking protocols. Moreover, it uses raw files as storage level, supporting only CSV files. Moreover, Hive is not flexible enough for Big Data problems, because it is not able to understand the structure of Hadoop files without some catalog information. Scope [7] provides a declarative scripting language targeted for massive data analysis, borrowing several features from SQL. It also runs atop a distributed computing platform, a MapReduce-like model, therefore suffering from the same problems: lack of flexibility and generality, although being scalable.

## 7. REFERENCES

[1] The tpc-h benchmark. `http://www.tpc.org/tpch/`, 1999.

[2] Namespaces in xml 1.1 (second edition). `http://www.w3.org/TR/xml-names11/`, August 2006.

[3] Xquery update facility 1.0. `http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/`, June 2009.

[4] S. Bächle. *Separating Key Concerns in Query Processing - Set Orientation, Physical Data Independence, and Parallelism.* PhD thesis, University of Kaiserslautern, 12 2012.

[5] S. Bächle and C. Sauer. Unleashing xquery for data-independent programming. *Submitted*, 2011.

[6] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[8] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide.* Oreilly Series. O'Reilly Media, Incorporated, 2010.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] T. Dory, B. Mejhas, P. V. Roy, and N. L. Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.

[11] L. George. *HBase: The Definitive Guide.* O'Reilly Media, 2011.

[12] T. Härder. Dbms architecture - new challenges ahead. *Datenbank-Spektrum*, 14:38–48, 2005.

[13] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there, 2008.

[14] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[15] F. Mattern. Virtual time and global states of distributed systems. In C. M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[17] C. Sauer. Xquery processing in the mapreduce framework. Master thesis, Technische Universität Kaiserslautern, 2012.

[18] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[19] N. Walsh, M. Fernández, A. Malhotra, M. Nagy, and J. Marsh. XQuery 1.0 and XPath 2.0 data model (XDM). `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/`, January 2007.

[20] T. White. *Hadoop: The Definitive Guide.* O'Reilly Media, 2012.