TECHNISCHE UNIVERSTÄT
KAISERSLAUTERN

Master Thesis

XQuery processing over NoSQL stores

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

in the

Databases and Information Systems (AG DBIS)
Computer Science

Author:
Henrique Valer

Supervisors:
M. Sc. Caetano Sauer
Prof. Dr.-Ing. Dr. h. c. Theo Härder

February 8, 2013

Abstract

This master thesis addresses the usage of NoSQL stores as storage layer for the execution of declarative query processing using XQuery, thus providing 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 thesis 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 thesis 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. Furthermore, this work also describes the computational model of MapReduce, and how an execution of XQuery on top of it is possible. Finally, this work analyzes the performance of the system in several scenarios.

Zusammenfassung

Diese Masterarbeit befasst sich mit der Nutzung von NoSQL-Stores als Speicherschicht für die Ausführung deklarativen Abfrageverarbeitung wie XQuery. Dadurch wird eine höherwertige Schnittstelle angeboten, um Daten in einer optimierten Weise zu verarbeiten. NoSQL bezieht sich auf eine Menge von neuen Stores, die wesentliche ACID-Eigenschaften für höhere Verfügbarkeit und Skalierbarkeit eintauschen. Dafür nutzen sie Techniken wie eventuelle Konsistenz, horizontale Skalierbarkeit, effiziente Replikation und schemalose Datenmodelle. Diese Arbeit stellt eine Abbildung der verschiedenen Datenmodelle von NoSQL-Stores—key/value, spalten- und dokument-orientierte—zum XDM-Datenmodell vor. Das ermöglicht Standardisierung und High-Level-Sprachen, wie XQuery, um NoSQL-Daten abzufragen. Diese Masterarbeit untersucht auch mehrere Optimierungsszenarien, um die Leistung auf diesen Stores zu verbessern. Darüber hinaus werden einfache CRUD-Funktionalitäten vorgestellt, durch die Updates in XQuery ermöglicht werden. Außerdem beschreibt diese Arbeit auch das MapReduce-Rechenmodell, und wie XQuery über MapReduce ausgeführt werden kann. Weiterhin analysiert diese Arbeit die Leistung des Systems in mehreren Szenarien.

Ich, Henrique VALER, versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema “XQuery processing over NoSQL stores” selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Datum: ______________________________

Henrique Valer: ______________________________

"A man needs to travel. By his means, not by stories, images, books or TV. By his own, with his eyes and feet, to understand what is his. For some day planting his own trees and giving them some value. To know the cold for enjoying the heat. To feel the distance and lack of shelter for being well under his own roof. A man needs to travel to places he doesn't know for breaking this arrogance that causes us to see the world as we imagine it, and not simply as it is or may be. That makes us teachers and doctors of what we have never seen, when we should just be learners, and simply go see it."

Amyr Klink

Acknowledgements

First, I would like to deeply thank Prof. Dr. Theo Härder, for the opportunity to work in his research group and realize this thesis under the Technical University of Kaiserslautern. Moreover, for giving me the opportunity of studying in a university with excellence and international prestige. The lessons I learned in this place opened doors in my life and contributed immensely to my growth, not only professionally but also personally. I also would like to thank all participants of the DBIS group, especially M. Sc. Caetano Sauer, which is co-advisor of this thesis, and was unbelievably helpful along this working process.

Next, I would like to thank my family: my parents Loreci Carlos Valer and Dirlei Teresinha Lamonatto Valer, for the respect and care with which they always treated me, and for the freedom of choices which was always given to me. An special acknowledgement to my father, which role model I always try to follow throughout my life.

Finally, to all others who I may have forgot to mention, who have directly or indirectly helped me through this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

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 not both scalable and 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 [40] 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 capabilities. 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 [54], Pig [49], and JAQL [18].

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.

1.2 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 [52]. 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 thesis: 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 *key/value* model is the simplest of these three. It stores data by associating a key to a value, which is simply an uninterpreted array of bytes. Put, get, and delete operations constitute the basic API of the model, and they work on a single key/value pair at a time. The store we chose to represent this family on our work is Riak [45]. Riak is a distributed key/value database with strong fault-tolerance regards inspired by the Dynamo system [28].

While Riak does not interpret values, the *column-based* model introduces a *tabular* structure to the model—something similar to columns in RDBMS's. Adding columns is quite inexpensive and is done on a row-by-row basis. It maintains a global ordering on the keys, resembling even more a relational table. The implementation we chose is HBase [33], based on BigTable [24]. Besides the intrinsic scalability, it provides built-in support for versioning and compression.

The *document-based* model stores documents in a key/document fashion. Documents are ordered sets of keys with associated values, and can contain nested structures, like other documents. The implementation we have chosen is MongoDB [25]. It is a document database that enforces no schema and allows for ad-hoc queries, among other built-in capabilities, like indexing.

The remainder of this thesis is organized as follows. Section 2 introduces the NoSQL model and its characteristics. It also details the CAP theorem, and describes technical details of the three NoSQL implementations used: Riak, HBase, and MongoDB. Section 3 explains both MapReduce and XQuery in more details. It also describes the used XQuery engine, Brackit, and the execution environment of XQuery on top of the MapReduce model. Section 4 describes the main contributions of this thesis: 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 thesis and discusses future work.

Chapter 2

Key/Value Storage: NoSQL

2.1 Introduction

For years, Relational Database Management Systems have been *de-facto* option for data storage, and this scenario will probably endure for more years to come. Nevertheless, more and more people are starting to discover alternative storage options. These alternatives trade off well-known *ACID* properties for higher availability or scalability by using techniques such as eventual consistency, horizontal scalability, efficient replication, and schema-less data models.

These set of characteristics are essentially known as *NoSQL*. The user accesses a low-level record-at-a-time API instead of any SQL—regardless of the genre of the store: key/value, columnar, or document-oriented—therefore minting the therm "NoSQL".

Advantages of the approach lie usually in performance—conflicting with scalability problems of RDBMS—and flexibility—conflicting with rigid relational schemas of RDBMS. The question of whether or not to use these NoSQL systems is not for us to answer, but in that regard, several different aspects should be taken into consideration. For example, that of the flexibility of the system: for flexibility on the data side, NoSQL is a better fit. It provides schema-less data structures, for situations where data relationships are not needed. On the other hand, for systems desiring flexibility on the query side, SQL suits better because of its great expressive power, and ad-hoc queries.

How these systems achieve high availability is somehow different from relational approaches. When RDBMS's face scalability challenges, they usually follow some traditional solutions: increase of cache, memory, and processors—scaling up vertically. Schema denormalization, pre-materialization of often-issued costly queries, dropping of secondary indexes, until eventually sharding data. On the other hand, NoSQL systems were designed to support scalability, therefore not degrading performance while doing it.

Fault-tolerance is another great characteristic of these systems, especially because they claim to work atop commodity hardware. Therefore, data replication—the act of storing the same data in multiple places—plays an essential role. We will see how these systems implement replication by choosing between consistency or availability.

This work focuses on three different types of NoSQL stores, namely *key/value*, *columnar*, and *document-oriented*, represented by Riak, HBase, and MongoDB, respectively. We make the differences between them explicit over the next sections. We discuss how each of them handles data, their overall architecture, how they provide for scalability, and how they achieve fault-tolerance, among other characteristics.

Before exploiting details of each implementation we better define *consistency*, *availability*, and *partition tolerance*. The so-called *CAP* theorem states that it is impossible for a distributed computer system to provide consistency, availability, and partition tolerance at the same time. In the next section we show the reasons for leaving the benefits of the ACID model behind.

2.2 From RDBMS to a broken ACID

The relational model is the one thought of when comes to data stores, regardless of people's acquaintance with databases. Writing queries using Structured Query Language (SQL) is the usual interaction, like the one present in Figure 2.1. RDBMS's statically type data values —therefore every column has a data type. Moreover, data may be of various sorts, like numeric, strings, date, among others.

```
SELECT id, type, details FROM contacts
UNION
SELECT 2, 'web-page', 'http://www.brackit.org'
```

Figure 2.1: Example of SQL query.

ID	Type	Details
1	e-mail	valer@cs.uni-kl-de
2	web-page	http://www.brackit.org

Table 2.1: Result of the SQL query.

Normalization makes data management easier by organizing data into small tables, which are related to each other. This process isolates data—allowing for modifications to be executed on a single point—and then propagates it to related tables, thus reducing redundancy and data dependency. On the other hand, relational databases suffer from the well-known impedance mismatch, introduced when storing objects from any object-oriented programming language. Alternatives for solving this problem, such as object oriented databases, or a whole technology stack based on XML—for example, XForms on the client side, XML databases on the storage level, and XQuery as query language—are beyond the scope of this work, and for that we relate to [51] and [55] respectively.

Two other characteristics are cornerstones for the power of the relational model: first, its mathematical background. RDBMS are built on top of set-theory called *relational algebra*, which essentially specify how RDBMS behave. Relational algebra is built on top of selections (*WHERE*), projections (*SELECT*), cartesian products (*JOIN*), set union (*GROUP BY*), set difference, and rename. Second, a whole plethora of facilities such as efficient join operations, indexes, aggregate functions, grouping operations, window functions, stored procedures, triggers, views, rewrite rules, string matching techniques ("like" and regular expressions), full-text searches, multidimensional data, among others.

Nevertheless, even with all above mentioned advantages of RDBMS, some cases still flee its jurisdiction. Whenever data requirements involve flexibility, like in XML, rigid schema requirements of RDBMS are not the best fit. Besides that, very high-volume reads and writes might not be satisfied using RDBMS.

Moreover, scalability and availability of RDBMS's is rather problematic. The CAP theorem constraints the former. RDBMS follow the so-called ACID properties—Atomicity, Consistency, Isolation and Durability—but on today's highly distributed world, where systems became distributed and need partition tolerance, either availability or consistency need to be sacrificed. The latter is cumbersome because the system is centralized. Moreover, when in a crash scenario, recovery mechanisms make the system unavailable for a long time.

2.3 CAP

The CAP theorem states that any network shared-data system can have at most two of three desirable properties: consistency, high availability, and tolerance to network partitions [20]. We will shortly define these terms, and see how it can be proved that achieving the three of them at the same time is a contradiction.

Consistent objects are objects on which every operation is atomic, therefore the intermediate state of operations is not visible to current transactions. Under this guarantee, a total order of all operations should exist, such that each operation is completed at a single instant, following the “all or nothing” guarantee. In a distributed system, this is equivalent to requiring that distributed requests act as if they were executed in a single node.

Availability dictates that objects must be available at all time. There are no constraints in the model regarding how much time the answer for a request must take, it only states that every request must terminate.

Network partition is defined as: given two partitions A and B of a network, such network is considered partitioned if all messages from A to B—or from B to A—are lost.

Given the above definitions, the work in [36] proves the impossibility to achieve availability and consistency when there are partitions in the network. Their proofs are for different network models, asynchronous or partially synchronous, with or without losing messages.

In asynchronous networks with message losses, the existence of an algorithm that meets the three CAP criteria is assumed and a contradiction strategy is developed. They assume a network of at least two nodes, which is divided into two disjoint, non-empty sets G1,G2. Assuming that (i) all messages between G1 and G2 are lost, then (ii) data is updated in G1, followed by (iii) a read operation over the same data in G2. The read step (iii) cannot return the result of the previous update operation (ii), therefore contradicting the existence of an algorithm that meets all three CAP criteria.

In asynchronous networks without message losses, the first part of the proof plus the fact that there is no way to determine if an asynchronous message was lost or simply delayed is used. Therefore, if there was an algorithm that guarantee atomic consistency without message loss, there would also exist one with message loss, which contradicts the first proof.

In the partially synchronous model, every node has an internal clock, which is not guaranteed to be synchronous with every other node’s clock. In this scenario, it is still impossible to achieve CAP criteria, regardless of message loss. The proof follows the already discussed example of asynchronous networks with message lost. Assume the same disjoint, non-empty sets G1, G2. The first update request, followed by an acknowledgement happens in G1. G1 then communicates with G2 but, again, all messages to G2 are lost. G2 follows, after some reasonable long time (enough for the write/acknowledge cycle of G1 to complete) with a read, that is promptly answered based on the availability condition. Finally, by superimposing both executions, we still have inconsistent data. For the case where messages are not lost, the example is still valid, and the only necessary change is to intercalate the requests before the communication between G1 and

G2 happens, therefore contradicting atomicity. For more details about the theorem's complete proof, we refer to [36].

It is possible though, to achieve any two of the three properties, and we will see with more details how each system deals with this situation. The next section starts this discussion with Riak.

2.4 Riak

Riak is a distributed key/value database with strong fault-tolerance concerns. It provides load balancing features for scalability, and data replication for high availability. It is inspired by the Dynamo system [28], and therefore it is a "web-driven" database: queries are made using URLs through HTTP get/set requests, and Riak has the potential to answer many requests with low latency.

On the other hand, Riak pays the price for its flexibility. By design it presents problems when referencing values together, in the relational sense of it, and does not present much support for ad-hoc queries.

Data Schema

Riak is a pure key/value store, therefore 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.

Physical Architecture

Riak is a part of a bigger technology stack. The overall architecture of the system can be seen in Figure 2.2, and we will go through its details in the following discussion.

Riak Core—depicted in Figure 2.2—represents a decentralized key/value store. It is an application that hosts services to build any number of distributed applications. It provides node tracking capabilities, as well as membership, and monitoring. Besides, it

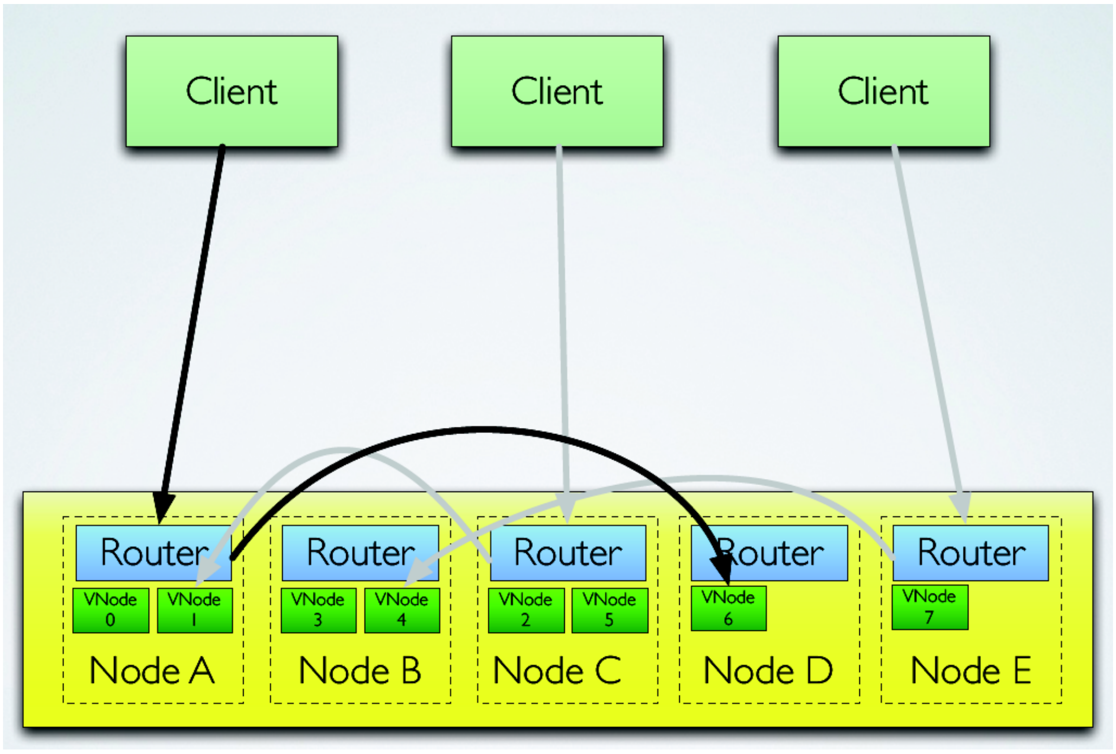


FIGURE 2.2: Riak Core architecture.

also provides partitioning and distribution of work among nodes, on a master-worker configuration that manages the execution of working units.

Riak Core essentially routes commands to nodes. It adds another level of indirection between clients and nodes, using a consistent hashing mechanism, which is better detailed in Section 2.4. Riak places the hashing structure (router) within each node of the system.

Riak also provides other services. *Riak KV* is a local key/value store. *Riak Search* is a distributed full-text search engine. Finally, *Riak Pipe* is a data processing for Map/Reduce capabilities. These components all work atop the Riak Core.

The storage components are place underneath Riak Core: *Bitcask*, *LevelDB* [34], or *Merge Index*. Riak is a distributed key-value store based on single-node key/value stores, therefore how key/value pairs are persisted into disk is not really relevant. We will discuss why Bitcask was implemented in Section 2.4, given the fact that other key/value stores already existed. One of these stores is actually made available for usage within Riak: LevelDB. It is an open source project started by Google, which addresses size limitations present in Bitcask.

On top of these core components, Riak also enables a whole plethora of services: replication and monitoring tools in a low-level API, as well as a Rest Toolkit, and *Protocol*

Buffers—the leverage of Google's buffer-serialization framework. Finally, on top of all this technology stack, various client APIs are available.

On the following sections we introduce the single-node key/value stores, namely Bitcask and LevelDB.

Bitcask

Low-level storage components need characteristics like low latency, high throughput, external memory support for Big Data, backup and restore facilities, among others. Several key/value stores fit this description, like Berkeley DB [48], or Tokyo Cabinet [42]. Nevertheless, Riak developers decided to build their own key/value store, motivated by the belief that they could do better than LSM-trees.

The model proposed for Bitcask is rather simple: a directory that cannot be accessed in parallel. At any given time, the directory—which can be seen as a "database server"—has only one active file, with a maximum size s. When the file reaches this threshold, it is closed and a new file is created. Once a file is closed, it becomes permanent, and is never overwritten again. Files are only written with append operations. Deletions are special write cases, as well as updates. For unused data, a cleaning process exists. It scans and removes old unused data from not-active files, thus improving space consumption.

A mapping of *key* $\rightarrow$ *value* is maintained in main memory, thus allowing for fast access of values. The whole performance gain comes from such structure.

This rather simple system works for limited sized datasets, and that comes with the predictable problem of the size of the mapping maintained in main memory. The system does not support a mapping that grows bigger than the available RAM, and that tends to be the case for applications working in the realms of Big Data. Because the structure of the mapping is not freely available, we cannot exemplify how much data is still comported for some given RAM parameters, but it is clear that such mechanism does not scale horizontally after the RAM threshold is crossed. This also explains the fact that LevelDB is also available as storage facility for the Riak key/value store.

LevelDB

The overall description of LevelDB is very similar to Bitcask: LevelDB is a persistent key/value store, where keys and values are arbitrary-long byte arrays, and keys are orderly-stored. But when it comes to details, LevelDB solves Bitcask's limitations.

LevelDB is an open source implementation of BigTable [24], and we will shortly discuss its layout. Three modules compose LevelDB: *SSTables*, *LSM-tree* and *caching*. SSTables (Sorted String Tables) are exactly what the acronym describes: each SSTable is a file with sorted key/value pairs. LevelDB streams data into main memory up to a threshold s. Once this limit is reached, it flushes data to disk, therefore minimizing the chance that reads and writes dominate run-time. The cache structure is called *MemTable*, and writes are always done primarily in main memory.

The flushing part essentially creates another SSTable: every SSTable has its own index, called *SS index*, which is also kept in main memory. An overview of the structure can be seen in Figure 2.3.

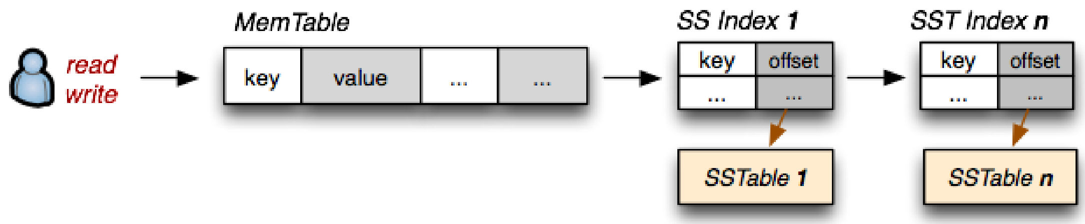


FIGURE 2.3: LevelDB scheme.

The random *reads* are solved by default, because we have the mappings in main memory. Data access for the worst case scenario needs two look-ups to main memory—essentially non-expensive—and one to disk. Random *writes* are the villain of the whole key/value story, because fast read/write access for petabyte datasets are not manageable within main memory. The MemTable structure in main memory provides the solution: writes are always orderly appended to such in-memory structure, and, once written to disk, they become immutable. Indexing structures are then updated in main memory, therefore random writes are effective. The process we just describe is essentially implemented by the LSM-tree structure [50]. The comparison between it and another well-known storage structure, the B+tree, is presented in Section 2.5.

Scalability

The *Riak Ring* is essentially how Riak deals with partition tolerance: it uses multiple logical partitions (virtual nodes—vnodes), each of which storing data. It divides partitions among nodes—physical nodes—and interprets each partition as a virtual node. On top of this architecture, Riak handles a so-called Riak Ring: it divides data among up to 2^{160} positions, represented by a 160-bit number, and interprets these positions as if placed in a circle, thus giving origin to the term *ring*.

Considering the default configuration of Riak, with 64 partitions, and supposing we are dealing with three nodes (A,B, and C), the ring helps discovering which node—more precisely, which virtual node—contains the value of a hashed key. The scheme works essentially by building partitions by hashing, and distributing them using round-robin. Each virtual node represents an interval of hashed keys. At boot-time, each physical node claims partitions in turns around the ring, thus dividing the partitions per node accordingly. Problems such as load balancing are not present, because by hashing the keys we guarantee an even distribution of values per virtual node. To get a value, the system simply hashes the key and searches the ring to find in which virtual node it is located.

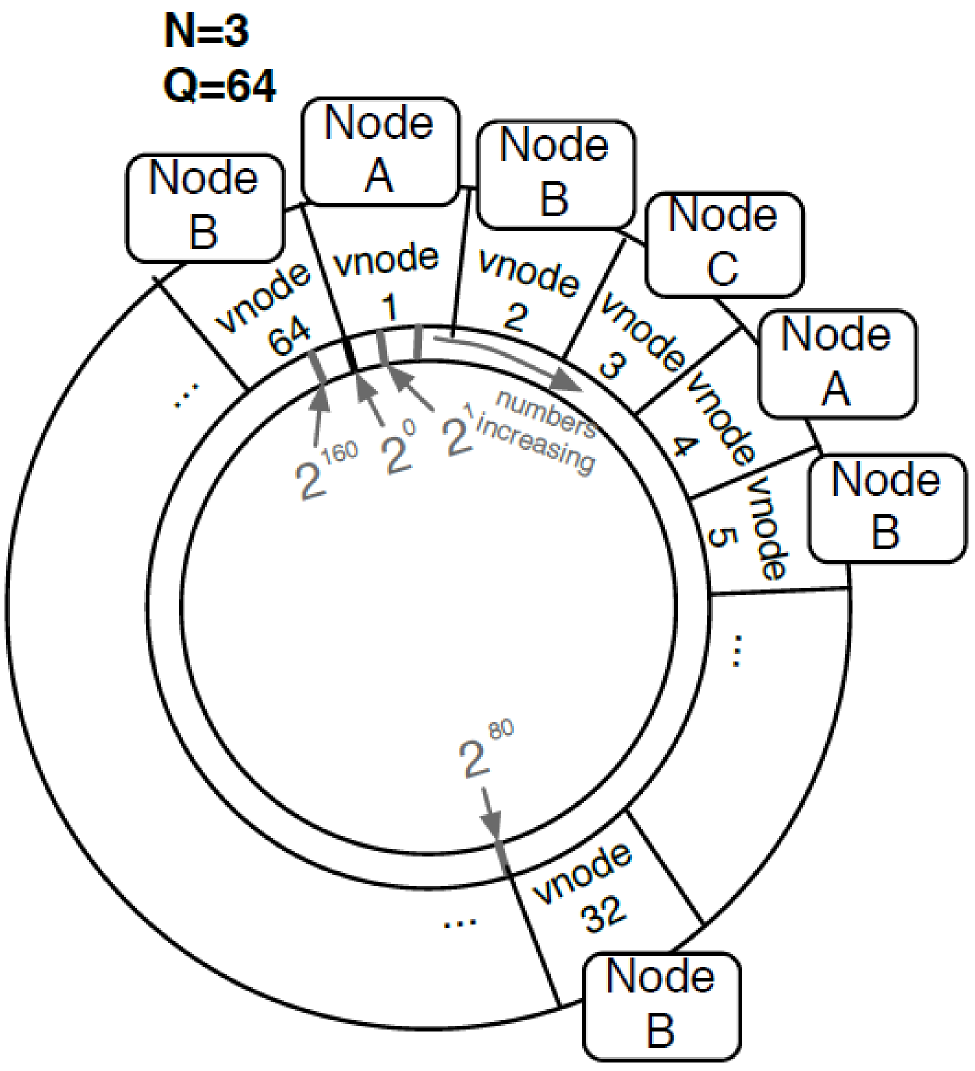


FIGURE 2.4: The Riak Ring [45].

A visual representation of the ring can be seen in Figure 2.4. The figure shows the exemplified configuration with 64 virtual nodes (Q), three physical nodes (N), and the virtual-node distribution per nodes. We can see that 64 virtual nodes were divided among the three physical nodes in a round-robin fashion, and that data is distributed—ranging from 2^0 to 2^{160}—within each virtual node.

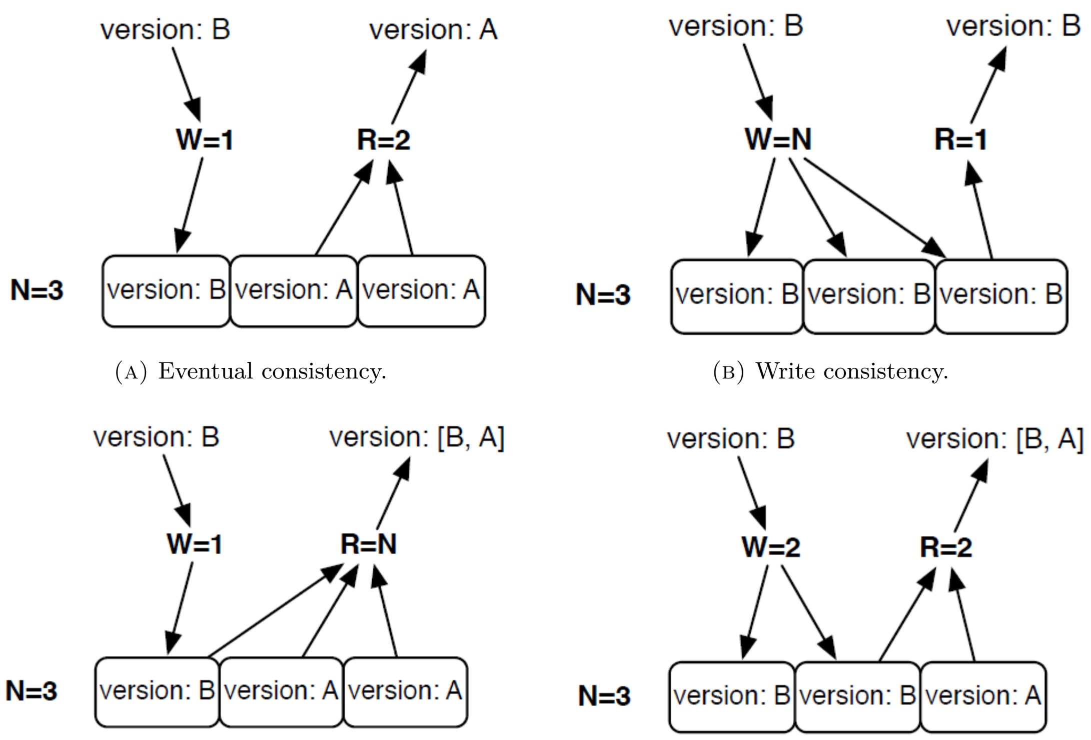
Riak uses a *gossip protocol* in order to keep consistency. It communicates ring-state information and bucket properties around the ring. Whenever a node changes its references on the ring, it "gossips" this information around, thus maintaining relaxed-consistency requirements among a very-large group of nodes. A gossip protocol is simple in concept: each nodes sends out some data to a set of other nodes. Data propagates through the system node-by-node, like a rumor. Eventually, data propagates to every node in the system, therefore providing *eventual consistency.* For more details about gossip protocols we refer to [30].

Replication

When comes to satisfying the CAP theorem, Riak provides the most flexibility: it allows to trade off availability and consistency on a per-request basis. It achieves such behavior by allowing reads and writes with three different parameters: (N)odes, (W)rites, and (R)eads. N represents the number of nodes some data will be replicated on. W is the number of nodes that must be written successfully before a response is issued. R is the number of nodes that some data must be read from in order to reply a request.

The amount of flexibility that this mechanism allows is huge, but in order to see the most common case scenarios, we will shortly discuss some usual situations. The first one represents *Eventual Consistency*, and is depicted in Figure 2.5a. This example shows a scenario where the system is more responsive than consistent. Imagine the following: a user issues a write operation, changing data to "version: B". Riak writes data into one virtual node. Before the virtual node had had the chance to synchronize data, another user issues a read operation on the same data, with $R = 2$. Riak replies the read request using the other two virtual nodes of the system, thus using an inconsistent version of the data, "version: A".

We can use a mechanism to guarantee that data reads will always be up-to-date. We ensure strong consistency by setting $W = N$, and $R = 1$, as can be seen in Figure 2.5b. In this case Riak works as if it had a transparent locking mechanism, guarantying that whenever a write is issued, it will be consistently written to all virtual nodes, and only then, read requests will be answered. This configuration speeds-up reads, but slows-down write operations terribly.



(A) Eventual consistency. (B) Write consistency.

(C) Read consistency. (D) Quorum consistency.

FIGURE 2.5: Riak consistency options using N,W, and N [45].

In another scenario, we can speed-up writes and slow-down reads by writing to a single node at a time, and reading from all virtual nodes. We can achieve that using W = 1, and R=N, as can be seen in Figure 2.5c. The downside of this approach is how to determine which of the read values is the newest one. For that Riak uses *vector clocks*, and we will briefly discuss them later in Section 2.4.

Finally, there is also a quorum possibility, where W and R can be set to a majority of nodes, usually defined as N/2+1—in our example, 2. This approach is the minimum required for consistency, and somehow balances the replication between reads and writes, as can be seen in Figure 2.5d.

The approaches depicted in parts "b" and "c" still suffer from another problem: writes to all virtual nodes, where one of them is down, will not work. The same is valid for reading situations. While attempts to read from all nodes will miserably fail in these situations, Riak provides a mechanism to allow writing to all nodes even when some nodes are down. The mechanism is called *Hinted Handoff*, and essentially writes to a near-by node while the failed node is still unresponsive. More advanced mechanisms for load-balancing, or dealing with a cascading failure situation—node A fails, node B receives the load from A. Node B becomes a bottle neck and fails. Node C receives the load from both A and B, and eventually fails. And so on, and so forth—are not supported. Therefore, these techniques are usually error prone on the long run.

Vector Clocks

Vector Clocks is an algorithm for generating a partial ordering of events in a distributed system [47]. Riak uses vector-clock tokens to keep the ordering of key/value operations (write, update, and delete). Each operation is tagged with a vector-clock token—simply *token* from now on—which is composed of a client ID and a version number—a counter, incremented at sight, for each key. Riak chooses Vector Clocks because a "timestamp" solution would require perfectly synchronized clocks—which is in most of the cases an unfeasible requirement. Moreover, a "centralized clock" solution goes against the peer-to-peer nature of Riak, and also represents a single point of failure.

In order to understand how Vector Clocks works, let's think of an example:

> "Hypothetical friends Henrique, Caetano, Pedro, and Marcos are planning to meet and play some poker. The planning starts with Henrique suggesting that they meet on Wednesday. Later, Caetano discusses possibilities with Pedro, and they decide for Thursday. Afterwards, Marcos exchanges e-mails with Caetano, and they change it for Tuesday. When Henrique finally calls everyone to check for confirmation for Wednesday, he gets mixed responses:
>
> Pedro claims to have settled on Thursday with Caetano, while Marcos claims it on Tuesday also with Caetano. Caetano is out of town for some days and can't be reached, therefore none of Henrique, Pedro, or Marcos know for sure when the poker will take place."

The overall problem is: we have two informations, "poker on Thursday" and "poker on Tuesday", and no way to decide which one is the most recent. Vector Clocks solves this situation following a simple rule-of-thumb: "make sure to include each process ID and the last token available for a given value whenever to store a modification" [45]. Individual conflict solving is then used in conflicting situations. Let's check the example in more details, in order to understand how it works:

> "Henrique suggesting they meet on Wednesday"

Results in the following entry, sent from Henrique to all other participants:

```
date: Wednesday
token: Henrique:1
```

“Caetano discusses possibilities with Pedro, and they decide for Thursday”

Caetano then produces the following entry, sent only to Pedro:

```
date: Thursday
token: Henrique:1, Caetano:1
```

Caetano left Henrique’s mark on the token, and added his own message with counter 1, meaning “this is the first time I see this message”.

Pedro then agrees to Caetano’s previous message, generates the following entry, and sends back to Caetano:

```
date: Thursday
token: Henrique:1, Caetano:1, Pedro:1
```

“Marcos exchanges e-mails with Caetano, and they change it for Tuesday”

First, Marcos generates and sends to Caetano the following entry:

```
date: Tuesday
token: Henrique:1, Marcos:1
```

At this moment, Caetano has two conflicting tokens:

```
a. date: Thursday
   token: Henrique:1, Caetano:1, Pedro:1

b. date: Tuesday
   token: Henrique:1, Marcos:1
```

Tokens are in conflict when they do not descend from each other. A token *a* is descendant of another token *b* if, and only if, for every marker on *a*, *b* has its own marker with a value that is greater-than or equal to the one present in *a*. In this case, token *a* contains marker *Caetano*, but does not contain marker *Marcos*. Token *b* contains marker *Marcos*, but does not contain marke *Caetano*. Ergo, neither of them descend from each other. This provides a partial order on tokens.

Caetano now comes back from his trip, and has to resolve the conflicts. He than chooses the first token, because Thursday suits him better. Caetano creates a new token using the rule-of-thumb, thus including all markers already used, and sends back to Marcos.

```
        date: Thursday
        token: Henrique:1, Marcos:1, Caetano:2, Pedro:1
```

Now Marcos has two tokens, but we easily see that the token *d* is descendant of the token *c*, therefore being the most recent value.

```
        c. date: Tuesday
           token: Henrique:1, Marcos:1

        d. date: Thursday
           token: Henrique:1, Marcos:1, Caetano:2, Pedro:1
```

Riak's utilization of the Vector Clocks mechanism is quite simple: requests can contain a header "X-Riak-ClientId" that contains the node's marker, and another one "X-Riak-Vclock" that contains the token. The behavior of the system is just as depicted in the previous example, where for Riak the "poker day" represents a key, and the decision taken represents a value.

The biggest disadvantage of the Vector Clocks algorithm is that it suffers from a well know size problem: tokens grown indefinitely. Therefore, Riak allows for configuring the maximum size of a token, and the aging factor of tokens. Based on these parameters, Riak prunes tokens over time.

Links

Riak is a pure key/value store. It has stored keys, which are divided among *buckets*, and each key is used to access a value. The problem that arises with such description is that of relationships between values. Opposed to RDBMS foreign keys, key/value stores do not foresee support for relationships among data items, and Riak's base model Dynamo does not support any mechanism for that either. Riak enables the usage of a linking technique, thus enhancing Dynamo's model in this regard.

The underlying reason is functionality: even the simplest application might need linked data, and therefore a mechanism to support it was necessary. Riak provides *links*, which are metadata establishing one-way relationships between key/value pairs. Each link has also a user-specified *tag*. For example, when designing an e-commerce application with Riak, products could be linked with carts, while purchased by users, representing a shopping cart.

Together with links, Riak provides a *link-walking* query mechanism, which essentially traverses the links of a key/value pair. When traversing links, an optional parameter might be specified to narrow down the links that will be followed, based on the link's tag name. This is the mechanism that will help us applying optimization techniques, such as filter pushdown, when querying Riak with XQuery later on.

2.5 HBase

HBase implements a so-called *columnar store*, based on BigTable [24]. It is a distributed system geared towards read and write performance. It provides powerful built-in functionalities: besides the intrinsic scalability, it offers versioning, compression, garbage collection, and in-memory tables. Moreover, it allows for pushdown projections using a filter mechanism.

Regarding the CAP theorem, HBase has chosen consistency and partition-tolerance over availability. It makes strong consistency guarantees. For example, whenever a client writes a value, the next request will read the updated value, up to a point where HBase would refuse requests, because there are no more nodes available to provide the response.

On the other hand, it provides no declarative query language—just scanning APIs, and has limited support for transactions—rows are atomic and read-modify-write operation are available atop of them.

Data Architecture

HBase is a column-oriented database designed for fault tolerance, because node failures are quite common in large clusters—especially when using commodity hardware. It uses write-ahead logging, distributed configuration, and also takes advantage of complementary tools. It is built on top of Hadoop, a scalable computing platform with MapReduce capabilities.

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. An overview of HBase's data model can be seen in Figure 2.6.

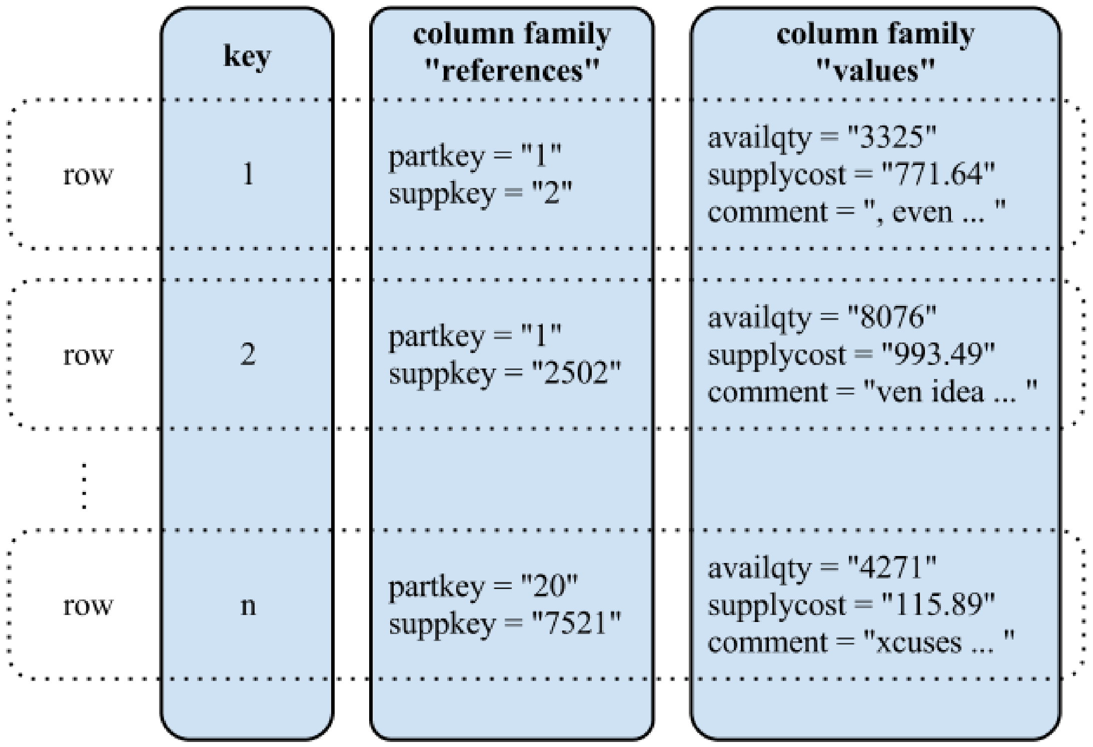


Figure 2.6: Architecture of an HBase table.

In this figure, we have a representation of the TPC-H [2] table *partsupp* mapped to an hypothetical representation within HBase. We created two column families: *references* and *values*. The table has several rows—denoted by dashed boxes—identified by their row keys: *1*, *2*, and so on. In this simplification, all rows present two qualifiers in the column family *references*, namely *partkey* and *suppkey*, and three qualifiers in column family *values*: *availqty*, *supplycost*, and *comment*. Note that the qualifiers are not defined at table creation time, in contrast to column families. Therefore, different rows in an HBase table can have different qualifiers within the same column family. The combination of row key, column family, and qualifier creates an address for locating values. In this example, the tuple {1/values:availqty} points us to the value *"3325"*.

Note that HBase does not use datatypes. All values are treated as uninterpreted arrays of bytes, thus showing no distinction between integers, strings, dates, and so on.

Physical Architecture

HBase is part of a larger infrastructure, and the general architecture of the system can be seen in Figure 2.7. The overall communication flow works as follows: first, clients contact the distributed coordination service, *ZooKeeper*, to localize data. ZooKeeper responds by pointing which physical node is storing the desired data. Second, clients request data from specific nodes of the system. Each physical node is called a *RegionServer*. As can be seen in the figure, each RegionServer accesses data regions with an object wrapper called *HRegion*. It maintains a *Store* instance for each column family on the data. Each Store instance has one or more wrappers—called *StoreFiles*—over each of the physical files. Finally, HDFS stores the physical files, called *HFiles*.

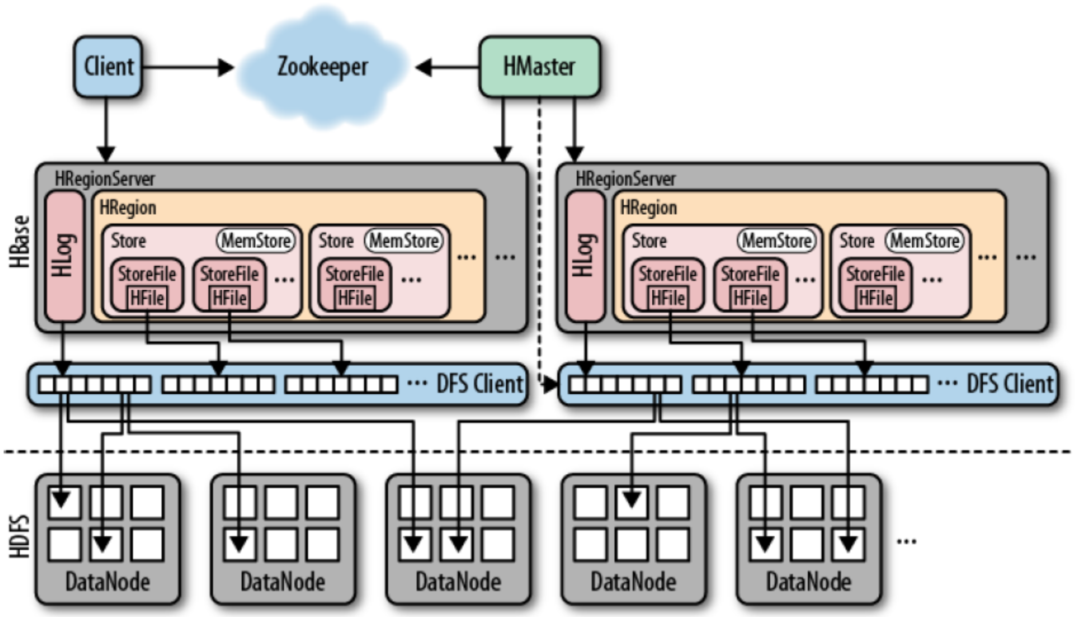


Figure 2.7: HBase architecture [33].

Lookups on RegionServers are based on two structures: the *ROOT* and the *META* catalogs. ROOT refers to all META regions, and the architectural design considers only one root. Therefore root regions are never split, and a B+tree lookup scheme with three levels is guaranteed: the first level contains the location of the ROOT catalog which, due to high usage, is always cached in main memory. The second level is a lookup on the ROOT catalog, also always cached, at least partially. And the third is the retrieval of the data-region location from the META catalog.

We can see that this scheme favors division of data based on column families: the less column families we inquire, the less physical HFiles will be accessed. It also allows for interesting scenarios: whenever using multiple column families, and less qualifiers, we create a columnar-based data access. On the other hand, with a single column family and multiple qualifiers, we provide a row-based data access. We explore this architectural design in our query engine later on.

Write-Ahead Log—WAL

HLog is the write-ahead log used. WAL is a logging mechanism shared by all regions of HBase. It is essentially a central logging facility, as can be seen in Figure 2.8. The overall process works as follows: first, clients issue data modification requests. These modifications are wrapped in key/value objects and sent to the nodes. Modifications are then routed to the proper data region by the node (RegionServer), and are also written to the log (WAL). The log uses HDFS as file system, so data is properly replicated [33]. Whenever the updates are persisted to disk, logs can then be thrown away.

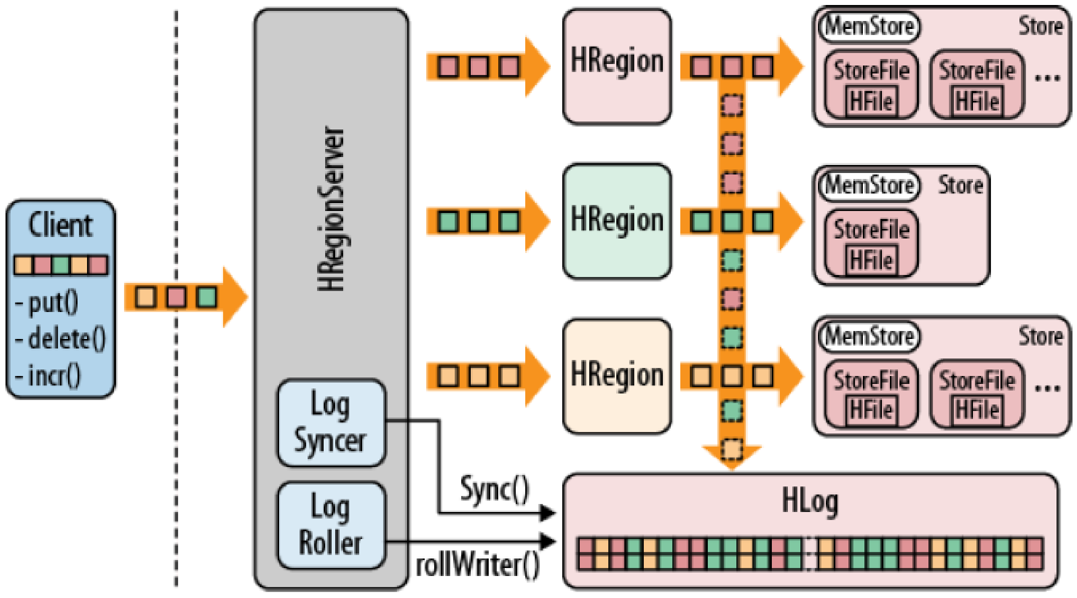


FIGURE 2.8: Write-Ahead Log [33].

A white-ahead log is needed for recovery purpose. RegionServers keep data in-memory until a threshold s is reached. When this limit is overpassed, writes are flushed from main memory to disk. The problem is that when working with distributed systems, with possibly many nodes, the chances of a single failure in one of the nodes are considerably high, therefore causing loss of any not yet persistent data.

HDFS

On the lower part of Figure 2.7, resides *HDFS*—Hadoop Distributed File System. It is based on [35] and designed for storing very-large files, with streaming capabilities. The definition of "large" is somehow complicated, but we will assume large as hundreds of megabytes, gigabytes, or terabytes in size.

HDFS is built on top of a “write-once, read-many-times” mechanism, and is aimed at high data throughput. Therefore, support for multiple file writers, or tons of small files, is not the main appeal of HDFS. What we see in the picture are several *DataNode* components, each one representing a node of the distributed file system. The white squares depicted within a “DataNode” are *blocks* .

As in any traditional file system, HDFS also applies the concept of block, but whereas in regular filesystems blocks are usually a few kilobytes big, in HDFS they are much bigger. The default size of a block is 64MB, and each block is stored as an independent unit. This default block size corroborates the fact that HDFS is not suited for small files. Nevertheless, in order to avoid the foreseeable waste of space in such cases, HDFS does not occupy a full block’s size when dealing with files smaller than the specified block size.

HDFS has two kinds of nodes, and they work in a master-worker pattern. One, depicted in Figure 2.7 is the *DataNode*—representing workers. The other one, omitted from the picture, is the *NameNode*—representing the master.

Workers store and retrieve information, and periodically propagate a list of stored blocks to the master. The master manages the file system tree, and file metadata. HDFS offers two distinct mechanisms to handle failures: either it stores metadata in multiple workers, or it executes a secondary master along with the primary. It just receives updates in an “up-and-running” state, in case something goes wrong with the master node. Both techniques have pros and cons, and for a better comparison we refer to [33].

The most recent versions of HDFS are deployed by default with a so called *HDFS-HA*, High Availability extension, that works accordingly to the second approach: two master nodes are executed by default, one as simple shadow of the primary. A shared log is maintained, so in the case of a failure in the master node, the shadow one assumes its place—but not before assuring its state is synchronized with the log.

B+ tree versus Log-Structured Merge Trees

HBase uses Log-Structured Merge Trees as storage structure, whereas most RDBMS’s use B+ trees. We will shortly elaborate on why. B+ tree is a type of tree that represents sorted data, allowing for efficient insertion, retrieval, and removal of records. The leafs of the tree store records, and internal node capacity is measured by a so-called “branching factor”, *b*. The number of children, say *m*, of a node has the following property:

$$b/2 <= m <= b \tag{2.1}$$

The root is an exception, where the number of children varies as:

$$1 <= m <= b \tag{2.2}$$

Leaf nodes store records in “buckets”, and follow a similar equation:

$$b/2 <= m <= b - 1 \tag{2.3}$$

Using these segments based on the branching factor, B+Trees achieve a much higher fanout when compared to other structures, like binary trees. The result is a much lower number of I/O operations when looking for keys, which is desirable for efficient lookups. The “b-1” upper capacity limit of leafs differs from intermediary nodes, on equation 2.3, because leaf nodes are linked and represent an in-order list of all keys, thus avoiding more costly tree traversals. A scheme of such B+tree can be seen in Figure 2.9.

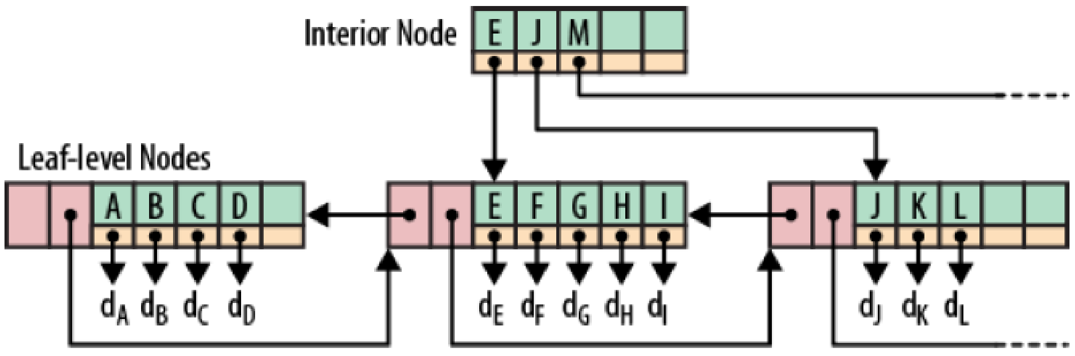


FIGURE 2.9: B+tree [33].

The B+tree depicted in Figure 2.9 shows an example of tree with branching factor 5. The first two fields in red represent the linked-leaf nodes. Intermediary nodes—like the one in the upper part of the image—don’t have them.

Operations on B+trees are relatively complex, and involve splitting or merging these node references, and updating the references on parent nodes. Such procedure must be executed until the tree is stable, because updating the reference list on parents might cause another split or merge. As a consequence, we cannot guaranty data locality when updates that re-arrange the three happen.

HBase uses Log-Structured Merge Trees (LSM), a disk-based data structure designed to provide low-cost indexing for files with high-record insert and update rates.

An LSM-tree is composed of two or more tree-like data structures. For clarity, we will first discuss the two-component variation (2LSM-tree). A 2LSM-tree has two three-like components: a *C0*, smaller and memory-resident, and a *C1*, bigger and disk-resident.

When rows are generated, the first step is a write ahead log insertion. Second, indexes are added to C0. Whenever C0 reaches a size threshold, a "rolling merge" process deletes some entries from C0 and merges them into C1 on disk.

C1's structure is similar to a B-tree, but optimized for sequential access. Its nodes are always full, and sequences of leaf nodes are packed together in contiguous multi-page disk blocks for efficient reads. This is optimal for HBase, because it provides only scan capabilities, and not any special method for punctual access.

HBase uses a similar approach, but instead of a two-component LSM-tree, each time the in-memory component reaches its threshold, a new Cn+1 file is created, named "store file". At store-file creation time, log updates can be thrown away, because data has became persistent.

Store files follow the C1 structure description, and as more and more store files are created, a background process aggregates them into a larger file, therefore disk seeks are limited to the number of store files. The process can be seen in Figure 2.10, and the result Ck-tree, depicted in the right hand side of the picture, is equivalent to the C1 tree structure of the original 2LSM-tree.

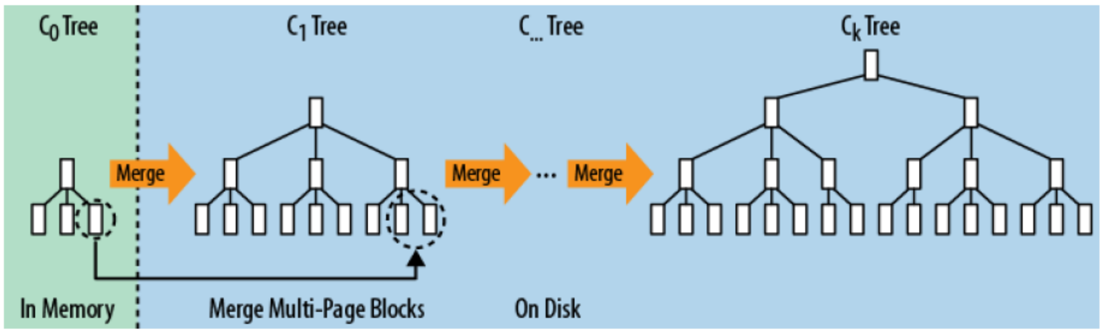


FIGURE 2.10: LSM-tree [33].

The option of HBase for LSM-tree is justified when comparing both structures. B+-trees deliver good performance until there are too many modifications, because the faster random writes are issued, the faster pages become fragmented, until eventually it takes longer to rewrite the existing structure for optimal access than the time-window we have to use the structure. LSM-trees, on the other hand, work at disk transfer rates, therefore scaling much better. And because data is always orderly merged, the number of disk seeks to access a key is also predictable. Moreover, reading data after a key access does not imply extra seeks.

Scalability

HBase provides built-in shardling capabilities. It splits records into horizontal partitions, across multiple files or servers. One of the biggest problems with shardling is *re-shardling*: when the used stores become so full that they must be re-configured, the re-writing operation is usually very costly. As any scalability issue, shardling is usually an afterthought, therefore being completely left to the operator on most RDBMS's implementations.

As we have seen, HBase works with data regions, which have a pre-configured size s. When too much data is inserted into a given table, and this size threshold is reached, an *auto-shardling* operation is triggered. First, it splits overloaded regions into two new regions—called *daughters*—each hosting half of the original region's content. The RegionServer configures the newly-created regions, and sends metadata information to the master node, who writes it into the META files. Afterwards, the master schedules for moving the daughters to other nodes, due to load balancing reasons: even though the new regions are fully functional, they are still physically stored within the same node.

Replication

On the upper part of Figure 2.7, there is a cloud component called *ZooKeeper*. ZooKeeper, as the name infers, coordinates the "zoo": it is the distributed coordination service. It provides a set of tools to handle partial failures.[1] The storage structure of ZooKeeper is a tree of nodes, called *znodes*. Data is atomic, znodes are referenced by absolute canonical paths—represented as slash-delimited Unicode character strings—and each znode stores an ACL list of permissions, besides data.

ZooKeeper achieves high availability through replication, and provides data as long as a majority of nodes are up. Thus, the task of ZooKeeper is to replicate data to a majority of upstanding znodes. However, implementing consensus is not that simple, and opposed to many expectations, ZooKeeper does not follow its predecessor's implementation. ZooKeeper is based on the Chubby Lock Service [21], but whereas Chubby uses Paxos as a consensus algorithm [46], ZooKeeper uses Zab [44]. Zab is similar to Paxos, and works in two distinguished phases: first, there's a master election, finished once a *leader* is chosen and synchronized with a majority of *followers*—the so called *quorum*. Second, during run-time, all write requests are forwarded to the leader, which broadcasts the update to the followers. Once a majority of nodes have persisted the information,

[1]Partial failures are situations in which nodes of the system don't even know for sure if a failure happened, like the classic dilemma: is the message taking too long to be delivered or is the receiver down?

the leader atomically commits the change. More details about consensus algorithms go beyond the scope of this work, and for that we refer to [38].

2.6 MongoDB

Introduction

MongoDB is a document-oriented database. It stores *collections* of *documents*. When compared to RDBMS, *tables* represent *collections*, and the concept of *row* is replaced with a more flexible unit: a *document*.

MongoDB provides built-in sharding and load-balancing capabilities, as well as many of the most useful features of relational databases, like secondary indexes, range queries, and sorting. It also provides replication and high availability using a master-slave mode, where failover is automatically handled by slave nodes in case of a master-node failure.

On the other hand, MongoDB still does not provide all relational capabilities. Server side joins are not supported, thus making clients responsible for it. Complex transactions—those between multiple documents—are also missing. MongoDB guarantees atomicity using a document as granularity.

Data Schema

A *document* 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.

MongoDB uses JSON [1] as data model for documents, but extends its model in regarding supported data types by adding types like: 32-bit integer, 64-bit integer, symbol, date, binary data, and embedded documents. This type system will be taken into consideration later on when querying MongoDB using XQuery.

MongoDB also supports indexing. It uses B-tree indexes, allows for compound-key indexes, and the general behavior is essentially the same as of any RDBMS's index capability. One example of document within MongoDB can be seen in Figure 2.11.

```
{
        "_id" : "some random UUID",
        "first" : "Hello"},
        "second" : "World!!!"
}
```

FIGURE 2.11: Example of a MongoDB document.

The representation of documents varies, but uses a data structure that is a natural fit, such as a map, hash, dictionary, or objects. The depicted example represents a simple document. All documents have a uniquely identified field, *_id*. The *first* and *second* are keys and they access “Hello” and “World!!!” respectively.

Physical Architecture

On a simplified scenario, a replicated MongoDB instance has at least two nodes: a *master* and a *slave*. The master handles requests, while the slave sporadically talks to the master and mirrors his actions on its copy of the data. The master node keeps track of operations through a log, the *Oplog*. This log keeps track only of operations that change the state of the database, like inserts and deletes, and not of queries. The log also works as a reference for slave nodes when recovering.

Operations are by default issued in a *fire-and-forget* manner, which essentially means that the client will not wait for a response from the server after issuing an operation. This sort of mechanism matches perfectly for applications like logging, or real-time analytics, for example when multiple updates occur every time a given web-paged is hit. The overall working architecture of MongoDB can be seen in Figure 2.12, and we will take a deeper look at its details.

Clients, depicted in the lowest part of the figure, make requests to MongoDB servers. Each MongoDB server is represented in the picture by a component called *mongos*. Each of these servers consults the configuration server and routes requests to specific virtual partitions—or *shards*. Sharding is a built-in capability in MongoDB, and we will go into details about it later. *Config* servers hold copies of the metadata, thus knowing where data resides. Each shard holds a portion of the data, and works in a master-slave fashion—called *Replica Sets*—better detailed later.

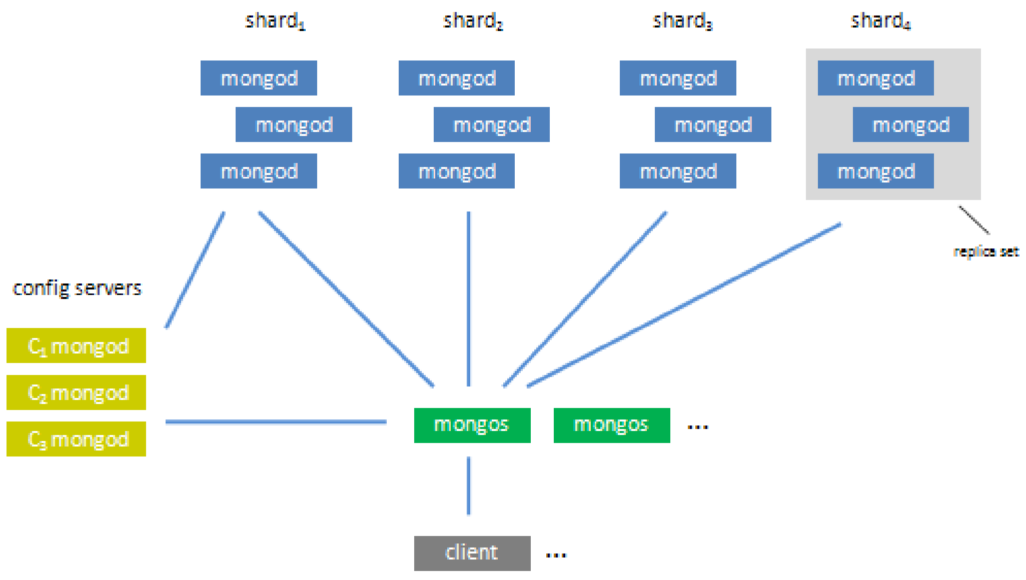


FIGURE 2.12: MongoDB architecture.

Storage Structure

MongoDB uses a memory-mapped engine. All of MongoDB's data files are mapped to main memory, and the responsibility for both (i) flushing data to disk, and (ii) paging mechanisms is completely advocated to the operating system. Each database has a single reference file, suffixed with the *.ns* extension, and monotonically increasing numeric-extent data files. For example, for the database *test*, the following files would exist: *test.ns*, *test.0*, *test.1* ... *test.n*.

MongoDB divides databases among *namespaces*, each of which storing an specific data type, and storing its metadata in the *.ns* configuration file. It groups data files for namespaces within *extents*, whose size doubles for each new file, up to a maximum size of 2GB. MongoDB preallocates these data files in the background, ensuring performance. Therefore, MongoDB will always try to keep an extra, empty data file for each database to avoid blocking on space allocation. An overview of this scheme can be seen in Figure 2.13.

What we see in this figure is a database called *foo* with several *namespaces*, namely: *test*, *bar*, and *baz*. Each namespace has several *extends*—not necessarily contiguous on disk. MongoDB uses a special namespace called *$freelist* to keep track of unused extents. The figure also shows preallocated-disk space filled with 0's.

In the 32-bit version, MongoDB implementation is limited to a practical total of 4GB of data per database, because all of the data must be addressable using only 32 bits. They

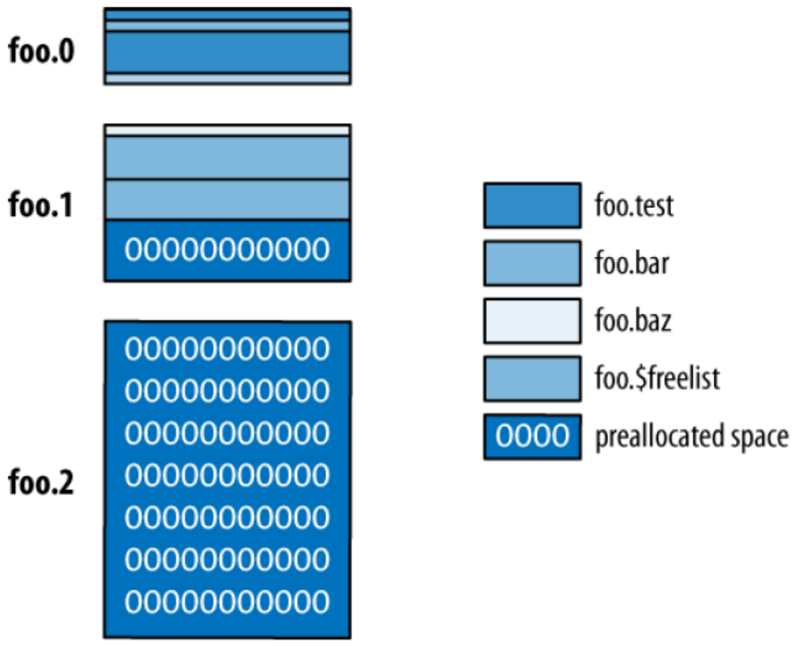


Figure 2.13: MongoDB storage structure.

authors claim this is due to code simplicity and easiness of development, but in practice this results in a huge limitation for all 32-bit clients.

Scalability

Auto-sharding is MongoDB's path to scaling out to huge data sets, and to achieve high-rate loads. Data is split among shards—represented on the upper part of Figure 2.12—and each shard is, therefore, responsible for a subset of the data. The "shard" node *per se* can be any implementation in a master-slave fashion, or using Replica Sets. The overall process is transparent for the client because the *mongo* server is still the one used for connection, regardless of where the data physically is.

The sharding process is based on *shard keys*. A shard key is a key from a collection, upon which the data will be split. In a collection representing people, we could use key *name* to split data, creating shard containing names staring with letters A-H, I-O, and P-Z. MongoDB automatically balances the load of data when shards are added or deleted. Some attention should be given to choosing a proper shard key, because we might create bottle-necks. Consider using timestamps as sharding keys: all inserts will be going to the same shard, therefore not taking advantage of the mechanism. In order to achieve uniform distribution of writes across shards, usually good candidates for sharding keys are hashed values.

Replication

MongoDB allows replication with an alternative for *master-slave* called *Replica Sets*. Essentially they work in the same way: a master node—in MongoDD called a *primary node*—is responsible for answering queries, and slave nodes—in MongoDB called *secondary nodes*—periodically update their references by reading logs from the primary node.

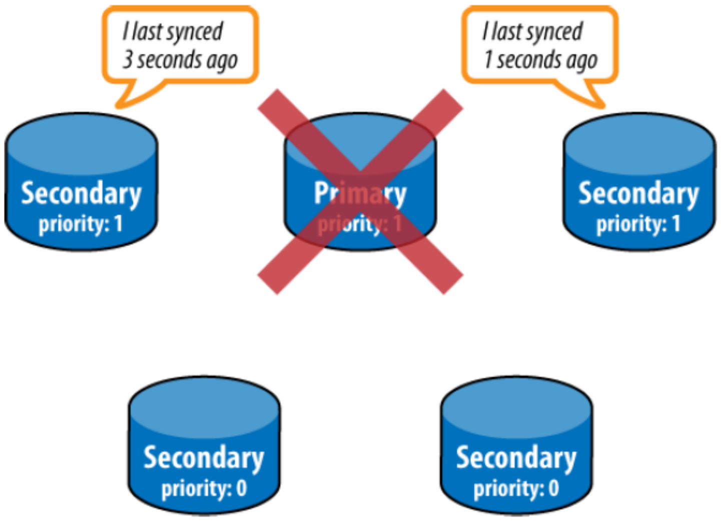


FIGURE 2.14: MongoDB's Replica Sets scheme.

Replica Sets enhances master-slave scheme with failover capabilities. Whenever a primary node is down, one of the secondary nodes is elected as new primary. The election works as follows: at any given time, a secondary node that cannot reach a primary can call for an election. Nodes in the system are classified by a priority scheme that ranges from 1 (high) to 0 (low). The new primary must be elected by a majority of secondaries, and in case of secondary nodes with the same priority, the most up-to-date node will became primary (based on an increasingly ordinal included on each log entrance). If the old master comes back to life, it will still act as a secondary node and update its data according to the new master's log. An overview of the election process can be seen in Figure 2.14, where the node in the upper right part of the picture, with the most up-to-date version of data was elected as new primary.

Besides been used for failover and data integrity, replicas can be used for scaling out reads and writes. When scaling out reads, secondary nodes will respond requests for reading data. Therefore it is important to notice that replication is asynchronous, and there is always a time interval between (i) a write request reaching the primary node, and (ii) the read request reaching a secondary node, where data will be inconsistent. When scaling out writes, secondary nodes will accept blocking operations without consulting the primary node. In this case, data replicated from the primary node will always take preference over locally written data, therefore updates to secondary nodes might be unused due to replication.

2.7 Summary

Riak is the simplest model we dealt with: it is a distributed key/value system that 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 when dealing with CAP limitations, 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.

HBase enhances Riak's data model by allowing columnar data, which works essentially as a map of maps. Terms are more close related to those of RDBMS, because well-known entities like tables, cells, rows, and columns are also present in HBase. What happens in comparison to RDBMS is a confusion of terms: tables in are not relational, rows do not act like records, column families are essentially namespaces, and qualifiers are the actual representation of relational columns. Data relations exist from column family to qualifiers, and operations are atomic on a per-row basis. When comes to CAP, 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, and 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. "Worker" instead of "slave" is justified because MongoDB allows to twist a little CAP's rules, and use worker-nodes to scale out reads and writes.

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 interrelated 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.

These stores are representative of the NoSQL movement. Almost all genres of available stores were used: besides the relational model—used throughout this work as comparison—we presented a key/value, a columnar, and a document-oriented.

The next chapter will go into details about the MapReduce model, how to use XQuery atop of it. Later on we will discuss how to take advantage of all the technical details of each store in order to query their data.

Chapter 3

XQuery over MapReduce

3.1 Introduction

After discussing how to scale the storage level, while being distributed and highly available, we still have to process huge amounts of data. In that regard, the problem is rather simple: speed access was not able to keep the pace with which storage capacity has increased. While a drive in the 90's could store 1,370 MB and had a transfer speed of 4.4 MB/s—scanning all data in about 5 minutes, todays drivers reach easily 1 terabyte of data, with transfer rates of 100MB/s—scanning all data in about 2 and a half hours. Moreover, CPU speeds also increased in a faster rate than memory speeds [14]. Memory latency become a bottleneck over the last decades and nowadays CPUs are capable of consuming more data than systems can delivered.

Parallelization is the idea, and in order to finish computation in a reasonable amount of time, we have distributed systems to read and compute data. *MapReduce* provides a linearly-scalable programming model for processing and generating large-data sets. It expresses computations with two functions: *map* and *reduce*. The former processes key/value pairs generating an intermediary set of key/value pairs. The latter aggregates intermediary results based on keys.

Hadoop [59] is the standard open-source implementation of the MapReduce model, which was firstly published in [27]. It automatically parallelizes and executes the model over a distributed system of nodes, thus hiding parallelization details, fault-tolerance, and distribution aspects from the programmer.

Despite being an important step towards introducing easy parallelization over Big Data problems, MapReduce still suffers from the lack of proper tools to enhance its querying capabilities. Moreover, when executed atop raw files—which don not follow any data structure—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 [54], Pig [49], and JAQL [18].

These approaches require learning separated query languages, each of which specifically made for the implementation. Besides, some of them require for schemas, like Hive and Pig, thus making them quite inflexible. On the other hand, there exists a standard that is flexible enough to handle all possible data types from different stores, whose compilation steps are directly mappable to distributed operations on MapReduce, and is been standardized for over a decade: *XQuery*. XQuery was designed by the W3C as the XML query language [57] and its capabilities vary from selecting, filtering, searching, joining, sorting, grouping, aggregating, updating, and scripting data within XML documents, to transforming and restructuring XML documents, as well as performing arithmetics or string manipulation functionalities.

Over the next sections we will see more details about the MapReduce model, about the XQuery standard, and a mapping of XQuery to the programming model of MapReduce.

3.2 MapReduce

Programming Model

MapReduce processes input key/value pairs and generates sets of output key/value pairs. It divides the computation in two functions: *map* and *reduce*. Map processes input key/value pairs generating intermediary sets of key/value pairs. MapReduce groups these intermediary results by key, *k*, and passes them to the reduce function. Reduce receives the key *k* and the set of values associated with such a key and merges these values, thus producing usually one or zero output values. The reduce function is usually employed to compute aggregation functions, like *sum*, *avg*, etc. Equations 3.1 and 3.2 show a mathematical representation of these functions.

$$map(k1, v1) \to list(k2, v2) \tag{3.1}$$

$$reduce(k2, list(v2)) \to list(v2) \tag{3.2}$$

Grouping takes place in between the map and reduce phases: *"The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function"*[27]. Therefore, a three-step chain of functions expresses the overall programming model of MapReduce: *map*, *group*, and *reduce*.

In order to make the model more clear, lets think of an example: given the data in Table 3.1 we want to find the highest temperature for each city over various measurement days.

ID	City	Temperature
1	Paris	20
2	Berlin	15
3	New York	17
4	Rome	2
5	Berlin	8
6	Rome	20
7	Berlin	9
8	New York	19
9	Berlin	17
10	Paris	13

Table 3.1: Temperature measurements per city.

The map function emits each key (city) plus an associated value (temperature), as can be seen in the left most part of Figure 3.1. The reduce function calculates the maximum temperature for a particular city, as can be seen in the right most part of the figure. The group function groups intermediary results based on a key (city), as can be seen in the middle part of the figure. Note that map and reduce are user-defined functions, applied to single key/value pairs, while group is a built-in function, applied over all input pairs at once.

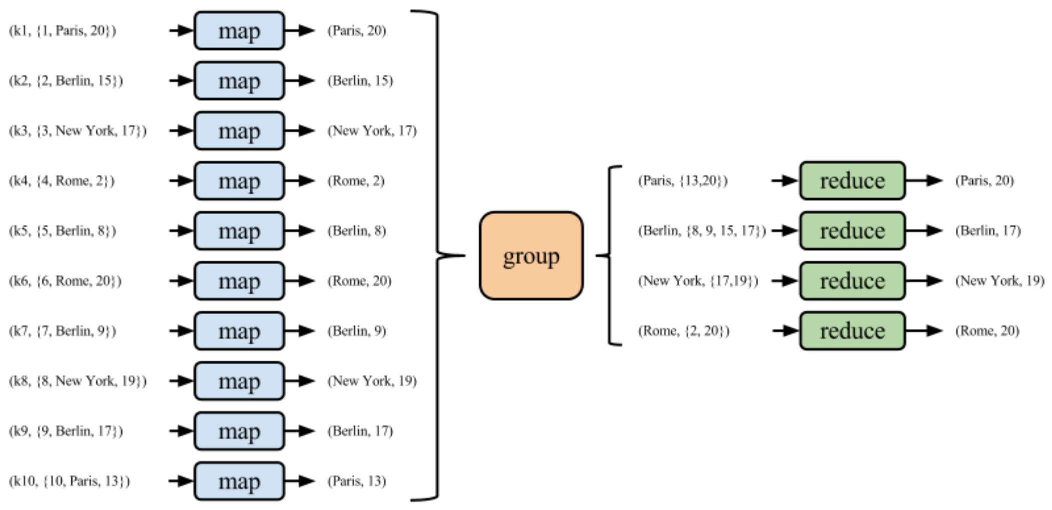


Figure 3.1: MapReduce computation model.

Execution Model

Nodes in a system distribute the execution of MapReduce. Both map and reduce invocations are distributed by partitioning data. The former partitions input data into parts, called *splits*. The latter partitions intermediary data. Nodes process partitioned data in parallel, regardless of being on the map or reduce phase. Figure 3.2 shows the overall execution flow of a MapReduce instance—usually referred to as *job*.

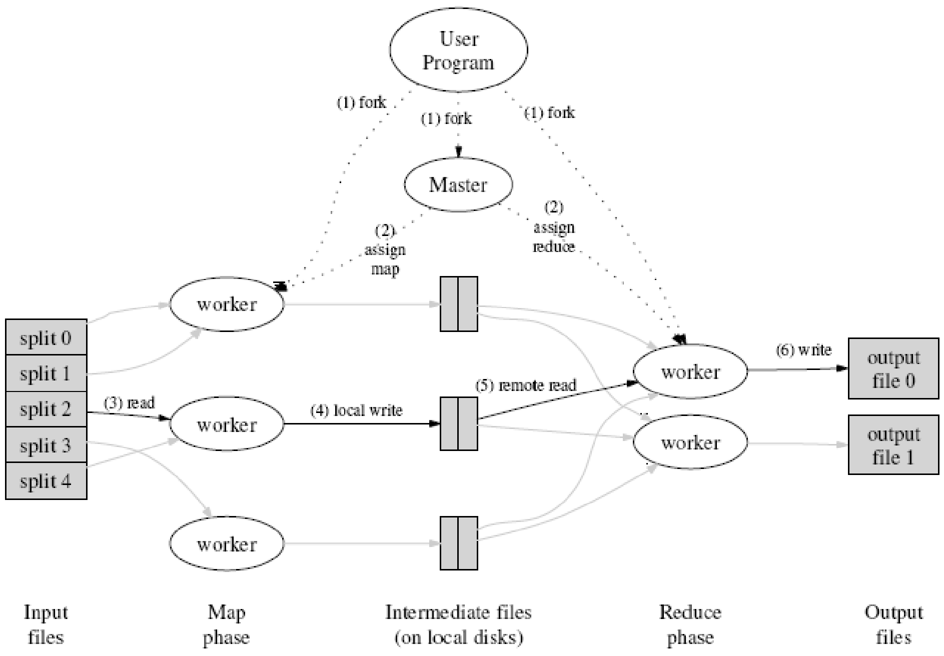


FIGURE 3.2: MapReduce execution model [27].

First, input data is split into n pieces, and different nodes in the distributed system execute several copies of the job (1). One special copy, called *master*, coordinates execution among idle workers by assigning map and reduce tasks (2). Each worker processing a map task reads the split content, parses its key/value pairs, executes the map function for each pair, and finally stores intermediate results in memory (3). Eventually, a node locally flushes data to disk, and informs the master node about the new-data location (4). Workers use RPC to read the intermediary data and, before actually executing the reduce job, they transparently group data, thus processing the middle step of previous Figure 3.1 (5). Reduce workers iterate over the sorted/merged-intermediary data applying the reduce function and appending each function's output to an output file (6). Finally, when all tasks are over, the master node finishes execution (7).

The above explained scenario directly maps to the discussed stores. Table 3.1 represents a bucket in Riak, a table in HBase, or a document in MongoDB. On Figure 3.2, splits represent Riak's partitions, HBase's regions, or MongoDB's shards.

The next section will introduce the XQuery language, allowing us to start thinking about how it can be leveraged to be executed atop MapReduce capabilities.

3.3 XQuery

Introduction

Over the past few years, the use of XML exploded. A considerably vast amount of information is represented using XML, and stored either on XML native databases or as documents in the filesystem. The design of XML emphasizes simplicity, and is defined on both human and machine-readable fashion. Moreover, given the boom in usage, several XML-based languages flowered, like XHTML, RSS, atom, and SOAP. The success of XML goes beyond the limits of academia, and the corporative world also uses XML format on commercial products, like Microsoft with the Office tools, Apple on several products, among others.

But how was XML adopted by so many so fast can be summarized to one word: flexibility. XML is a metamarkup language. The *markup* part indicates that data, in this case represented by strings of text, is surrounded by a markup language, very similar to what is available with HTML. As in HTML, the most basic unit of data is called an *element*. For how these markup tags are defined, delimited, acceptable names, and so on we recommend [41] as reference. More inportant than the markup part is the *meta* part. By been a metamarkup language, XML does not have a fixed set of tags and elements that are allowed, like keywords on a programming language. As a creativity exercise, writers and developers can create and invent as they please, therefore been as flexible as necessary. Moreover, it emphasizes the human readable aspect of it. For those against the anarchy of flexibility, XML allows for so-called "XML applications". These are not applications using XML, but rather sets of rules for XML tags and elements naming, for specific domains, like music or books.

On the downside, whenever we think about XML, creation and storage phases seam relatively simple, but what happens when data, present in an XML document, must be queried is where XQuery comes into the picture. XQuery was designed by the W3C as the XML query language [57] and its capabilities vary from selecting, filtering, searching, joining, sorting, grouping, and aggregating data within XML documents, to transforming and restructuring XML documents, as well as performing arithmetics or string manipulation functionalities.

Moreover, several extensions are either recently finished or still in progress, like Scripting Extensions [10], Update Facilities [9], among others, increasing even more the power of this language, that should not be seen as a simple query language, but as a universal programming language. XQuery can also be used as replacement for SQL in the database

```
<catalog>
  <book>
    <title>The Time of the Hero</title>
    <artist>Jorge Mario Pedro Vargas Llosa</artist>
    <country>Peru</country>
    <company>Hispania</company>
    <price>11.99</price>
    <year>1963</year>
  </book>
  <book>
    <title>Lord of the Flies</title>
    <artist>William Golding</artist>
    <country>United Kingdom</country>
    <company>Perigee Books</company>
    <price>9.99</price>
    <year>1954</year>
  </book>
  <book>
    <title>Waiting for Godot</title>
    <artist>Samuel Beckett</artist>
    <country>Irland</country>
    <company>Grove Press</company>
    <price>22.00</price>
    <year>1977</year>
  </book>
  <book>
    <title>One Hundred years of Solitude</title>
    <artist>Gabriel Garcia Marquez</artist>
    <country>Colombia</country>
    <company>Penguin</company>
    <price>10.10</price>
    <year>1967</year>
  </book>
</catalog>
```

Figure 3.3: XML book catalog example.

layer, and, after reading this thesis, the reader will be convinced it can also work as a better substitute for querying data over NoSQL databases.

Finally, one might still argue why not use SQL instead of XQuery. Even though SQL is a much more used querying language, and some NoSQL databases also provide a table-like environment for storing data, SQL cannot cope with schema-less data stored in NoSQL stores. Besides, there is a multitude of research showing that XQuery can be issued on top of relational databases, basically compiling a XQuery into a relational-query plan [29]. Therefore, playing on the safe side, XQuery is a better choice.

The next few sections talk about specific parts of the XQuery standard that are relevant to this work, such as: *FLWOR expressions*, *XDM*, and *path expressions.*

FLWOR expressions

FLWOR expressions—commonly pronounced "flower"—are the heart of XQuery. The acronym stands for *For*, *Let*, *Where*, *Order by*, and *Return.* It allows for manipulation, transformation, and sorting of data. Figure 3.4 shows a FLWOR expression returning all books published before 1964. The queried document is depicted on Figure 3.3.

Query:

for $book **in** doc(“catalog.xml”)/catalog/book
let $year := $book/year
where $year < 1964
order by $book/title
return $book/title

Returns:

```
<title>Lord of the Flies</title>
<title>The Time of the Hero</title>
```

FIGURE 3.4: Example of FLWOR expression

The *for* clause iterates over the elements returned by the path expression, in this case book elements. The *$book* variable is bound once to each book in the sequence, in this case four times. The rest of the FLWOR expression is evaluated once for each book. The *let* clause simply binds the *$year* variable to $book/year element. The *where* clause filters elements from the result, in this case reducing the number of results to two. The *order by* clause sorts elements by title. Finally, the *return* clause returns the filtered title of each book element.

Data Model

XQuery operates atop *XDM*: the XQuery Data Model. It is a logical structure, described in details in [58], based on *XML Schema.* It works atop *sequences* and *items.* A sequence is an ordered collection of zero or more items, and there is no distinction between an item and a singleton sequence containing that item. Sequences can contain nodes, atomic values, or any mixture of nodes and atomic values.

XDM is vast and complex, and defines all possible values of expressions in XQuery, including values used during intermediate calculations. Two distinct type hierarchies usually represent this type system, whose root elements are *item* and *xs:anyType.* Nevertheless, all primitives are derived from *xs:anyType*, something like the *Object* class for the Java language. Elements defined as being *xs:anyType* can contain data in any of the XDM primitives, as well as any other complex type defined in separated schema documents. The *xs:anySimpleType* is a base type for all simple types, like atomic, list, and union types. It works as a wildcard, therefore not adding any constraints to the lexical segment. The *xs:untyped* represents the dynamic type of an invalid element node. Figure 3.5 represents the first part of this hierarchy.

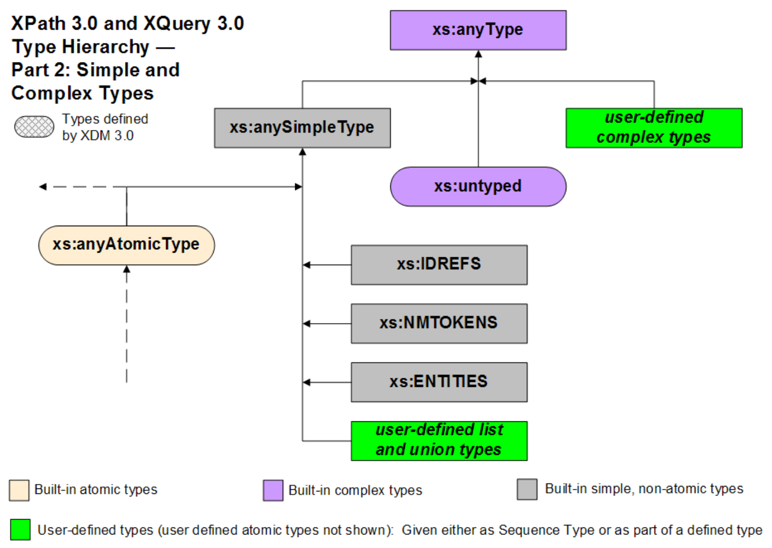

XPath 3.0 and XQuery 3.0 Type Hierarchy — Part 2: Simple and Complex Types
Types defined by XDM 3.0
xs:anyType
xs:anySimpleType
user-defined complex types
xs:untyped
xs:anyAtomicType
xs:IDREFS
xs:NMTOKENS
xs:ENTITIES
user-defined list and union types
Built-in atomic types
Built-in complex types
Built-in simple, non-atomic types
User-defined types (user defined atomic types not shown): Given either as Sequence Type or as part of a defined type

FIGURE 3.5: xs:anyType type system hierarchy.

XDM also has an hierarchy for *item*. An item is the abstraction of either *nodes*, *functions*, or *atomic types*. Nodes can be from types *attribute*, *comment*, *document*, *element*, *namespace*, *processing-instruction*, or text. Nodes have unique identity (not necessarily valued, but comparable), and are ordered relative to each other (document order). Serialization is done by a pre-order, left deph-first traversal of the tree.

Functions' main type—from which they all derive—is *function(*)*. As any other data type, functions have subtype relationships, and follow the *SequenceType Subtype Relationships* definition present on [7]. Atomic types represent common data types used on the XQuery language, such as strings, data, numbers, and times. This set of types is called *built-in types*. Among built-in types, nineteen of them are *primitive types*, and have individual value spaces, describing valid values, as well as a set of lexical representations for each value in the value space. The rest of the built-in types are derived either by restriction or extension of any of these primitive types. For more details about it, we refer to [58]. Figure 3.6 shows a demonstration of the available primitive types, as well as their hierarchical organization.

Path Expressions

XQuery uses XPath expressions to navigate and select nodes within XML documents. Path expressions are made of possibly many steps—separated by single or double slashes. Each expression is evaluated relative to a particular context node—serving as start point to the relative path—and produces a sequence, used as context to evaluate following steps. Figure 3.4 shows an example of XPath expression: *doc("catalog.xml")/catalog/book*.

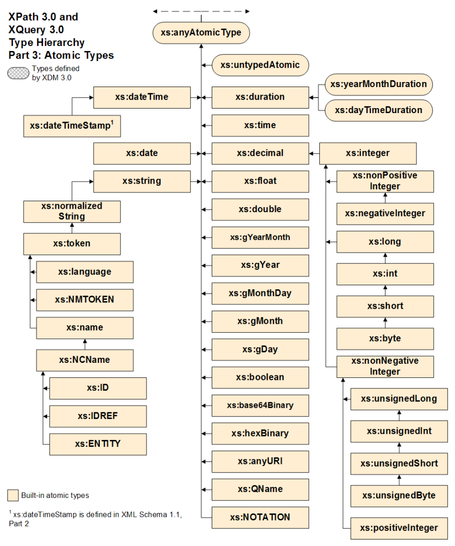


Figure 3.6: xs:anyAtomicType type system hierarchy.

Each step consists of an *axis*, a *node test*, and zero or more *predicates*. Axis specify navigation direction and relationships within the XML tree. They can be of any of thirteen types: *ancestor*, *ancestor-or-self*, *attribute*, *child*, *descendant*, *descendant-or-self*, *following*, *following-sibling*, *namespace*, *parent*, *preceding*, *preceding-sibling*, or *self*. Their behavior is really straight-forward, and a better visualization can be seen in Figure 3.7. Each *node test* indicates which of the nodes to select along the specified axis. The specification can be by name or more general expressions, like "*" for any node name. Finally, optional *predicates* can be used to further filter expressions.

The next section introduces Brackit: an open source XQuery processor that implements the concepts we have been discussing up until now.

3.4 Query Processor: Brackit

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. Examples of the former are Saxon [4]—a utility written in Java that performs queries on documents or collections without an XML database; Zorba [12]—a general purpose XQuery processor written in C++ providing full W3C specifications: *XPath*, *XQuery*, *Update*, *Scripting*, *Full-Text*, *XSLT*, *XQueryX*, among others. Zorba also provides a store API, allowing to process XML stored in different locations, but originally it ships with a main memory based storage, so similar to the Saxon's approach. On the other hand, examples of native XML databases are XBird [8]—a light-weight XQuery processor and database system written in Java; ExistDB [5]—an open source

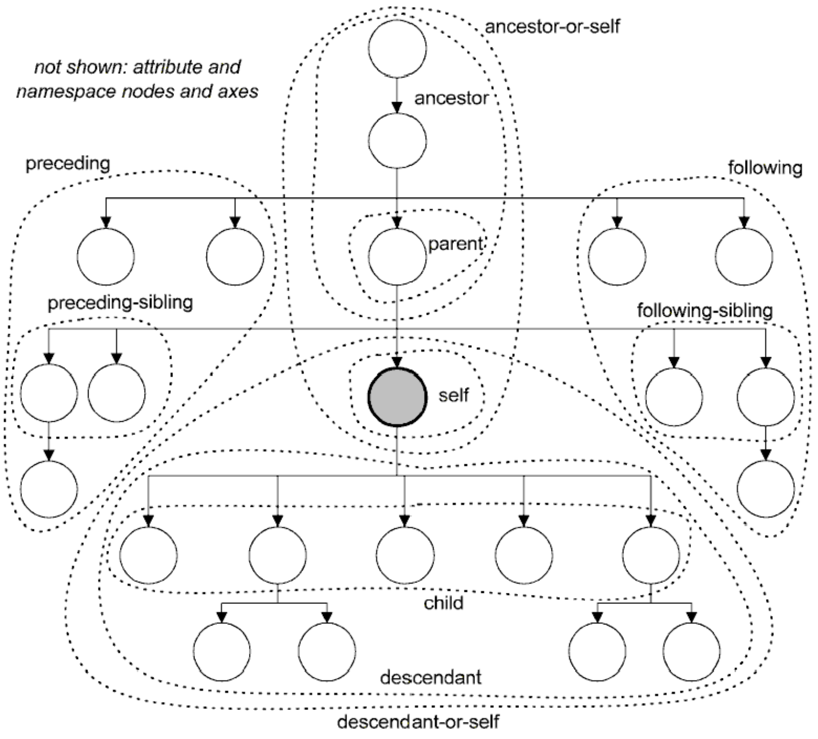
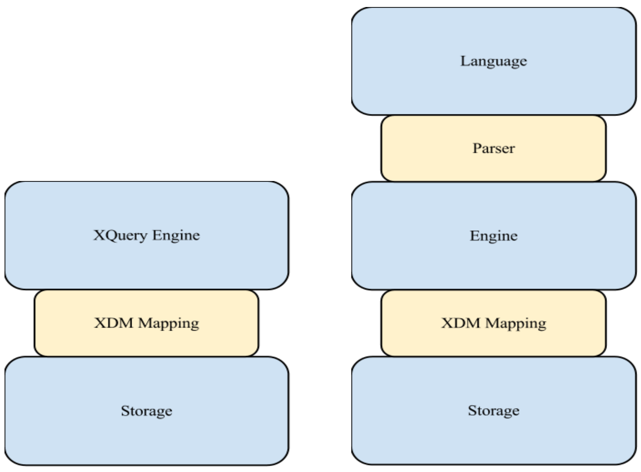


FIGURE 3.7: XPath axis hierarchy.

XML database management system that also supports W3C standards like XQUery, XPath, and XSLT; and MarkLogic[3]—which provides a database, search engine, and application server.

Usually, light-weight XQuery processors lack any sort of storage facility. On the other hand, native XML storage systems are just not flexible enough, because of the built-in storage layer, therefore not being suited for the purpose of this thesis. We decided to use Brackit [11]. It provides intrinsic flexibility, allowing for different storage levels to be “plugged in”, without lacking the necessary performance when dealing with XML documents [16].

Figure 3.8 depicts the flexibility of Brackit’s architecture. We require modularity in the sense that the XQuery engine needs to be detached from the storage, as can be seen in the first part of the figure. The related approaches—which intrinsically integrate the XQuery engine to a XML database—do not match our requirements. On the other hand, some approaches that deliver only the upper part of the figure—a XQuery engine without any possible connection or API to a database—would be just as unfit as the just related approaches.



(A) Needed engine structure.

(B) Brackit Engine modular structure.

FIGURE 3.8: Comparison between needed modular sctructure and the one offered by Brackit.

What we can see on the second part of the picture is how Brackit handles and provides flexibility and modularity. 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. The *engine*, depicted on the right part of the picture, realizes our access optimizations, such as filters and predicates push-down. The *XDM mappings* are the main contribution of this thesis: a mapping from NoSQL data model to XDM, thus perfectly fitting Brackit's architecture. Moreover, this layout allows us to take full advantage of already existent engine optimizations, and allows us to add more optimizations as needed.

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 [15]. 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 [52]. 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 3.9.

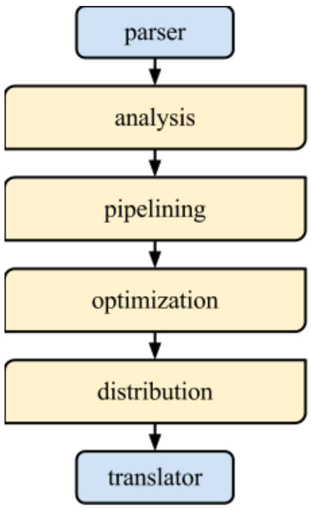


FIGURE 3.9: Compilation process in Brackit [17].

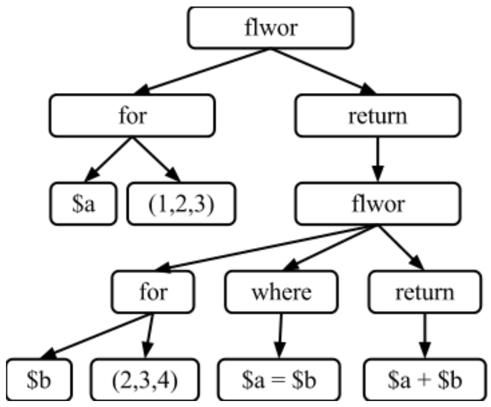
FLWOR Pipelines

Brackit provides a hybrid processing model, when comes to compiling XQuery. It combines characteristics of both *iterative* and *set-oriented* processors. On the former, processors evaluate queries sequentially. They evaluate ASTs recursively, therefore evaluating subexpressions before creating result sequences. As a result, these processors usually present poor scalability because they provide an unmaintainable scenario for parallel execution. The latter computes independent subexpressions more efficiently or even in parallel, not necessarily sequentially. The trade-off is main memory: it usually leads to memory overheads, because explicit representations of the individual state of the dynamic context are required.

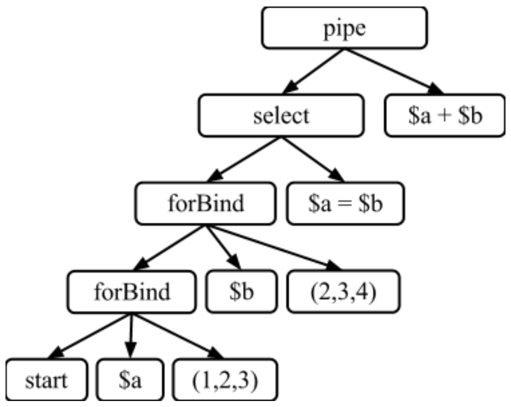
Brackit compiles queries into a tree of expressions, as in the iterator model [37], but parts of queries that might profit from set-oriented processing, such as FLWOR expressions, are extracted from the AST. *Pipe* expressions represent FLWOR expressions, and it contains a top-down pipeline of operators. Therefore, a query is still a tree of expressions evaluated recursively, but with one difference: one or more of the expressions might be a pipeline, thus being evaluated in a parallel-set-oriented style.

Each pipeline—or *FLWOR pipeline*—realizes the tuple-stream semantics of XQuery [32]: it produces a stream of context tuples for the final return clause of the FLWOR expression. The pipeline is processed with cursors, in an open-next-close fashion. A *start* feeds the context tuple to the pipeline. For each input tuple a *for bind* cursor evaluates and binds the items of their bind sequences Finally a *select* filters the results. An example of these expressions can be seen in Figure 3.10.

```
for $a in (1,2,3)
return for $b in (2,3,4)
          where $a = $b
          return $a + $b
```



(A) Expression AST.



(B) Expression tree with operator pipeline.

FIGURE 3.10: XQuery example with original AST and the resulting operator pipeline.

The expression in the upper part of Figure 3.10 consists of two FLWOR expressions, both containing *for-loop* clauses. The first one—*for $a in (1,2,3)*—binds values to the variable *$a*. The second one—*for $b in (2,3,4)*—executes once for each value of *$a*, and also binds values to another variable *$b*. An select operator is then executed, comparing both variables with an equality predicate. Finally, values that were not filtered by the predicates are added. In this case resulting in {4,6}.

Note that the rewriting expressed in Figure 3.10b is not a direct translation of the AST depicted on Figure 3.10a. It as been unnested for optimization reasons. This rewriting does not improve performance, but allows for well-known relational techniques to be applied. *Sorts* might be merged or considered unnecessary, parallelization is possible because context tuples decouple state from computation, and *selects* can be pushed down to reduce the number of selected tuples. Actually, because the tree AST is top-down, these selects are actually being pulled-up on the logical plan. Nevertheless, the physical plan is bottom-up, so we keep the references to the pushdown term.

After discussing both models, MapReduce and XQuery, next section explains an implementation of how to execute XQuery atop the MapReduce computational model.

3.5 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 [52]. 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 [39] 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 [1]. 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 [59], and therefore we will not go into details here.

[1]Note that the term *split* used here as operator for MapReduce does not have any relation to the term *split* in Figure 3.2.

3.6 Summary

MapReduce provides a scalable-parallel programming model, expressing computations through *map* and *reduce* functions. The functions process key/value pairs, or sets of key/value pairs, and achieve parallelization by being evaluated on different nodes and different chunks of data.

XQuery, as the XML query language, provides more than just querying capabilities, allowing for a whole plethora of functionalities. Moreover, we saw how XQuery transparently leverages the MapReduce model and executes in parallel without loss of flexibility.

On the next chapter we provide an XDM mapping implementation for the stores introduced in Chapter 2, allowing them to reuse the compilation and distributed execution logic of BrackitMR.

Chapter 4

XQuery over NoSQL

4.1 Introduction

After discussing several NoSQL stores, the MapReduce model, and some characteristics of the XQuery language, this section presents the main idea of this thesis: leverage NoSQL stores to work as storage layer for XQuery processing. First, we present mappings from NoSQL data models to XDM. Then, we present mappings from relational data to NoSQL data models. We also add XDM-node behavior to these data mappings. Afterwards, we introduce the optional usage of metadata, enabling access optimizations. We discuss these optimizations, together with data-filtering techniques. Finally, we show how we provided for data updates, and how we integrate our approach to a framework for development using XQuery: BrackitAS.

4.2 XDM mappings

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, already explained in Section 2.4. A key/value pair *kv* represents an XDM *element*, and key/value pairs linked to *kv* are addressed as children of *kv*.

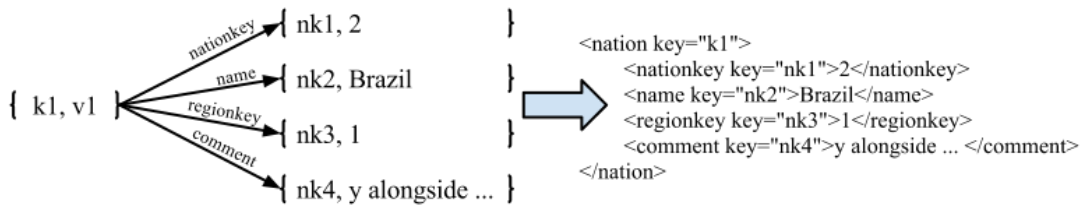

{ k1, v1 }
nationkey
name
regionkey
comment
{ nk1, 2 }
{ nk2, Brazil }
{ nk3, 1 }
{ nk4, y alongside ... }
<nation key="k1">
<nationkey key="nk1">2</nationkey>
<name key="nk2">Brazil</name>
<regionkey key="nk3">1</regionkey>
<comment key="nk4">y alongside ... </comment>
</nation>

FIGURE 4.1: Mapping between a key/value tuple and an XDM instance.

Figure 4.1 shows the overall idea of the mapping. We map key/value tuples as XDM elements. The name of the element is simply the name of the bucket[1] it belongs to. We create one bucket for the element itself, and one extra bucket for each link departing from the element—in Figure 4.1 five buckets coexist: *nation*, *nationkey*, *name*, *regionkey*, and *comment*. 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 load on the system. Besides, each element has an attribute *key* which Riak uses to access key/value pairs on the storage level.

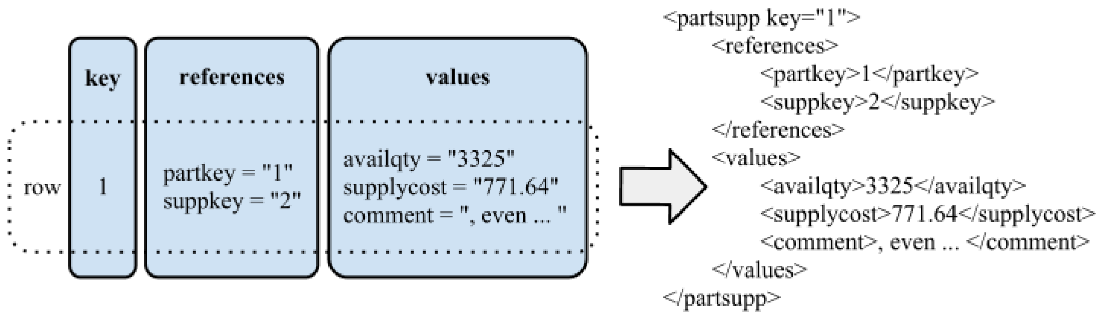
It allows access using key/value as granularity, because every single element can be accessed within a single *get* operation. Full reconstruction of the element depicted in Figure 4.1 requires five different accesses. Besides, Riak provides atomicity using single key/value pairs as granularity, therefore 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 4.2 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*.

Figure 4.2 depicts this mapping. 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.

[1]Remember that a bucket is equivalent to a table in a RDBMS.



```
<partsupp key="1">
    <references>
        <partkey>1</partkey>
        <suppkey>2</suppkey>
    </references>
    <values>
        <availqty>3325</availqty>
        <supplycost>771.64</supplycost>
        <comment>, even ... </comment>
    </values>
</partsupp>
```

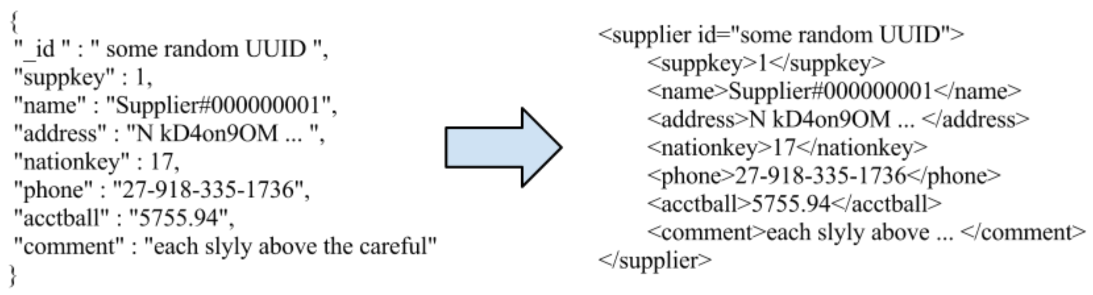
FIGURE 4.2: Mapping between an HBase row and an XDM instance.

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. Low-level details about how MongoDB stores each document's fields are not accessible on a high-level API, therefore we do not have any flexibility on trying to create more efficient mappings, as we do with HBase.

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. Figure 4.3 shows the overall idea of the mapping.



```
{
"_id " : " some random UUID ",
"suppkey" : 1,
"name" : "Supplier#000000001",
"address" : "N kD4on9OM ... ",
"nationkey" : 17,
"phone" : "27-918-335-1736",
"acctball" : "5755.94",
"comment" : "each slyly above the careful"
}
```

```
<supplier id="some random UUID">
    <suppkey>1</suppkey>
    <name>Supplier#000000001</name>
    <address>N kD4on9OM ... </address>
    <nationkey>17</nationkey>
    <phone>27-918-335-1736</phone>
    <acctball>5755.94</acctball>
    <comment>each slyly above ... </comment>
</supplier>
```

FIGURE 4.3: Mapping between a MongoDB document an XDM instance.

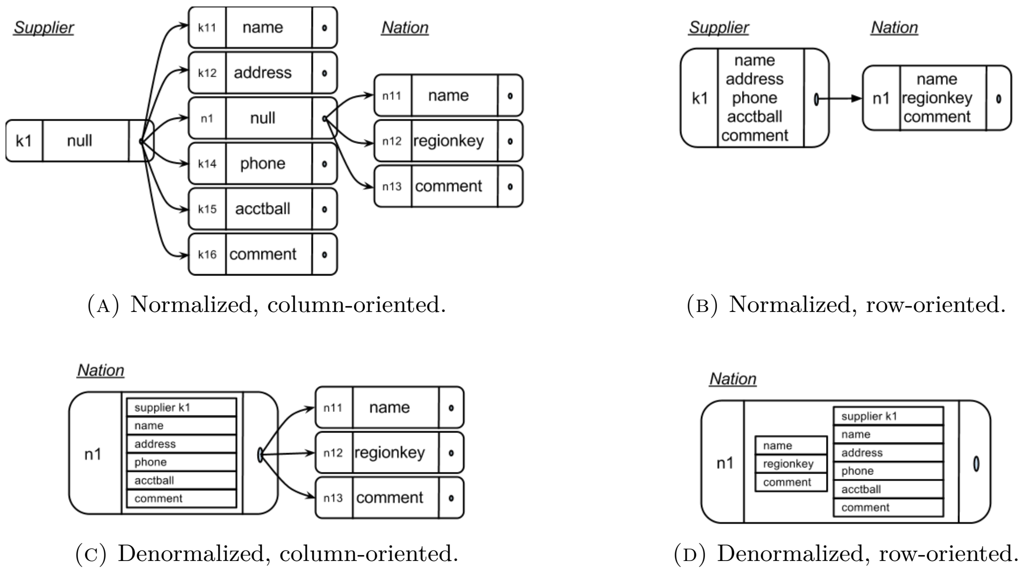
Figure 4.3 shows the mapping for a JSON document to XDM. The left part of the figure represents the document stored within MongoDB. Note that, because MongoDB interprets values, *suppkey* and *nationkey* are actually integers, and not strings. On the right-hand side of the picture we represent the resulting XDM instance.

4.3 Relational mappings

After mapping the data model of the three stores to XDM, we introduce relational-data mappings. We discuss other possibilities for how to map relational data for Riak and HBase. We use TPC-H's [2] relational data for the mappings, because it is the same dataset we use for our experiments later on. No further mapping strategy is explored within MongoDB due to lack of flexibility on the storage level.

Riak

We introduce a record structure for storing relational data within key/value tuples, combined with Riak's linking capabilities. We investigate possibilities atop two dimensions: *data location* and *orientation*. The former refers to where physically place data: either using a denormalized- or denormalized-like mechanism. The latter refers to data-orientation: either *column-oriented* or *row-oriented*. Figure 4.4 shows the possible combinations.



(A) Normalized, column-oriented.

(B) Normalized, row-oriented.

(C) Denormalized, column-oriented.

(D) Denormalized, row-oriented.

FIGURE 4.4: Relational mappings possible with Riak.

In order to better grasp how the mappings work, let us consider the one-to-many (1:n) relationship between tables *nation* and *supplier* of the TPC-H schema [2]. Figure 4.4a shows the *normalized, column-oriented* mapping, which is essentially the mapping already explained in Section 4.2. Each relational row is a key/value *k1*, where relational columns are values linked to k1. Each column is stored in a separated key/value. Figure 4.4b shows the *normalized, row-oriented* mapping. Each relational row is a key/value *k1*, relational columns are represented within the value, and only relational references are stored in a separated key/value.

Figure 4.4c shows the *denormalized, column-oriented* mapping. Each relational row is a key/value *n1*, where relational columns are values linked to n1, and only relational references are denormalized and stored within the *1* side of the 1:n relationship. Figure 4.4d shows the *denormalized, row-oriented* mapping. Each relational row is a key/value *n1*, where both relational columns and relational references are denormalized and stored within the *1* side of the 1:n relationship. In both denormalized cases, each nation appears multiple times within suppliers' data, therefore supplier is stored within nation.

When accessing data from both normalized versions, depicted in the upper-part of Figure 4.4, we need a two-operation access pattern: suppose we want to access the *nation* of the supplier with key *k1.* We would access the supplier key/value with one get operation, and use the nation key *n1* to get the nation value. Therefore, two get operations. To optimize the access pattern, we use a composite key to identify the n-sided table of the 1:n relationship. If *k1'* is used as key, and it is a composition of k1 with n1, we can individually access supplier or nation with one get operation. Both denormalized versions, depicted in the lower-part of Figure 4.4, improve reading-data performance, but creates an unmaintainable scenario for updating data, because they need to update the data replicated among different key/values.

Column-oriented mappings rely on Riak's linking mechanism to represent data structures. This is visibly in Figure 4.4a, where three levels of links are depicted. On the other hand, row-oriented mappings need schema information to represent the record stored within the key/value. In order to explore all options, we chose the two extremes for our experiments, namely *Normalized, column-oriented* and *Denormalized, row-oriented.* We investigate results of both approaches more deeply on our experiments, in Section 5.2.

HBase

We present two additional mappings to store relational data within HBase: mapping relational columns to (i) column families or (ii) qualifiers within HBase. The former takes full advantage of data organization, because query projections can be filtered on storage level. The latter increases row-oriented data locality, because data from different columns will be stored close together, allowing for optimized scan operations. Both approaches are shown in Figure 4.5.

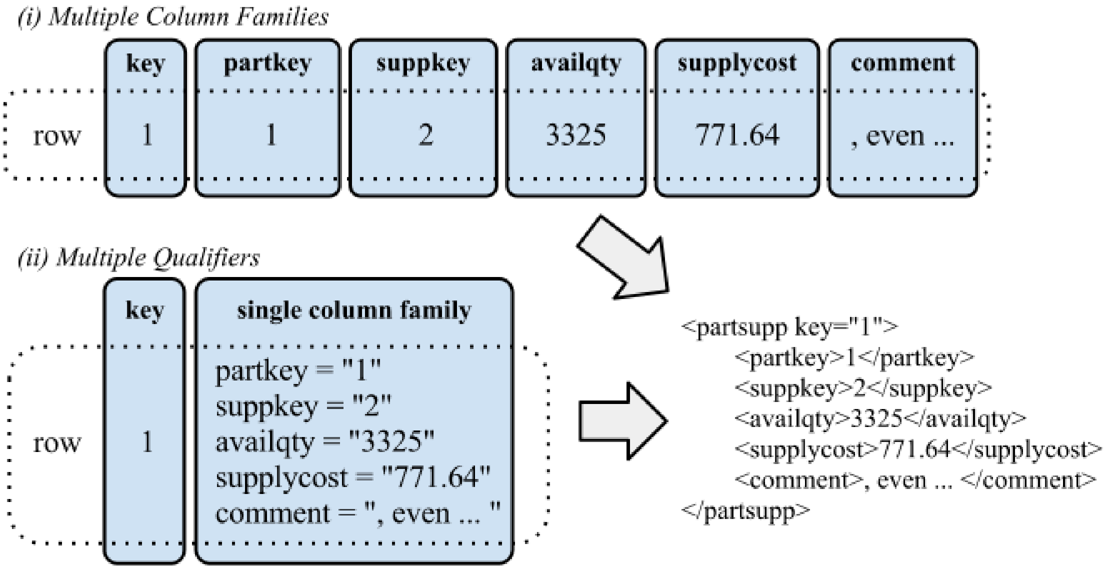


FIGURE 4.5: Mappings (i) and (ii) between an HBase row and an XDM instance.

On the upper part of Figure 4.5, we can see (i) the Multiple Column Families mapping. Each blue-rounded rectangle represents a column family within HBase, and wraps data from a relational column. On the figure's lower-left part, we see (ii) the Multiple Qualifiers mapping. We create a single-extra-generic column family and store values therein. We also create multiple qualifiers, represented on the picture by every line within the rectangle. Both mappings produce the same resulting XDM instance, depicted in the rightmost-bottom corner of the figure. The name of the created element is simply the name of the table it belongs to, and we store key to access it as an attribute on the element. Children elements are either column-family names and values in (i), or qualifier names and values in (ii).

Other strategies for specific configurations of column family/qualifiers go beyond the scope of this work, nevertheless we experiment the described configurations on Section 5.3.

4.4 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 [58], Namespaces [6], and Xquery Update Facility [9] 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*.

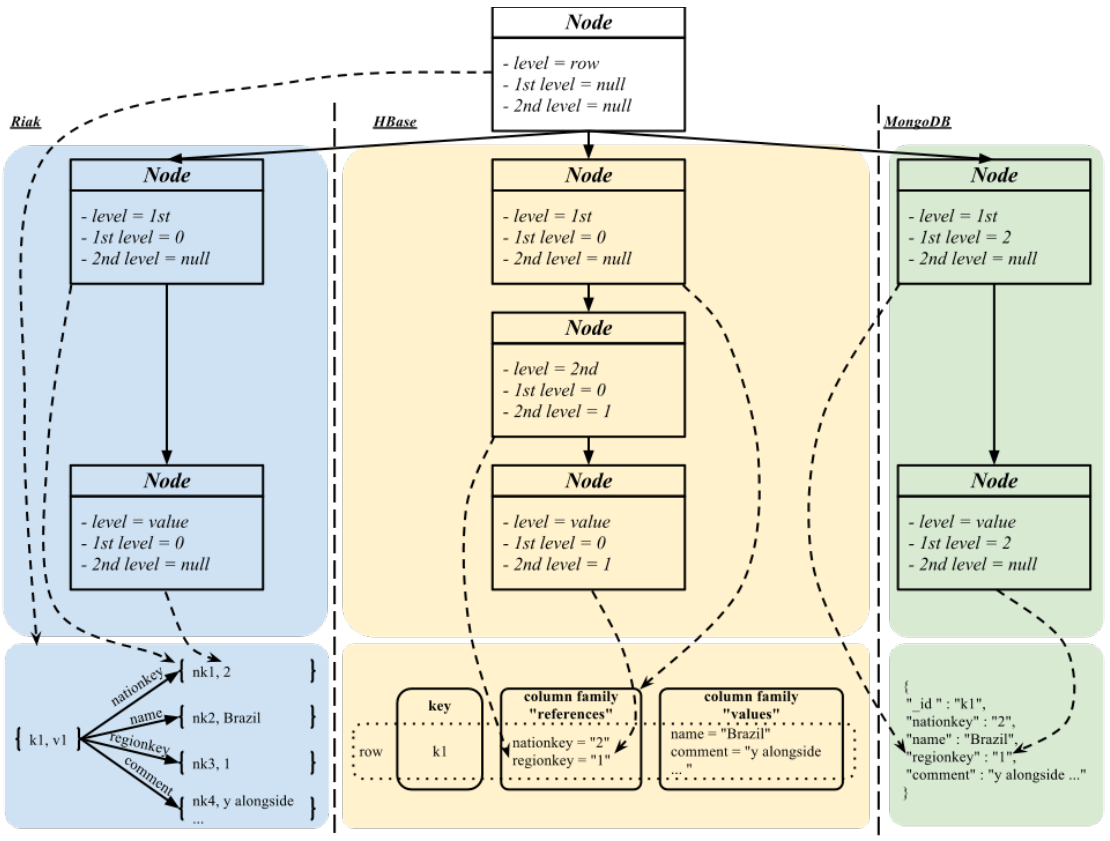


Figure 4.6: Nodes implementing XDM structure.

Overall, each instance of these objects represents one unit of data from the storage level. In order to better grasp the mapping, we describe an abstraction in more details. *Buckets* from Riak, *tables* from HBase, and *documents* from Mongo, 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*. A *link* from Riak, *column family* from HBase, and *field* from MongoDB represent the same level of abstraction within the mapping, from now on referenced as *first-level-access*. HBase *qualifiers* represent a *second-level-access*, which is not present in other store's representations. Finally, *values* from all three stores 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 4.6 depicts the mappings. We divided the figure in three parts, where the left-most part represents Riak's structure. The middle part represents HBase's structure. Finally, the right-most part represents MongoDB's structure. 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. Note that 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*. Its semantic is the same as a *first-level-access* of the other two stores, in this example accessing the wrapper of *regionkey = "1"*. Finally, lower-level nodes with *level = value* access values from the structure.

4.5 Metadata

The node structure presented so far does not require any metadata for its execution. Riak only provides metadata about the keys that are stored, therefore links between key/value pairs are completely independent. HBase only stores information about column families, thus supporting a fixed number of column families per table, but a heterogeneous set of qualifiers among rows. MongoDB allows for heterogeneous documents to be stored within the same collection.

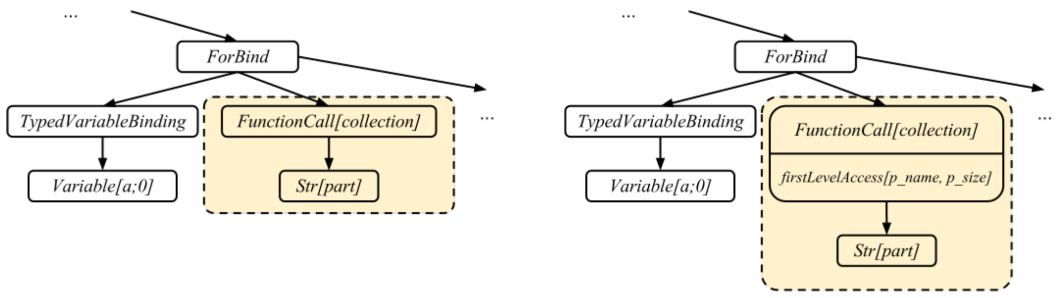
Nevertheless, we used schema information, because in practice we implemented and tested atop relational data. We store informations about data structures within a central *catalog*. It collects table names, column names, and primary and secondary reference keys. We also use these informations for access optimizations at a query level, for example: instead of navigating through a node's child elements—like when looking for a given predicate of a path expression—we already know if it exists and its position. Next section shows how this is done.

4.6 Optimizations

The next subsections discuss several optimizations techniques. We introduce projection and predicate pushdowns, besides data-access 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. Experiments on Section 5 show the performance impact of these techniques.

Projection pushdown

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 of data passed up to the query processor. We traverse the AST, generated in the beginning of the compilation step, looking for path expressions (*PathExpr*). Whenever we find a path expression that represents a *child* step from a collection node, we annotate the collection node on the AST with this information. We do the same for *descendants* of collection nodes, because in the HBase implementation we have more than one access level within storage. Figure 4.7 depicts the overall process.



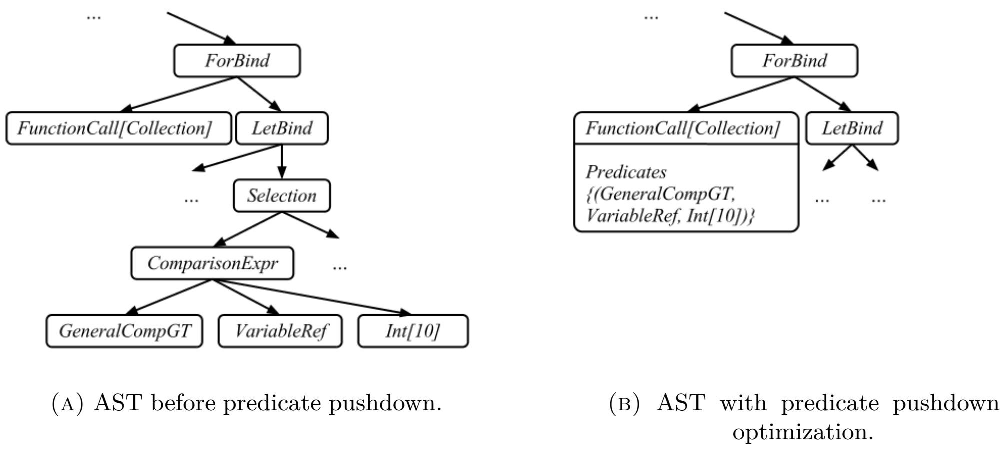
(A) AST with collection node. (B) AST with annotated collection node.

Figure 4.7: Partial ASTs for collection representations.

The AST depicted above shows the collection node for the XQuery of Figure 4.9. Figure 4.7a shows the regular collection node, not yet annotated. When we process this node, it accesses the stores and retrieves complete rows from the storage level, which the query processor filtrates afterwards. Figure 4.7b shows the same collection node, but with an attribute annotated: a *firstLevelAccess*. It contains a vector of projections, which are names that represent the query projections. When we access the collection in the storage level, we use the stores API to filter which data will be passed to the query processor. Note that the amount of data flowing from storage to process level is, on the worst case scenario—projections from all the columns—the same as with no projection pushdown.

Predicate pushdown

Predicate pushdown 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. We traverse the AST—generated in the beginning of the compilation step—looking for general-comparison operators, such as *equal*, *not equal*, *less than*, *greater than*, *less than or equal to*, and *greater than or equal to*. When found, we look for the collection that the operator acts upon, and add a reference.



(A) AST before predicate pushdown.

(B) AST with predicate pushdown optimization.

Figure 4.8: Partial ASTs for collection representations.

When we access the collection on the storage level, we use the marked predicates to filter data, without further sending it to the query engine. Figure 4.8 shows the effects of predicate pushdown in practice. On the first part of the picture, 4.8a, we have the original AST before the optimization. Note that the query consists of a collection access, in which a selection is performed. In this case, we are using the *greater than* operator to compare a variable bound to the collection, to the integer value 10.

The processing layer would turn this AST into a comparison. It would then analyze and filter tuples matching the selection criteria. The second part of the picture, 4.8b, shows the resulting AST, after applying the predicate-pushdown optimization. Instead of having a comparison operation, we have simply annotated the collection node—*FunctionCall[Collection]*—with the comparison information. Moreover, we have removed the selection node. Note that if the selection had more comparison operators, we would replace the comparison node with a boolean node valued *true*.

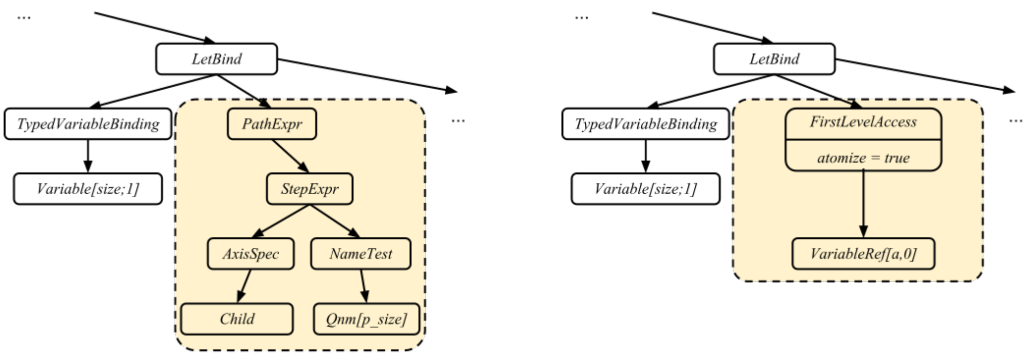
Data-access optimization

As described previously, the Nodes interface provides several capabilities, among them navigation. Usually, navigation within XDM elements works atop *iterators*, because of the lack of schema mechanisms. For example, we cannot access the first child of an element *a* and guarantee that, when accessing the first child of a sibling of *a*, let us say *b*, both children's element type will be the same. Figure 4.9 shows a simple example, and we discuss how to optimize its navigational access.

```
for $a in collection("part")
where $a/p_size > 10
let $size := $a/p_size,
    $name := $a/p_name
return
<result>
        <name>{$name}</name>
        <size>{$size}</size>
</result>
```

FIGURE 4.9: XQuery example for navigational access.

Direct access to a field in a record is not a built-in capability. We must iterate over the children of each document of collection *part* in order to find the *p_size* child. An obvious optimization would be to access the child directly on a given position, and with the help of the metadata structure described in Section 4.5 we can.



(A) AST with iterable navigational access.

(B) AST with optimized navigational access.

FIGURE 4.10: Partial ASTs for navigational access.

Figure 4.10a shows the AST of the *let* binding of Figure 4.9. It consists of a variable creation, called *$size*, and the value bound to it—which is a path expression on top of the already declared variable *$a*. In order to use a direct access, we replace the path expression with our *firstLevelAccess* expression, as can be seen in Figure 4.10b.

What we do is, in a step of the compilation—more precisely the *optimization* step, depicted in Figure 3.9—we traverse the AST looking for path expressions (*PathExpr*) that bound step expressions (*StepExpr*)—such as a child access—to XQuery-bounded variables. We then replace the default StepExpr—which would navigate using traditional iterators—by a so-called *firstLevelAccess* expression. This new expression uses the metadata to directly access child elements. As already discussed in Section 4.4, first-level elements have different semantics, depending on the store used: Riak's are *links*, HBase's are *column families*, and MongoDB's are *fields*.

4.7 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* [9] 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.

We developed supplementary XQuery functions, allowing the user to directly interact with the underlying stores. We created these functions to fulfill the needs for data modifications, like inserts, updates, and deletions. For better organization, we used the concept of namespaces, achieving better classication of these extra features. We added the *db* namespace, comprising database-related functions.

The *insert* function stores a value within the underlying store. It receives as parameters the table—by table we generically refer also to buckets, collections, or any other name that stores use for the data-storing units—as string, and the XML document to be stored. It returns a boolean flag representing success or error results. The specified XML content must be a well-formed XML node.

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

We also provide an alternative signature for the insert function, where a *key* is needed. It essentially allows the user to insert the value using the given key as reference. It also provides *update* semantics, for when an already existent key is used, the store will replace the key/value entry with the new value. In both cases, whenever the table does not exists, it will be automatically created.

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

The *delete* function deletes a given value from the store by its key. It receives as parameters the table as string, and the key value referenced for the deletion. It also returns a boolean flag indicating success or failure on the deletion operation.

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

Besides deleting an specific key, we provide a signature where the user can drop the table. We overloaded the delete function with a delete-table semantics. Whenever no specific key is passed as argument, the whole table will be deleted.

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

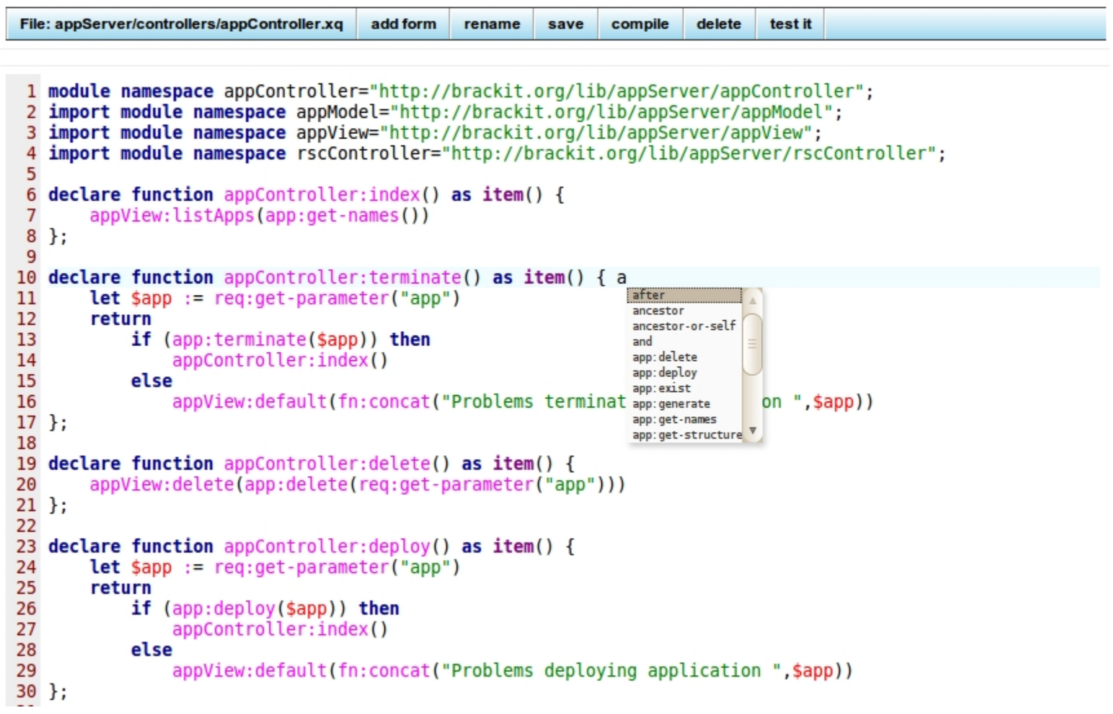
These are development-enabling functions, and next section presents an approach that leverages XQuery to work as full-fledged development languages, thus taking advantage of the just described functions.

4.8 Brackitas

Brackit Application Server (BrackitAS) [56] leverages XQuery to work as full-fledged development language. Its architecture provides intrinsic scalability: it allows for a pluggable-storage layer, therefore using NoSQL stores for persistence was straight-forward.

Moreover, it provides a lightweight, browser-based application and development toolkit, which aims at dynamic scenarios where applications need to grow flexibility with changing requirements regarding UI, functionality, interoperability and scalability. It follows a “do only what you need” philosophy, providing wizard-guided solutions to quickly setup an application for common tasks, and offers a complete development environment to enrich and modify it using XQuery. Figure 4.11 shows the browser-based development toolkit of BrackitAS.

As can be seen in the picture, BrackitAS manages files through a single interface: on the upper part of the picture, compilation, testing, saving and renaming functionalities



```
module namespace appController="http://brackit.org/lib/appServer/appController";
import module namespace appModel="http://brackit.org/lib/appServer/appModel";
import module namespace appView="http://brackit.org/lib/appServer/appView";
import module namespace rscController="http://brackit.org/lib/appServer/rscController";

declare function appController:index() as item() {
    appView:listApps(app:get-names())
};

declare function appController:terminate() as item() { a
    let $app := req:get-parameter("app")
    return
        if (app:terminate($app)) then
            appController:index()
        else
            appView:default(fn:concat("Problems terminat           on ",$app))
};

declare function appController:delete() as item() {
    appView:delete(app:delete(req:get-parameter("app")))
};

declare function appController:deploy() as item() {
    let $app := req:get-parameter("app")
    return
        if (app:deploy($app)) then
            appController:index()
        else
            appView:default(fn:concat("Problems deploying application ",$app))
};
```

FIGURE 4.11: BrackitAS browser-based development toolkit.

are available. It also provides code-highlighting, as well as code-completion. Besides, it works completely out of the box by just accessing the web UI.

Application development

Consider a simple e-commerce website example: users go online, look for products, add them to their shopping cart, pay, check discount options, etc. On the management side, sellers keep track of most sold products, relate items bought together with one another, offer similar offers to clients looking for these products, etc. These situations essentially describe both OLTP- and OLAP-like operations that any modern selling platform works with everyday.

Using XQuery as high-level query language—besides BrackitMR as processing engine, and BrackitAS as application server and development environment—provides the mechanisms to run these queries on any underlying store, in a scalable manner, without any further knowledge.

Suppose our e-commerce website registered a new sale. We use Riak for an OLTP-like query, where a new sell for client with id="4711" is added. Figure 4.12 shows the execution using both Riak's API and XQuery.

We chose Riak for this example because of the simple get/st interface for data manipulation. Nevertheless, we can clearly see how counter-intuitive the whole process

Riak insertion:

```
IRiakClient riakClient = RiakFactory.pbcClient();
Bucket b = riakClient.fetchBucket("sales").execute();
IRiakObject result = b.store("4711", saleInformation)
                   .w(Quora.QUORUM).execute();
```

XQuery insertion:

```
db:insert("sales", "4711",  saleInformation)
```

FIGURE 4.12: Insertion query.

is. Besides, the code above does not show how the Java API handles the result value, fetched from the database if needed. On the other hand, XQuery's approach is intuitive, clean, and self-contained.

Moreover, consider the case where the e-commerce owner holds data about suppliers, regions, orders, etc, pretty much following the schema of [2] for data relationships. We wants to know which *supplier* should be selected to place an *order* in a given *region*, a very OLAP-like query, depicted in Figure 4.13.

```
for  $p in collection("part"),
     $s in collection("supplier"),
     $ps in collection("partsupp"),
     $n in collection("nation")
where $p/p_partkey = $ps/ps_partkey
  and $s/s_suppkey = $ps/ps_suppkey
  and $s/s_nationkey = $n/n_nationkey
  and $ps/ps_supplycost = (
    for $ps in collection("partsupp"),
        $s in collection("supplier"),
        $r in collection("region")
    where $p/p_partkey = $ps/ps_partkey
      and $s/s_suppkey = $ps/ps_suppkey
      and $s/s_nationkey = $n/n_nationkey
    return
      min($ps/ps_supplycost)
    )
order by $s/s_acctbal descending
return
  element minimum_cost_supplier {
    $s/s_acctbal,
    $s/s_name
  }
```

FIGURE 4.13: OLAP-like query.

The above picture represents the intuitive, high-level version, using XQuery. Using Riak's API—or any of the NoSQL stores' API—would require writing the whole query from scratch, using cursors—when available—to iterate over data, managing joins by hand, etc. The usability abyss between high-level languages and low-level record-at-a-time APIs is more than clear.

4.9 Summary

This section presented the main work of this thesis: it showed how to use NoSQL stores as storage layer of BrackitMR: a MapReduce-based XQuery engine. Moreover, it showed how to optimize data access, and how to leverage XQuery to be used as a full-fledged programming language. Next section shows the results of these implementations and optimization techniques, together with some technicalities of the implementation.

Chapter 5

Experiments

5.1 Introduction

The framework we developed in this thesis 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 [2]. TPC-H is a well know benchmark that includes ad-hoc queries, complex OLAP transactions, a wide variety of operators and constraints, thus properly evaluating database's performance. The dataset size we used has 1GB, and we essentially scanned the five biggest tables on TPC-H: *part*, *partsupp*, *order*, *lineitem*, and *customer*. Tables *region*, *nation*, and *supplier*, do not offer much data, containing approximately 0.01% of the whole set. The TPC-H schema can be seen in Figure 5.1.

Our approach so far takes into consideration data already present in NoSQL stores, and provides a powerful and generic tool for querying this data. Nevertheless, in order to make these tests feasible, the relational data of TPC-H has to be modeled for storage. Each section of this chapter will briefly introduce how TPC-H data has been persisted into different stores.

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 [26] and [31].

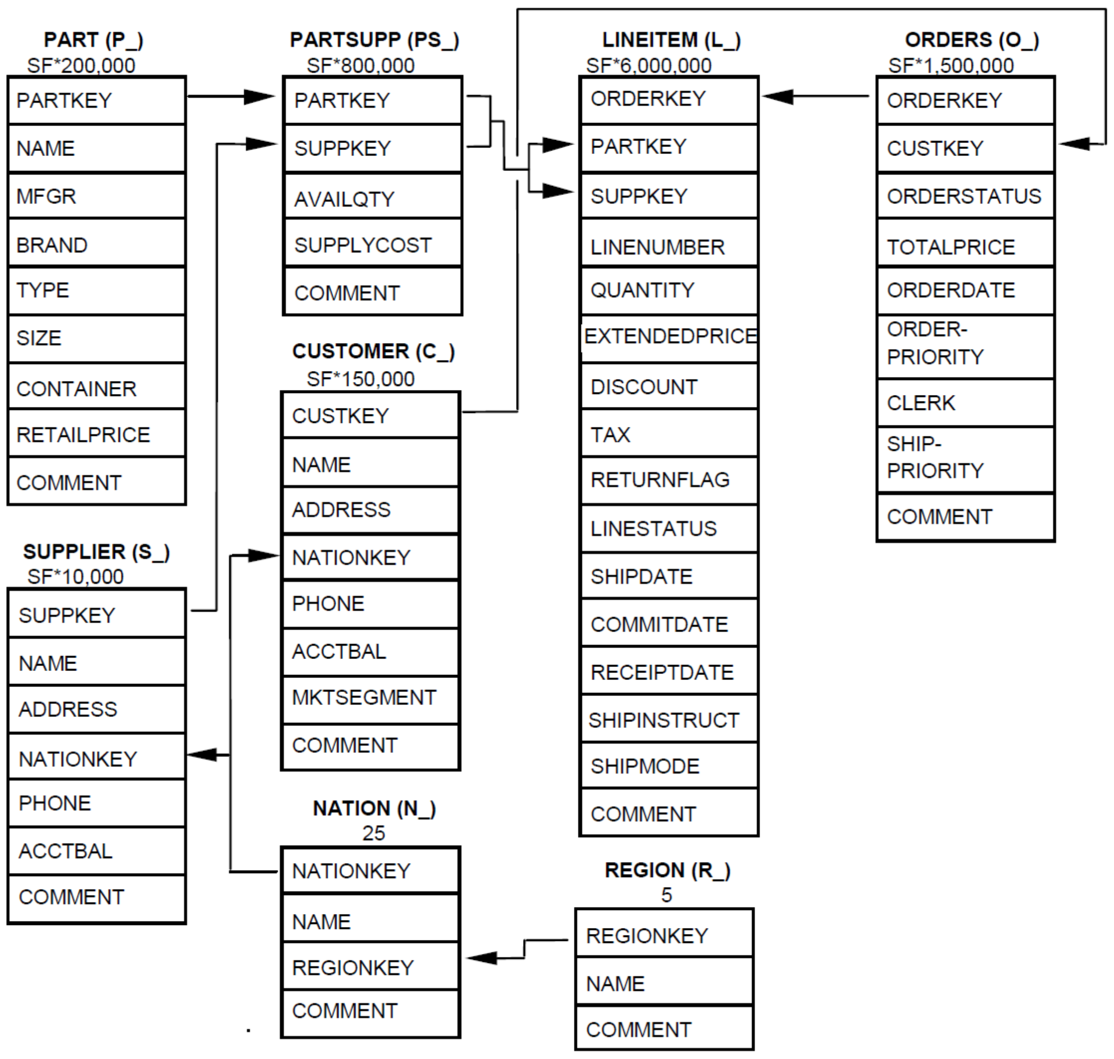


FIGURE 5.1: TPC-H schema.

5.2 Riak

Riak has the simplest data schema of our stores: a simple key/value representation. We described four possible mappings: *(a) normalized, column-oriented*, *(b) normalized, row-oriented*, *(c) denormalized, column-oriented*, and *(d) denormalized, row-oriented*. In practice, we used only (a) and (d)—from now on *(i) column-oriented* and *(ii) row-oriented*, respectively—because they are representative of all possible singular configurations. A more detailed discussion about the mappins is presented in Section 4.3.

The former approach has the advantage of being able to create filtered access to data. Think of a simple example: a scan over a single column of the table data. While the first approach would be able to simply return all key/values for a specific column, the second approach would obligatory transfer all data from the store to the XQuery engine, and only then filter it. The latter has the advantage of creating less RPC calls between client and server. Because Riak does not provide any scanning capabilities, each access to a key is a single RPC call. Therefore, with (ii) we have a single RPC call to each row, instead of a RPC call for each column. We can see the results of both techniques in Figure 5.2.

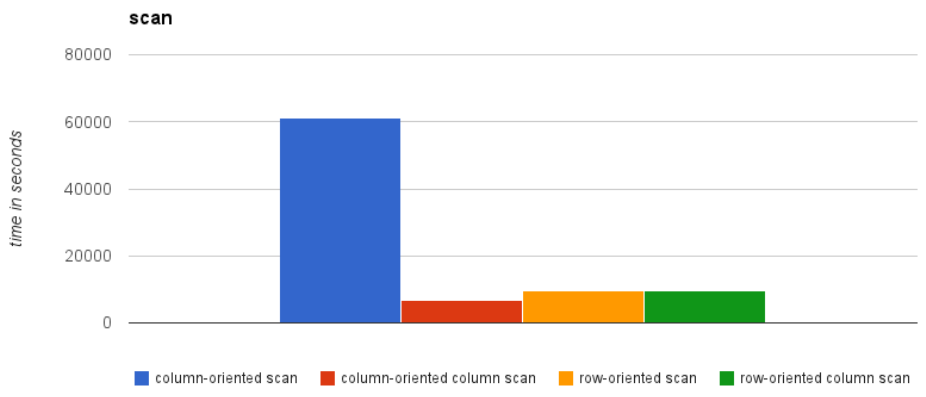


FIGURE 5.2: Latency of Riak's scan.

The picture shows average scan results over 1GB-sized TPC-H tables, on two types of queries: one scanning a whole table (*scan*), and another scanning a single column of a table(*scan/column*). First, we can see that the scan performance of column-oriented scheme is reasonably worst than the row-oriented version. Note that, on the column-oriented version, each column of TPC-H is stored within a different key/value pair. Ergo, in order to scan a whole table, a total amount of *number of columns* RPC calls is needed. On the row-oriented version, all columns are stored within a single key/value pair, therefore minimizing the RPC overhead. Because Riak does not provide any scan capability, this number of RPC calls cannot be further languished.

Figure 5.2 also shows the performance gain on column-oriented schemes when scanning a single column of the table—red column, taking 6733 seconds—in comparison with scanning the whole table—blue column, taking 61032 seconds. On the row-oriented configuration, this number almost remains the same: it varies from 9564 seconds for the full-table scan, to 9531 seconds on the single-column scan. On the row-oriented version, techniques such as filter and projection pushdown do not influence the result. Therefore, in both cases the whole data will flow from storage level up to processing level, only there being filtered. On the other hand, the column-oriented version can take full advantage of these optimizations. Nevertheless, the column-oriented version has shown an overall worst performance than the row-oriented version. Thus, creating less RPC calls has shown to be a better design choice for Riak configurations.

Troublesome environment

Experiments with Riak have shown to be more difficult in practice than expected, because of some bugs (still not fixed in version 1.2.1). First, the pipe mechanism for inputs is not able to keep up the pace of inserts. Essentially, Riak's writing processes stale when receiving too much input. It then forwards these inputs to other writing processes, who also fail, cascading the errors. This is triggered much faster when working with a lower number of nodes. We implemented a pacing-down-writes solution, which is not very effective, but was the only possibility. How slow the pace of writes must be in order to be tolerable is a matter of trial-and-error, and will not be discussed here.

Another bug is in Riak's merging process: Riak K/V stores each *vnode* partition as a separate store directory. Each of these directories contain multiple files with key/value data, because Riak uses partial writes to allow for recovery of data that is not yet fully synchronized to disk. In practice, this data management strategy trades disk space for operational efficiency. Significant storage overhead happens until a threshold is met, then unused space is reclaimed through a process of merging. The merge process traverses data files and reclaims space by eliminating out-of-date versions of key/value pairs, writing only current data to files within the directory.

To understand how this situation manifest itself on our first approach, the algorithm for key/value insertion within Riak is depicted in Algorithm 1.

Algorithm 1 Riak's key/value storage algorithm

```
procedure RIAKINSERT
    sKey ← getNextKey()
    while sKey $\neq$ null do
        IRiakObject key ← keyBucket.store(sKey, sKey)
        key ← bKey.fetch(sKey)
        for (String s : sKey.columns) do
            valueBucket.store(s)
            key.addLink(valueBucket, s)
        end for
        keyBucket.store(key
    end while
end procedure
```

We can see that this layout creates a potential data-thrashing scenario, because every single key needs to be inserted twice. The first insertion is almost instantly discarded, because a second insertion with additional links is issued. Riak's API does not allow for links to be added at non-persistent data, so this double-insertion pattern is necessary.

In practice this becomes even more troublesome: when inserting lots of data, in our case more than 1GB, the merging mechanism is activated, and data is actually written three times: the first insert, the linked-insert, and the merged-insert. The writing processes cannot complete their writing tasks and start dropping inputs. Meanwhile, the whole system does not respond to requests, therefore timing-out each request.

Eventually, the server crashes. It needs to be manually restarted. When restart is issued, Riak starts a recovery protocol: it reads logging information and apply recovery steps for not-yet-persistent data. During recovery, the high volume of insertions again causes thrashing.

5.3 HBase

For testing with HBase mapping schemes, we created three different testing scenarios, already explained in Section 4.2. First, (i) creates one column family within HBase for each column in TPC-H tables. Second, (ii) creates a single column family within HBase which stores all columns from TPC-H tables, and (iii) uses a data scheme where two column families were created, namely *references* and *values*, and each column of TPC-H table is a qualifier within HBase.

The first experiments are the basis for our HBase case. They measure the average latency of the system, thus providing us comparable data for further experiments. An average value of 248 seconds was achieved for scanning all data, with individual measurements staying within standard deviation. The results of this first experiment can be seen in Figure 5.3.

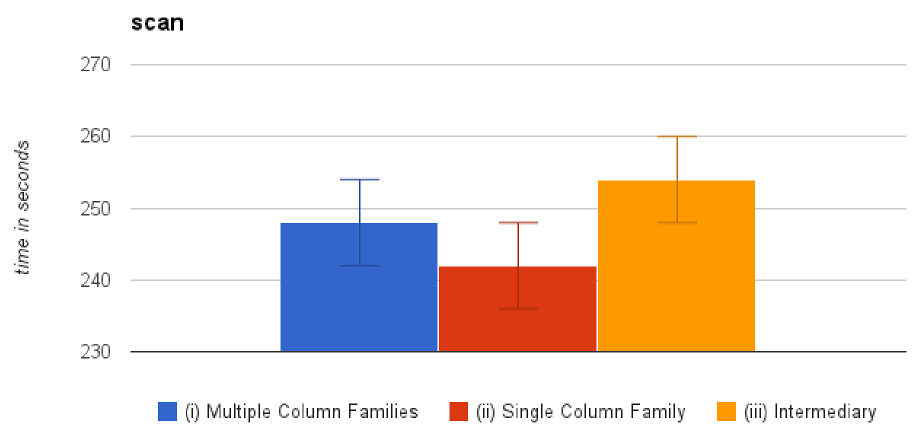


FIGURE 5.3: Latency of HBase's scan.

Not that, according to Figure 5.3, different mapping strategies did not affect performance: (i) scanned all data in 248 seconds; (ii) scanned all data in 242 seconds; (iii) scanned all data in 254 seconds. We cannot see any major difference in this case.

Therefore, when scanning all data, the dataset size dictates latency, regardless of which configuration we chose for column families and qualifiers.

A second and more interesting experiment can be seen in Figure 5.4. Here we pushed down filters, thus minimizing the amount of data transported from store to XQuery engine and resulting in a reasonable performance gain.

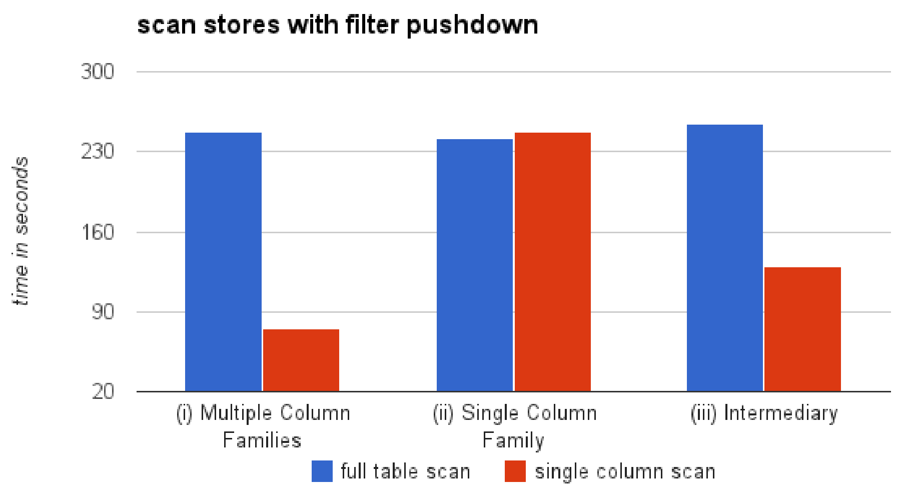


FIGURE 5.4: Latency of HBase's scan using filters.

Figure 5.4 shows latency times for queries scanning 1GB of TPC-H data. Blue columns represent *full table scan*, while red columns represent *single column scan*. The latency improves significantly from 248 to 75 seconds, when using the *Multiple Column Families* configuration. Latency remains almost the same, varying from 242 to 248 seconds, when using *Single Column Family*. Finally, latency improves reasonably, from 254 to 127 seconds, when using the *Intermediary* configuration.

The different mapping schemes have shown meaningful results: filtering data on the storage level instead of filtering it on the processing level explains the collected data. With mapping scheme (i) *Multiple Column Families*, a better performance is achieved than with the other mappings. It uses a file-per-column-family configuration to persist data, which in this case represents the whole columnar data. On the other mappings, data from multiple columns share files within the file system. Therefore, at querying time, it is more time-consuming to respond to queries involving only one column. Note that *(iii) Intermediary* also shows improvement, thus reinforcing that by better separating data in the storage level, less overhead is necessary when scanning this data.

5.4 MongoDB

For testing with MongoDB, only one mapping was possible due to lack of flexibility on MongoDB's storage level. Three types of queries were tested: a *table scan*—which simply return all data from a given table; a *column scan*—which scans a given column; and a *predicate column scan*—which returns data from a given column if the predicate is satisfied. Predicates for the latter were added in order to provide as results an average of half the data returned for the column scan, thus allowing for better comparison. The results can be seen in Figure 5.5.

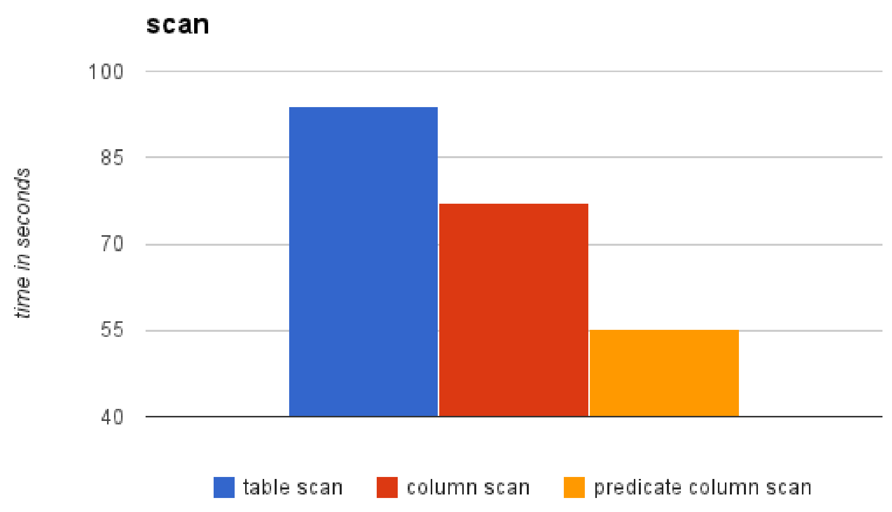


FIGURE 5.5: Latency of MongoDB's scan.

Figure 5.5 shows the performance improvements using filter and predicate pushdown techniques on MongoDB. The first column of the graph—*table scan*—shows the latency when scanning all data from TPC-H tables, taking approximately 93 seconds. The second column—*column scan*—represents the latency when scanning a simple column of each table, taking approximately 77 seconds. 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 third column—*predicate column scan*—represents the latency when scanning a single column and where results were filtered by a predicate. As stated before, we have chosen predicates to cut in half the amount of resulting data when compared with *column scan*. It took approximately 55 seconds to scan the data—a further improvement in latency, reducing the query time in approximately 30%. It does not reach the 50% theoretically-possible-improvement rate, essentially because of processing overhead. Nevertheless, it shows how efficient the technique is.

5.5 Summary

This section has shown in practice our different mapping schemes implemented for different stores. The test cases show how to query data already present on any of the stores, and the results of the implemented optimization techniques. Figure 5.6 summarizes the results.

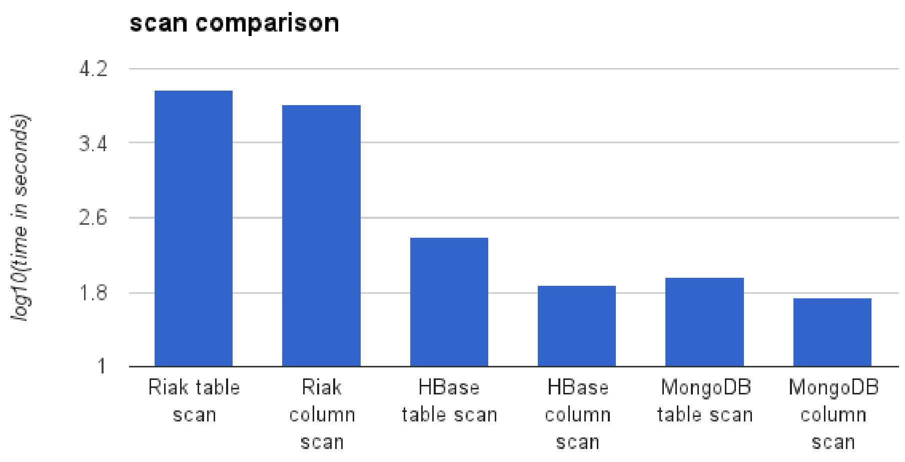


FIGURE 5.6: Latency comparison among stores.

Figure 5.6 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.

In scanning scenarios like the ones on this thesis, 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.

Our testing cases took into consideration only data already present in the stores, therefore we do not intent on comparing data-insert rates on the different scenarios. Moreover, we have not compared results of random read and writes. This affects mostly the Riak implementation when compared to HBase or MongoDB, because Riak only provides a key/value at-a-time API, whereas both other approaches provide scanning mechanisms.

More complete benchmarks on the area are still missing, but we refer to two of them as references. The first one is presented in [26], where systems throughput under read, write, and update operations are compared. Another benchmark is presented on [31] that measures not only the scalability of a couple NoSQL stores, but also the *elasticity*—the ability to expand or contract resources in order to meet the exact demand—of them.

Chapter 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. Moreover, we provided punctual access to update data within these stores.

This chapter concludes this thesis. It presents an analysis of the different technologies used in Section 6.1. Section 6.2 briefly discusses related approaches. Section 6.3 provides a brief overview of the open challenges highlighted by this work.

6.1 Technology Analysis

SQL vs NoSQL

Throughout this thesis, we have avoided as much as possible to get into the SQL vs NoSQL dilemma. It is our understanding that RDBMS have been *de-facto* technology for data storage, and this scenario will probably endure for more years to come. Nevertheless, it seams that more and more users will be willing to trade off ACID guarantees for scalability or availability. Clients tend to tolerate airline over-bookings, rejected orders, inexistent appointments, etc. *"The world is not globally consistent"* [22].

On the SQL side, if RDBMS can indeed scale as NoSQL stores, choosing the technology that provides not only scalability, but standard query language and ACID guarantees seems to be the obvious choice. The robustness of RDBMS is indisputable, and RDBMS systems have been adapted to work for specific applications in the past. There is no evidence up until now showing that they cannot do it again. Besides, many of the NoSQL implementations are definitely not ready for business, working much more as a proof of concept than anything else. On the other hand, there is still a lack of benchmarks showing that SQL can scale as efficiently as NoSQL. Scenarios requiring schema-less or more flexible data models cannot be satisfied by the relational model, and a natural consolidation of some implementations will appear, therefore creating reliable options for the NoSQL costumer. Both will probably coexist in the predictable future.

Riak

Riak distributes data easily, and takes full advantage of this fact for focusing on simple-to-manage clusters. Riak has shown to be horizontally scalable, but not extremely fast. It provides no single point of failure, and in case of disasters, any node can still handle incoming requests. It is completely extensible in the sense that new buckets and schema changes are completely dynamic.

The key/value model is particularly suited for problems with few data relationships. For example, users' session data or shopping carts are largely unrelated to each other. On the other hand, the lack of indexes and scanning capabilities make Riak only interesting when the span of operations will not get any more complex then basic CRUD—Create, Read, Update, Delete—operations. Moreover, it is one of the biggest examples of implementations not yet ready for the market, because of bugs such as the one described in Section 5.2. Nevertheless, it is based on Dynamo, which is in full operation within Amazon's systems.

HBase

With increasing functionality when compared to the key/value model, HBase uses key-matching for querying values—like a key/value store; and groups values among columns—like an RDBMS. Column manipulation is inexpensive, versioning is trivial, and there is no real storage cost for unpopulated values—because values are stored by column, instead of keeping them by row. It is suited for Big Data problems, which scale easily. It also provides built-in support for compression and versioning. The classic usage example is for web-sites indexing, where content is highly textual—beneficing from compression; and changes over time—beneficing from versioning.

On the other hand, there is a confusion of terms in comparison to RDBMS: tables are not relational, rows don't act like records, column families are essentially namespaces, and qualifiers are the actual representation of relational columns. It allows for interesting scenarios, because different designs using column families and qualifiers present different performance in practice, as we saw in Section 5.3. Whenever using multiple column families, and less qualifiers, it provides a column-based data access. On the other hand, with a single column family and multiple qualifiers, it provides a row-based data access. Nevertheless, the schema must be decided upon beforehand, trading-off flexibility.

MongoDB

Document databases are suited to problems involving highly-variable domains. Moreover, documents map well to object-oriented programming models easing the task of moving data between storage and processing layers. MongoDB has potential. It puts in check the impedance-mismatch claim of XML-centric applications by natively speaking JSON, thus allowing JavaScript applications to directly interact with stored documents. Nevertheless, XQuery processors—such as Brackit—already process JSON. Therefore, when using MongoDB as storage layer it also minimizes impedance-mismatch.

On the other hand, both joins and highly-normalized data are not available. Moreover, the 32-bit version is limited to a practical total of 4GB of data per database, because all of the data must be addressable using only 32 bits. They authors claim this is due to code simplicity and easiness of development, but in practice this results in a huge limitation for all 32-bit clients.

We still argue that XQuery is a more robust language for searching, transforming, and manipulating semi-structured content. Nevertheless, XQuery's complexity is sometimes higher then what developers need. We don't believe that MongoDB will completely replace XML databases, but it might split the market in more data-centric applications—specially those focused on JSON interfaces.

Store Comparison

Besides the performance comparison analyzed in Section 5.5, that showed a much more efficient execution for MongoDB in comparison with Riak and HBase, this section compares some characteristics of the stores, like the concurrency mechanism, the replication method, and the transaction mechanisms of the systems. Table 6.1 shows the overall comparison results.

System	Concurrency Control	Replication	Transaction
Riak	MVCC	Asynchronous	Not supported
HBase	Lock	Asynchronous	Locally supported
MongoDB	Lock	Asynchronous	Locally supported

Table 6.1: Stores comparison

Regarding concurrency control, Riak uses *Multi-Version Concurrency Control*, implemented by the Vector Clocks mechanism. HBase provides locks using a row as granularity, and MongoDB provides locks using fields as granularity. Regarding replication, they all implement asynchronous replication mechanisms: Riak uses gossip-protocols, HBase uses a distributed coordination service, and MongoDB uses a master-slave mechanism. Regarding transactions, HBase and MongoDB support them in a local scope. The latter allows transactions using a single document as granularity. The former uses a row as granularity, but allows to increase it to a data region.

Overall they all present similar characteristics, and the table reveals a clear technological gap, already widely discussed in the CAP theorem: either consistency or availability can be satisfied at once. While Riak choses availability, using Vector Clocks and therefore different data version that are always available. Both HBase and MongoDB chose consistency using locking mechanisms, and as a bonus, they are able to support transactions in some intermediary-level of granularity.

MapReduce

MapReduce provides a linearly-scalable programming model for processing and generating large-data sets. It hides parallelization details, fault-tolerance, and distribution aspects 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.

XQuery

XQuery was originally designed as a query language for XML data. It provides expressive power like SQL and supports XML-specific operations such as navigation in hierarchical data. It is being extended by a number of additional features, like XQuery Update Facility [9], and XQuery Scripting Facility [10], thus becoming a full-fledged programming language. Overall, XQuery is perceived as slow and complicated, typically because of XML's markup characteristics, and because XQuery processors do not perform as well as processors geared towards proprietary formats. XQuery has not reached a breakthrough yet, and has not achieved the maturity of more used programming languages, like Java or C.

6.2 Related Work

This section discusses related work. Three approaches are highlighted: *Hive*, *HadoopDB*, and *Scope*. The first two approaches handle data atop a MapReduce framework—Hadoop—and use HDFS, or more general storage level technologies, to store data. The latter presents a new scripting language *SCOPE*—Structured Computations Optimized for Parallel Execution—aiming at large-scale data analysis.

Hive

Hive [54] is a framework for data warehousing on top of Hadoop. It supports queries using *HiveQL*—a SQL-like declarative language—and compiles them into MapReduce jobs, executing them in Hadoop. The language includes typed tables, primitive types, collections, and compositions of just cited mechanisms. It comprises a subset of SQL, adding extensions like *MAP* and *REDUCE* clauses. A catalog holds schema details, like informations about tables, partitions, columns, and data types, physical locations, statistics informations, etc.

Create table statement:

create table user (id **bigint**, name **string**, active **boolean**)
partitioned by (familyName **string**)
stored as sequencefile;

Select statement:

select user.*
from user
where user.familyName = “Valer”;

Figure 6.1: Example of HiveQL query.

HiveQL creates tables and queries data as presented in Figure 6.1. HiveQL creates a partition for each distinct value of the referenced column, in this case of *familyName*. The query part, represented by the *select* statement is similar to an SQL statement. In this query, HiveQL prunes results based on the partitioning clause.

On the downside, Hive 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. In order to access data in NoSQL stores, the user must implement Hive's serialization interface.

Hive provides an extension called *Hive Storage Handlers*, which allows Hive to access data stored by other systems, including in the moment HBase and Hypertable [43]. Nevertheless, this is a working in progress, not at all finished, and with minimal support for other projects than HBase at the moment. Moreover, Hive is not flexible enough for Big Data problems. It is not able to understand the structure of Hadoop files without the catalog information, and therefore is not usable on scenarios where data is already present within other NoSQL stores.

HadoopDB

HadoopDB [13] extends Hive. It uses MapReduce on the communication layer atop nodes running DBMS instances. It translates SQL queries to MapReduce—a much similar approach appears later on in [52], mapping XQuery to MapReduce, and is used on this thesis—and dispatches work to DBMS nodes. Therefore, queries are executed in parallel among different nodes, but as much work as possible is pushed down to the DBMS instances.

A *Database Connector* connects different database systems to HadoopDB. It receives an SQL query from the MapReduce jobs, connects to the database, executes the query, and finally returns the values as key/value pairs. A *Catalog* maintains informations about the databases, like physical location, partitioning, replica locations, etc. The SQL to MapReduce process is done by extending HiveQL, but now instead of connecting to tables stored as HDFS files, DBMS's are used. Therefore, more complex optimizations are possible due to more sophisticated cost-based optimizers present in DBMS's. These are very specific operations, and up until now only possible for RDBMS's, like MySQL and PostgreSQL. Furthermore, it enables point access.

Nevertheless, the system still suffers from the same limitations of Hive: lack of general solution—supporting only SQL systems—and lack of flexibility on the storage layer, exactly because of the use of a DBMS. Moreover, the implementation available so far does not provide sufficient performance when compared to parallel databases [13]. It was one order of magnitude slower on the overall query time—mainly because they use MapReduce, not because of their architectural decisions.

Scope

Scope [23] provides a declarative scripting language targeted for massive data analysis. It borrows several features from SQL: data is modeled as sets of rows, with typed columns, and uses well-defined schema on a per rowset basis. Figure 6.2 shows an example of query using Scope.

Scope step-by-step query:

```
e = extract query
from "search.log"
using LogExtractor;

s1 = select query, count(*) as count
from e
group by query;

s2 = select query, count
from s1
where count > 1000;

s3 = select query, count
from s2
order by count desc;

output s3 to "qcount.result";
```

Scope SQL-like query:

```
select query, count(*) as count
from "search.log" using LogExtractor
group by query
having count > 1000
order by count desc;
output to "qcount.result";
```

Figure 6.2: Example of Scope query [23].

The right part of Figure 6.2 represents an SQL-like query, possible to write using Scope. It looks for the most used "queries" inside a search log. The command is really similar to an SQL select statement, being the *using* clause an exception. In this example, it uses the *LogExtractor*—which parses each log and extracts requested columns—as built-in extractor. On the left-most part of the picture we have the same query, at least regarding functionality, in a step-by-step manner. Each command takes the previous output as input. This example first extracts data from the log file. *S1* counts query occurrences. *S2* filters occurrences with less than 1001 repetitions. *S3* sorts the result. Finally, *S4* writes the result to a file.

The system runs atop a distributed computing platform called *Cosmos*. It provides replication and data distribution atop commodity hardware. Data is stored in an append-only fashion. Programmers use a high-level API for development, and programs are modeled as a directed-acyclic graph (DAG) for execution. It complements the SQL model by providing MapReduce capabilities: *process*, *reduce*, and *combine* provide the same functionality as the MapReduce model—using a row as granularity, instead of key/value pairs.

Scope has shown experiments on which they scale linearly with cluster size, and with data input size. Nevertheless, they are just as inflexible regarding storage layer—using Cosmos files for storing data—and provide no generic solution, supporting only a MapReduce-like model mixed with SQL: the Scope scripting language. Moreover, even though their approach scales, there is a lack of comparison with other approaches concerning performance: it seams clear that the goal is either provide better performance than parallel databases, or performance similar to parallel databases plus scalability. Scope has shown only scalability so far, as any MapReduce-based system.

VoltDB

VoltDB [53] is an in-memory database. It is ACID-compliant and uses a shared nothing architecture. It supports SQL access from within pre-compiled stored procedures written in Java. It uses stored procedures as unit of transaction—procedures are commited if they are successful, otherwise they are rolled back—and executes them sequentially.

VoltDB relies on horizontal partitioning to scale. Each VoltDB database is divided in a number of partitions over a number of nodes. Each partition is single-threaded, thus eliminating locking and latching overheads. It stores each row from tables across partitions, and it creates partitions based on table's primary keys.

VoltDB uses synchronous replication, and uses snapshots to persist data. Snapshots are needed because VoltDB is an in-memory database. Therefore, in unexpected shutdowns, memory-resident data will be lost. At fixed intervals, VoltDB persists snapshots do disk. In case of disasters, it also uses them for recovery.

Although VoltDB provides some scalability without compromising data consistency, some scenarios will definitely not scale efficiently. We can think of two for now: (i) operations and (ii) transaction that span trough many nodes. For example, (i) joins over many tables, will not scale efficiently with data partitioning techniques. Likewise, (ii) transactions will be very inefficient due to communication and two-phase commit overheads.

Hyracks

Hyracks *is* a platform for data-intensive applications. It provides a partitioned-parallel model to run data-intensive computations on shared-nothing clusters. It stores large collections of data as local partitions across nodes in a cluster.

Hyracks provides a programming model to divide computations on large data collections: it expresses computations using DAG of data operators (nodes) and connectors (edges). It provides an API allowing to describe operators' behavioral and resource-usage characteristics. Operators consist of one or more activities. At run-time, each of these activities is realized as a set of identical tasks, each of which operates over individual partitions of the data.

Users can write jobs without the need to write processing logic as map and reduce functions, because Hyracks provides a higher-level query language (AQL) to query and analyze data. Moreover, to facilitate the migration for Hadoop users, Hyracks provides an Hadoop-like interface. It receives Haddop jobs and processes them atop Hyracks platform.

When comes to fault tolerance, it improves MapReduces's strategy: while MapReduce persists intermediate results to disk, Hyracks follows two different techniques: for medium- to big-sized jobs, it follows the same persisting-to-disk strategy. For smaller- to medium-sized jobs, it simply restart all jobs impacted by failures. This strategy has shown to be effective for such environments [19].

Overall, Hyracks improves MapReduce computational model, by allowing a more flexible programming model and providing a higher-level query languages. Nevertheless, it presents problems similar to Scope and leaves the question of *"performance better than parallel databases, or performance similar to parallel databases plus scalability"* still open.

6.3 Future work

Working with different data models and NoSQL implementations has open our eyes to several unfinished solutions and open challenges. Like (i) *column/family versus qualifier* configurations, or a (ii) *"store" plan optimization.*

The former is strictly related to columnar stores, so in this case we will talk directly about HBase. The current HBase implementation advises for as less column families as possible, essentially because of their flushing and compaction scheme. So-called *minor compactions* happen reasonably often within HBase. They follow file-selecting rules based on size, and rewrite together usually the two smallest matching files—files that store the same column family. The more column families we have, the more the compaction mechanism will work, and the more data will be uneven distributed among column families. If one column family is carrying the bulk of the data, the adjacent families will also be flushed through disk, even though the amount of data they carry is small. For example, If column family a has 1 million rows and column family b has 1 billion rows, data from a will likely be spread across many regions, which makes scans for column family a less efficient. On the other hand, this means that data is clustered by column, and partitioned by row. Therefore, data being spread across many regions also means more parallelism.

Column families are a compromise between row-oriented and column-oriented access. HBase provides column families as an optimization that supports column operations, therefore column families are more about performance than about schema. We have show how different column family configurations provide different results in practice in Section 5.3. In this regard, schema design of column families for Hbase—or any columnar NoSQL store—that better suits different application scenarios are needed, and their automation is an open and promising area.

The latter essentially means choosing the best underlying store to answer a request. Think about the final steps of a compilation process in most RDBMS's: after generating different query plans, the processor would estimate their costs and chose the "best" query plan to answer the query. As with cost-based query optimizers, we could describe the costs of physical operators in different stores, as well as average access times, thus gathering information. A *decision layer* is placed atop of stores instances, centralizing—does not mean that it cannot be a distributed system—requests. It reads the request and, based on the gathered information, chooses the best underlying store to answer it.

We already show in this work that mappings from stores to XDM are possible, therefore allowing for the same data to be stored among different stores. Moreover, the *parallel databases versus NoSQL stores* question could be answer by choosing the underlying data-store on a per-request basis: for performance—up to a size threshold—parallel databases answer the requests; for scalability—from the size threshold onwards—NoSQL-scalable stores answer the requests.

Bibliography

[1] Json - javascript object notation. http://www.json.org/, 1999.

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

[3] Marklogic operational database. http://www.marklogic.com/, 2001.

[4] Saxon xslt and xquery processor. http://www.saxonica.com/, 2004.

[5] exist-db xml database management system. http://exist-db.org/, 2005.

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

[7] Xml path language. http://www.w3.org/TR/xpath-30/, 2007.

[8] Xbird xquery processor. http://code.google.com/p/xbird/, 2009.

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

[10] Xquery scripting extension 1.0. http://www.w3.org/TR/xquery-sx-10/, 2010.

[11] Brackit xquery engine. http://brackit.org/, 2011.

[12] Zorba xquery processor. http://www.zorba-xquery.com/, 2012.

[13] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[14] F. Alted. Why modern cpus are starving and what can be done about it. *Computing in Science Engineering*, 12(2):68 –71, march-april 2010.

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

[16] Sebastian Bächle and Caetano Sauer. Unleashing xquery for data-independent programming. *Submitted*, 2011.

[17] Sebastian Bächle and Caetano Sauer. Unleashing xquery for data-independent programming. *Submitted*, 2011.

[18] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12): 1272–1283, 2011.

[19] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767921. URL http://dx.doi.org/10.1109/ICDE.2011.5767921.

[20] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45:23–29, 2012. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2012.37.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298487.

[22] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. ISSN 0163-5808. doi: 10.1145/1978915.1978919. URL http://doi.acm.org/10.1145/1978915.1978919.

[23] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1454159.1454166.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

[25] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. Oreilly Series. O'Reilly Media, Incorporated, 2010. ISBN 9781449381561. URL http://books.google.de/books?id=BQS33CxGid4C.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell 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. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL http://doi.acm.org/10.1145/1323293.1294281.

[29] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 623–634, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872832. URL http://doi.acm.org/10.1145/872757.872832.

[30] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM. ISBN 0-89791-239-X. doi: 10.1145/41840.41841. URL http://doi.acm.org/10.1145/41840.41841.

[31] Thibault Dory, Boris Mejhas, Peter Van Roy, and Nam Luc Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.

[32] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, 2005. URL http://www.w3.org/TR/xquery-semantics/.

[33] L. George. *HBase: The Definitive Guide.* O'Reilly Media, 2011. ISBN 9781449315221. URL http://books.google.de/books?id=nUhiQxUXVpMC.

[34] S Ghemawat and J Dean. Leveldb key/value store. http://code.google.com/p/leveldb/, 2011.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL http://doi.acm.org/10.1145/1165389.945450.

[36] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[37] Goetz Graefe. Query evaluation techniques for large databases. *ACM COMPUTING SURVEYS*, 25:73–170, 1993.

[38] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132867. URL http://doi.acm.org/10.1145/1132863.1132867.

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

[40] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there, 2008.

[41] E.R. Harold and W.S. Means. *XML in a Nutshell.* In a Nutshell. O'Reilly Media, 2004. ISBN 9780596007645. URL http://books.google.de/books?id=NBwnSfoCStAC.

[42] Mikio Hirabayashi. Tokyo Cabinet: a modern implementation of DBM. http://fallabs.com/tokyocabinet/, 2009. URL http://tokyocabinet.sourceforge.net/index.html.

[43] Hypertable Inc. Hypertable. http://hypertable.org/, 2012.

[44] F.P. Junqueira, B.C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245 –256, june 2011. doi: 10.1109/DSN.2011.5958223.

[45] Rusty 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. ISBN 978-1-4503-0516-7. doi: 10.1145/1900160.1900176. URL http://doi.acm.org/10.1145/1900160.1900176.

[46] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL http://doi.acm.org/10.1145/279227.279229.

[47] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[48] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC ’99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268708.1268751.

[49] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew 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. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726. URL http://doi.acm.org/10.1145/1376616.1376726.

[50] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996. ISSN 0001-5903. doi: 10.1007/s002360050048. URL http://dx.doi.org/10.1007/s002360050048.

[51] C.S.R. Prabhu. *Object-Oriented Database Systems: Approaches and Architectures*. Eastern Economy Edition. Prentice-Hall Of India Pvt. Limited, 2005. ISBN 9788120327894. URL http://books.google.com.br/books?id=Yo6jGojiW4cC.

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

[53] L.M. Surhone, M.T. Timpledon, and S.F. Marseken. *VoltDB*. VDM Publishing, 2010. ISBN 9786132129666. URL http://books.google.de/books?id=Ta1RYgEACAAJ.

[54] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[55] Henrique Valer. Xquery-based application development. http://hdl.handle.net/10183/31022, 2011.

[56] Henrique Valer and Sebastian Bächle. Brackitas: A lightweight xquery-based application server. *Submitted*.

[57] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007. ISBN 0596006349.

[58] Norman Walsh, Mary Fernández, Ashok Malhotra, Marton Nagy, and Jonathan Marsh. XQuery 1.0 and XPath 2.0 data model (XDM). http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/, January 2007.

[59] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012. ISBN 9781449338770. URL http://books.google.de/books?id=Wu_xeGdU4G8C.