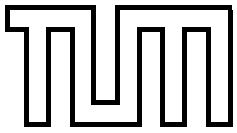
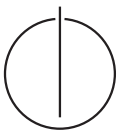


TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK



Forschungs- und Lehreinheit XIX:
Software Engineering for Business Information Systems

Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer

Analyse und Klassifikation von NoSQL Datenbanken und Bewertung ihrer Eignung zur Ablösung einer objektrelationalen Persistenzschicht

Master's Thesis

Author:	Kai Orend
Supervisor:	Prof. Florian Matthes, Ph.D.
Advisor:	Thomas Büchner, Ph.D.
Submission Date:	14.04.2010

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis only supported by declared resources.

München, den 14.04.2010

Kai Orend

Abstract

This work deals with distributed databases from the NoSQL movement, which gained popularity since April 2009. It explains some of the basic problems of distributed databases and the technologies that are used to solve them. Furthermore, a selection of NoSQL databases is analyzed and classified. In order to evaluate their ability to replace an object-relational persistence layer, their feature sets were compared to the requirements of an already existing web collaboration and knowledge management software. This application was then enhanced with a support layer for a NoSQL database and later on compared with the equivalent implementations for relational databases.

Zusammenfassung

Diese Arbeit beschäftigt sich mit Datenbanken aus der erst seit April 2009 bestehenden NoSQL Bewegung. Sie erklärt einige der grundlegenden Probleme verteilter Datenbanken und die Technologien die verwendet werden um diese zu lösen. Außerdem wird eine Auswahl von NoSQL Datenbanken analysiert und klassifiziert. Um ihre Eignung zur Ablösung einer objektrelationalen Persistenzschicht zu beurteilen, wurden ihre Fähigkeiten mit den Anforderungen einer bereits existierenden Web Collaboration und Knowledge Management-Software verglichen. Diese Anwendung wurde dann mit der Unterstützung für eine NoSQL Datenbank erweitert und später mit den entsprechenden Implementierungen für relationale Datenbanken verglichen.

Contents

1 Introduction

Some of today's web applications face the big challenge of serving millions of users, which are distributed all over the world and who expect the service to be always available and reliable. Successful web services do not only have big user bases, they are also growing faster than the performance of computer hardware is increasing. So at one point or another a web application which attempts to become big needs the ability to scale.

Scalability of a web application means that any workload or any amount of data can be processed in a defined time if the application can run on enough servers, which will be referred to in this text as nodes. In the ideal case the relation between workload, time and needed machines would be linear. A more general definitions of scalability can be found under [43] and [42].
Every application has different requirements towards scalability. Some need the ability to serve a large number of users at the same time and others might need to be able to scale with the amount of data.
Some applications like social driven applications even need to scale in both ways.

So the idea is to distribute a web application over as many nodes as necessary. To see how this can be done, the infrastructure of Amazon is used in this work as an example for a scalable web application.

1.1 Example: Amazon

Amazon started 1994 as an Online Bookstore and is now one of the biggest marketplaces in the Internet with an average of 70.000.000 visitors per month in the year 2009 on amazon.com alone [22]. They started with a monolithic web server which held the business logic and generated the pages for amazon.com and one database. This architecture was changed in 2001 to a Service Oriented Architecture (SOA) to enable the application layer to scale [8].

1.1.1 The Architecture of Amazon

Requests are handled in Amazon's SOA by front-end servers which are responsible for the generation of the requested pages. For this the rendering components have to send requests to approximately 150 services. Each of these services can have dependencies to other services, which can result into call graphs with more than two layers [26].

Amazon distinguishes between services with states and stateless aggregation services, which are used to distribute the workload to other services and to merge the results, so they only need caching and no persistence, see figure 1.1.
The other services can use several different kinds of data stores, dependent on the requirements of their task. For example, a service with strong requirements for consistency, like a service involved in payment, can use a Relation Database Management System (RDBMS), whereas other services with weaker consistency requirements can use one of the distributed storage solutions like Amazon's Dynamo, S3 or SimpleDB, which are described later on in this work.

1.1.2 Observations

One possible solution to distribute a web application over different nodes is to split it into several services which all have their own database. This kind of scaling can only be done until the services cannot be split anymore.

So to be able to scale without restrictions, the database layer must also be scalable. In order to achieve this, Amazon uses distributed non relational databases like Amazon S3 and Dynamo.

1.2 The NoSQL Movement

In the past, relation databases were used for nearly everything. Because of their rich set of features, query capabilities and transaction management they seemed to be fit for almost every possible task one could imagine to do with a database.

But their feature richness is also their flaw, because it makes building distributed RDBMSs very complex. In particular it is difficult and not very efficient to make transactions and join operations in a distributed system. This is why, there are now

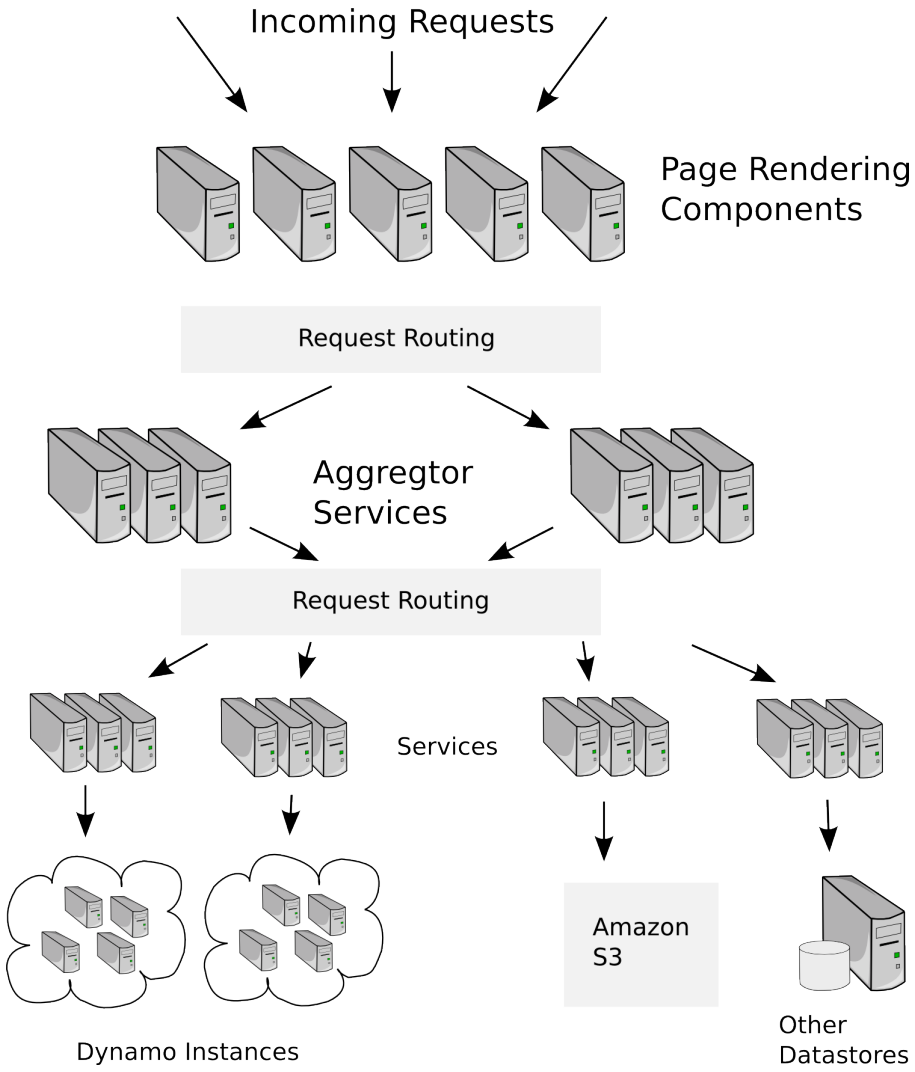


Figure 1.1: Overview of Amazon's Service Oriented Architecture.

some non relational databases with limited feature sets and no full ACID (see 2.2) support, which are more suitable for the usage in a distributed environment.

These databases are currently called **NoSQL databases**. This term just reached a noticeable awareness level in April 2009 as it can be seen in the Google Trends result in figure 1.2, even though the term was first used 1998 as a name of a database [46].

The name first suggests that these databases do not support the SQL query language and are not relational. But it also means "Not Only SQL", which is not so aggressive against relational databases. This stands for a new paradigm: One database

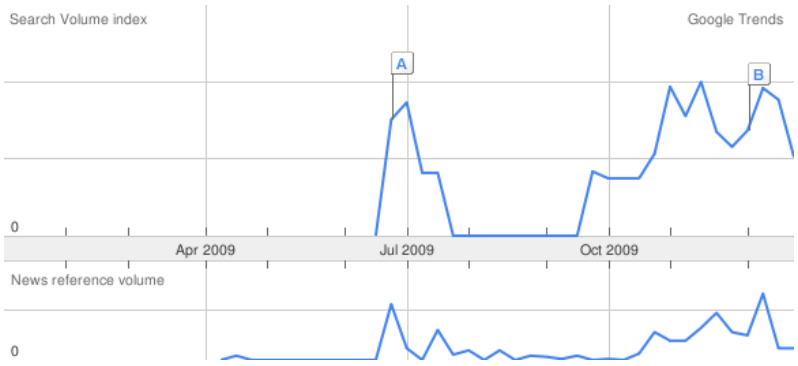


Figure 1.2: Google Trends result for the term **NoSQL**.

technology alone is not fit for everything. Instead it is necessary to have different kinds of databases for different demands, like in the architecture of Amazon.

Most NoSQL databases are developed to run on clusters consisting of commodity computers and therefore have to be distributed and failure tolerant. To achieve this, they have to make different trade-offs regarding the ACID properties, transaction management, query capabilities and performance. They are usually designed to fit the requirements of most web services and most of them are schema free and bring their own query languages.

The goal of this work is to give an overview of the current state of NoSQL databases and how they can be used for existing web applications.

1.3 Reader's Guide

The first part of this work intends to give an overview of the current landscape of NoSQL databases and explains some of the basic problems of distributed databases and commonly used technologies to solve them, to better understand the restrictions of current implementations. Furthermore some NoSQL databases and their concepts are examined and compared to each other.

The second part evaluates if a NoSQL database can be used for a persistence layer of a collaborative web application. This is done in this work while extending the persistence layer of an already existing web application to support a NoSQL database.

Chapter 2 : Explains the problems and trade-offs of distributed databases and some of the technologies that are used for their implementation.

Chapter 3 : Examines a selection of currently available NoSQL databases.

Chapter 4 : Compares the in chapter 3 examined databases against each other.

Chapter 5 : Analyzes the database requirements of Trica, a collaborative web platform with an object relation persistence layer.

Chapter 6 : Describes the implementation of a NoSQL support layer for Tricia.

Chapter 7 : Evaluates the implementation from Chapter 6.

Chapter 8 : The conclusion of this work.

2 Basics

The intention of this chapter is to explain the problems of distributed databases and the technologies that are commonly used to solve them. It shows how databases can be scaled and why distributed database have to make several trade-offs to achieve this. Furthermore, the MapReduce pattern is explained, which simplifies distributed computing and is used in some NoSQL databases for complex aggregations.

2.1 Scaling through Sharding and Replication

A database can be scalable in three different ways. It can be scalable with the amount of read operations, the number of write operations and the size of the database. There are currently two technologies that are used to achieve this: Replication and Sharding.

2.1.1 Replication

Replication in the case of distributed databases means that a data item is stored on more than one node. This is very useful to increase read performance of the database, because it allows a load balancer to distribute all read operations over many machines. It is also very advantageous that it makes the cluster robust against failures of single nodes. If one machine fails, then there is at least another one with the same data which can replace the lost node.

Sometimes it is even useful to replicate the data to different data centers, which makes the database immune against catastrophic events in one area. This is also done to get the data closer to its users, which decreases the latency.

But the downside of data replication are the write operations. A write operation on a replicated database has to be done on each node that is supposed to store the respective data item. A database has basically two choices for doing this: Either

a write operation has to be committed to all replication nodes before the database can return an acknowledgment. Or a write operation is first only performed on one or a limited number of nodes and then later send asynchronously to all the other nodes. The choice of one of this two options decides the availability and consistency properties of the database, which will be explained later in this work with Brewer's CAP Theorem.

2.1.2 Sharding

The term Sharding derives from the noun 'shard' as it means that the data inside a database is splitted into many shards, which can be distributed over many nodes. The data partitioning can, for example, be done with a consistent hash function that is applied to the primary key of the data items to determine the associated shard.

This implicates that a table (if the database uses a concept comparable to tables) is not stored on one single machine, but in a cluster of nodes. Its advantage is that nodes can be added to the cluster to increase the capacity and the performance of write and read operations without the need to modify the application. It is even possible to reduce the size of a sharded database cluster when the demands decreases.

The downside of sharding is that it makes some typical database operations very complex and inefficient. One of the most important operations in relational databases is the *join* operator, which is used to materialize the relations of data items. A join operation works on two sets of data items, a left one and a right one, which are connected with a pair of attributes. A distributed join in a sharded database would require that the database would have to search in the right set for all items that are associated to each item in the left set. This would require many requests to all machines that store data items from one of the two sets, which would cause a lot of network traffic. Because of this, most sharded databases do not support join operations.

The more nodes are used inside a sharded database cluster the more is the probability increased that one of the machines or one of the network connections fails. Therefore is sharding often combined with replication, which makes the cluster more robust against hardware failures.

2.2 The ACID Properties

One very important feature of relational database is their ability to ensure the **ACID** properties for transactions. ACID is an acronym for Atomicity, Consistency, Isolation and Durability [49].

Atomicity means for a transaction that all included statements are either executed or the whole transaction is aborted without affecting the database.

Consistency means that a database is in a consistent state before and after a transaction. If the changes of a transaction violate a consistency rule then all changes of the transaction must be revoked to ensure that only valid data is written to the database.

Isolation means that transactions can not see uncommitted changes in the database. This makes transactions unaware of other concurrently running transactions on the system.

Durability requires that changes are written to a disk before a database commits a transaction so that committed data cannot be lost through a power failure. In relational databases this is usually implemented with a write ahead log file. The log file allows the database to redo transactions that were committed but not applied to the database.

Atomicity, Consistency and Isolation are usually implemented in RDBMS using a central lock-manager. The lock-manager allows transactions to get a lock on the data they want to read or modify. If a transaction attempts to access data that is already locked by another transaction it has to wait until the other transaction releases the lock.

For a distributed system a central lock-manager would be the bottleneck, because all database nodes would need to contact the lock manager for every operation. This is why some NoSQL databases are using optimistic replication with Multiversion Concurrency Control, see section 2.5.

2.3 Eventual Consistency

Later in this chapter the CAP theorem will be introduced, which has the implication that distributed databases can either be strongly consistent or available.

Consequently, most of the NoSQL databases can only provide eventual consistency, a weaker type of strong consistency. To explain the different types of consistency the following terminology is used:

A, B, C	Unrelated processes that intend to read or modify the database.
x	The data item x
$x1, x2, x3$	Different values of the item x in the database.
$write(Item, Value)$	A write operation of a certain process on the database.
$read(Item) = Value$	A read operation of a certain process on the database.

Strong consistency means, that all processes connected to the database will always see the same version of a value and a committed value is instantly reflected by any read operation on the database until it is changed by another write operation, see figure 2.1.

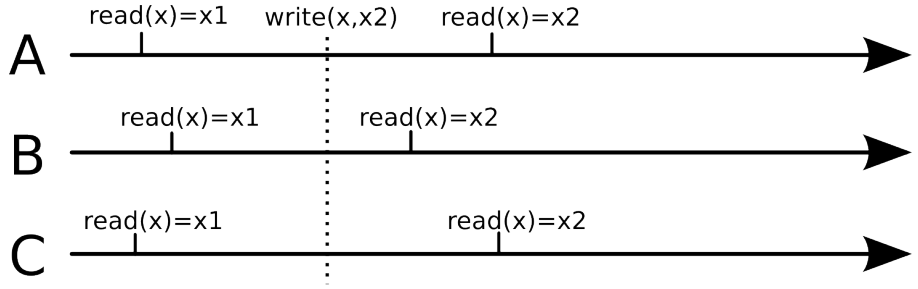


Figure 2.1: Strong consistency: The processes A, B and C are always seeing the same version of the data item.

Eventual Consistency is weaker and does not guarantee that each process sees the same version of the data item. Even the process which writes the value could get an old version during the inconsistency window. This behavior is usually caused by the replication of the data over different nodes [48].

When process A writes $x2$ it takes some time until the new value is replicated to each node responsible for this data item.

If the database would wait with the return of the write operation until the new value is completely replicated, then it would blockade the writing process A and could cause bad latencies for the client. Furthermore, it could happen that not all nodes are accessible because of connection problems or hardware failures, which could blockade the whole application. This alone would not completely solve the

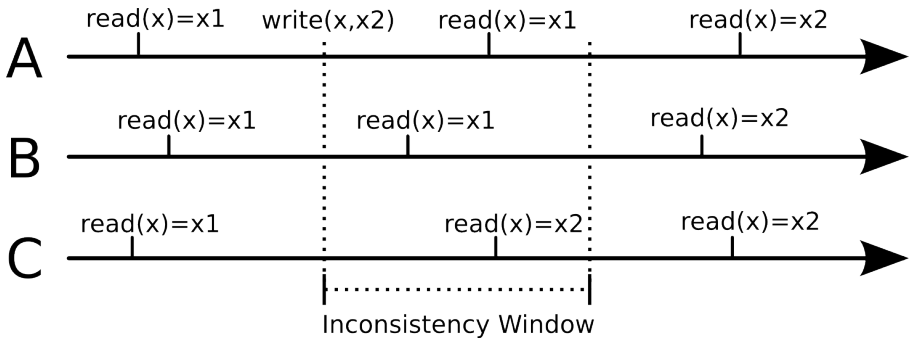


Figure 2.2: Eventual consistency: The processes A, B and C can see different versions of a data item during the inconsistency window, which is caused by asynchronous replication.

problem that the state of the nodes would be inconsistent during the replication, unless something like the two phase commit protocol is used [45].

Some distributed databases can ensure that a process can always read its own writes. For this, the database has to connect the same process always to nodes that already store the data written by this process. Figure 2.3 shows the behavior of a system that ensures *read-your-own-writes consistency*.

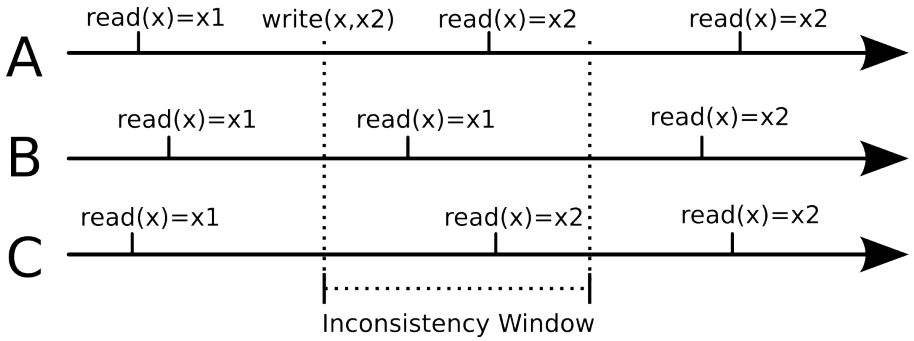


Figure 2.3: Read-your-own-writes consistency: The writing process A always reads its new version of the updated data item x, while other processes might see an older version during the inconsistency window.

A subtype of read-your-writes consistency is *session consistency*. Thereby it is only guaranteed that a process can read its own written data during a session. If the process A starts a new session, it might see an older value during the inconsistency window, see figure 2.4.

Another variant of eventual consistency is *monotonic read consistency*, which assures that when a newly written value is read the first time, all subsequent reads on this data item will not return any older values, see figure 2.5. This type of

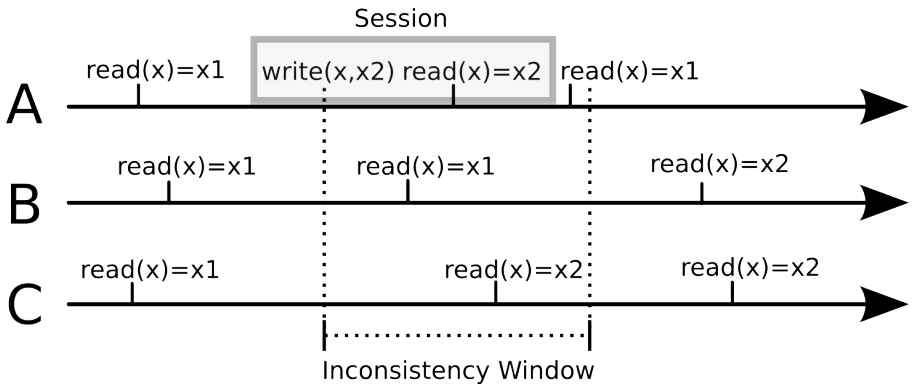


Figure 2.4: Session consistency: The writing process A sees all updates that have occurred during a session

consistency allows the database to replicate newly written data, before it allows the clients to see the new version.

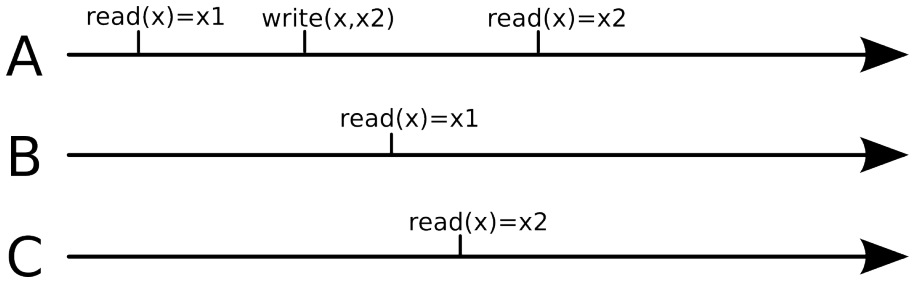


Figure 2.5: Monotonic read consistency: All processes alway see the latest version of a data item that was read from the database.

2.4 The CAP Theorem

The CAP Theorem was introduced by Dr. Brewer in a keynote [7] addressing the trade-offs in distributed systems and was later formalized by Gilbert and Lynch [29]. It states that in a distributed data storage system only two features out of availability, consistency and partition tolerance can be provided.

Availability means in this case that clients can always read and write data in a specific period of time. A partition tolerant distributed database is failure tolerant against temporal connection problems and allows partitions of nodes to be separated.

A system that is partition tolerant can only provide strong consistency with cutbacks in its availability, because it has to ensure that each write operation only finishes if the data is replicated to all necessary nodes, which may not always be possible in a distributed environment due to connection errors and other temporal hardware failures.

To describe this in more detail, the following terminology is used:

N	The number of nodes to which a data item is replicated.
R	The number of nodes a value has to be read from to be accepted.
$W3$	The number of nodes a new value has to be written to before the write operation is finished.

To enforce strong consistency $R + W > N$ must be complied to ensure that a read operation always reads from at least one node with the current value [48].

Figure 2.6 shows an example configuration with $N = 3$, $R = 2$ and $W = 2$. The new value $x2$ is written to two nodes and read from two nodes, which results in an overlap at one node, so that the read operation always has a consistent result.

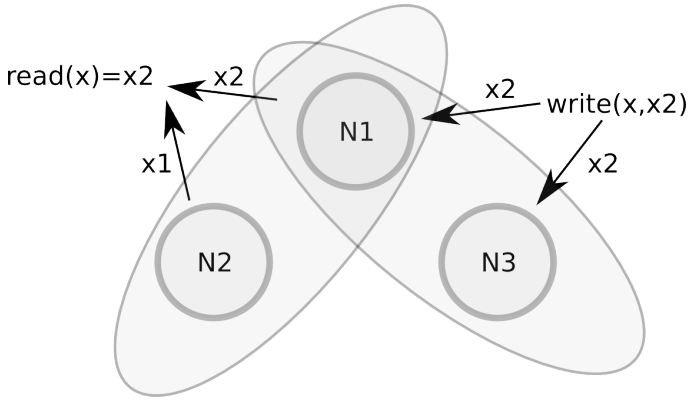


Figure 2.6: If a data item has to be written to at least two of three nodes and needs to be read from at least two, then one of the nodes from which the data item is read must contain the latest version.

But the situation changes if the nodes are separated into two partitions, see figure 2.7. This could for example, happen through a connection problem. In this case the database has two options to deal with the problem: Either disallow all write or read operations that could cause inconsistencies or allow inconsistencies to enforce availability. In the second case the database needs a way to resolve the inconsistencies

when the nodes are not separated anymore, which is usually done with multiversion concurrency control (MVCC), which will be explained in the next section.

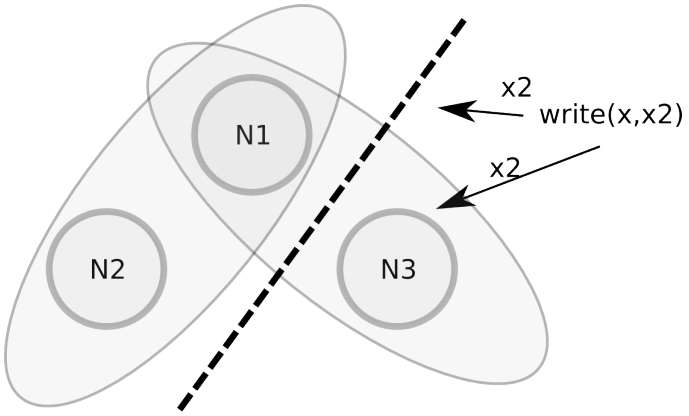


Figure 2.7: If the nodes in the example configuration are separated into two partitions, the databases becomes unavailable.

Systems with eventual consistency only enforce $R+W \leq N$, to be partition tolerant and available. Thereby some systems can be tuned with different values for R and W to customize the database for different usage scenarios. A big R and a low W is good for applications, which write more than they read. And a low R and a big W are good for applications with more read operations. Lower R and W enhance the availability, but reduce the consistency, so systems that strictly provide optimistic replication have $R = 1$ and $W = 1$.

2.5 Multiversion Concurrency Control (MVCC)

MVCC is an efficient method to let multiple processes access the same data in parallel without corrupting the data and the possibility of deadlocks. It is an alternative to the Lock based approaches, where every process first has to request an exclusive lock on a data item, before it can be read or updated. MVCC is used in some relational databases as well as in most distributed databases. It was first described in a dissertation by Reed [44] and was introduced to the database world through the implementation of InterBase [47].

Instead of letting each process access the data exclusively for a certain amount of time, MVCC allows processes to read the data in parallel, even if a process is updating the data. To maintain consistency, each data item has some kind of time stamp or revision. If a process reads a data item, it does not only get the value of it, the process also retrieves the revision of the data item. So if this process attempts

to update this data item, then it writes the new value with the previously read revision number to the database. If the actual revision in the store is the same, then the new value is written and the revision of the data item is incremented. But if the revision in the store is not the same as the revision read by the writing process, then there must have been another process which has updated the data item in the meantime. In a relational database, the writing process would be a transaction, which would in this case be aborted or restarted. Figure 2.8 shows an example for the creation of a conflict in a MVCC system.

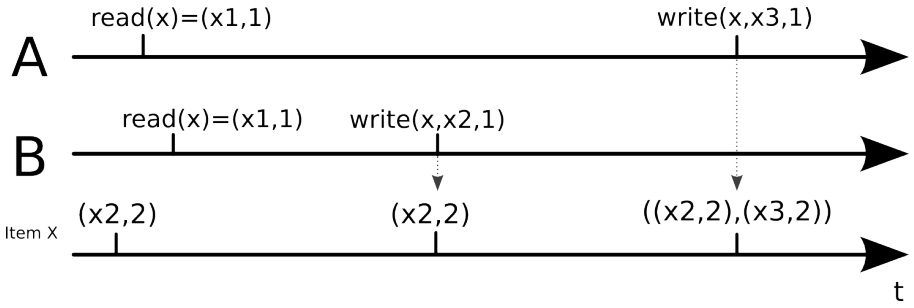


Figure 2.8: This figure shows the creation of a conflict in a database with MVCC. Process A and B are both trying to write a new value to the data item x. Both read at first item x including its current revision number. Process B is the first to write its new version to the database. Process A tries to write a new version of x based on an older version than the current version in the database and thereby generates the conflict.

In distributed databases, there are at least two cases for such a conflict: The first one is that two processes are attempting to write the same data item on the same node. In this case, the database could detect the conflict during the write operation and abort it, so the client would need to reread the data item and retry its desired update, like the behavior of a RDBMS in this situation.

Another case is that multiple clients update the same data item on different nodes. If the distributed database uses asynchronous replication, then this conflict can not be detected and handled during the write operations. The nodes first have to be synchronized before they can handle the conflict. The conflict resolution can happen during the replication or during the first read operation on the conflicting data item.

Some databases, which implement this, store all conflicting revisions of the data items and let the client decide how the conflict should be handled. In such systems a read request returns all conflicting versions of the value and the client has to choose one or has to merge the versions and to write the corrected revision back to the database.

2.5.1 Revisions in distributed systems

For optimistic replication it is necessary to determine if multiple versions of the same data item were created in parallel or in a serial order. Furthermore, the database needs to know which version the newest is. A simple solution for this would be to use timestamps, but this would require that all nodes are time synchronized and it could happen that two nodes write the same item exactly at the same time. To avoid this problems some distributed databases use time independent mechanisms to track the versions of the data. Two of them are described in the following.

2.5.2 Vector Clocks

Vector Clocks are based on the work by Lamport [40] and are used by several databases to determine if a data item was modified by competing processes. Each data item contains a vector clock, which consists of tuples with a separate clock for each process which had modified the data item. Each clock starts at zero and is incremented by its owning process during each write operation. To increment its own clock, the writing process uses the maximum of all clock values in the vector and increments it by one. When two versions of an item are merged, the vector clocks can be used to detect conflicts by comparing the clocks of each process. If more than one clock differs, there must be a conflict. If there is no conflict, the current version can be determined by comparing the maxima clocks of the competing versions.

Figure 2.9 shows an example for Vector Clocks including the data item x and the two processes A and B. In the beginning the item x holds the Vector Clock ($[A, 0]$), because it was initially written by process A. Process B updates the data and starts its own clock for the item. Since the last maximum version of x was 0, the process sets its own clock in x to 1. Because A and B were writing to the item serialized, there is no conflict detected. In the next step both processes are writing in parallel to x. Parallel does not necessarily mean that the processes are writing to x at the same time. It can also be that they write to different replicas, which may be synchronized at a later time. Both processes increment the vector clock in the way previously described. At the next synchronization, the two vector clocks have differences at two positions, so there is a conflict detected, which then needs to be handled by the client.

The disadvantage of vector clocks is that they get bigger with each process that writes to a data item.

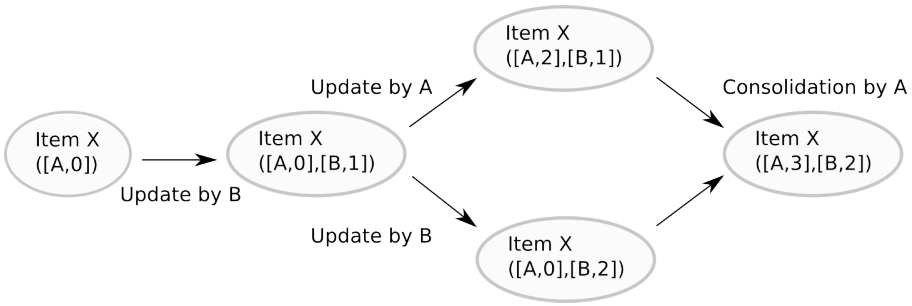


Figure 2.9: Conflict detection with vector clocks.

2.5.3 Hash Histories (HH)

Hash Histories are described in the paper by Kang, Wilensky and Kubiatowicz [37] and have the advantage that the version size is independent of the number of nodes performing write operations on a data item. However hash histories grow with the number of updates. A HH contains a list of versions of the item. Each version consists of a hash value calculated from the content of the data item and a revision number. The revision number is initially zero and then incremented during each write operation.

The order of write events can be determined by comparing the histories. If the history of two competing versions contains entries with the same revision number but an unequal hash value, then there must be a conflict. Figure 2.10 shows the example from above for Hash Histories.

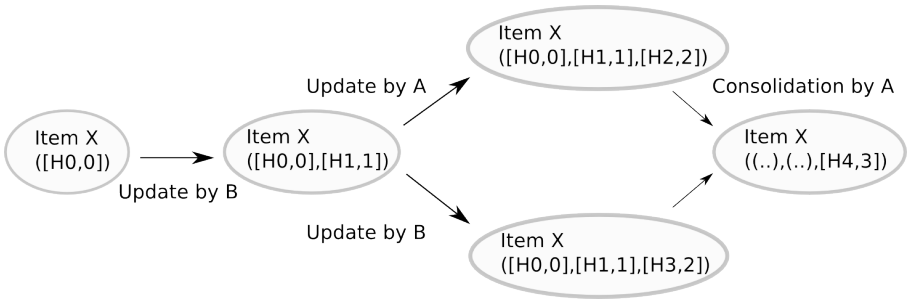


Figure 2.10: Conflict detection with hash histories.

One positive side effect of using a hash function for revision identification is that false conflicts can be detected. This is if two processes make exactly the same changes to the same revision of an item then no conflict is created.

2.6 MapReduce

MapReduce is a programming model for distributed computation developed by Google and is described in a paper by Dean and Ghemawat [25]. Applications that are written inside the MapReduce framework can automatically be distributed across multiple computers without the need for the developer to write custom code for synchronization and parallelization. It can be used to perform tasks on large datasets, which would be too big for one single machine to handle.

In order to instrument a MapReduce framework, the developer has to specify a map function and a reduce function. The map function is invoked with a key/value-pair for each data record and returns intermediate key/value pairs, which are used by the reduce function to merge all intermediate values with the same intermediate key. So each reduce invocation gets an intermediate key and a list of values belonging to the key and returns a smaller data set or a single value as final result.

Google's MapReduce framework has the following method signatures:

$$\begin{aligned} map(key, value) &\rightarrow list(ikey, ivalue) \\ reduce(ikey, list(ivalue)) &\rightarrow list(fvalue) \end{aligned}$$

Thereby $(key, value)$ is one record of input data and $(ikey, ivalue)$ an intermediate key/value pair, and $fvalue$ a final result value.

2.6.1 Example

The MapReduce paper [25] gives a simple example written in pseudo-code for a MapReduce job, which takes a list of documents and counts the occurrences of each word, see listing 2.1.

The map function is called for each document and emits a key/value pair for each word in the document. This result is forwarded to the reduce function, which is called for each word in this example and gets a list of occurrences, which are summed up and then returned as a single value.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Listing 2.1: Word counting example for a MapReduce task.

2.6.2 Architecture

The paper does not only describe the MapReduce programming model, it also describes Google's implementation of a MapReduce framework, which is optimized for a cluster consisting of computers build with commodity hardware. The described framework has the requirement to be able to handle failures of single machines.

It consists of a master, which assigns the map and reduce tasks to the available nodes and is responsible for the failure management. The framework is designed to work together with GFS, Google's distributed storage system. Figure 2.11 shows the execution schema of a map reduce task.

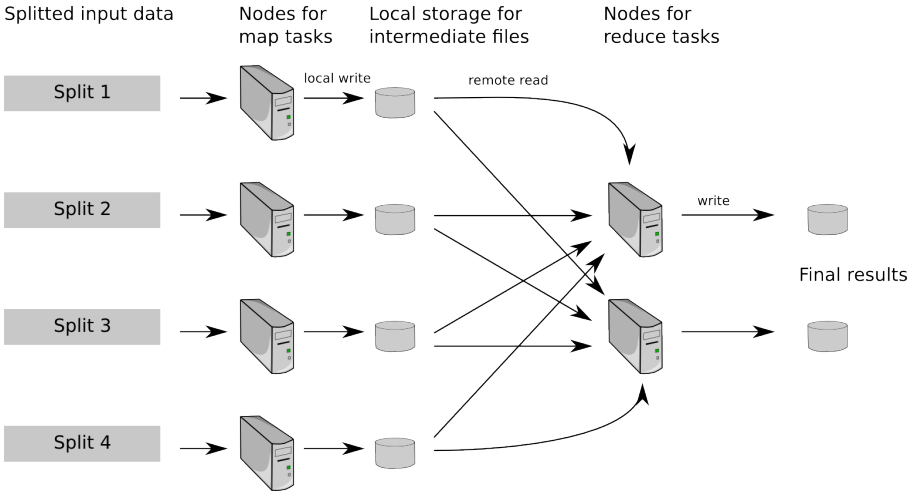


Figure 2.11: Overiew of a MapReduce process.

Each node assigned with a map task gets an own partition of the input data. The results of the map function is stored on the local disk of the map node and ordered by the intermediate keys. The intermediate key-space is also split by the master and distributed over the reduce nodes. To get the intermediate data each reduce node has to ask the master for the locations of intermediate data with their assigned intermediate keys and read it from the responsible map node.

2.6.3 Extension

The shown architecture has one big flaw: For some tasks the reduce nodes would have to read massive amounts of data over the network. For instance, the shown example for counting word occurrences would produce a key-value pair for each single word occurrence, which would need to be transferred over the network to the responsible reduce node.

That is why the MapReduce framework has an additional method to map and reduce: A Combiner function, which can be invoked between the map and the reduce function to partially merge the intermediate data on the map node. The combiner function is usually the same as the reduce function, but produces intermediate data and no final results. This requires, that the reduce function is associative and idempotent.

This would mean for the example that the word occurrences would first be counted for each data split and then the reduce function would only have to sum up the intermediate results. So instead of transferring a key/value pair for each word occurrence over the network, only one key/value per word and data split would need to be remotely read by the the reduce nodes in the worst case.

2.6.4 Advantages

The MapReduce framework simplifies the development of distributed algorithms by hiding the actual distribution from the developer. But using the MapReduce framework alone does not guarantee that the performed task is maximal parallelized. It is necessary for the developer to think about, which parts of a task can be more efficient solved with the usage of a combiner function. When this is done, the MapReduce framework is able to minimize the required network traffic drastically.

2.6.5 Implementations

Aside from Google's own MapReduce implementation there are several other ones. The most popular implementation in the open source world is Hadoop [13] with its own distributed file system HDFS.

MapReduce is also implemented with some variations in various NoSQL databases to perform queries or to build indexes.

3 NoSQL Databases

This chapter introduces a selection of NoSQL databases with different capabilities and purposes. The first three examined databases Dynamo, S3 and SimpleDB are from Amazon. Dynamo is only a key-value store without any range query capabilities and is only used internally at Amazon and would therefore be not of much interest for this work. But Dynamo uses some very interesting technologies to achieve a robust peer to peer architecture, which is described in the Dynamo paper [26]. This paper had a great influence in the NoSQL community and describes a lot of ideas by which other development teams were inspired, because of this is Dynamo presented in this chapter but is not part of the comparison in the next chapter.
SimpleDB was chosen, because it is part of Amazon's cloud service, which is currently one of the biggest. Since SimpleDB has some limitations, which require to use Amazon's Simple Storage Service S3 in combination with it, it is also briefly presented even though S3 is also only a key-value store.

The next two examined databases BigTable and App Engine datastore are from Google, which makes them interesting because Google is obviously able to scale with them. BigTable is not directly usable by other than Google, but there are several publicly available databases, which implement the ideas described in the BigTable paper. The App Engine datastore is interesting because of two things: First it is the only database that can be used inside Google's Software as a Service platform App Engine and is therefore interesting for everyone who intends to develop an application for this platform. The second fact that makes it interesting is that the Datastore uses BigTable only as a key-value store with range query capabilities and provides nevertheless many features and query capabilities that go beyond this. So the Datastore is a good example for how a database with only some primitive query features can be used for more complex tasks.

The last two databases MongoDB and CouchDB are the only ones in this survey which are open source and are not only usable as a service. They also have in common that they are document oriented and schema free. Both projects have gained a lot of momentum in the last years and are used in production by large internet services. However, they have slightly different objectives. CouchDB aims more for data integrity and optimistic replication whereas MongoDB is built for high performance.

For a better comparison, the databases in this chapter are analyzed with several aspects in mind. The first one is the **Data Model**, which defines how a database stores data and how it handles concurrent access. The second one is the **Query Model**, in which the surveyed databases differ the most. This category contains the power of the used query language, its restrictions and how it can be used.

The next two examined facets are **Sharding** and **Replication**, to determine how the databases can be distributed other many machines. And in the **Consistency** category is analyzed, which consistency level the databases achieve and which trade-offs they are making.

Some of the technical details of the implementations are taken care of in the **Architecture** sections. Another important aspect is how the databases can handle various types of failures, like single node failures and network problems, this is discussed under the term **Failure Handling**.

Some databases in this survey are not examined under all this aspects, due to the lack of available information or a limited feature set.

There is currently a lot of development in the NoSQL area, therefore there are many other projects which are not described in this work.

3.1 Dynamo

Dynamo is a distributed key-value storage system that is used internally by Amazon for its own services [26]. It provides a simple query API which allows clients to retrieve a value for a unique key and to put key-value pairs into the storage with values smaller than one megabyte.

The system is designed to work in a network of nodes build from commodity hardware and assumes that each node and every network connection can fail anytime. This network of nodes can incrementally be enhanced with new nodes and allows nodes with different capacities to be added without manual partitioning. The workload is distributed proportional to the capabilities of the nodes. Each node in this system has the same responsibilities so there are for example no dedicated nodes for routing, replication or configuration. To achieve immunity against disasters affecting complete data-centers, every key-value pair is replicated with a geological distribution over several data-centers around the world.

Dynamo can be tuned to achieve different trade-offs between availability, consistency, cost-effectiveness and performance to be suitable for different kinds of services. Like some other distributed databases, Dynamo uses optimistic replication with multiversion concurrency control (MVCC) to achieve a type of eventual consistency.

To meet all these requirements, Dynamo utilizes a mix of existing technologies from distributed databases, peer to peer networks and distributed file systems such as consistent hashing for replication and key partitioning.

3.1.1 Query Model

Dynamo offers two API functions to access the store:

get(*key*): Returns an object with the value represented by this key or a list of conflicting versions and context objects for each value. A context object contains the metadata for a value including its revision.

put(*key*, *context*, *value*): Saves a value attached to a key in the storage. The *context* is needed for Dynamo to be able to determine if the value is based on the current revision stored in the database.

3.1.2 Sharding

The data is partitioned in Dynamo with a variant of consistent hashing. Each node is subdivided into a number of virtual nodes, which are associated with a random position in the key space. The key space is a ring of numbers into which every key can be mapped with a hash function. A node is responsible for all keys between its own position and its predecessor. For example in the ring of figure 3.1 the virtual node B is responsible for the key k.

The usage of virtual nodes has the advantage that each real node can be associated with a number of virtual nodes depending on its capacity and other capabilities to allow heterogeneity of the underlying hardware.

3.1.3 Replication

Dynamo allows to specify how many replicas a key should have with the number N. The responsible node for k replicates k to its $N-1$th successors. In Figure 3.1 B is responsible for the replication of k to C, D. So each node has to store all keys between their own position and their Nth predecessor.

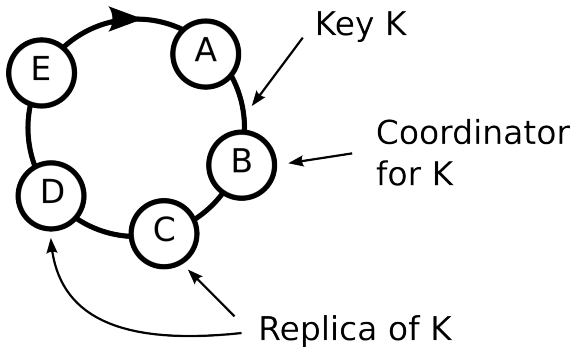


Figure 3.1: The replication schema of Dynamo.

Dynamo uses MVCC to allow the asynchronous synchronization between the replica. Each update generates a new revision. If this generates a new branch, then the *get*() operation will return a list of the conflicting revisions to allow the application layer to merge the different branches. The revision control in Dynamo is implemented with vector clocks.

3.1.4 Consistency

Dynamo uses a configurable quorum like approach. The client can specify from how many replicas R a key must be read and how many replicas W must be produced during a write. With $R + W > N$ this would be equivalent to a quorum system with strong consistency, but the clients can also set $R + W < N$ in favor of latency and availability over consistency.

3.2 Amazon S3 Simple Storage Service

S3 is a distributed and scalable key-value storage system comparable to Dynamo. But unlike Dynamo, S3 can be used by third parties. It can store values up to a

size of 5 GB. Simple Storage organizes the data into Buckets. A Bucket is used as a container or a name space for the data. The client can specify a geological location of a bucket (currently either U.S. or Europe) and can specify access restriction for it [9].

All data in S3 is replicated to be save against failures of single nodes. Like most distributed databases it assumes that every node and network connection can fail any time.

3.2.1 Query Model

Because S3 is a key-value store it has a map like query interface:

Create Bucket:	Creates a Bucket with a unique name.
Delete Bucket:	Deletes a Bucket.
List Keys:	Lists all keys stored inside a Bucket.
Read:	Reads the data belonging to a unique key from a Bucket.
Write:	Writes the data associated with a unique key into a Bucket.
Delete:	Deletes an key and its data from the storage.

3.2.2 Consistency

In S3 a write operation is atomic, but there is now way to make atomic write operations over multiple keys. It does not provide a locking mechanism, if there are concurrent writes on the same key, then the last one wins.

After the commit of a write or a delete operation it may take a while until the changes are reflected by the results of the read and list keys operations, because S3 first applies the changes to all replicas and lets them invisible until all data for the key is replicated.

3.3 SimpleDB

SimpleDB is a schema free and distributed database provided by Amazon as a web service. Its main feature is its ease of use: SimpleDB does not need a schema,

decides automatically which indexes are needed and provides a simple SQL like query interface.

All data stored in SimpleDB is replicated onto different machines in different data centers to ensure the safety of the data and to increase the performance [10]. It does not support automatic sharding and consequently can not scale with the amount of data, unless the application layer performs the data partitioning by itself.

3.3.1 Data Model

SimpleDB lets the client organize the data into domains, which can be compared with tables in relation databases, with the difference that a domain can contain a different set of attributes for each item. All attributes are byte arrays with a maximum size of 1024 bytes. Each item can have multiple values for each attribute. Due to restrictions in the Query Model it is impossible to model relations of objects with SimpleDB without creating redundant information. So the developer can either denormalize its data or handle relations in the application layer.

Queries can only be processed against one domain, so if a client needs to aggregate data from different domains, this must also be done in the application layer. But it would be unwise to put everything into one domain, because domains are used in SimpleDB to partition the data, which results in restrictions in the size of one domain. Furthermore, the query performance is dependent on the size of such a domain.

3.3.2 Query Model

SimpleDB allows the client to retrieve data with a SQL like select statement against one single domain with the following format:

```
select output_list or count(*)
from domain_name
[where expression]
[sort_instructions]
[limit limit]
```

output_list:	The attributes which should be retrieved by this query. If replaced by *count*(∗), the query will return the number of items that would be in the result.
domain_name:	The name of the domain.
expression:	The conditions items must fulfill to be in the result list. An expression can be a comparison of an item attribute with a constant value or expressions connected by logical operators (or, and, not).
sort_instructions:	The name of one attribute and the sort direction: Either asc for ascending or desc for descending.
limit:	The maximum number of items to return.

The Query Model of SimpleDB does not allow joins and no comparisons of items inside a query that is why all comparisons are restricted to be between one attribute and one constant value. This restrictions may simplify the automatic generation of indexes and could allow SimpleDB to parallelize the queries, but it is unknown to the public how SimpleDB works internally.

Because SimpleDB only supports byte array attributes, the sorting is lexicographic. This must be accounted for when numbers or dates are used for sorting.

3.3.3 Consistency

SimpleDB provides eventual consistency, but without MVCC, so there is no way to detect conflicts with SimpleDB on the client side.

3.3.4 Limitations

SimpleDB has currently some restrictions for the size of domains, query time, and number of domains. These limits may change during the lifetime of the service.

Parameter	Limitation
Domain Size	10 GB per domain / 1 billion attributes per domain
Domains per Account	100
Attribute value length	1024 bytes
Maximum items in Select response	2500
Maximum query execution time	5s
Maximum response size for one *Select* statement	1MB

For a complete and up to date list see [11].

3.3.5 Example: Simple Forum

This example is intended to show how a simple forum could be implemented with SimpleDB in combination with Amazon S3. The example forum organizes the posts into categories and threads.

Domain ForumPosts:

ID	Category	Thread	User	Time
23	Cars	Motors	Tom	
33	Cars	Motors	Joe	
44	Boats	Fishing	Bob	
55	Boats	Fishing	Alice	
34	Cars	Gears	Joe	
45	Planes	Wings	Alice	

Because of the limitations for the size of the values, the contents of the posts are not stored in SimpleDB but in Amazon S3, so the application has to use the Item ID as a key for S3 to store and retrieve the content. Because the two databases may not always be consistent, the application needs to be fault tolerant to posts with contents that are not ready to be retrieved from S3.

The example uses a denormalized structure to avoid joins in the application layer. If a Category, Thread or User changes, all affected rows must be updated. SimpleDB allows to change attributes only for single items, so the applications layer would need to load all affected rows, change them and then write them back to the database. This operation would be very expensive and would leave the database

in an inconsistent state in the meantime, because only each single PUT operations would be atomic. So changes of the Category, Thread or User should either be not allowed or the application would need another data structure, which would require joins in the application layer and more requests to the database.

To allow the names of Categories,Threads and User to be changed, they need to be stored in separate items and referenced in the domain ForumPosts by their ID. This would require the application to make one additional request for each Post to get the author name. The Categories and Threads might also be requested very frequently, so it would be a good idea to cache them, which would reduce the cost of client side joins for them.

To decide which model fits best, the developer needs to know how often the values of certain attributes might be changed to be able to determine if the expensive changes of redundant data might be amortized by the fewer requests needed to retrieve the desired information.

3.4 BigTable

BigTable is one of Google's solutions to deal with its vast amount of data. It is build on top of Google's distributed file system GFS [28] and is used by Google for several applications with different demands on the latency of the database. Some of them require near real time behavior to satisfy the users of Google's web services and other ones are more throughput oriented batch applications [20].

BigTable is intended to be scalable, distributable and therefore tolerant to hardware failures. Google's own implementation is not publicly available, besides the usage inside Google's App Engine, but there are some open source implementations of BigTable, like Hypertable [16] and Hbase [14], which is based on Hadoop [13].

Another interesting implementation of the ideas described in the BigTable paper is Cassandra [12], which uses approaches from both BigTable and the Dynamo paper [26]. BigTable is furthermore the basis for the Datastore in Google's App Engine.

3.4.1 Data Model

Basically BigTable is comparable to a key-value store, but instead of mapping one key to one value, BigTable maps a tuple consisting of a row key, a column key and

a timestamp to a value. A row in Google's BigTable consists of a row key and a variable number of columns. Each column contains a version history of its content, which is ordered by a timestamp. The rows are stored in lexicographic order of the row key and are used to partition a table for the distribution to different machines.

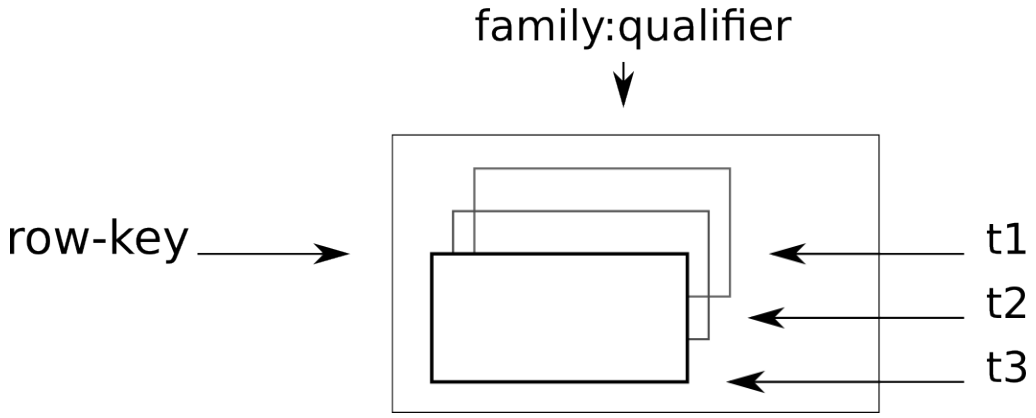


Figure 3.2: A value in BigTable is referenced by its row key, the column key consisting of the column family name and the qualifier and a timestamp. Each row can have any number of columns.

Columns are organized into column families, which is comparable to the concept of having a list or map inside of each column. BigTable is so designed that every row can have any number of columns but it assumes that the number of column families is limited for a table and that the names of the column families are static. Figure 3.2 shows the structure of a BigTable row with one column. A column key in BigTable has the syntax *"family:qualifier"*, thereby the qualifier is the key of the column inside its family.

The access control of BigTable is managed on the column family level, which allows to have columns with different access restrictions for different applications inside a row. BigTable only differs between two data types: Strings for the values and key names and int64 for the timestamps of the version history.

To reduce the number of stored versions of a data cell, it can be defined that either only the last n versions are stored or all versions inside a certain time range.

3.4.2 Query Model

BigTable creates indexes over the row-key, the column keys and the timestamps of the version histories. BigTable does not support any kind of joins, so relations need to be handled by the application layer, if needed.

BigTable provides a C++ client library, which can be used to make queries against a BigTable. It provides an abstraction called scanner, which lets the application define several filters for a row key, column families, column key and timestamps and allows the iteration over the results.

The architecture of BigTable requires that each query restricts the number of the rows on which the scanner has to look at, because the row key is used to determine the locations of the necessary tablets, which are explained in the following subsection. The scanner allows to filter the row key by defining prefixes and ranges for it. The data model implies that the results are ordered by the row key, therefore it is not possible to order the results by different columns.

In addition to the client library of BigTable there are bindings for Google's MapReduce framework, which allows to use BigTable as data source and target for MapReduce tasks.

3.4.3 Sharding

The tables inside a BigTable databases are partitioned by their row-key into several key-ranges, which are called *tablets* in BigTable terminology. Each tablet is assigned to only one tablet server at a time. A master server stores the meta information about the current assignment of each tablet and assigns the currently unassigned tablets to currently available tablet servers.

A tablet server can lose its assigned tablets, if the tablet server goes down or the network is partitioned due to network failures or other events.

Client requests for accessing a certain key-range require to lookup the meta information from the master server, to avoid having a bottleneck at the master server, the client library caches the meta information and only updates it, when a tablet can not be found by the client under its last known location. This takes a lot of load from master server and makes the cluster more robust against temporal unavailabilities.

3.4.4 Replication

BigTable does not directly replicate the database because one tablet can only be assigned to one tablet server at a time, but it uses Google's distributed file system

GFS [28] for the storage of the tablets and log files, which handles the replication on the file level.

3.4.5 Consistency

Since each tablet is only assigned to a single tablet server, BigTable can provide strong consistency and therefore sacrifices availability due to the eventual restricted accessibility of tablets during the recovery from the loss of a tablet server.

BigTable allows to make atomic operations on one row and provides the ability to make transaction inside a row. But there is no way in BigTable to make transactions over multiple rows.

3.4.6 Architecture

BigTable uses a kind of master oriented Architecture. The master in a BigTable cluster is responsible for the assignment of the tablets to the tablet servers. A tablet server can loose its tablet assignments, if it looses its exclusive lock in Google's distributed lock manager Chubby [17]. The Master monitors the state of the lock files of each tablet server and reassigns the tablets of the server, which have lost their lock. Figure 3.3 shows an overview of a BigTable cluster.

The Chubby Service consists of five servers and uses the quorum based protocol Paxos [39] to solve the distributed consensus problem. Chubby lets its clients store small files and provides exclusive locks for them. BigTable uses Chubby for the election of a master server and the tablet servers use it to find their master. This makes chubby a possible single point of failure.

The meta-data with the information about the location of the tablets in GFS and their current assignments is stored in a tree with three levels. The first level consists of a Chubby file with the meta-data of the root tablet, which contains the meta-data about the tablets that store the actual meta-data of the BigTable. The BigTable client library caches the meta information and only refreshes them if it discovers that the cache locations are outdated.

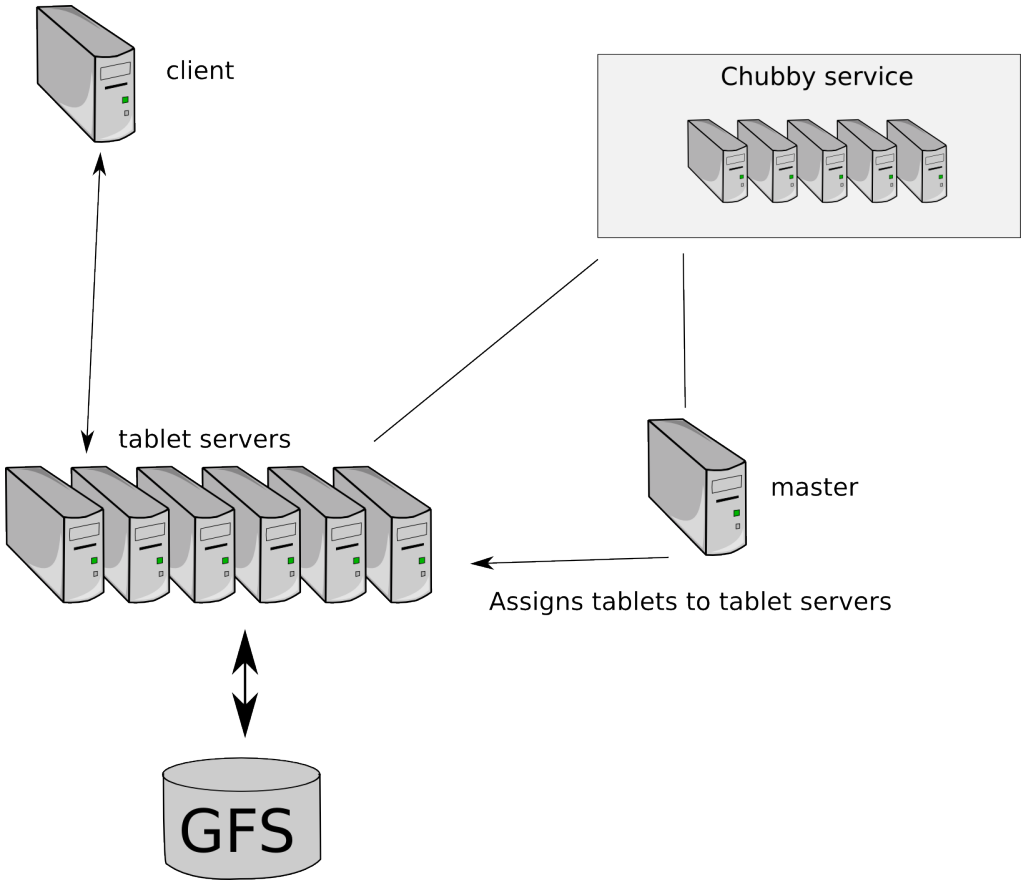


Figure 3.3: The Architecture of a BigTable cluster.

3.4.7 Failure Handling

If a tablet server fails it loses its exclusive lock in the chubby service, this is detected by the master server, which reassigns the tablets of the lost tablet server to the currently available servers. Because the tablets are stored in GFS, the tablet server can read their newly assigned tablets from it, but it is not ensured that all write operations that were committed on the failing tablet server were already written to the tablet files. To solve this problem BigTable uses log files for all write operations which are comparable to the transaction log files of relational databases. Each tablet server writes its own log file to GFS, hence during the recovery phase each tablet server that is assigned with a tablet of a previously lost tablet server has to read its log file and search it for write operations on the tablets that are now assigned to the new responsible tablet server.

Another failure scenario is that the master fails. If this is the case, then the tablet servers elect a new master server using the Chubby service and restore the meta-data tablets using the root tablet file in the Chubby service.

So the real single point of failure in BigTable is Chubby. If the Chubby service fails, then the tablet server will not be able to coordinate themselves. A temporary failure in the Chubby service is not a problem, as long as no tablet server or the master server fails in this time period and no new tablet server needs to be added to the cluster. To ensure the availability of the Chubby service, it consists of five servers and it requires at least three of them to get an exclusive lock on a Chubby file. Because of that it would require more than two servers to fail to break the Chubby service.

3.5 Google App Engine datastore

The Google App Engine datastore provides applications that are running inside Google's App Engine with a queryable and schema free datastore. Google's App Engine is a service that lets third party applications run on Google's infrastructure inside a sandbox with some restrictions [30].

The datastore is built on top of BigTable and simplifies the usage of it by hiding the denormalization that is needed to use BigTable for queries against different attributes of the same object. Furthermore, the datastore provides transactions over multiple entities. To simplify the usage of the datastore even more, persistence APIs for Java and Python are built into the App Engine sandbox.

3.5.1 Data Model

The datastore uses a schema-less data model and supports all primitive data types of the supported programming languages. Objects in the datastore are stored in a serialized form inside a BigTable with their identifier as row key. This makes the data model of the datastore very similar to the one of the document oriented databases.

The datastore supports transactions over multiple entities. This requires all objects in a transaction to be inside the same entity group. An entity group consists of a root entity, which has other entities as children. Each newly created entity which is not added to another entity group is per default the root object of its own entity

group. If an entity is supposed to be in a entity group then it is necessary to add the entity to the desired entity group before it is written to the database, because this can not be changed afterwards.
An entity is added to an entity group by setting the entities parent to an entity of the group. Because of this, every entity has a path of ancestors, which is used by the datastore as a prefix for the row key of the entity in the BigTable. This has the effect that entities with the same parents are stored near each other, which allows the datastore to efficiently manage transactions inside an entity group [15]. The entity groups can be used to model all relations that can be described as a tree.

3.5.2 Query Model

Depending on the used programming language, the datastore can be queried with either the query language GQL for Python programs or JDOQL for programs that use the Java JDO API. Both are very similar to SQL, but they provide only a subset of the features provided by relational databases [33].

The datastore query APIs allows to define query conditions over different attributes, but does not allow to compare entity properties with each other. The conditions inside a query can only be combined with a logical AND operator, so if an application has to use an OR operator the queries must be split and the results must be combined by the application itself.

Another restriction is that inequality operators like $<, <=, >=, >, ! =$ can only be used on one property per query. And if a query uses inequality operators and sort orders, then the first sort order must be the property to which the inequality operators are applied to. Sort orders for queries with an equality filter are ignored. These restrictions are necessary because queries in the datastore are materialized views, which are stored inside a BigTable which supports only one key for sorting and filtering [31].

For the same reason it is necessary to define an index for all queries, which use more than one property. Queries over one property do not need an extra index, because the datastore automatically indexes every property that is not marked as unindexed.

The datastore does not have support for any aggregations functions, other than counting the result set of a query. In addition to that, the number of query results is limited to 1000.

3.5.3 Architecture

The datastore uses four BigTables for all applications in the App Engine to store the entities and indexes [31]. The first table is the entity table. Each entity is stored in a BigTable row as a serialized object in Google's Protocol Buffer format [32]. Because all applications share the same BigTables, the row key of an entity begins with the application name, which is followed by the ancestor path and the type and identifier of the entity. For a simple example with the entity types *BlogPost* and *BlogComment* the entity key for a comment with a post as parent would have the form:

$$BlogApplication/BlogPost : ParentBlogPost/BlogComment : CommentId$$

Because the datastore stores the entities in serialized form, the structure of the BigTables do not need be changed to reflect the entity properties. The three other tables used by the datastore are filled with the indexes. Two tables are used for the automatically generated single property indexes. One of these tables is used to store the index in ascending order and one in descending order. A single property index consists only of a row key, which consists of the kind of the entity, the property name and the value. For the blog comment example, an index entry for the property author would look like:

$$BlogApplication/BlogComment/author : AuthorName$$

The fourth table is used for all queries that affect more than one property. These kind of queries are called *Composite Queries* in Google's terminology. To be able to make Composite Queries against the datastore, the developer has to specify an index before runtime which contains all properties and sort orders. When a Composite Query is made, the datastore query analyzer first searches for an index that matches the requirements of the the query and then scans the composite index table only in the range with valid results for the query. This makes querying against the datastore very fast, because the tablet machine that contains the query results can constantly read the results without having an additional seek operations on the hard drive.

An entry for the composite index consisted of the entity type, the entities ancestor type if required and the property values. For example, an index entry which would allow to query for all blog comments of a certain post written by a certain author inside a time range would look like:

$$BlogApplication/BlogComment/BlogPost : ParentBlogPost \\ /author : AuthorName/writeTime : 22.01.2010 : 08 : 20$$

This index allows the query analyzer to transform a query like:

```
WHERE ANCESTOR IS :ParentBlogPost && author=AuthorName && writeTime
    > 12.01.2010:05:20 && writeTime < 12.01.2012:05:20 ORDER BY
    writeTime
```

into a scan from

$$BlogApplication/BlogComment/BlogPost : ParentBlogPost \\ /author : AuthorName/writeTime : 12.01.2010 : 05 : 20$$

to

$$BlogApplication/BlogComment/BlogPost : ParentBlogPost \\ /author : AuthorName/writeTime : 12.01.2012 : 05 : 20$$

3.5.4 Consistency

Like BigTable, the datastore is strongly consistent and therefore has a restricted availability. Because a write operation on an entity requires to write to four different BigTables and to at least one tablet server for each index affecting the entity, many machines and network connections are involved in one write operation, which all can fail. So an application must be able to handle situations in which a certain entity can temporarily not be written.

If this happens, the application can either give the user an error message and stall the current operation or it can queue the write operation to be tried again later. If an application uses the second approach, then it circumvents the strong consistency of the datastore and makes it eventual consistent for this entity, because there are then at least two versions of the same entity until the queued write operation can finally be committed.

```
{
    "_id": "154c48b23c8f6d14efcf3659ed8c7312",  //identifier
    "name": "Hugo",
    "hobbies": [
        "sport",
        "movies",
        "programming"
    ]
}
```

Listing 3.1: Example MongoDB document as JSON object.

3.6 MongoDB

MongoDB is a schema less document oriented database developed by 10gen and an open source community [3]. The name MongoDB comes from "hu**mongo**us". The database is intended to be scalable and fast and is written in C++.
In addition to its document oriented databases features, MongoDB can be used to store and distribute large binary files like images and videos.

3.6.1 Data Model

MongoDB stores documents as BSON (Binary JSON) objects, which are binary encoded JSON like objects [2]. BSON supports nested object structures with embedded objects and arrays like JSON does. Listing 3.1 shows an example for such a document. MongoDB supports in-place modifications of attributes, so if a single attribute is changed by the application, then only this attribute is send back to the database.

Each document has an ID field, which is used as a primary key. To enable fast queries, the developer can create an index for each query-able field in a document. MongoDB also supports indexing over embedded objects and arrays.

For arrays it has a special feature, called "multikeys": This feature allows to use an array as index, which could for example contain tags for a document. With such an index, documents can be searched by their associated tags.

Documents in MongoDB can be organized in so called "collections". Each collection can contain any kind of document, but queries and indexes can only be made against one collection. Because of MongoDB's current restriction of 40 indexes per collection and the better performance of queries against smaller collections, it is advisable to

use a collection for each type of document. Relations in MongoDB can be modeled by using embedded objects and arrays. Therefore, the data model has to be a tree. If the data model cannot be transformed into a tree, there are two options: Denormalization of the data model or client side joins.

The first option would imply that some documents would be replicated inside the database. This solution should only be used, if the replicated documents do not need very frequent updates.
The second option is to use client side joins for all relations that cannot be put into the tree form. This requires more work in the application layer and increases the network traffic with the database.

3.6.2 Query Model

Queries for MongoDB are expressed in a JSON like syntax and are send to MongoDB as BSON objects by the database driver. The query model of MongoDB allows queries over all documents inside a collection, including the embedded objects and arrays [4]. Through the usage of predefined indexes queries can dynamically be formulated during runtime.

Not all aspects of a query are formulated within a query language in MongoDB, depending on the MongoDB driver for a programming language, some things may be expressed through the invocation of a method of the driver.

The query model supports the following features:

1. Queries over documents and embedded subdocuments
2. Comparators ($<, \leq, \geq, >$)
3. Conditional operators (equals, not equals, exists, in, not in, ...)
4. Logical operators: AND
5. Sorting by multiple Attributes
6. Group by
7. One aggregation per query

In addition to this, MongoDB allows the expression of more complex aggregations with a variation of MapReduce. The results of a MapReduce operation can either be stored as a collection or be removed after the results have been returned to the client.

The MongoDB version of MapReduce is a bit different to the original one. Instead of having a Map, Reduce and a Combiner function, it uses a Map, Reduce and a Finalize function and requires that the Reduce function is associative and idempotent, because MongoDB calculates the reduction iteratively. So the Reduce function in MongoDB is more like the Combiner function in Google's MapReduce model.

The Finalize function is optional and is invoked for each record of the reduction result and can be used, for example, for a division of the end result, which enables the calculation of averages. The finalize function is needed in MongoDB for some cases, because the reduce function has to be associative and idempotent.

The Map Reduce model of MongoDB has the following signatures:

$$\begin{aligned} map(key, value) &\rightarrow list(ikey, ivalue) \\ reduce(ikey, list(ivalue)) &\rightarrow list(rkey, rvalue) \\ finalize(rkey, rvalue) &\rightarrow list(fkey, fvalue) \end{aligned}$$

For performance reasons, MongoDB allows to specify a query for a MapReduce task, so that only the result set of this query is used as input for the map function. This might be more efficient in some cases, than invoking the map function for each element of a collection, because the query can be performed with the usage of previously generated indexes [5].

3.6.3 Replication

MongoDB does not use MVCC and can, therefore, not provide optimistic replication and requires to have only one master node with write capabilities at a time.

Replication in MongoDB is implemented with the usage of a log file on the master node containing all high-level write operations performed on the database. During the replication process, the slaves ask the master for all write operations since their last synchronization and perform the operations from the log on their own local database. All operations in the log can be performed repeatedly, to be able to perform the replication without endangering the consistency of the slave database even if the slave is not sure about its local database state after a failure. MongoDB supports the following replication setups:

Master Slave
One master with write capabilities and a transaction log, which is asynchronously forwarded to the slave.

Replica Pairs
MongoDB supports the configuration of replica pairs, which is basically a master slave configuration, but with the extension that the paired nodes can negotiate themselves which one the master is. If the current master of the pair fails, then the current slave will become the new master.

If the network connection between the pair fails, then they have to ask an arbiter server. The node which can reach the arbiter first becomes the master. Without an arbiter node, the paired nodes would not know if the other node is down or if they are only disconnected.

Limited Master-Master
MongoDB also supports a restricted master-master setup. In this configuration only insert and delete of the write operations are allowed. The reason for this is that any updates on existing documents would compromise the integrity of the database, due to the lack of an distributed lock manager or a multi version concurrency mechanism.

3.6.4 Sharding

MongoDB supports automatic sharding, which is currently in an alpha stage and consequently has some limitations. A MongoDB cluster consists of three components: Shard nodes, configuration servers and routing services called mongos in MongoDB terminology. Figure 3.4 shows an overview of an MongoDB cluster.

Shard nodes are responsible for storing the actual data. Each shard can consist of either one node or a replication pair. In future versions of MongoDB one shard may consist of more than two nodes for better redundancy and read performance. The config servers are used to store the meta data and routing information of the MongoDB cluster and are accessed from the shard nodes and from the routing services.

Mongos, the routing processes are responsible for the performing of the tasks requested by the clients. Depending on the type of operation the mongos send the requests to the necessary shard nodes and merge the results before they return them to the client. Mongos for themselves are stateless and therefore can be run in parallel.

The documents in a MongoDB cluster are partitioned by their owning collection and by a user specified shard key. A shard key in MongoDB is very similar to an index and can contain multiple fields.

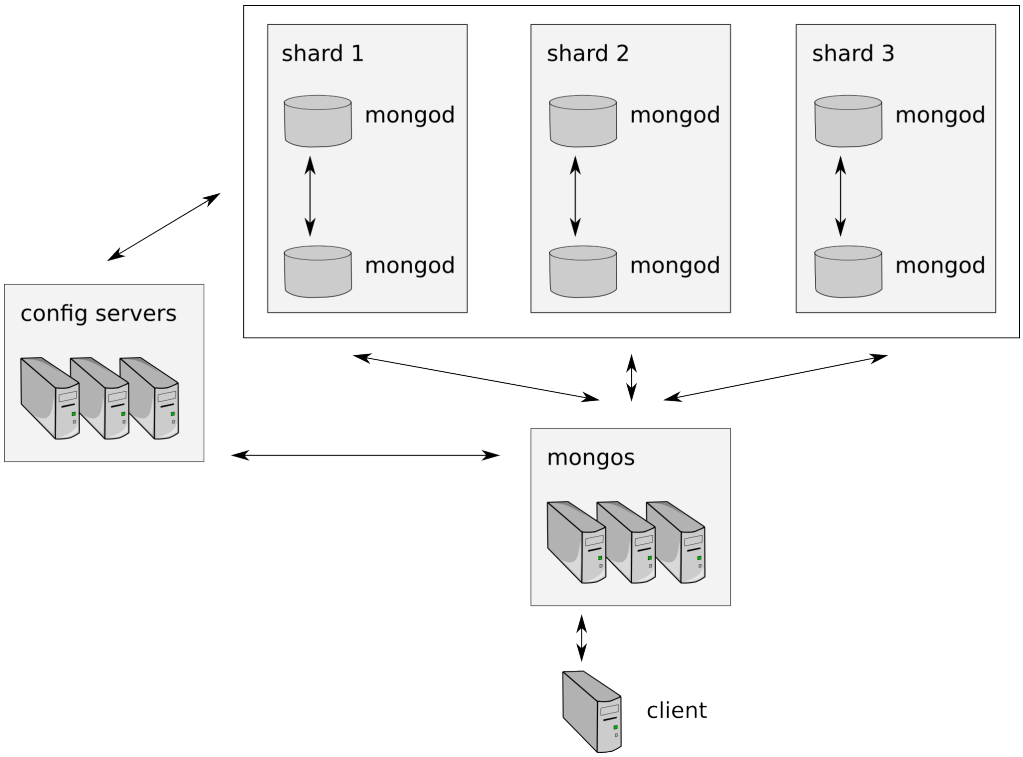


Figure 3.4: Typical architecture of a MongoDB cluster.

This shard key is used to partition the whole collection into shards. Each shard stores its assigned documents ordered by this key. The documents inside a shard are organized into chunks. Each chunk contains an ordered set of documents from one start shard key to a specific end shard key. If a chunk gets too big, it will be split. Chunks are used by MongoDB for the automatic rebalancing of the shards. If the size of one shard is too big, some of its contained chunks are migrated to other shards. Chunks are also used to redistribute the data when new nodes are added or removed.

The config servers store a record for each chunk in the cluster, consisting of the start and end key of the chunk and its assigned shard. This information is used by the mongos to decide which shard nodes are needed for which requests. Depending on the type of the operation either only one chard or nearly all shards need to be consulted to fulfill a request. For example can a simple find query to search for a document with a specific id be routed only to the shard that stores this document, if the used id is also the shard key. But queries that can not be restricted with the usage of a shard key need to be send to each shard node of the cluster.

3.6.5 Architecture

MongoDB is implemented in C++ and consists of two types of services: The databases core **mongod** and the routing and autosharding service **mongos**.

For storage MongoDB uses memory-mapped files, which lets the operating system's virtual memory manager decide which parts of the database are stored in memory and which one only on the disk. This is why MongoDB cannot control, when the data is written to the hard disk.
The motivation for the usage of memory mapped files is to instrument as much of the available memory as possible to boost the performance. In some cases this might eliminate the need for a separate cache layer on the client side. But there is currently an alternative storage engine for MongoDB in development, which will allow MongoDB more control over the timing of read and write operations. Indexes are stored in MongoDB as B-Trees like in most other databases.

3.6.6 Consistency

MongoDB has no version concurrency control and no transaction management. So if a client reads a document and writes a modified version back to the databases it may happen that another client writes a new version of the same document between the read and write operation of the first client.

MongoDB provides only eventual consistency, so a process could read an old version of a document even if another process has already performed an update operation on it.

3.6.7 Failure Handling

MongoDB does not use a transaction log to ensure the durability of newly written data and because of the usage of memory mapped files it performs lazy writes. So if a MongoDB node crashes, some data might be lost. Because of interrupted writes during a crash or hardware failure, some of the database files may be corrupted. If a crashed node comes back online, it will require that maintenance utilities of MongoDB are used to search for corrupted database files and to fix them.

If a single shard node crashes and it is part of a replication pair, then the other member of the pair will overtake the complete workload of that shard until the

broken node is back online or replaced. If all nodes of a shard fail, then the cluster will be unable to perform operations on data in this shard.

If one of the config servers fails, the MongoDB cluster will be unable to perform any kind of split or migrate operations on the data chunks until the lost config server is back online or replaced.

3.7 CouchDB

CouchDB is a schema free document oriented database with an optimistic replication mechanism [36]. The project is part of the Apache Foundation and is completely written in Erlang. Erlang was chosen as programming language, because it is very well suited for concurrent applications through its light-weight processes and functional programming paradigm.

CouchDB itself is currently not a distributed database by itself, but it can be used as such in combination with a proxy layer, which handles the sharding and node management, but this features might also be integrated in CouchDB in later releases.

CouchDB is not only a NoSQL database, but also a web server for applications written in JavaScript. The advantage of using CouchDB as a web server is that applications in CouchDB can be deployed by just putting them into the database and that the applications can directly access the database without the overhead of a query protocol.

3.7.1 Data Model

Data in CouchDB is organized into documents. Each document can have any number of attributes and each attribute itself can contain lists or even objects. The Documents are stored and accessed as JSON [24] objects, this is why CouchDB supports the data types String, Number, Boolean and Array. Listing 3.2 shows a CouchDB document.

Each CouchDB document has a unique identifier and because CouchDB uses optimistic replication on the server side and on the client side, each document has also a revision identifier. The revision id is updated by CouchDB every time a document is rewritten.

```
{
    "_id": "154c48b23c8f6d14efcf3659ed8c7312",  //identifier
    "_rev": "1-4492d097e7826b4e1019441344d88ede",  //revision id
    "name": "Hugo",
    "hobbies": [
        "sport",
        "movies",
        "programming"
    ]
}
```

Listing 3.2: Example CouchDB document as JSON object.

Update operations in CouchDB are performed on whole documents. If a client wants to modify a value in a document, it has first to load the document, make the modifications on it and then the client has to send the whole document back to the database. CouchDB uses the revision id included in the document for concurrency control and therefore can detect if another client has made any updates in the meantime.

CouchDB has no built in support for document types or equivalent to tables, hence developers have to build the type distinction by themselves. One solution is to have a type attribute in every document, which specifies the kind of document. But one could also use the *What walks like a duck, is a duck* principle in CouchDB. This means that the kind of a document is only determined by its attributes and not by a dedicated type attribute. For example, does CouchDB allow to define queries that can find any kind of document with a tag attribute. Both concepts can be mixed in CouchDB as they are needed.

One other interesting feature in CouchDB is the option to specify a validation functions in JavaScript. If a document is updated or created, then CouchDB lets all user specified validations functions check if they approve with the new document and gives the validation functions the opportunity to reject not well formed ones.

Such a validation function gets as parameters the new document, the old document and the user who is trying to make the modification. The user information can be used by the validation functions to make security checks.

Because of CouchDB distributed architecture, validation functions are not allowed to have side effects and can only read the documents which are given as argument. For this reason, it is not possible to use validation functions to enforce, for example, foreign key constraints. Listing 3.3 shows a validation function, which only allows documents with less than 10 hobbies to be written into the database.

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.hobbies && newdoc.hobbies.length >= 10 ) {
      throw({forbidden : 'too many hobbies'});
  }
}
```

Listing 3.3: A validation function which restricts the number of hobbies someone can have.

3.7.2 Query Model

The query model of CouchDB consists of two concepts: Views which are build using MapReduce functions and a HTTP query API, which allows clients to access and query the views.

A View in CouchDB is basically a collection of key-value pairs, which are ordered by their key. Views are build by user specified MapReduce functions, which are incrementally called whenever a document in the database is updated or created. This is a difference to other distributed databases in which the MapReduce model is used to instrument already existing indexes for aggregations.

Views should be specified before runtime, as introducing a new View requires that its MapReduce functions are invoked for each document in the databases. This is why CouchDB does not support dynamic queries.

The MapReduce pattern in CouchDB is implemented with only a Map and a Reduce function, but the Reduce function in CouchDB is used as both the Combiner and the Reduce function in Google's MapReduce paper.

Map functions are invoked with a document as argument. The function can only access this one document and cannot have any side effects nor access the results of any views. The only thing the function can do is to emit key values pairs based on a given document. If a reduce function is specified, then these key values pairs are forwarded to the reduce function, otherwise the emitted pairs are put into the view. The map function can emit any number of key-value pairs for a single document.

As mentioned before, the Reduce function in CouchDB is a combination of the Combiner and the Reduce function as described in the MapReduce paper. The function is invoked with a key and a subset of all the values belonging to it.

A reduce step in CouchDB is separated into two phases: The reduce phase in which the function is invoked with the output of the map step and a rereduce phase which

gets the results of the reduce phase. The Reduce function can determine in which of this two phases it is, by a binary argument that is passed through to the function.

During the reduce phase, the reduce function is invoked for different subsets of the values associated with the same key, hence the MapReduce implementation of CouchDB cannot be used for problems that cannot be separated into small subsets without changing the result. This means that CouchDB assumes that the following is true for every reduce function:

$$reduce(values) = rereduce(reduce(values1), reduce(values2), ..., reduce(valuesN))$$

With *values* being the value set for a certain key and *valuesN* being a subset of *values*. The reduce function in CouchDB, for example, can be used for aggregations like counting or calculating other kinds of statistics like averages. The reduce function is used in CouchDB for problems which are in SQL usually handled by the group statement and the aggregation functions.

Listing 3.4 shows a MapReduce function that counts the occurrences of hobbies. The Map function emits a key value pair for each hobby in the document, with the hobby as key and 1 as value. The reduce function gets the subsets of the values grouped by the key, so in the reduce phase it only has to return the number of values in the set and then to sum up the intermediary result in the rereduce phase. The reduce function could be simplified by summing up the values in both the reduce and the rereduce phase, but that would not show the difference between the two phases. Figure 3.5 shows a possible execution of the example MapReduce task.

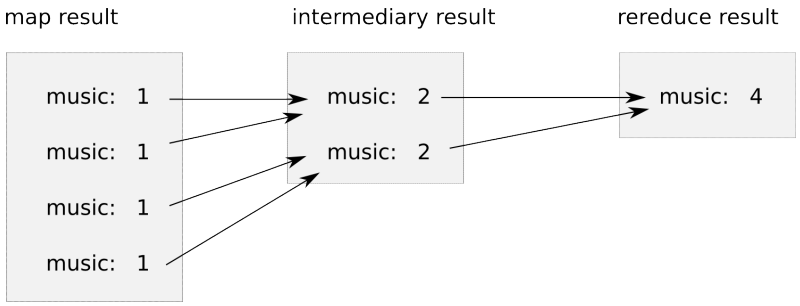


Figure 3.5: Example execution of the MapReduce task.

One feature of CouchDB is the possibility to use complex keys, which means that a key of a view does not need to consist of only one attribute. A key in CouchDB can be any kind of JSON object, it is especially possible to use an array with different attributes in it. This allows to order the documents in a view by different attributes and enables a more powerful usage of the reduce function, which will be explained later. The rules by which CouchDB orders complex keys are not well documented,

```
//Map function
function(doc) {
  if(doc.hobbies) {
    for(var  hobby in doc.hobbies) {
      emit(doc.hobbies[hobby], 1);
    }
  }
}

//Reduce function
function(keys, values, rereduce) {
  if (rereduce) {
    return sum(values);
  } else {
    return values.length;
  }
}
```

Listing 3.4: A view that calculates the popularity of hobbies.

but it is known that the first entry is used as first ordering criteria and the second as second and so on. The ordering rules are sensitive to the data types of the key, so CouchDB orders numbers as numbers and not as strings.

The usage of complex keys does not only have an impact on the ordering, it also affects the reduce function. If a complex key is used then CouchDB stores results of the reduce function for every level of the key. This means for an example view with a simple document counting reduce function and a map function, which emits a complex key consisting of an attribute A and B that it is possible to query the view via the HTTP API for the number of documents in the view, the number of documents with a instance of key A as well as the number of documents with a certain complex key $[A, B]$.

CouchDB's HTTP API allows simple queries on views by their key. Basically a query against a view consists of a key range. It is possible to define a startkey and an endkey or to define only a startkey and a maximal number of search results.

In addition to this, the API allows to get the results of the reduce function for the different levels of the key, like previously described. Other options allow to invert the search order in the view, which means that CouchDB would start the search at the startkey and then would go backwards from there in the index until it reaches the result limit or the endkey.

The usage of complex keys in CouchDB provides the functionality to order a view by different attributes and grouping over different attributes, but it can only be used if the attribute one wants to query are not in conflict with the desired ordering.

3.7.3 Replication

CouchDB uses optimistic replication for both the client side and the server side. Each CouchDB database can be synchronized to another instance. This can either be triggered manually or used continuously and can be used unidirectional as well as bidirectional.

The optimistic replication feature of CouchDB enables it to be used for very different scenarios. For example, it is possible to implement any kind of master-master replication server setups with this kind of replication. In a master-master setup, both nodes can be used for writing and reading. Due to the optimistic replication, such a setup is partition tolerant. If the connection between the two nodes is lost for a while, then both can operate for themselves until the connection is reestablished. Because a CouchDB can be synchronized with more than one other database, any kind of replication topologies can be built.

Another usage scenario is to use the replication not only on the server side, but also on the clients side. Users can have a replica of a database on their own machines and can work on it without having a connection to the original one and can synchronize it any time they like. This usage scenario is currently implemented with the Ubuntu One Service [41]. This is a synchronization service for Ubuntu desktops, which allows users to synchronize their files, calendars and bookmarks from different desktops. Application that support this feature store their data in a CouchDB database and let the Ubuntu One Service and CouchDB handle the synchronization. So CouchDB can not only be used for big databases, but also for small desktop applications.

The downside of optimistic replication is that a system which instruments it must be able to handle conflicts between different revisions of the same object. CouchDB can detect possible conflicts between document revisions and passes them forward to the application. The conflict handling of CouchDB is handled while the application reads a document. Then the application has to decide if it wants to choose one specific version as the correct one or to merge the conflicting documents.

The MVCC in CouchDB is implemented with the usage of hash histories. Because of this, every document in CouchDB has a revision id, which consists of the number of updates that had been applied to the document and a hash key of the content.

CouchDB uses the advantage of hash histories to detect duplicate versions of the same document, this means that an update of a document only changes the revision id, if the update really changes the document.

The replication mechanism in CouchDB is implemented using a changes feed which can be accessed with the REST API. The feed contains a list with the ids of the documents that were last modified. If a CouchDB instance is replicated to another one, then the feed is used to determine which documents had been updated since the last replication. The changes can also be used by the client application, which allows the client to fire events for document updates and to remove outdated documents from its cache.

3.7.4 Sharding

CouchDB itself has no built in sharding mechanism yet, but there are currently two projects which provide sharding support for CouchDB. Because CouchDB's query model is based on incremental MapReduce functions, the documents inside a CouchDB database can be partitioned over different databases. This only requires that the application layer or a proxy has to send queries against the views to all nodes and then has to merge the results. Despite this, the different shards do not need to communicate with each other.

The first project is *The Lounge* [38], a tool set that contains proxies, which can distribute the data to different CouchDB instances. The Lounge is developed and used by Meebo.com.

The Lounge consists of two proxies: The dumbproxy, which can be used to read single documents and to write to the cluster. The dumb proxy can decide which CouchDB instance is responsible for a certain document, by applying a consistent hash function onto the document identifier, and then can forward the read and write request to the responsible machine.

The other proxy is the smartproxy, which is responsible for requests against the views of the database. For this, the request is send to all nodes in the cluster and the results are merged by the proxy. Because the view results are already sorted, they can be merged in $O(n)$.

The other project that provides data partitioning for CouchDB is Cloudant [21] which provides a hosting service with automatic sharding functionalities for CouchDB databases.

In addition to the automatic sharding this service provides a quorum based replication feature like Dynamo, which is planned to be configurable. Currently it runs with the default setting $R = 1, W = 3, N = 3$, which means that each document is written to at least three nodes and read from at least one node. This configuration enforces strong consistency.

3.7.5 Architecture

CouchDB consists of one server application, which contains the database service and the in CouchDB integrated web server. User generated functions such as the MapReduce and Validation functions are per default JavaScript applications. But CouchDB has an pluggable language support, which allows to use other programming languages for the user functions as well.

3.7.6 Consistency

The type of consistency of a CouchDB database is dependent on how CouchDB is used. If CouchDB's built in replication feature is used in a master-master configuration, then CouchDB provides eventual consistency. But the replication can also be used in a master-slave setup, which would provide strong consistency.

Another case is the usage of something like Cloudant's dynamo inspired replication mechanism, which can also be configured to provide strong consistency.

3.7.7 Failure Handling

Because CouchDB itself is not distributed, CouchDB has no mechanism for dealing with failures of nodes. Currently it only provides the replication mechanism, which can be used as a tool to build a robust infrastructure. However, the management of multiple nodes and fail over mechanism needs to be realized with other services. Because CouchDB uses a HTTP API, the infrastructure can be build using any kind of HTTP proxies and load balancers. If sharding is used the fail over mechanism and node management might be part of the used sharding framework.

CouchDB uses an append only B+Tree as file structure to be robust against power failures and to minimize the number of seek operations of the hard drive while writing into the database [34]. If a document is updated, then CouchDB creates

a new copy of the document and a copy of the complete path in the B+Tree and changes the pointer to the new root node after the write operation of the part of the tree is complete. The copy-on-write behavior of CouchDB makes sure that the files on the disk are always in a consistent state, even if a power failure occurs. Because of this, CouchDB does not need a transaction log file and can always be interrupted without the risk of corrupting the database files.

4 NoSQL Database Comparison

All examined NoSQL Databases share the same restrictions, which are derived from their distributed architecture. All of them are victims of the CAP Theorem and therefore can either only provide eventual consistency or sacrifice some amount of availability. One weak spot of all databases is that they do not provide a functionality to model relations on the database side, so in almost all cases this has to be done in the application layer. The line between the different NoSQL databases is very thin, but they have still some small but very significant differences.

4.1 Sorting

Providing a sorted view of a given data set is a typical task for a database. All examined databases can handle the ordering of objects by a single attribute, but only the both document oriented databases MongoDB and CouchDB and the App Engine datastore allow the ordering by multiple attributes. All three of them require to specify an index to be able to perform the ordering efficiently.

SimpleDB does not offer the possibility to order a query result by different attributes, this may be because SimpleDB does not require nor allow the developer to specify indexes manually. BigTable orders its objects after their primary index, because there can only be one primary index, BigTable does only support the ordering by one attribute.

A sometimes important feature of databases is the ordering of a set by the result of an aggregation, like a count operation or summation. This is for example required for Top N lists. If an application requires such a feature, one possible solution would be to let the client handle the creation of top n list in a periodically batch process, it could even use the database to store and order the result. Table 4.1 shows the capabilities of the different databases.

	Order by a single attribute	Order by multiple attributes	Order by aggregation result
SimpleDB	yes	no	no
BigTable	yes	no	no
Datastore	yes	yes	no
MongoDB	yes	yes	no
CouchDB	yes	yes	no

Table 4.1: Comparison of the sorting capabilities of the examined NoSQL databases.

4.2 Range Queries

A range query is a query which defines a upper and/or a lower limitation for at least one attribute. A range query, for example, would be to retrieve all objects with an age attribute between 16 and 44. All databases considered here can handle such a query. However BigTable only allows this on the primary index.

Range Queries over multiple attributes connected with a logical operator can be handled by the Datastore, SimpleDB and MongoDB. CouchDB also allows range queries over different attributes at the same time, but it uses its complex keys for this task, this means that the developer has to specify an index with a key consisting of the attributes on which the range query should be performed. On such an index, the application makes queries consisting of one startkey and one endkey.

Furthermore, the attributes on which the range queries are performed in CouchDB are always the same as the attributes which are used for sorting. The App Engine shares this restriction with CouchDB, because both databases work with materialized views with a single index for sorting and range querying.

Table 4.2 shows the differences in the expression power for range queries of the different databases.

4.3 Aggregations

Most of the examined databases have the ability to use MapReduce for aggregations tasks such as counting, calculating of averages and other statistic calculations. SimpleDB and the App Engine datastore only support counting without grouping and do not support any other aggregation functions.

	Range queries on single attributes	Range queries on multiple attributes	Range queries on aggregation results
SimpleDB	yes	yes	no
BigTable	yes	no	no
Datastore	yes	yes	no
MongoDB	yes	yes	no
CouchDB	yes	yes	no

Table 4.2: Comparison of the range querying capabilities of the examined NoSQL databases.

Aggregations in BigTable and CouchDB can only be implemented with the usage of MapReduce functions, but both do this differently. BigTable allows to use Google's MapReduce framework on it, which can use BigTable as data input source and output target. In contrast to this, CouchDB's incremental MapReduce system stores the results of MapReduce tasks as index which is incrementally updated whenever a document is written. The advantage of CouchDB's system is that the results of a MapReduce task always reflect the current state of the documents in the database. But the downside of this is that it is not possible in CouchDB to chain multiple MapReduce tasks to perform more complex calculations than possible with one single MapReduce step.

A MapReduce task on something like BigTable might not be well suited for real time tasks. That is why the typical use case of non incremental MapReduce is batch processing, which can be done periodically. This kind of MapReduce is more suitable for large data warehouses, with more complex calculations, which do not require to be always up to date.

MongoDB provides extra functions for simple aggregations operations like counting, summations and averages and group by statements comparable to the ones in SQL. This functions can be used efficiently in combination with the indexes that can be specified in MongoDB. In addition to this, MongoDB also supports the usage of MapReduce functions. Its implementation is like the one for BigTable meant for batch processing. One advantage of MapReduce tasks in MongoDB is that they can instrument previously specified indexes for better performance. Especially, a MapReduce task can be bound to a query, which can be used to reduce the amount the number of documents, the MapReduce task has to look at.

Table 4.3 shows the differences of the aggregation functionalities of the databases.

	Counting	Aggregation primitives like Summations and Averages	Complex aggregations with MapReduce
SimpleDB	yes	no	no
BigTable	MapReduce	MapReduce	yes
Datastore	yes	no	no
MongoDB	yes	yes	yes
CouchDB	MapReduce	MapReduce	yes

Table 4.3: Comparison of the aggregation functionalities.

4.4 Durability

Durability means that all write operations that are committed to a database can never be lost. This requires that new data is written in some way onto the hard disk before a database can return the acknowledgement of a successful commit operation, so that committed data can not be lost even in the event of a power failure. In RDBMS, durability is usually ensured through the usage of a transaction log file. In this file, all transactions are written, before they are committed. If a failure like a power outage or some other kind of software or hardware failure happens, the database can use this log to recover committed but not completely applied transactions.

For distributed databases the term of durability can even be more extended, because some databases can not only make sure that committed data is always persistent on a single machine, but they can also make sure that the data is replicated to at least a specific number of machines or even to different data centers distributed around the world to be safe against geological disasters and permanent failures of single machines.

However, ensuring durability is very expensive. If a database attempts to ensure that committed data is always written on at least one hard disk then the hard disk with the log file or the database files becomes the bottleneck. In addition to this, it is necessary that the write caches of the used operating system and the ones of the hard disk are disabled to ensure that committed data is really written to the disk and not only into the write cache, whose data could be lost during a power failure. This all slows down the write performance of the database.

The other solution to ensure durability is to make sure that committed data is always replicated to multiple machines, which both reduces the availability and the

partition tolerance of the database.
This is why the examined databases make different trade-offs between durability, availability and partition tolerance. MongoDB, for example, aims mainly for performance and does not make any attempts to make the newly added or changed data durable. It currently uses memory mapped files and lets the operating system decide when a file has to be written to the disk. MongoDB's replication mechanism does only lazy replication and therefore cannot provide reliable durability.

The same is the case for CouchDB with its default settings to cache all write operation, but if turned off CouchDB can make sure that committed data is always written to at least one disk. Because CouchDB also provides only lazy replication, this mechanism does not ensure that new data is already replicated to different nodes after it is committed.

For SimpleDB were no concrete information available about the durability besides that the data is replicated to different machines and different data centers.

BigTable and the BigTable based Datastore ensure durability and can therefore only provide limited availability.

	Written on at least one hard disk after commit	Replicated to different machines after commit	On disk integrity
SimpleDB	?	?	?
BigTable	Yes, in a commit log file stored in GFS	yes	Log file
Datastore	Yes, in a commit log file stored in GFS	yes	Log file
MongoDB	no	No, only lazy replication	Recovery may cause data loss and requires consistency checks of the database files
CouchDB	Not in the default configuration, but can be enabled.	No, only lazy replication. But improvements in this area are on the way.	Append only B+Tree lets the database files always be in a valid state.

Table 4.4: Comparison of the the durability properties.

4.5 CAP Properties

This section tries to compare the CAP properties consistency, availability and partition tolerance of the examined databases with each other. Figure 4.1 shows the relative distribution of the databases inside these three dimensions.

BigTable and the App Engine datastore are in the left part of the graph, because they are strongly consistent and not always available and because the datastore is based on BigTable they have only marginal differences. The datastore might be a little less available than BigTable, because the datastore depends on four different BigTables.

The other databases provide eventual consistency and therefore are in the right part part of the graph. CouchDB is the database with the greatest partition tolerance because of the optimistic replication features. CouchDB, MongoDB and SimpleDB are all on the same level of availability, because they all use asynchronous replication. SimpleDB is hard to position in the graph because of the lack of information, but it can be assumed that is comparable with MongoDB and CouchDB.

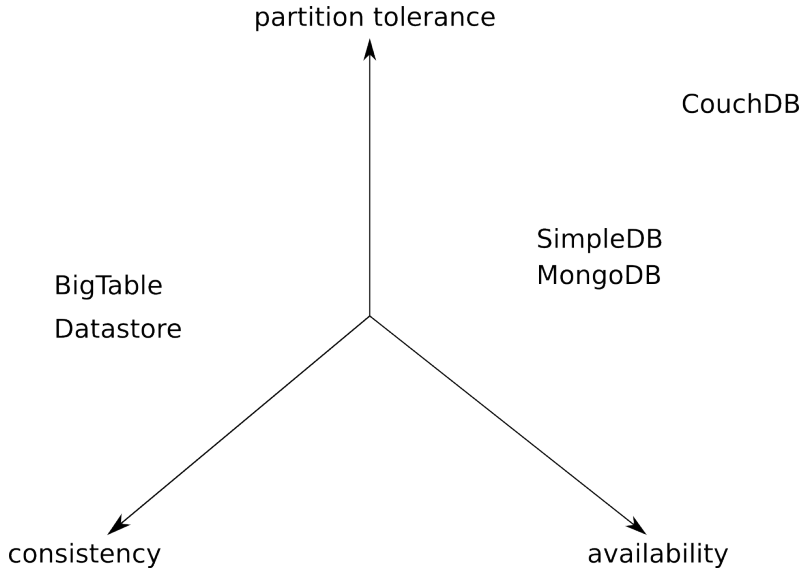


Figure 4.1: Relative positions of the NoSQL databases in the CAP theorem.

5 Database Requirements of Tricia

One goal of this work is to determine if NoSQL databases can be used for the persistence layer of a collaborative web application. To do this, Tricia and its persistence layer is used as an example to determine the requirements for such a system and to implement a prototype. Furthermore, this chapter deals with the selection of a suitable NoSQL database for the prototype implementation in the next chapter.

Tricia is an Open Source Web Collaboration and Knowledge Management Software, which is developed by Infoasset [35]. It is an implementation of the Introspective Model-Driven Development paradigm presented in [19] and [18].

Tricia has a persistence layer, which allows to model the data independent from the underlying database. Queries in Tricia are formulated as Java objects, which are translated by Tricia to the query language of the underlying database. Because of this architecture, it is possible to add support for a NoSQL database to Tricia without having to reimplement every query.

To determine the databases features required to run Tricia, its persistence layer was modified to record all queries made against a relational database. Then, several tests with a big code coverage were invoked. This should have covered almost any queries used by Tricia, but there is small probability that a few queries were not caught by this analysis.

5.1 Relations

The results of this analysis has shown that queries used by Tricia are not very complex, for instance, there are no queries with more than one join operation. This is a very interesting result, because Tricia supports many to many relations, which are in relational databases usually modeled with a table for the relation, which stores a foreign key for each side of the relation, like the group member relation in

figure 5.1 . Figure 5.2 shows the implementation of this relation with tables in a relational database.

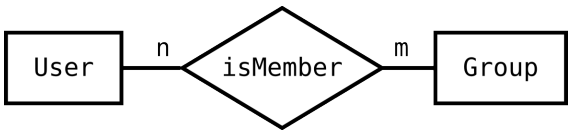


Figure 5.1: ER diagram of a simple many to many relationship.

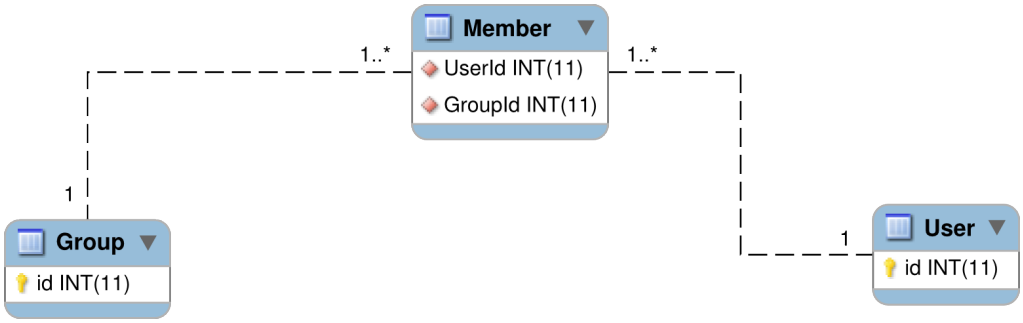


Figure 5.2: Implementation of the many to many example as tables in a RDBMS.

To query, for example, all members of a group, it could expected that a join operation over three tables would be required to get, for instance, all users belonging to a group, like in this SQL query:

```
select u.* from `User` u, Member m, `Group` g where g.id = m.GroupId
    and u.id = m.UserId and g.id = GroupOfInterest
```

Thereby *GroupOfInterest* is a placeholder for the identifier of the group from which all the members are to be retrieved. Because there is only one group involved in this query, the join operation over the Group and the Member table is unnecessary. So the query can be reduced to:

```
select u.* from `User` u, Member m where u.id = m.UserId and
    m.GroupId = GroupOfInterest
```

This is very important, because NoSQL databases generally do not support join operations on the database side, because join operations are very expensive for distributed databases to perform.
So to implement this with a NoSQL database, the last join operation would either need to be handled by the application layer itself or the data model would need to be denormalized. For most cases denormalization is not an option, because it requires the management of redundant information, which can be expensive and can cause consistency problems.

Therefore, the most common solution is to let the application layer handle the joins. This implies that the persistence layer has to get all required relations from the relation table and then it has to make an extra request for each element in the first result set to get the objects for the final result.
For the group member example the application would first make a query to get the identifiers of each member of the group and then it would need to fetch the user objects afterwards.

This can be optimized by using a cache on the application side. Instead of making a request for each entry in the result set, the persistence layer first looks into the cache and only fetches the objects that are not already cached from the database.

5.1.1 Ordering with joined queries

Another important result is that Tricia uses sort orders on joined queries only on the first table and not on the final result. So in the membership example this would mean that Tricia would only make sorting instructions for the member table, but not for the user table. Assuming that the member table would have an additional column for a timestamp, Tricia could make a query like this:

```
select u.* from `User` u, Member m where u.id = m.UserId and
    m.GroupId = GroupOfInterest order by m.date
```

But Tricia would not make a query with a sort instruction for the final result like (assuming the table User would have the additional attribute name):

```
select u.* from `User` u, Member m where u.id = m.UserId and
    m.GroupId = GroupOfInterest order by u.Name
```

The first query can easily be divided into two queries and the join operation can be performed by the application like it was previously described, with the difference that the sort instruction for the member table must be added.

But if Tricia would make queries like the second one, then this would not be enough. If the application would first query for all members with the required groupId and then would fetch the related user objects from the database with the sorting instruction for the name of the user, then the final result would still be unsorted. So in this case the application would need to sort the final result by itself and this would require to load all results into memory even if the final result set would have a limited size. This problem could be solved in a few different ways: The first solution would be to denormalize the schema by writing the name of the member

also into the member table. Then both parts of the query could be ordered by the user name. The down part of this solution is that it requires that the name to be written into two different places every time it is changed.

The other solution would be to perform the join operation the other way around. So the application would start to iterate other all users ordered by their names and would just skip the ones, which are not members of the group of interest. The downside of this solution is, that this would not scale for many users.

Which solution should be used depends on the size of the tables and how often the attributes are changed. In most cases the denormalization would be the most efficient.

But because Tricia does not make queries like the second one, the implementation of an NoSQL layer for Tricia is a lot easier and more efficient.

5.1.2 Other Requirements

Tricia makes queries which include sort instructions for up to two different attributes per query. From the aggregation functions, Tricia only uses the *count* operation to determine the size of the result sets.

Tricia uses only logical ANDs as operators between the conditions, except for queries which include conditions on the type of the returned objects. Some objects in Tricia have a type attribute, which is used to describe class inheritances. So queries with this type of conditions could look like:

```
select u.* from `User` u, Member m where u.id = m.UserId and
    m.GroupId = GroupOfInterest  and (u.type=2 or u.type=3 or
    u.type=4)
```

Besides this type of query are no logical ORs used. This also makes the implementation easier, because this is a special case of an OR relation which can be handled more efficiently in some NoSQL databases than other cases of OR relations.

During the survey of Tricia's database requirements only one range query was detected, so range queries do not seem to be very important for collaborative web services like this one. However, range queries are not a problem for most NoSQL databases.

Tricia also uses some database constraints, but because they are also checked in the application layer, they should never be violated. Because NoSQL databases have either none or a very limited feature set for constraints, they are ignored for the prototype implementation in this work.

5.2 The choice of a NoSQL database for the prototype

The possible choices of NoSQL database for the following implementation can be narrowed down with the information from this and the last chapter. Because most queries from Tricia include multiple conditions, the possible choices are reduced to SimpleDB, App Engine datastore, MongoDB and CouchDB. Because some queries use sort instructions for up to two attributes, SimpleDB is also eliminated as an option.

The App Engine datastore has the additional restriction that it can only return up to 1000 results per query. This could be a problem in some cases, especially if join operations are handled by the application layer. So the possible choices are reduced to CouchDB and MongoDB.

CouchDB has the advantage of a better data integrity and durability than MongoDB and the optimistic replication feature would allow some interesting new use cases for Tricia. But it also has the disadvantage that it is currently not as distributed as MongoDB. Another difference between them is the query interface: CouchDB uses views, which are generated by MapReduce tasks. So the usage of CouchDB would require the implementation of a query analyzer, which would generate the MapReduce tasks automatically. In contrast to this is the very easy to use API from MongoDB, which allows dynamic queries and provides a very good integrated database driver for Java. This is why MongoDB was selected for the example implementation in this work.

6 Prototype Implementation

This chapter describes the implementation of a MongoDB support layer for Tricia. The persistence layer supports expandability for different databases. A module that enables the usage of a certain database is called **Store** in Tricia. Figure 6.1 shows an overview of the store package with two examples of the already existing relational database store packages and the new package for the MongoDB support.

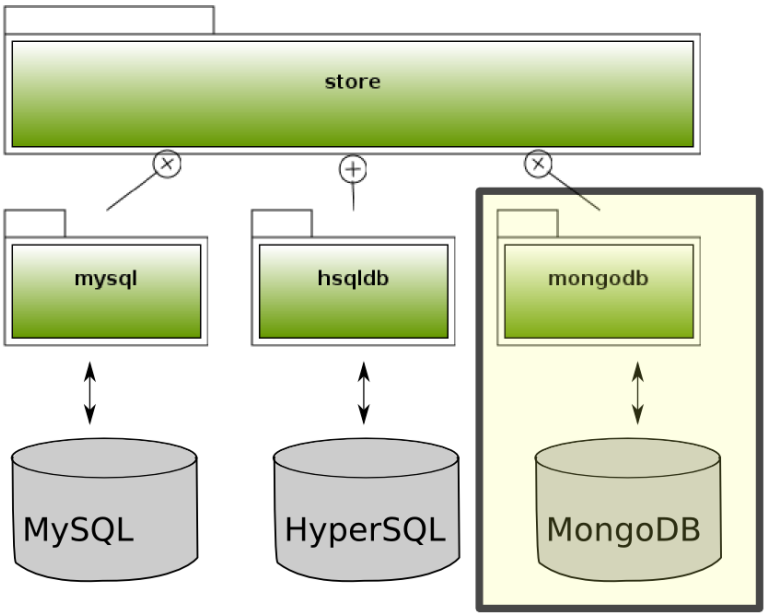


Figure 6.1: Overview of the store package, which will be extended with a store for MongoDB.

6.1 The Proceeding

Replacing the persistence layer of an already existing application comes with various difficulties. The first one with Tricia was that the architecture was designed for

relational databases and therefore had to be changed to support data stores that are not based on SQL.

The second problem was that Tricia can only run on a complete store, so using a store that is not fully implemented yet would result in a lot of runtime errors, which would be hard to trace back. So to make the development of the first prototype easier, the existing persistence layer for relational databases was modified step by step to perform all operations in parallel on a relational database and MongoDB. Thereby it was possible to first implement some basic features like saving and loading objects to MongoDB and to test them without the need to implement everything else first.

Running Tricia on two databases in parallel during the development also enabled the comparison of the query results of the two databases directly during runtime. This made the development a lot easier, because this showed instantly most bugs of the new implemented code.

After most of the features of the first prototype were implemented and tested, the architecture of Tricia's persistence layer was changed to support stores that are not based on SQL. This enabled the implementation of a stand alone store for MongoDB with the code of the first prototype.

To still be able to compare the behavior of the new store to a relational store a generic store was developed, which can run two other stores in parallel. This parallel store does not only allow the testing of stores against each other, it also allows to copy the content of one store to another one, which enables for example the conversion of the data from an existing database to a MongoDB instance. Figure 6.2 shows the class diagram of the *ParallelStore*. This store encapsulates two other stores and relays all operations to both of them. In order to this, the class *ParellelContainer* encapsulates two other containers and *ParallelContent* two contents.

6.2 Changes in the Architecture of the Persistence Layer

Each Store extends the class *Store* and if necessary the classes *Container* and *Content*. An instance of *Container* represents a table and a *Content* object the content of a row. Figure 6.3 shows a class diagram of the old architecture for two example stores.

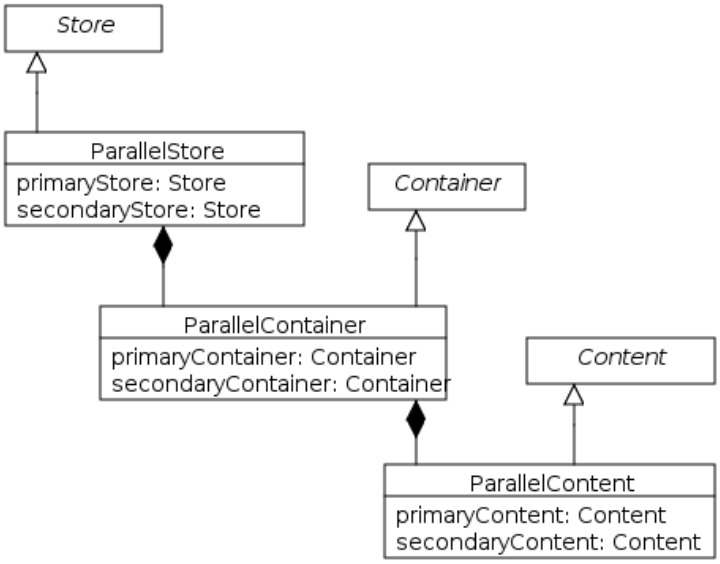


Figure 6.2: Overview of the *ParallelStore*.

Because all existing supported databases were relational databases, some shared functionalities for relational databases were inside the classes *Store*, *Container* and *Content*. To enable the integration of a NoSQL store these functionalities were pulled down into the new classes *RdbStore*, *RdbContainer* and *RdbContent*. Figure 6.4 shows the new architecture including the MongoStore. The new architecture also enabled the development of stores that encapsulate other stores like the *ParallelStore* and the *MonitorStore* which is used in the next chapter.

6.3 The MongoDB Query API for Java

The MongoDB driver for Java is not a JDBC driver and provides a completely different interface. The biggest difference between the MongoDB driver and the JDBC interface is that the MongoDB driver provides more database functionalities over a Java interface, whereas most functionalities of relational databases are accessed with string commands in SQL syntax, which are passed to the JDBC driver. Figure 6.5 shows on overview of MongoDB Java API.

This architecture is very similar to the one of the Tricia Stores. The class *DB* is the equivalent to the *Store* class in Tricia and *Collection* is the pardon to the *Container* class. For the counterparts *DBObject* and *Content* the similarities are even more obvious, because they both have a map like interface. This makes not

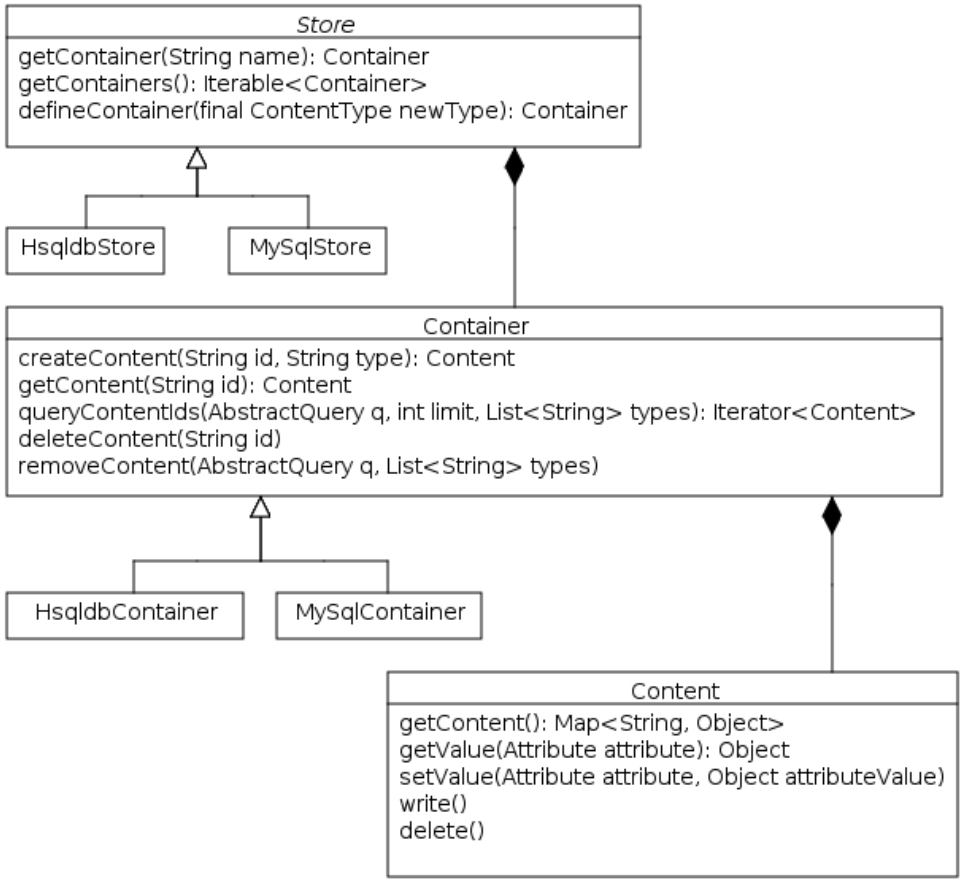


Figure 6.3: Class diagram of the base store classes and two example stores.

only the implementation of a MongoDB Store for Tricia easier and more elegant than the one for SQL based databases, it can also be said that the MongoDB Java API is already a persistence layer for itself.

The following code shows how a connection can be established to a local MongoDB database "tricia" without authentication.

```
Mongo m = new Mongo();
DB db = m.getDB( "tricia");
```

The *DB* class represents the database and, for example, allows to get a collection for a certain name. If no collection with the given name exists, then a new one is created.

```
DBCollection userCollection = db.getCollection("user");
```

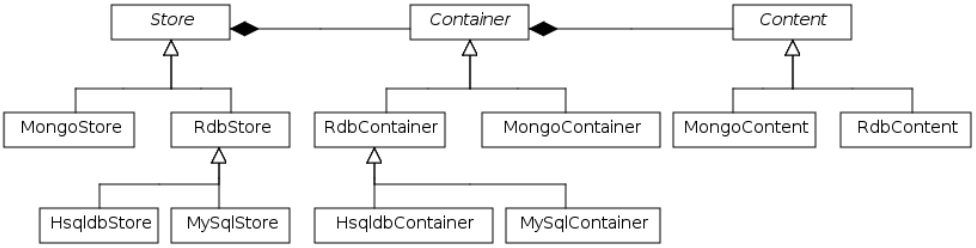


Figure 6.4: New architecture of the persistence layer.

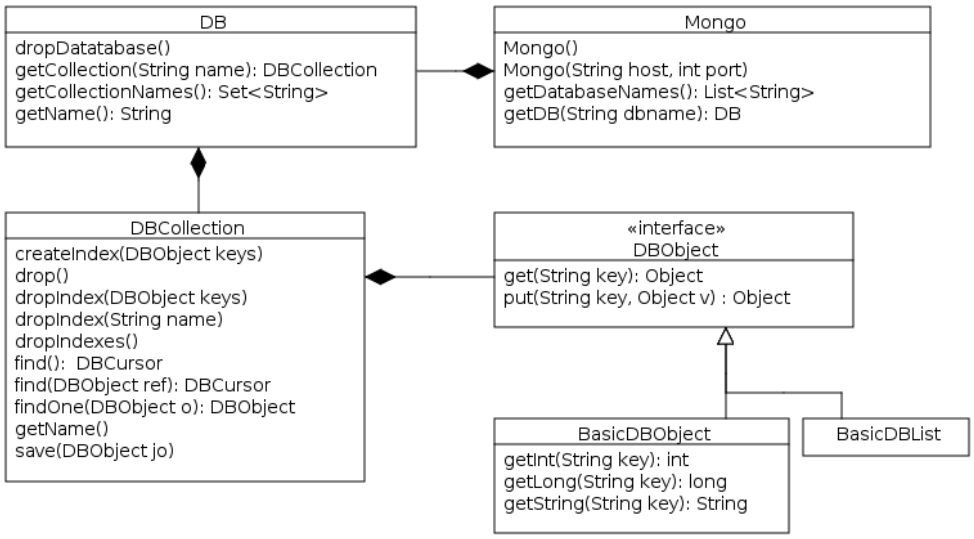


Figure 6.5: Overview of the MongoDB Java API classes.

Performing the the same operation with a JDBC driver would require to build a string in SQL syntax, which then would be passed to the database. The *DBCollection* class represents a collection and is used for all operations on this collection. Getting for example the size of the user collection can be done with this:

```
userCollection.getCount();
```

To do the same with a SQL database and a JDBC driver would need a little more code as it can be seen in listing 6.1.

MongoDB documents are represented through the class *BasicDBObject*, which provides a map like interface. Listing 6.2 shows how this can be used to insert a new user into the collection.

```
try {
    Statement stmt=dbConnection.createStatement();
    ResultSet set = stmt.executeQuery("select count(*) from user");
    set.next();
    int count = set.getInt("count(*)");
    stmt.close();
    } catch (Exception e) {
    }
```

Listing 6.1: Determining the size of table with the JDBC interface.

```
BasicDBObject newUser = new BasicDBObject();
      user.put("name", "Hugo");
      user.put("age", 35);
userCollection.insert(user);
```

Listing 6.2: Creation of a new user document.

The same class is also used to make queries, then it is used as a description for the desired objects. Listing 6.3 shows how all users with the name “Hugo” can be retrieved and iterated. If a query needs to contain a range condition instead of a certain value for an attribute, then it is possible to add a condition to an attribute instead of a value, like in listing 6.4.

Some queries made by Tricia contain conditions that are connected with an *OR* operator, which is not directly supported by MongoDB. But these queries use the *OR* only for conditions on the same attribute and look like this in SQL:

```
select * from user where (type = 'person' or type = 'admin' or type
   = 'moderator')
```

This special case can be expressed in MongoDB with the *in* operator, which is used as a condition to restrict the allowed values to a specific set. Listing 6.5 shows how this can be used to express the previous SQL statement for MongoDB.

```
BasicDBObject queryUser = new BasicDBObject();
      queryUser.put("name", "Hugo");

DBCursor cur = userCollection.find(queryUser);
while(cur.hasNext()) {
          System.out.println(cur.next());
}
```

Listing 6.3: Query for all users with the name "Hugo".

```
BasicDBObject queryUser = new BasicDBObject();
      queryUser.put("age",new BasicDBObject("$gt", 30));

DBCursor cur = userCollection.find(queryUser);
while(cur.hasNext()) {
          System.out.println(cur.next());
}
```

Listing 6.4: Query for all users with a age bigger than 30.

```
BasicDBObject queryUser = new BasicDBObject();
BasicDBList types = new BasicDBList(); // A List of types
       types.put(1,"person");
             types.put(2,"admin");
             types.put(3,"moderator");

queryUser.put("type", new BasicDBObject("$in", typeList) );

DBCursor cur = userCollection.find(queryUser);
while(cur.hasNext()) {
          System.out.println(cur.next());
}
```

Listing 6.5: Query for all user with the types person, admin or moderator.

If Tricia would use the *OR* operator in other cases, then the persistence layer would need to split the query for each *OR* operator and would have to merge the results in the application layer.

Sort orders in MongoDB are not expressed inside the query itself, instead they are added to the cursor. Listing 6.6 shows how all users can be retrieved in order of their age.

```
DBCursor cur = userCollection.find();
//Set the the cursor to sort by age in ascending order
cur.sort(new BasicDBObject("age", 1) );
while(cur.hasNext()) {
          System.out.println(cur.next());
}
```

Listing 6.6: Retrieve all users ordered by their age.

6.4 Implementation of the MongoStore

The main functionalities of the MongoStore are in the class *MongoQuery*. This is because all operations that are based on a query, like counting query results, deleting with a query and querying itself require that the query is first transformed into statements for MongoDB. Not being supported by MongoDB, join operations must be handled by the application itself, which is in this case implemented in the class *MongoQuery*.

A *MongoQuery* is instantiated with a Tricia query and the container on which the operation should be performed. The resulting MongoQuery object can then be used to perform operations such as counting, deleting or building an iterator with the query results. The source code of the *MongoQuery* class can be found in Appendix A.

6.4.1 Query Building

Queries in Tricia are formulated using Java objects. Figure 6.6 shows an overview of the classes that represent queries in Tricia.

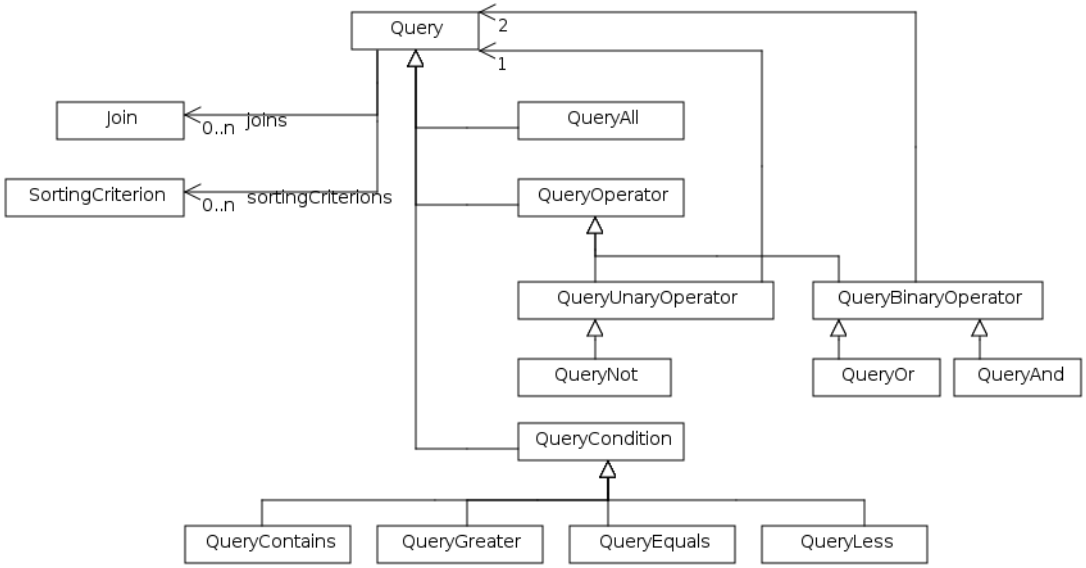


Figure 6.6: Overview of the classes that represent queries in Tricia.

To transform a Tricia query to MongoDB queries, the method *buildQuery* traverses the object tree of the query with a visitor. In most cases this method only has to put the attributes of the query conditions into MongoDB objects. If the given query contains a join operation, then the algorithm has to build two separate queries and has to remember the attribute names that are used for the join operation. The part of the query that has to be performed first is stored in the variable *joinedQuery*.

There are two other special cases besides the handling of joins. The first one is the transformation of *or* operators into one *in* operator for queries on the *type* attribute, which is already explained in the previous chapter. The other one is the handling of the *not* operator. Because the *not* operator is used by Tricia only to check if an attribute is not null, it can be transformed into MongoDB's *exists* operator. If the *not* would be used otherwise, then MongoDB's *not* operator could be used or it could be eliminated by negating all conditions it is applied to.

6.4.2 Iterators

Query results are returned by the *Container* class as Iterators and it supports querying for the identifier only or for the complete content, the *MongoQuery* class provides an *IdIterator* and a *ContentIterator*. Those two iterators can encapsulate either a *SimpleIterator* or a *JoinIterator* if the query contains a join operator.

The *JoinIterator* is the most complex one, because it has to handle the join operations. The iterator uses two MongoDB cursor for the query, one for the first part of the query and one for the second part. The second cursor must be rebuild every time the first cursor is moved one position forward. And the first cursor is moved forward, when the second cursor has reached its end.

To be able to determine if the iterator has a next element, the *JoinedIterator* always needs to be a step ahead. So the *next* methods returns the last fetched object and then searches for the following element.

6.4.3 Count and Delete

The count and delete operations are very straight forward for queries without a join operation. For queries with a join, they have to first iterate over the results of the *joinedQuery* and then build the query for the delete or count operation for each element in the first result set in order to perform the join.

7 Evaluation

This chapter compares the performance of the MongoDB store against the stores for MySQL [23] and the in memory version of HSQL [27]. In order to do this a *MonitorStore* was implemented, which is a generic Tricia store that can encapsulate any other store and monitors the performance of all operations and queries. This does not only allow the comparison of the overall performance of the stores but also the comparison of the efficiency of different operations. Figure 7.1 shows a class diagram of the *MonitorStore*.

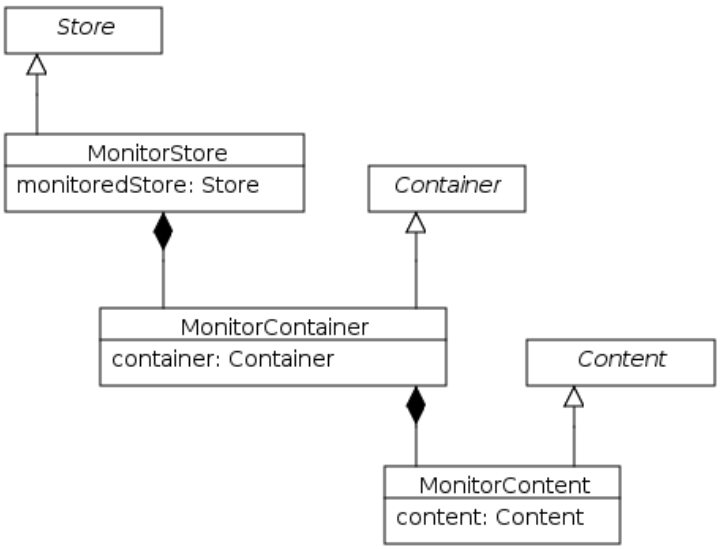


Figure 7.1: Overview of the *MonitorStore*.

7.1 Performance for the test suits

Tricia has a set of test suits that are used to test the current implementation for bugs. This test cases are not designed as a performance benchmark, they do for example not fill the databases with a big amounts of data. But they cover almost

anything of the database functionalities that are used by Tricia. So the results of this test can only give a first impression of the performance of the different databases and does not reflect the performance with real data. This test was performed on a single machine with an Intel(R) Core(TM)2 CPU 4300 clocked at 2,7 GHz and 4GB memory running with Ubuntu 9.10.

Table 7.1 shows some of the results of this test. The biggest difference is the write performance, which is not a big surprise, because MySQL writes everything first into a transaction log file on the hard disk, whereas HSQL and MongoDB are writing asynchronously to the disk.

Furthermore, HSQL and MongoDB can also perform the queries a lot quicker than MySQL, this might also be because they can store everything in memory. The fact that the join operations were performed inside the application layer while using MongoDB did not have a strong impact during this test, probably because of the small amount of data.

	MySQL	HSQL	MongoDB
Total time spent for db operations (exluding constraint creation)	30min	51s	41s
Slowest Operation with average time	Writing one object into WikiPage in 202ms	Getting one object from Job in 3ms	Getting one object from Job in 11ms
Average time for writing one object into WikiPage	102ms	2ms	1ms
Average time for getting one object from WikiPage	2ms	1ms	2ms
Average time of the slowest count operation	2ms	1ms	3ms
Average times of the five slowest queries in ms	13, 5, 5, 4, 2	3, 3, 1, 1, 1	2, 1, 1, 1, 1

Table 7.1: Performance results of the first test.

7.2 Performance with real world data

For testing the read performance of the MongoDB store with real world data, the database content of a Tricia installation with approximately 440.000 rows was used. The test itself consisted of 20 simulated users who vised the start page, some wikis and group pages simultaneously. Each test run made 1600 requests to the Tricia instance and was performed on a virtual machine with Windows Vista 64 Bit on an Intel Xeon processor with 2,5GHz and 1,5GB of RAM.

Table 7.2 shows some of the results of this test. MongoDB performed most queries faster than any of the two other databases. The query with the slowest average time to MongoDB took a little bit longer than the slowest query in MySQL, however MongoDB performed the next slowest queries a lot faster than MySQL. In total MongoDB was more than three times quicker as MySQL and more than two times faster than HSQL, despite that the join operations were not performed inside the database. The results of the MonitorStore showed that the join operations performed by the MongoQuery class did not have a noticeable impact on the performance.

The total times in table 7.2 show how much time was spent for the five database operations, which consumed the most time during the test.

	MySQL	HSQL	MongoDB
Total time spent for db operations	3375s	2315s	948s
Average times of the five slowest operations in ms	1140, 890, 775, 763, 578	1461, 777, 648, 547, 491	1306, 208, 142, 139, 109
The five greatest total times in s	1137, 494, 404, 297, 136	970, 270, 209, 186, 121	433, 150, 105, 90, 56

Table 7.2: Performance results with real world data.

8 Conclusion

This work has shown that there are a lot of differences between current NoSQL implementations. This is not a surprising result, because none of the examined databases aim to be the perfect solution for every problem. Instead each database has its own properties and features, so that developers have to choose the right database depended on the requirements of their project. Therefore this work compared the trade-offs of the examined database implementations and explained why these are necessary.

Furthermore was an Object-relational Persistence Layer based on a NoSQL database for a Web Collaboration and Knowledge Management Software successfully implemented. This showed that NoSQL databases can fulfill the requirements of such an application. It has also shown that the usage of a NoSQL database can increase the performance significantly, even on a single machine. Aside from this was the development process for the replacement of a persistence layer of an already existing application outlined.

Using MongoDB for the persistence layer also has some disadvantages compared to established relational databases like MySQL. The biggest problem with MongoDB is the durability. By using memory mapped files, MongoDB becomes very vulnerable to power failures and other kinds of system failures [6].
The other problem is the administration, because MongoDB has no built in graphical configuration tools or graphical query browsers yet. There are some projects which attempt to provide graphical user interfaces for the administration of MongoDB databases [1], but they are currently not in a state comparable to the ones of the already established databases.

The prototype implementation instruments join operations that are performed by the client to realize the relations between the entities. Even though this implementation was faster then the counterparts for MySQL, the performance could be even more increased through the usage of nested documents in MongoDB. But this would have required more fundamental changes in the architecture of the persistence layer.

Another interesting topic for future works could be the implementation of support layers for other NoSQL databases for Tricia, which would give the opportunity to compare the performance of them with real world data.

A Listing of MongoQuery.java

```
package de.infoasset.platform.store.mongodb;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;

import com.google.common.collect.Maps;
import com.mongodb.BasicDBList;
import com.mongodb.BasicDBObject;
import com.mongodb.DBCursor;

import de.infoasset.platform.services.Loggers;
import de.infoasset.platform.store.AbstractAttribute;
import de.infoasset.platform.store.AbstractQuery;
import de.infoasset.platform.store.AbstractQueryVisitor;
import de.infoasset.platform.store.Container;
import de.infoasset.platform.store.Content;
import de.infoasset.platform.store.Join;
import de.infoasset.platform.store.Query;
import de.infoasset.platform.store.QueryCondition;
import de.infoasset.platform.store.QueryEquals;
import de.infoasset.platform.store.QueryGreater;
import de.infoasset.platform.store.QueryGreaterOrEqual;
import de.infoasset.platform.store.QueryLess;
import de.infoasset.platform.store.QueryLessOrEqual;
import de.infoasset.platform.store.QueryNot;
import de.infoasset.platform.store.SortingCriterion;

public class MongoQuery {

    /** The Container against this query is made. */
    MongoContainer container;

    /** The query as mongodb object. */
    BasicDBObject mongoQuery = new BasicDBObject();

    /** The second part of the query, if the query contains a join
        operation. */
    BasicDBObject joinedQuery = new BasicDBObject();

    /* True if the query contains a join operation. */
```

```
    boolean join = false;

    /* The container that is used for the joinedQuery. */
    MongoContainer joinContainer;

    /*
     * The name of the attribute which is used to join the queries on
        the left
     * side of the query.
     */
    String leftJoinAttribute;

    /*
     * The name of the attribute which is used to join the queries on
        the side
     * of the joined query.
     */
    String rightJoinAttribute;

    /* The ordering of the query. */
    BasicDBObject orderBy = new BasicDBObject();

    /* The ordering of the join part of the query. */
    BasicDBObject joinOrderBy = new BasicDBObject();

    public MongoQuery(MongoContainer mongoContainer, AbstractQuery q,
            List<String> types) {
        this.container = mongoContainer;
        buildQuery(q,types);
    }

    /* Analyzes queries and prepares them for mongodb. */
    private void buildQuery(AbstractQuery q, List<String> types) {
        if (!(q instanceof Query)) {
            if (Loggers.mongoLog.isDebugEnabled()) {
                Loggers.mongoLog.debug("unsupported query type "
                        + q.getClass());
            }
        }
        Query query = (Query) q;
        if (Loggers.mongoLog.isDebugEnabled()) {
            Loggers.mongoLog.debug("buildQuery for: " + query
                    + " in " + container.getTableName());
        }

        // Create type conditions
        if (types != null) {
            if (types.size() == 1) {
                mongoQuery.put("type",types.get(0));
            } else if (types.size() > 1) {
                // use the $in operator for multiple types
```

```
90                  BasicDBList typeList = new BasicDBList();
                    int i = 0;
                    for (String type : types) {
                        typeList.put(i++,type);
                    }
95                  mongoQuery.put("type",new BasicDBObject("$in",
                            typeList));
                }
            }

100         // visit all parts of the query
            query.visit(new AbstractQueryVisitor() {
                // this map stores all already visited equals conditions
                HashMap<QueryEquals, QueryEquals> used = Maps
                        .newHashMap();
105
                @Override
                protected void visitOp(Query q) {
                    if (used.containsKey(q))
                        // do not look again on an already visited query
110                     return;
                    // handle join operations
                    for (Join j : q.getJoins()) {
                        Container cont1 = castToMongoContainer(j
                                .getFirstAttributeSignatures()
115                             .getContainer());
                        Container cont2 = castToMongoContainer(j
                                .getSecondAttributeSignatures()
                                .getContainer());
                        if (joinContainer != null) {
120                         if (Loggers.mongoLog.isDebugEnabled()) {
                                Loggers.mongoLog
                                        .debug("Queries with more than one
                                             join operation are not
                                            supported!");
                            }
                            Thread.dumpStack();
125                         if (MongoContainer.stopOnError())
                                System.exit(-1);
                        }

                        if (cont1 != container) {
130
                            joinContainer = castToMongoContainer(cont1);
                            leftJoinAttribute = AbstractAttribute
                                    .getColumnName(j
                                            .getSecondAttributeSignatures());
135                         rightJoinAttribute = AbstractAttribute
                                    .getColumnName(j
                                            .getFirstAttributeSignatures());
                        } else {
```

```
                joinContainer = castToMongoContainer(cont2);
                rightJoinAttribute = AbstractAttribute
                        .getColumnName(j
                                .getSecondAttributeSignatures());
                leftJoinAttribute = AbstractAttribute
                        .getColumnName(j
                                .getFirstAttributeSignatures());
            }
            // convert the id names to MongoDB id names
            if (rightJoinAttribute.equals("id"))
                rightJoinAttribute = "_id";
            if (leftJoinAttribute.equals("id"))
                leftJoinAttribute = "_id";
            // this query obviously has a join operation
            join = true;
        }
        // currentQuerry points either to mongoQuerry or to
            the
        // joinedQuerry
        // depending on the container of the condition
        BasicDBObject currentQuery = mongoQuery;
        if (q instanceof QueryCondition) {
            QueryCondition c = (QueryCondition) q;

            Container cont = c.getAttributeSignature()
                    .getContainer();

            if (!cont.getTableName().equals(
                    container.getTableName())) {
                join = true;
                // this condition belongs to the joinedQuerry
                currentQuery = joinedQuery;
            }

            // translate the different types of conditions
                into MongoDB
            // conditions
            if (q instanceof QueryEquals) {
                QueryEquals qe = (QueryEquals) q;
                currentQuery.put(AbstractAttribute
                        .getColumnName(qe
                                .getAttributeSignature()),qe
                        .getValue());
            } else if (q instanceof QueryGreater) {
                QueryGreater qr = (QueryGreater) q;
                BasicDBObject b = new BasicDBObject();
                b.put(AbstractAttribute.getColumnName(qr
                        .getAttributeSignature()),qr
                        .getValue());
                currentQuery.put("&gt",b);
            } else if (q instanceof QueryGreaterOrEqual) {
```

```
                        QueryGreaterOrEqual qr =
                            (QueryGreaterOrEqual) q;
                        BasicDBObject b = new BasicDBObject();
                        b.put(AbstractAttribute.getColumnName(qr
                                .getAttributeSignature()),qr
                                .getValue());
                        currentQuery.put("&gte",b);
                    } else if (q instanceof QueryLess) {
                        QueryLess qr = (QueryLess) q;
                        BasicDBObject b = new BasicDBObject();
                        b.put(AbstractAttribute.getColumnName(qr
                                .getAttributeSignature()),qr
                                .getValue());
                        currentQuery.put("&lt",b);
                    } else if (q instanceof QueryLessOrEqual) {
                        QueryLessOrEqual qr = (QueryLessOrEqual) q;
                        BasicDBObject b = new BasicDBObject();
                        b.put(AbstractAttribute.getColumnName(qr
                                .getAttributeSignature()),qr
                                .getValue());
                        currentQuery.put("&lte",b);
                    } else {
                        if (Loggers.mongoLog.isDebugEnabled()) {
                            Loggers.mongoLog.error("condition type "
                                    + q.getClass()
                                    + " is not supported");
                        }
                    }
                } else if (q instanceof QueryNot) {
                    // handle not operations, which are only supported
                    // if they are used to determine if an attribute
                        is set
                    // and can therefore be transformed into an
                        $exists
                    // condition
                    QueryNot qn = (QueryNot) q;
                    if (qn.getTerm() instanceof QueryEquals) {
                        QueryEquals qe = (QueryEquals) qn.getTerm();
                        currentQuery.put(AbstractAttribute
                                .getColumnName(qe
                                        .getAttributeSignature()),
                                new BasicDBObject("$exists",true));
                        used.put(qe,qe);
                    } else {
                        Loggers.mongoLog
                                .error("cannot process not operator");

                    }
                }
            }
        });
```

```
        // Handle the sort order:
        for (SortingCriterion sc : query.getAllSortingCriterions()) {
            String name = AbstractAttribute.getColumnName(sc
240                 .getAttributeSignature());
            // convert id attribute name to MongoDB
            if (name.equals("id"))
                name = "_id";

245             if (!sc.getAttributeSignature().getContainer()
                        .getTableName().equals(container.getTableName()))
                            {
                    // this ordering instruction is for the joinedQuery
                    if (sc.isAscending()) {
                        joinOrderBy.put(name,1);
250                 } else {
                        joinOrderBy.put(name,-1);
                    }
                } else {
                    // this one for the mongoQuery
255                 if (sc.isAscending()) {
                        orderBy.put(name,1);
                    } else {
                        orderBy.put(name,-1);
                    }
260                 if (join) {
                        Loggers.mongoLog
                                .error("Sorting in the second part of a
                                    joined query is not supported");
                    }
                }
265         }

        // some debug code:
        if (Loggers.mongoLog.isDebugEnabled()) {
            Loggers.mongoLog.debug("created mongoQuery: "
270                     + mongoQuery);
            if (join) {
                Loggers.mongoLog.debug("created joinedQuery: "
                        + joinedQuery);
                Loggers.mongoLog.debug("join left: "
275                     + leftJoinAttribute + " right: "
                        + rightJoinAttribute);
                Loggers.mongoLog.debug("orderBy: " + orderBy);
            }
        }
280 }

    /*
     * @return An iterator over the content objects in the result set
        of this
```

```
     * query
     */
    public Iterator<Content> getContentIterator(int limit) {
        if (join) {
            return new ContentIterator(new JoinIterator(limit));
        } else {
            DBCursor cursor = container.getCollection().find(
                    mongoQuery);
            if (limit > 0)
                cursor.limit(limit + 1);
            cursor.sort(orderBy);
            return new ContentIterator(new SimpleIterator(cursor));
        }
    }

    /* @return An iterator of the idententifiers of the query results.
     */
    public Iterator<String> getIdIterator(int limit) {
        if (join) {
            return new IdIterator(new JoinIterator(limit));
        } else {
            DBCursor cursor = container.getCollection().find(
                    mongoQuery);
            if (limit > 0)
                cursor.limit(limit + 1);
            cursor.sort(orderBy);
            return new IdIterator(new SimpleIterator(cursor));
        }
    }

    /* Deletes all objects in the database that are in the query
       result set. */
    public void delete() {
        if (join) {
            DBCursor cursor = joinContainer.getCollection().find(
                    joinedQuery);
            while (cursor.hasNext()) {
                BasicDBObject obj = (BasicDBObject) cursor.next();
                mongoQuery.put(leftJoinAttribute,obj
                        .get(rightJoinAttribute));
                try {
                    container.getCollection().remove(mongoQuery);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }

        } else {
            container.getCollection().remove(mongoQuery);
        }
    }
```

```
    public MongoContainer castToMongoContainer(Container cont) {
        if (cont instanceof MongoContainer) {
            return (MongoContainer) cont;
        } else {
            return (MongoContainer) container.getStore()
                    .getContainer(cont.getTableName());
        }

    }

    /* @return The size of the result set of this query. */
    public int count() {
        if (join) {
            DBCursor cursor = joinContainer.getCollection().find(
                    joinedQuery);
            int sum = 0;
            HashMap<String, String> used = Maps.newHashMap();
            while (cursor.hasNext()) {
                BasicDBObject obj = (BasicDBObject) cursor.next();
                String id = obj.getString("_id").toString();
                if (!used.containsKey(id)) {
                    mongoQuery.put(leftJoinAttribute, obj
                            .get(rightJoinAttribute));
                    sum += container.getCollection().find(mongoQuery)
                            .count();
                }
            }
            return sum;
        } else {
            return container.getCollection().find(mongoQuery).count();
        }
    }

    /* An Iterator for queries without joins. */
    private class SimpleIterator implements Iterator<BasicDBObject> {
        DBCursor cursor;

        public SimpleIterator(DBCursor cursor) {
            this.cursor = cursor;
        }

        @Override
        public boolean hasNext() {
            return cursor.hasNext();
        }

        @Override
        public BasicDBObject next() {
            BasicDBObject result = (BasicDBObject) cursor.next();
            return result;
```

```
        }

        @Override
        public void remove() {
            cursor.remove();
        }

    }

    /* This Iterator is used for queries with a join operation. */
    private class JoinIterator implements Iterator<BasicDBObject> {
        /* Cursor over the results of the joinedQuery. */
        DBCursor joinedCursor;

        /* Cursor over the final result sets. */
        DBCursor cursor;

        /* The object that was last returned by next(). */
        BasicDBObject current;

        /* The next object that will be returned by next(). */
        BasicDBObject next;

        int limit;

        int n = 0;

        /* Already used objects from the joinedCursor. */
        HashMap<String, String> used = Maps.newHashMap();

        public JoinIterator(int limit) {
            if (joinContainer == null) {
                Loggers.mongoLog.error("joinContainer null");
            }
            // init the cursor of the joinedQuery
            joinedCursor = joinContainer.getCollection().find(
                    joinedQuery);
            // some debug code
            if (Loggers.mongoLog.isDebugEnabled()) {
                Loggers.mongoLog.debug("JoinIterator: cursor "
                        + joinedQuery + "  cursorcount: "
                        + joinedCursor.count());
                Loggers.mongoLog.debug("JoinIterator: cursor on "
                        + joinContainer.getCollection().getName());
                Loggers.mongoLog
                        .debug("JoinIterator: cursor collection size "
                                + joinContainer.getCollection()
                                        .getCount());
            }

            joinedCursor.sort(joinOrderBy);
```

```
435             /* Find the first element. */
                searchNext();
                this.limit = limit;
            }

440         @Override
            public boolean hasNext() {
                return next != null;
            }

445         @Override
            public BasicDBObject next() {
                current = next;
                searchNext(); // the iterator is always a step ahead to
                    determine if
                // there is a next element
450             n++;
                return current;
            }

            /*
455          * Find the next element. In order to do this, this method has
                 to
             * perform the join operation.
             */
            private void searchNext() {
                next = null;
460             if (cursor == null || !cursor.hasNext()) {
                    // if the cursor cannot return anymore elements, then
                        increment
                    // the joinedCursor
                    // until the joinedCursor has reached its end or a new
                         cursor is
                    // found with
465                 // more result elements
                    while (joinedCursor.hasNext()
                            && (cursor == null || !cursor.hasNext())) {
                        String value = joinedCursor.next().get(
                                rightJoinAttribute).toString();
470                     if (!used.containsKey(value)) {
                            used.put(value, value);
                            mongoQuery.put(leftJoinAttribute, value);
                            if (Loggers.mongoLog.isDebugEnabled()) {
                                Loggers.mongoLog
475                                     .debug("JoinIterator: cursor "
                                            + mongoQuery);
                            }
                            cursor = container.getCollection().find(
                                    mongoQuery);
480                         cursor.limit(limit);
                        }
```

```
                }
            }
            // if the cursor has another element, then get it
            if (cursor != null && cursor.hasNext()) {
                next = (BasicDBObject) cursor.next();
                if (Loggers.mongoLog.isDebugEnabled()) {
                    Loggers.mongoLog.debug("JoinIterator: next "
                            + next);
                }

            }
        }

        @Override
        public void remove() {
            container.deleteContent(current.get("_id").toString());
        }
    }

    /* An Iterator that returns Content objects. */
    private class ContentIterator implements Iterator<Content> {

        Iterator<BasicDBObject> iter;

        public ContentIterator(Iterator<BasicDBObject> iter) {
            this.iter = iter;
        }

        @Override
        public boolean hasNext() {
            return iter.hasNext();
        }

        @Override
        public Content next() {
            MongoContent content = container.contentFactory();
            BasicDBObject dbObject = iter.next();
            content.initialize(container,dbObject.get("_id")
                    .toString(),null);
            content.read(dbObject);
            return content;
        }

        @Override
        public void remove() {
            iter.remove();
        }
    }

    /* An Iterator that returns only the identifiers of the objects.
       */
```

```
    private class IdIterator implements Iterator<String> {

        Iterator<BasicDBObject> iter;

        public IdIterator(Iterator<BasicDBObject> iter) {
            this.iter = iter;
        }

        @Override
        public boolean hasNext() {
            return iter.hasNext();
        }

        @Override
        public String next() {
            return iter.next().get("_id").toString();
        }

        @Override
        public void remove() {
            iter.remove();
        }
    }
}
```

Bibliography

[1] *Tim's Daft Junk*. http://blog.timgourley.com/post/453680012/tuesday-night-tech-mongodb-ui-edition, 2010. [Online; accessed 6-April-2010].

[2] 10GEN.COM: *BSON - MongoDB*. http://www.mongodb.org/display/DOCS/BSON, 2009. [Online; accessed 17-November-2009].

[3] 10GEN.COM: *Home - MongoDB*. http://mongodb.org/, 2009. [Online; accessed 17-November-2009].

[4] 10GEN.COM: *Querying - MongoDB*. http://www.mongodb.org/display/DOCS/Querying, 2009. [Online; accessed 17-November-2009].

[5] 10GEN.COM: *Querying - MongoDB*. http://www.mongodb.org/display/DOCS/MapReduce, 2009. [Online; accessed 20-November-2009].

[6] 10GEN.COM: *Durability and Repair - MongoDB*. http://www.mongodb.org/display/DOCS/Durability+and+Repair, 2010. [Online; accessed 6-April-2010].

[7] A.BREWER, DR.ERIC: *PODC keynote*. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, 2000. [Online; accessed 10-Oktober-2009].

[8] ACMQUEUE: *A Conversation with Werner Vogels*. http://queue.acm.org/detail.cfm?id=1142065, 2006. [Online; accessed 10-Oktober-2009].

[9] AMAZON: *Amazon Simple Storage Service*. http://docs.amazonwebservices.com/AmazonS3/latest/index.html?Introduction.html, 2009. [Online; accessed 14-Oktober-2009].

[10] AMAZON: *Amazon Simple Storage Service*. http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/, 2009. [Online; accessed 5-November-2009].

[11] AMAZON: *Amazon Simple Storage Service*. http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/index.html?SDBLimits.html, 2009. [Online; accessed 5-November-2009].

[12] APACHE SOFTWARE FOUNDATION: *The Apache Cassandra Project*. http://incubator.apache.org/cassandra/, 2009. [Online; accessed 25-November-2009].

[13] Apache Software Foundation: *Welcomde to Hadoop!* `http://hadoop.apache.org/`, 2009. [Online; accessed 25-November-2009].

[14] Apache Software Foundation: *Welcome to HBase!* `http://hadoop.apache.org/hbase/`, 2009. [Online; accessed 25-November-2009].

[15] Barrett, Ryan: *Under the Covers of the Google App Engine Datastore.* `http://sites.google.com/site/io/under-the-covers-of-the-google-app-engine-datastore`, 2010. [Online; accessed 20-Januar-2010].

[16] Blue Coast Web: *Hypertable An Open Source, High Performance, Scalable Database.* `http://www.hypertable.org`, 2009. [Online; accessed 25-November-2009].

[17] Burrows, M. et al.: *The Chubby lock service for loosely-coupled distributed systems.* In *Proc. of the 7th OSDI*, volume 11, 2006.

[18] Büchner, T.: *Introspektive modellgetriebene Softwareentwicklung.* VDM Verlag Dr. Müller, 2007.

[19] Büchner, T. and F. Matthes: *Introspective Model-Driven Development.* Lecture Notes in Computer Science, 4344:33, 2006.

[20] Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes and R.E. Gruber: *Bigtable: A distributed storage system for structured data.* In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDIâ06)*, 2006.

[21] cloudant.com: *Cloudant: CouchDB hosting.* `http://cloudant.com`, 2010. [Online; accessed 15-Januar-2010].

[22] Compete.com: *Site Profile of amazon.com.* `http://siteanalytics.compete.com/amazon.com/`, 2009. [Online; accessed 10-Oktober-2009].

[23] Corporation, Oracle: *MySQL :: The world's most popular open source database.* `http://www.mysql.com`, 2010. [Online; accessed 19-March-2010].

[24] Crockford, Douglas: *RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON).* `http://tools.ietf.org/html/rfc4627`, 2006. [Online; accessed 19-Dezember-2009].

[25] Dean, J. and S. Ghemawat: *MapReduce: Simplified data processing on large clusters.*

[26] DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels: *Dynamo: amazon's highly available key-value store.* ACM SIGOPS Operating Systems Review, 41(6):220, 2007.

[27] Development Group, The hsql: *HSQLDB*. http://www.mysql.com, 2010. [Online; accessed 19-March-2010].

[28] Ghemawat, S., H. Gobioff and S.T. Leung: *The Google file system*. ACM SIGOPS Operating Systems Review, 37(5):43, 2003.

[29] Gilbert, Seth and Nancy Lynch: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News, 33(2):51–59, 2002.

[30] Google: *Google App Engine - Google Code*. http://code.google.com/intl/de/appengine/docs/whatisgoogleappengine.html, 2010. [Online; accessed 20-Januar-2010].

[31] Google: *How Index Building Works - Google App Engine - Google Code*. http://code.google.com/intl/de/appengine/articles/index_building.html, 2010. [Online; accessed 20-Januar-2010].

[32] Google: *protobuf - Project Hosting on Google Code*. http://code.google.com/p/protobuf/, 2010. [Online; accessed 22-Januar-2010].

[33] Google: *Queries and Indexes - Google App Engine - Google Code*. http://code.google.com/intl/de/appengine/docs/java/datastore/queriesandindexes.html, 2010. [Online; accessed 20-Januar-2010].

[34] Ho, Ricky: *Pragmatic Programming Techniques: CouchDB Implementation*. http://horicky.blogspot.com/2008/10/couchdb-implementation.html, 2010. [Online; accessed 16-Januar-2010].

[35] Infoasset: *infoAsset : infoAsset - Tricia*. http://www.infoasset.de, 2010. [Online; accessed 23-February-2010].

[36] J. Chris Anderson, Jan Lehnardt, Noah Slater: *CouchDB: The Definitive Guide*. http://books.couchdb.org/relax/, 2009. [Online; accessed 17-Dezember-2009].

[37] Kang, B., R. Wilensky and J. Kubiatowicz: *The hash history approach for reconciling mutual inconsistency*. In *INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS*, volume 23, pages 670–677. Citeseer, 2003.

[38] Kevin Ferguson, Vijay Raghunathan, Randall Leeds: *Lounge*. http://tilgovi.github.com/couchdb-lounge/, 2010. [Online; accessed 15-Januar-2010].

[39] Lamport, L.: *Paxos made simple*. ACM SIGACT News, 32(4):18–25, 2001.

[40] Lamport, Leslie: *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM, 21(7):558–565, 1978.

[41] Ltd., Canonical Group: *Ubuntu One : Home*. https://one.ubuntu.com/, 2010. [Online; accessed 14-Januar-2010].

[42] Nussbaum, Daniel and Anant Agarwal: *Scalability of parallel machines.* Commun. ACM, 34(3):57–61, 1991.

[43] Project, The Linux Information: *Scalable definition by The Linux Information Project.* http://www.linfo.org/scalable.html, 2006. [Online; accessed 15-March-2010].

[44] Reed, D. P.: *NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM.* Technical Report, Cambridge, MA, USA, 1978.

[45] Skeen, Dale: *Nonblocking commit protocols.* In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM.

[46] Strozzi, Carlo: *NoSQL - A Relational Database Management System.* http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, 2010. [Online; accessed 6-April-2010].

[47] Stuntz, Craig: *Craig Stuntz's Weblog - Multiversion Concurrency Control Before InterBase.* http://blogs.teamb.com/craigstuntz/dtpostname/multiversionconcurrencycontrolbeforeinterbase/#2718, 2005. [Online; accessed 9-November-2009].

[48] Vogels, Werner: *Eventual Consistenz - Revisited - All Things Distributed.* http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, 2008. [Online; accessed 6-November-2009].

[49] Wikipedia: *ACID — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=ACID&oldid=343104240, 2010. [Online; accessed 10-February-2010].