Comparison of Distribution Technologies in Different NoSQL Database Systems

Studienarbeit

Institute of Applied Informatics and Formal Description Methods (AIFB)
Karlsruhe Institute of Technology (KIT)

Dominik Bruhn

Reviewer: Prof. Dr. Tai
2nd Reviewer: Prof. Dr. Studer
Advisor: Markus Klems

21 Feb 2011 - 21 May 2011

Dominik Bruhn
Willy-Andreas-Allee 7
76131 Karlsruhe

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Karlsruhe, den 6.8.2011

____
Dominik Bruhn

Contents

List of Figures

List of Tables

1 Introduction

1.1 History of Database Systems

Traditionally, most organizations used relational database management systems (*RDMS*) for handling their data [Stonebraker, 2010]. When the amount of data and users got bigger and bigger *scalability* and *distribution* came into focus.
In the 1970's, it surfaced that it is difficult to distribute a RDMS while retaining all the features and guarantees [Vogels, 2009]. With the rise of the large Internet systems, with their huge amount of data and requests, new solutions for the distribution of databases were needed. This is why within the last years alternative data management systems, so-called *NoSQL DMS* (or *NoSQL data stores*), have been created and are becoming more and more important. The term NoSQL is short for "Not Only SQL" and was introduced in 2009, when it was chosen as the title of a conference "for folks interested in distributed structured data storage" [Evans, 2009]. Although at this time, some of the major players in the NoSQL market already existed (like *Apache Cassandra*) or were in internal company use (like *Google BigTable*), a major boost in development has occurred since then.

There is currently no consistent definition of NoSQL in literature. Those systems can differ from classic RDBMS in various ways, whereas not all systems incorporate all of these specifications:

- The schema is not fixed.
- Join operations are not supported and must be implemented by hand.
- The interface to the database uses a more lower-level language.
- The data stores often do not attempt to provide all ACID guarantees.
- The systems are built to scale horizontally and are optimized for heavy read or write loads.
- The data is stored in a column- or document-oriented way.

Most of these NoSQL systems support some kind of distribution technology to achieve scalability, although, the technologies which are incorporated in NoSQL systems vary widely with respect to the used procedures and algorithms.
The objective of this work is to compare the distribution technologies used by various NoSQL database stores. Focus is on the analysis, how these technologies can provide scalability to these systems, their strengths and weaknesses regarding high availability and low latency demands.

1.2 Structure

This thesis is divided into six parts: In chapter 2 ideas from two papers published by Google and Amazon are introduced and discussed. Later in chapter 3 selected NoSQL database systems are explained and analyzed in detail. After this, in chapter 4, the systems explained previously will be compared and categorized using a unique and newly created categorization scheme. In chapter 5 some typical use cases for the systems will be discussed and a summary in chapter 6 marks the end of this thesis.

1.3 Scalability

Traditional database systems and the new NoSQL implementations both allow a client to store and retrieve data. The data which is saved in the database is structured as objects or rows.
A system is called scalable, if it can handle the addition of load without changing the fundamentals (e.g. used software or way of access) [Neuman and Neuman, 1994]. An important tool for building a scalable system is *distribution.* On a distributed database, the data is distributed over several machines or nodes. There are three mechanism which are used in a distributed database system to achieve scalability [cf. Neuman and Neuman, 1994]:

1. *Replication*: A replicated database has multiple identical persistent instances of its objects on many nodes, in particular, to guarantee availability.
2. *Fragmentation* means the separation of the data into smaller subsets and the distribution of those subsets onto nodes. This is sometimes also called *sharding.*
3. *Caching:* Copies of a object are temporarily saved on another node for faster access because of the increased locality.

Whenever replication is involved in database systems, attention has to paid to *consistency*: In a consistent database system, every copy of a object will be up-to-date and thus be the same, so that the client gets the same result whichever replica it queries. As it is very difficult to achieve consistency, some systems make lower guarantees: *Eventual consistency* means that the system does not guarantee that all replicas contain the most recent updates, so old data might be present in the system. Nevertheless, the systems guarantees that at some point in the future the updates will be commited on all nodes.

2 Influential papers

2.1 Amazon Dynamo

In 2007 nine Amazon[1] engineers published a paper titled *Dynamo: Amazon's Highly Available Key-value store* [DeCandia et al., 2007]. Amazon had, as many other Internet companies, the problem of a huge growth in data within the last years. This is why they were searching for a reliable and scalable data store.
Reliability is important because every second the data store is unavailable Amazon keeps loosing money, because their online store is also not available to customers.
Scalability was an issue because of the aforementioned grow in data and in accesses.
The problem the engineers soon realized was that scalability implies the usage of many devices, and if a system contains many devices, it is very likely that some of the devices will not work currently. These ever-present outages make it difficult to guarantee reliability.
Out of this dilemma they came up with a software-solution called *Dynamo*. In their paper they only describe the ideas behind the data store, but it was not published as software up to now. Nevertheless, the authors state that they implemented Dynamo using Java as programing language. Amazon uses Dynamo within the company for hundreds of their services, which all have different requirements. This is why the system needs to be flexible for different uses.
For Amazon is was clear that, because of the vast amount of data, this data had to be distributed over several machines. This led to the question, how these machines can be kept in a consistent state and still guarantee fast response times.

2.1.1 Design Considerations

- The data is distributed over many machines: For availability reasons, there need to be replicas on different machines in different data centers. This distribution must be somehow balanced, as it is important that there is no machine which has to handle a majority of data and accesses. Also, the load should be distributed according to the capabilities of the machine.
- Machines will fail and the system must be able to handle this.
- It must always be possible to write data (Dynamo is a so-called "always writable" data store), even if this leads into inconsistencies. This comes from the nature of

[1]`http://www.amazon.com`

Amazons business: It is better to provide a inconsistent service than not to provide a service at all (and loose money).

- Read and write accesses to the data store must be provided within a given time, it is not possible to wait for a request a very long time. Most of their services based upon Dynamo are latency sensitive.
- There is no need to enumerate or iterate the data store. All accesses to the data are made by an key. This key uniquely identifies each object stored in the system. The systems doesn't care what data is effectively stored, it only cares for the key. The data handled is relatively small (less than 1 MB).
- The system is symmetric and decentralized: This means that every node in the system has the same set of responsibilities, there are no "master"-nodes which would lead to a single point of failure. Each node can be queried for each object, if it is not stored on the node, the request will be forwarded to a appropriate node.

2.1.2 System Design

For choosing the nodes where to store the objects, Dynamo relies on a consistent hashing scheme [Karger et al., 1997]. This means that the output range of a hash function is treated as a ring (so the highest value of the hash function wraps around to the lowest value). Each node of the system randomly selects a value withing the range of the hash function. This value is called the "position" of the node on the ring.
As mentioned above, each object is identified by a key. By hashing the key, every data object is assigned to a position on the ring. The node where this object should be stored (the so-called *coordinator node*) is the first node found when walking the ring in clockwise direction. This makes each node responsible for the region of the ring between its predecessor and itself.

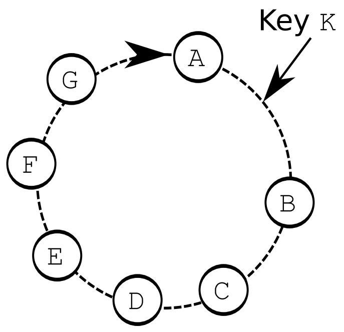

Figure 2.1: Node B is responsible for all keys starting from A up to B. This includes the key K. So the data with key K will be stored on node B.

Because each node in the ring must now on which nodes which objects are stored, there must be a protocol for exchanging information about responsibilities. Resulting from the decentralized structure of Dynamo, there can't be a "controller"-node which maintains the information which nodes are available and which nodes stores which object. For these reasons, the *gossip-based membership protocol* is used: Each node selects some nodes and asks them about their status and their knowledge about other nodes once per second.

2.1.3 Replication Design

Each object which should be stored in the system in replicated at N hosts, where N (the so-called *replication count*) can be configured by the developer. Some of these replicas are written in a asynchronous way, so the system returns after writing the first W copy and writes the remaining (up to $N - W$) copies in background. This so-called write-quorum W can also be configured by the user.
The first copy is stored on the coordinator node, the remaining $N - 1$ replicates are stored on the $N - 1$ clockwise successor nodes in the ring. This leads to the fact that a node is responsible for storing keys in the region between itself and its N^{th} predecessor.

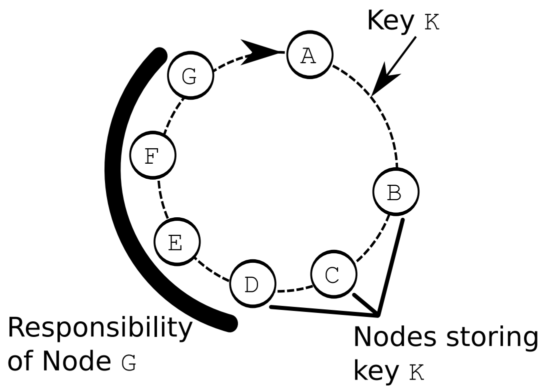

Figure 2.2: Key `K` is stored with a replication count of 3. It is stored on node `B`, while copies are kept on nodes `C` and `D`. Node `G` is responsible for all keys from `D` to `G` (left black arc).

2.1.4 Operation Design

Dynamo only support two basic operations: `put(key, value)` and `get(key)`. There are three configurable values which influence these operations:

1. N, the replication count. As mentioned above, this is the number of nodes that hold a copy of a object. The replication count can be at most the amount of nodes available in the system.
2. W, the write threshold. This specifies how many nodes must participate in a write operation to make it return successfully. W must be $\leq N$.
3. R, the read threshold. Similar, this specifies how many copies must respond with a value before the read request is answered. R must also be $\leq N$.

Different constellations of N, W and R have impact on performance and consistency. When setting R and W such that $R + W > N$, it is guaranteed, that the reader always sees the most recent update, so no inconsistencies can occur (because there is always one node which is in the read quorum and in the write quorum as well). The downside of this setting is that the read or write operations are limited by the slowest node. For this reason, normally R and W are usually less than N for better latency.

2.1.5 Handling Failures

Due to the asynchronous nature of the replication design inconsistencies can occur. Dynamo implements some mechanisms to solve these problems:

2.1.5.1 Conflict Resolution

Even under optimal operation conditions, so if there are no node or network failures, there might be some periods in which the replicas of an object are in an inconsistent state because some copies haven't received the most recent updates yet.
These inconsistencies lead into several different version being read. There are two possibilities for handling these differences: It might be possible for the system to determine the authoritative version without the interaction of the application using the database. This is the simpler case called *syntactic reconciliation.* Most times this happens by selecting the newer version and discarding the older one.
The harder case occurs, if version branching happens, which means that several versions of an object exist and it is not possible to order them by time. This happens if updates are made and there is a network partition (so there are two nodes which are running and accessible, but can't communicate which each other). Under this condition, Dynamo can't use syntactic reconciliation and returns all available versions to the application. Now the application can decide how to merge these versions back into one consistent state. This is called *semantic reconciliation.*

2.1.5.2 Vector Clocks

For finding out which versions can be reconciled on the system and which must be handled on the application side Dynamo uses *Vector Clocks* [see also Lamport, 1978]. In short, each object is assigned a list of $(node, counter)$ tuples. By examining the vector clocks, you can check whether two objects are in a temporal order or happened in parallel.

2.1.5.3 Hinted Handoff

When writing a object with a replication count of N, Dynamo tries to write to the N designated nodes in the ring. If one of these nodes is down or unreachable, Dynamo check the next node on ring until if finds a place where it can store the copy. This temporary replication node saves the copy in a special database together with a hint which node was the intended recipient of the replica. When the failing node recovers, the replica is transferred over to the original node and the temporary copy is deleted. This mechanism, which speeds up recovering after a temporary node failure, is called Hinted Handoff.
So in the example above (see figure 2.2), if node `D` is not available when trying to write a value for key `K`, the next node on the ring, node `E`, is chosen to store the third copy. When node `D` recovers, it moves the replica from node `E` back to itself.

2.1.5.4 Read Repair

If a node hasn't received the latest updates for some of its replicas, this gets detected upon read: If some nodes response with a older version to an read request, the system automatically sends these outdated nodes the most recent versions. This is done after returning the most recent version to the application and so does not slow down the normal operations.

2.1.5.5 Anti Entropy Protocol

If a node is was down and the hint from the hinted handoff is lost (possibly also due to another node failure), the recovering node has old versions. For detecting these inconsistencies the *Anti Entropy Protocol* is used. All nodes periodically exchange hash sums (so-called *Merkle trees*, see also Merkle [1988]) on their data. If the hashes don't match, this means that there are old versions on one of the nodes. These nodes then exchange the real data and so all nodes get the same versions again. Using hashes minimizes the amount of data that needs to be transferred.

2.2 Google BigTable

Google[2] ran into the same set of problems as Amazon. In the time between 2003 and 2006 several papers [Ghemawat et al., 2003; Burrows, 2006; Chang et al., 2006] were published by Google engineers. Although the overall problems, the huge amount of data and the problem of availability, were the same as with Amazon, Google had other priorities: They needed a data store to handle different-size entries, from a few kilobytes to some gigabytes. Also their focus was more on bulk processing than on real time queries. Dynamo is optimized for the later because these request occur the most time in web applications. Google calls his data store *BigTable*. It is used by various applications within the Google company.

2.2.1 Design Considerations

- Google needed a system flexible enough to provide a back end store for various tools, it must provide "scalability, high performance and high availability".
- BigTable focuses on consistency: If the network fails, some nodes or operations get unavailable.
- Data is uniquely identified by a key. The data it self is not unstructured as with Dynamo but instead can be structured using a so-called *column-oriented* design.

[2] `http://www.google.com`

2.2.2 Existing Components

BigTable uses two services for the creation of the distributed system which were developed seperately:

2.2.2.1 Google File System

The Google File System (*GFS* published in 2003 by Ghemawat et al. [2003]) is a distributed file system. This means that large amounts of data are stored on hundreds or even thousands of computers. Because of the need of availability and scalability, GFS distributes the files over several nodes, making replicas and splitting the files for faster access. The system typically stores file in the range from several hundred megabytes. Instead of a typical file system interface (for example via a POSIX-Layer), Google choose to implement the API as a library which is included in every application which accesses the file system.
A GFS cluster is made of a single master and several chunkservers. Each of these run on stock Linux machines. The files, which are stored on the file system, are divided into fixed-size chunks. Each chunk has a unique chunkhandle which identifies it globally. While the chunkserver keep the chunks in their memory and store them on hard disk, the master keeps track of the location of the chunks, their metadata and those of the files (including a mapping between files and their chunks). To increase availability the chunks are replicated over several chunkservers. When writing or reading files clients first ask the master for the appropriate chunkservers storing the file and then communicate with this chunkservers directly without further invocation of the master. This prevents the single master from becoming a bottleneck because no data is transferred through the master. The master keeps track of the replication, it creates new copies if nodes go down.
GFS uses a simple and relaxed consistency model: Two data mutations are supported: *record appends* and *writes*. Record appends means that a bunch of data (the "record") is appended to a file at least once at a position the server may choose. The server will try to append the record to a file a second time if the first write was made invalid by a concurrent write. Normal writes, where the client may specify the location in the file where the data should be written, don't guarantee that the data is written to the file at all. If a concurrent operation is made, the data of a write may be completely lost. GFS guarantees that successful writes (this is true for both, record appends and writes) are the same on all replicas except for a small time when the data is transferred between the nodes. The fact that some parts of a file might be invalid (either due to concurrent writes or due to a incomplete replication) must be handled by the application. Google uses checksums to find invalid records and skip them. As appends are the most common operation on a GFS, the missing guarantees on writes are not a problem, but the programmer must be aware of this. If the application needs to lock the access to a file to prevent concurrent access, it can use a lock service like *Chubby*.
Using a schema like this, GFS can be used to save files which can be accessed in a

fast way and can be read and written even if some node fails. This means that GFS implements a distributed file system.

2.2.2.2 Chubby Lock Service

The *Chubby lock service* (published in 2006 by Burrows [2006]) provides a system which is used within a distributed system. The purpose of the service is to allow its clients to synchronize their work and to exchange basic information. The clients can obtain coarse-grained locks from the service. This means that (in contrast to so-called fine-grained locks) the locks are typically held for a long duration (minutes and more). Each lock is identified by a path, similar to paths in a file system. It is possible to associate additional metadata to a lock which can be used by the clients to exchange information. The locks are "advisory", which means that the data for a lock can be read without acquiring it if the client wants to do that. There are some simple operations which the client can issue via a RPC to the server:

- `open()` and `close()`: Opens and closes a lock handle, but does not try to lock.
- `acquire()`: Acquires the lock which is represented by a handle.
- `getContents()`, `setContents()`: Returns and sets the metadata for a lock handle which is stored together with the lock.

Chubby is operated by a single master which stores the lock information. This master continuously replicates the information to some standby masters using the Paxos protocol [Lamport, 2001; Chandra et al., 2007]. If the master fails, a new one is elected and takes over. During this time, (usually between 3 and 30 seconds, the lock service is unavailable. The clients should be able to handle this outages.

2.2.3 BigTable Architecture

BigTable consists of three components: A library which is linked to the clients using the database, a master server and many so-called tablet servers.
The row range of the data is partitioned into many so-called tablets. Each of these tablets is assigned to a single tablet server. As result short row scans are efficient because all entries are stored together in the same tablet on the same server. This tablet server handles the read and write operations to the tablets it serves and splits and redistributes the tablets that have grown too large.
The master keeps track which server stores a specific tablet, it is responsible for tracking the removal, failure and addition of nodes to the system. As with GFS, no data is transferred through the master, it only points out which tablet server can be asked for the information and client can then retrieve the data from this node without intervention of the master. This heavily reduces the load of the master because the client library caches tablet locations. To track tablet servers BigTable uses Chubby: When starting up each tablet server creates and acquires a lock on a uniquely named file. The master monitors the directory where these files are created and can react to certain conditions:

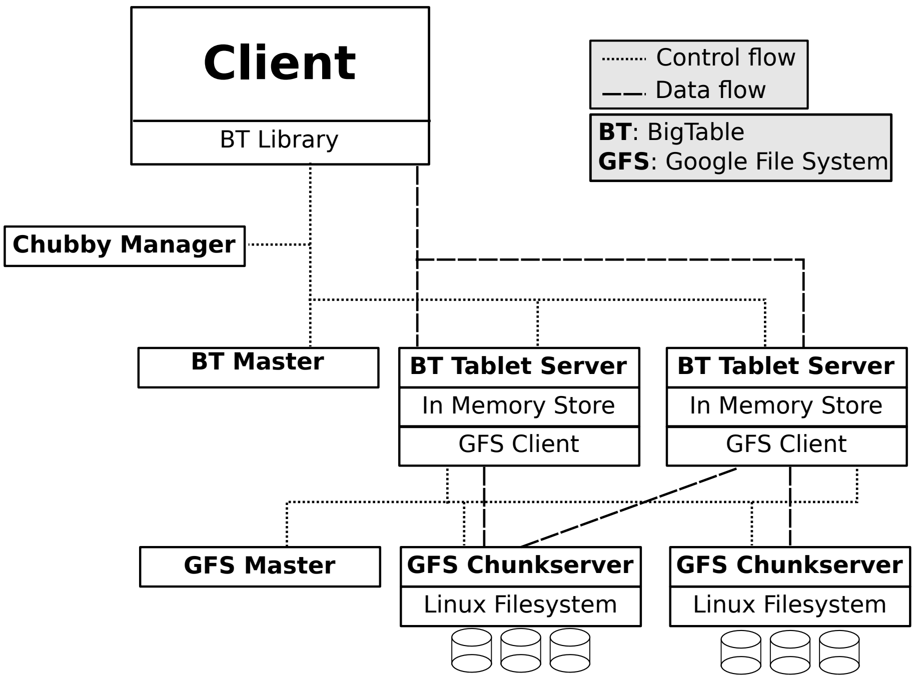

Figure 2.3: Architecture of Google BigTable: BigTable (BT) is created on top of two existing components: Google File System (GFS) and the Chubby lock service. While the control flow (dotted lines) is mainly between clients and the managers, the data (dashed lines) is transferred between the clients and the providers without the master.

If a new file is created (and a lock acquired) this means that a new tablet server is starting up and data can be assigned to this node. If a lock is released (because of a timeout), the master knows that a node lost connection and should be considered down. When starting up the Master itself scans the lock directory in Chubby and then ask every tablet server for their tablet assignments.
The tablet servers hold parts of their tablets which they store in memory while the whole tablet is stored on the distributed GFS. Write requests to a tablet are handled by the matching (single) tablet server. This server appends the write requests to its commit-log (stored on GFS) and then alters it data-set (in memory and on GFS). Read requests are answered by the tablet server either from memory or from the data-files on GFS. Periodically the node also dumps the current dataset onto the GFS and thus makes the commit-log shorter and easier to handle.

2.2.4 Handling Failures

Because of the single master concept BigTable is always consistent. For BigTable itself two components can fail:

1. **Master**: If the master is unreachable, the whole database fails and can't be ac-

cessed by new clients. Clients, which still have information about the tablet servers, can try to access them. This means that network partitions are no consistency but an availability problem for BigTable. Because of the desired usage pattern these kinds of failures are acceptable. To prevent major downtimes in case the master node fails, Google holds hot replicas of the master in standby which can take over after a short downtime.

2. **Tablet servers**: If the master detects the failure of a tablet server, it starts a new node and assigns the tablets from the old node to the new one. Because the data and the commit-log are stored in GFS, the new node can simply resume operation.

3 Implementations

3.1 Apache Cassandra

Apache Cassandra[1] [Hewitt, 2010; Lakshman and Malik, 2010] was released as open source software in 2008. It is based upon the Dynamo paper (see chapter 2.1) but also adds features from BigTable (see chapter 2.2). The first implementation was written and used at Facebook to save and search a large amount (150 terabytes) of message which uses sent to each other. Today, Cassandra is used by companies like Twitter[2] or Digg[3] to handle their large amounts of data.
Cassandras focus was always on real time queries which were on the one side small and easy queries but on the other side must be answered very fast because the result of a web service is dependent upon them.

3.1.1 Design Considerations

Cassandra is a monolithic software written in Java, containing every component needed to run a distributed database. It is constructed around some major ideas which will be explained next.

3.1.1.1 Decentralized

Cassandra is like Dynamo a decentralized system. This means that there is no single master concept like in BigTable or HBase. Every node in Cassandra functions the same way, each can answer the same queries and stores the same type of data. This means on the other hand, that there is no coordinator which controls the operation of the cluster. Instead of this, the nodes must do this on their own. This decentralization leads to two advantages: As each node is equal, its simpler to use and it avoids downtimes.

3.1.1.2 High Availability

Cassandra is highly available, which means that, even if nodes or the network fail, the data stored on the cluster is still available. It is also possible to replace nodes with no downtime to the system. Cassandra automatically detects failures and moves data to keep it save.

[1]`http://cassandra.apache.org/`
[2]A microblogging website, see `http://www.twitter.com`
[3]A social news website, see `http://www.digg.com`

3.1.1.3 Tunable Consistency

Cassandra is designed to serve as a data store for many different services. As each of these services might have other requirements in terms of performance and consistency, Cassandra makes it possible for the developers to decide how much performance they are willing to give up for a gain in availability. Cassandra can be everything from a strict consistent data store, where every read requests always gets the latest data, to a highly eventual consistent system, where old data is sometimes provided for a huge gain in availability and performance.

3.1.2 Data Schema

In contrast to Dynamo, which does not structure the data which is saved in the system, Cassandra uses the same data model as BigTable does: Each row of data is structured into columns. This schema is called column-oriented. Cassandra supports secondary indexes for faster queries when not selecting by the primary key. Multiple column families (what would be called "tables" in a RDMS) can be grouped together in one keyspace ("database" or "schema" in a RDMS). It is possible to iterate over the rows using range queries, although ordered range queries tend to get very inefficient.

3.1.3 Typical Use Cases

Cassandra is written for large deployments. It is not meant to outrange traditional RDMS when it comes to small, one node setups. Cassandra is also optimized for a huge write throughput, so it's ideal for storing user activities in a web application. Cassandra also supports geographic replication out of box: Data can be stored in multiple data centers to ensure availability even if a whole data center fails.

3.1.4 System Design

Cassandra clusters are structured in the same way than a Dynamo system: Each node is responsible for a part of a ring which represents the whole key range. A so-called *Toaken* assigned to each node before the first start-up. The node holds all data with keys in the range of its own token until the token of the next node on the ring. If a new node is added to a cluster, the node with the highest load gives away half of its range to the newly created one.
The placement of replicas can be configured per cluster. Several strategies are shipped together with Cassandra. Because the names were changed recently, the old names are provided as well.

- `SimpleStrategy` (former `RackUnawareStrategy`): Replicas are simply placed on the next nodes on the ring. This is what the Dynamo paper proposed as strategy.
- `OldNetworkTopologyStrategy` (former `RackAwareStrategy`): The first replica is placed in another data center, the second replica is stored in the same data center as the original version but in another rack. Eventually remaining replicas are

distributed on the next positions on the ring. This strategy guarantees, that the data can survive a rack outage and a data center outage.

- `NetworkTopologyStrategy`: This new strategy allows full configuration for the administrator where which replicas should be stored.

Just like with Dynamo, each node can handle all read and write operations and eventually forwards the queries to nodes which can answer the requests. The failure handling mechanisms which were proposed by Amazon (see section 2.1.5) like hinted handoff, read repair and anti-entropy protocol were implemented similarly in Cassandra. Cassandra does not support semantic reconciliation: If two conflicting versions of a object exist, the older one is discarded. This is why no vector clocks are needed (and thus implemented) and versioning is done by simple timestamps. Although timestamps sound like a simple solution, it is difficult to keep the hardware clocks of the nodes in sync over the whole cluster. Protocols like NTP can help to solve this problem.
Clients can access Cassandra using a Thrift-API [Hewitt, 2010, p. 156] which exists for various programming languages.

3.1.5 Scaling Mechanisms and Consistency

There are three main factors which have major influence on performance, availability and consistency:

3.1.5.1 Node Count

Adding more nodes to the ring helps to lower the load of heavily used nodes. This speeds up read and write accesses because a smaller region of the keyspace must be handled by each node. If a new node is added to the ring, it typically "steals" half of the key range which is used by the most used node. This is somehow problematic, as adding a new node reduces only the load on one specific machine and does not reduce the average load on all machines. By reducing the amount of data each node must store, the chance that the whole dataset fits into memory is increased. Accesses which can be answered directly from the main memory are multiple orders of magnitude faster than those which must read from hard disk. The more nodes you add, the higher you can set the replication count which increases availability. Cassandra supports adding nodes while keeping the cluster operational.
Nodes can be remove from the ring after they have been "decommissioned". This means that all their data is moved to another node in the ring. Also, dead nodes can be removed permanently. This is needed because normally Cassandra waits for a dead node to come online again after a failure.

3.1.5.2 Replication Factor

The replication factor (or replication count) N determines how many copies of a single data object are stored on the ring. The more copies you store, the faster read accesses

will get: If asking a node for a row it gets more likely that the node can answer the request directly without forwarding. On the other hand, write accesses might get slower: More copies must be updated, so more time is consumed and more load is put on the nodes.
The replication count can be configured per keyspace. Increasing the replication count is not supported without interference but decreasing can be done without influence on the service. A typical replication factor is three, where one copy can be stored in another data center for increased availability.

3.1.5.3 Consistency Levels

A application developer can specify ”consistency levels” for every request which is issued to the cluster. So in contrast to the replication factor, which is a fixed setting, the consistency levels can be set dynamically for every use case and request. These levels control how many replicas are queried when reading and how many copies are updated synchronously before returning on a write request. Choosing these levels has influence on consistency and performance.
For read requests these consistency-levels are:

- `ONE`: Immediately returns the data from the first node that responded to the query. This achieves the highest performance but the copy could be outdated which leads to inconsistencies.
- `QUORUM`: Issues a query to all nodes which hold a copy of the row. The system waits until at least half of the nodes has answered, so a majority of the nodes has responded. Out of the answers the most recent value is returned to the client. This setting provides a good tradeoff between performance and consistency. Inconsistencies can still happen, but are very unlikely. The system is still able to handle node outages.
- `ALL`: Query all nodes which hold a copy of the row. Return the row with the most recent timestamp. Guarantees strong consistency but if a node goes down, the request can’t be answered.

Similar levels can be configured for write requests although their meanings are different:

- `ZERO`: Immediately return to the client after the requests has been received. All write operation will happen in background. If the node fails before it has made the modifications the write is completely lost.
- `ANY`: Make sure that the value has been written to at least one node, hints (see section 2.1.5.3) count as a write.
- `ONE`: Returns if at least one of the designated nodes has written the request.
- `QUORUM`: The write must be done by a majority of the replicas.

- `ALL`: Ensure that all replicas have written the update.

Any configuration where $R+W > N$ is considered strong consistent. In this equation, R is the amount of replicas which are queried for a read operation, W is the same for writes and N is the replication count. For example, the following setup are strongs consistent:

- Read consistency `ONE`, write consistency `ALL`
- Read consistency `ALL`, write consistency `ONE`
- Read consistency `QUORUM`, write consistency `QUORUM`

3.2 Riak

Riak[4] is another implementation which follows the Dynamo paper. It was written by Akamai[5] engineers and is released both as a feature reduce open source version and a full featured commercial version. The company "Bashoo" sells the commercial version together with support and maintains and writes nearly all of the source code. Riak is written mostly in Erlang (a functional programming language) while some parts are written in JavaScript. Clients can communicate with Riak using a REST-API or via Google's Protocol Buffers, where the later is about two times faster but is harder to debug. One of the users of Riak is the Mozilla Foundation which stores crash reports generated by their popular Firefox Browser to a Riak database.
Riak is, like Cassandra (see section 3.1), based upon the ideas published in the Dynamo paper. Because of the same origins, only the differences between Riak and Cassandra are outlined here.

3.2.1 Data Model

- In regards to the data model Riak stays closer to the Dynamo proposal: Every row stored in the database is completely schema-less and interpreted as byte-stream. It is up to the client to separate columns or fields in the rows to structure the data.
- The so-called back end can be configured for each node: In this back end the nodes store their rows on hard disk. Multiple back ends are available and can be used depending on the type of information which should be stored on each node.
- There is a MapReduce [see White, 2011, p. 15] interface directly implemented in Riak: The user can ask the data store to run a MapReduce task written in JavaScript without any external tools or libraries.
- Riak allows to add so-called "links" between rows. These links can be followed automatically when reading a row or doing map-reduce jobs and can be used to model relations between rows.

[4]`http://www.basho.com/products_riak_overview.php`
[5]A content distribution network, see `http://www.akamai.com`

3.2.2 System Design

Before the first launch of a cluster, the `ring-creation-size` has to be set by the administrator. This number denominates in how many parts the ring should be split. After this splitting, every node gets some parts of the ring keyspace. If new nodes join, each of the old machines give away some of their parts. This makes it possible to reduce the load of all machines when adding a new node.
The replication count can be configured per bucket. A bucket is the same as a keyspace in Cassandra.
Riak chooses to implement Vector-Clocks, like the Dynamo paper intended. This makes clock synchronization like it is needed for Cassandra obsolete and simplifies the cluster setup.
Riak does not support configurable placement of replicas like Cassandra does. So in Riak the replicas simply get placed on the next nodes in the ring. There is no possibility to enforce the storage of copies in different data centers.
Consistency levels for operations do not differ much from those implemented in Cassandra. For a read request, the developer can supply the so-called read-quorum out of the following values:

- `one`, `quorum` and `all` with the same semantics as those in Cassandra
- Any integer r, which means that at least r nodes are queried and the most recent result of those nodes is returned to the client.

For a write request two different values can be specified: The write-quorum, which means, how many nodes must acknowledge the successful reception of the write request and the durable-write-quorum, which defines how many nodes must have saved the write request to their durable storage (so the hard disk) before returning. Both settings take values from the same set as the read request from above. A default value for each of these quorums can be configured for each bucket. Any operation can override the defaults be explicitly specifying them.

3.3 Project Voldemort

In 2009, LinkedIn[6] released their implementation of the ideas from the Dynamo paper called *Project Voldemort*[7]. They wrote the project in Java and stayed very close to the paper which leads to a big similarity with Riak. Voldemort also features different storage back ends which includes a MySQL data store.
The project has staled soon after its release. As only developers from LinkedIn are involved in the project, progress is very slow and no new releases have been made since June 2010. Due to the similarity with Cassandra, no further analysis has been made concerning Project Voldemort.

[6] A social network, see `http://www.linkedin.com/`
[7] `http://project-voldemort.com/`

3.4 HBase

The *Apache HBase*[8] (White [2011, pp. 411ff], Khetrapal and Ganesh [2008]) project is closely modelled after the Google BigTable paper (see section 2.2). The first release was made in 2007 when it was shipped bundled together with the Apache Hadoop project. In 2010 it became a Apache Top Level Project which is independently released from Hadoop but still dependent upon it. Today, major users like Facebook[9] contribute their efforts to make HBase better back to the project.

3.4.1 Subcomponents

While Apache Cassandra (see section 3.1) is a monolithic system, HBase consists of many modules and services which form the distributed database HBase together. Most of the concepts and the services used in BigTable can be found again in HBase, for example Chubby, GFS or the data model. Although all HBase modules are written in Java, it is only tested on Linux-Systems and not on Windows. In a first step the fragments which make up HBase are introduced and later their interaction is explained.

3.4.1.1 ZooKeeper

Apache ZooKeeper[10] (Hunt et al. [2010], White [2011, pp. 441ff]) is a service for storing information and accessing it from another distributed system. ZooKeeper stores information in a file-like structure where the files (so-called *znodes*) are limited to a size of 1 megabyte but the usual amount of data is much smaller. Typically ZooKeeper is used to store information which must be accessible by many nodes. For this ZooKeeper can be regarded as a distributed database (see figure 3.1) it-self although it is normally used to implement a database on top on it and only use it for information exchange between the nodes of the database. Example use-cases for ZooKeeper are:

- A configuration service, where the configuration for a distributed application is stored in a ZooKeeper ensemble for easier maintenance.
- A distributed file system, where ZooKeeper stores, which parts of the data are distribute on which file system-node.
- A lock service: Using numbered files as locks, distributed applications can assure that only one thread enters a critical section. This can be used for Master elections in distributed databases.

ZooKeeper stores the information in a hierarchical namespace where each znode can store information and can have further znodes as children. The znodes are addressed like in a file system using slashes as delimiters. You can list all children of a znode and

[8]`http://hbase.apache.org/`

[9]A social network, see `http://www.facebook.com`

[10]`http://zookeeper.apache.org/`

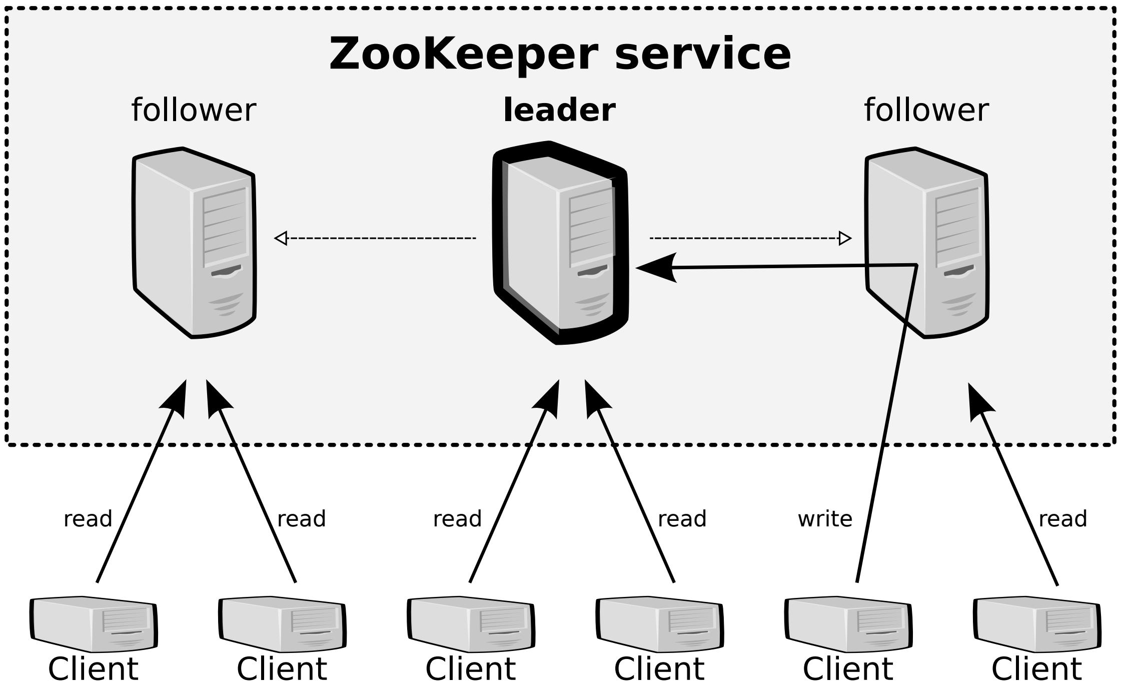

Figure 3.1: A typical ZooKeeper ensemble consisting of three nodes. The center node was elected as *leader*, the other two nodes are *followers*. While read requests by the clients can be handled at each node locally, write requests first get forwarded to the leader which then assures that every follower gets the updates.

the service can notify clients if children are created or deleted.
The ZooKeeper service is replicated of a set of machines, a so-called *ZooKeeper ensemble* or *ZooKeeper quorum*. Each of these nodes holds‘ the whole dataset in their memory to get a high throughput. This leads to the fact that the amount of data which can be stored in a ensemble is limited by the memory of a individual node. This is not a issue because as mentioned above, typically only very small files are stored. The nodes in a ensemble know each other and have a communication channel established. Due to the synchronization scheme involved, ZooKeeper guarantees that it can handle any node failure if a majority of the nodes in the ensemble are up. This means that if there are N nodes in the cluster it is still in working order if only $|\frac{N}{2}| + 1|$ nodes are available. The nodes elect a so-called *Leader*, all the other nodes become *Followers*. If the Leader fails, a new one is elected out of the remaining nodes.
A client can connect to each of the nodes and issue read or write requests. Write requests are captured by the contacted node and forwarded to the leader. The leader now forwards the write request to all the leaders and makes sure a majority of all nodes has written the file before acknowledging the write request. A read request is answered from the local memory of each node and thus is very fast. The fact that only half of the nodes might have received an update leads to the problem, that some read requests may

provide old results. If a client wants up-to-date information, it can issue a `sync` query before its read request which makes the node to "catch up" to the most recent state.
ZooKeeper has some similarities with the concepts of Google's Chubby (see section 2.2.2.2): The hierarchical structure and the notification system are some examples. But there are features which are not implemented by ZooKeeper but in Chubby: Most important the out of the box locking mechanism provided by Chubby, where a lock can be established on each file, is not implemented in ZooKeeper although this behaviour can be mimicked using sequentially numbered znodes.

3.4.1.2 Hadoop Distributed File System

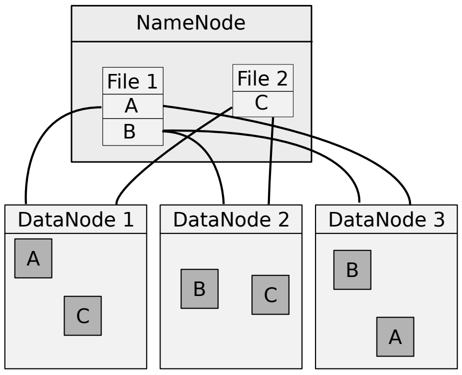

Figure 3.2: Architecture of the Hadoop Distributed File System: A single NameNode and three DataNodes. Files are splited into blocks (file 1 into block A and B, file 2 into block C). These blocks are distributed over the DataNodes: For example, blocks A and C are stored on DataNode 1. The NameNode keeps information where the blocks are stored.

Like BigTable, HBase also relies on the techniques which are implemented in a distributed file system. HBase uses its own file system called *Hadoop Distributed File system* or short *HDFS*[11] [White, 2011, pp. 41ff]. It is a subproject of Apache Hadoop and is based on the ideas of the Google File System paper (see section 2.2.2.1).
As a distributed system HDFS (see figure 3.2) consists of a single Master, the *NameNode*, and many *DataNode*s (these are called *Chunkservers* in GFS). Each file which should be stored on the file system is split into many equal-sized blocks. Typically these blocks are 64 megabytes large and are stored on the DataNodes. The NameNode holds the metadata which is:

1. the metadata for each file: Its name, size, ownership, permissions and timestamps
2. a association between the blocks and files: Which blocks belong to which files
3. block storage information: Which DataNodes store a specific block

This data is regularly dumped to hard disk of the NameNode. To be save from node outages, the blocks are replicated over many nodes. The replication count can be configured per file. Because of the large block size, HDFS is not suitable for many small files. HDFS can not be used without the Master, if it fails, the whole system gets unavailable. Using a hot standby NameNode, the downtime can be reduced. By assigning additional

[11] `http://hadoop.apache.org/hdfs/`

DataNodes to a HDFS cluster, more space is available and more replicas can be stored per file.
In contrast to ordinary file systems, HDFS has some limitations concerning file access:

- You can only append to a file, no random access (like the `seek` operation) is allowed.
- No concurrent access is supported: If one client has opened a file, no other client may access it until it is closed.
- It is not guaranteed that a immediate read after a write leads to the expected results. The client can manually wait for a sync if it needs up-to-date data.

Because of this limitations, access to HDFS is not offered via a POSIX-Layer but instead using a library which is linked to the program. Command-line tools which are shipped together with HDFS can be used to transfer files from the local file system to HDFS and back.
If a client wants to read a file from the file system, it first queries the NameNode for the nearest locations of all the blocks of the file. The client then contacts the appropriate DataNodes and reads the data from them. Again, like in BigTable and GFS, no data flows through the Master which keeps the load low. If a new file should be written, first the metadata is transmitted to the NameNode. Then, the file is split into blocks and for each block the NameNode is queried for a list of Nodes where the blocks should be stored. The client then transfers the block to the first DataNode which forwards the block to the next node. This is repeated until all replicas of the block have been written.

3.4.2 HBase Architecture

HBase is built on top of HDFS and supports real-time read and write random access to very large datasets. Data is grouped into tables (like in traditional RDMS). Each row in these tables is uniquely identified by a row-key. The row-data it self is stored in a column-oriented way (like in BigTable). Nevertheless the schema of HBase is out of scope of this thesis. Clients can access HBase from various programming languages using a API generated by Thrift [Hewitt, 2010, p. 156].
Tables are partitioned into so-called *regions*. Each region consists of all rows with their row key in specific bounds. Initially all data is stored in a single region. If the region grows to large, it is split into two regions and so forth. Regions are the unit of distribution, this makes it possible to store tables much larger than what a single machine can handle.
HBase consists of a single master (the *HBase Master*) and a cluster of slaves (called *Region Servers*). Each region is stored on exactly one Region Server. The Master takes the administrative work, it assigns regions to the Region Servers and recovers from the failure of them. The Master also splits regions if they grow to large. HBase uses a ZooKeeper ensemble for Master failover. This is important because using only one Master would impose a single point of failure. This is avoided by keeping several Masters in standby mode. Using a voting protocol on a ZooKeeper cluster, a new Master is elected

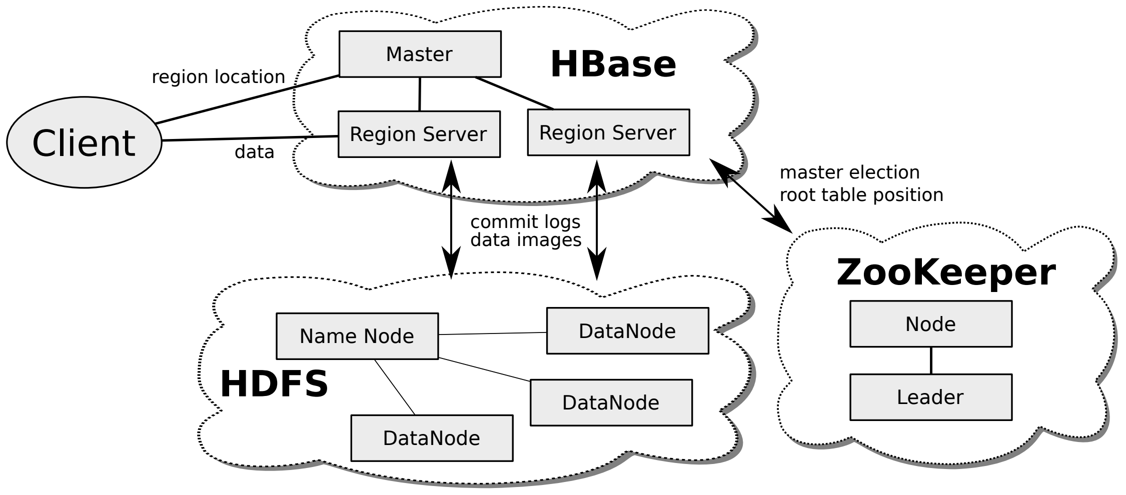

Figure 3.3: HBase components: HBase consists of a single Master and multiple Region Servers. Each of these Region Servers saves its commit logs and data images to a HDFS cluster. ZooKeeper is used for administrative operations. Multiple components can run on the same physical machine. For example, each node, which runs a Region Server could run a HDFS DataNode as well.

if the designated one fails.
The information which region is stored on which server is also stored in a HBase table. This leads into problems when trying to find out where this metatable is stored. This is a second point where the ZooKeeper service is used.
Typical operations by clients first connect to the Master, querying for the Region Server which stores the rows that should be read or modified. Then the Region Server is contacted by the client. The location-information is cached by the clients to keep the load of the Master low.
Each Region Server only handles access to the regions it stores. If a new write request is issued, the Region Server first writes a new log entry synchronously to the underlying HDFS, modifies its internal data structures and then acknowledges the operation. If the whole region does not fit into main memory, parts of the data can be stored on hard disk. Periodically, the commit-log is aggregated to a new data image in background on each Region Server.

3.4.2.1 Handling Failures

Two types of errors can occur on a HBase system (compare to the failure handling in BigTable in section 2.2.4)

- **Failure of the Master**: Like mentioned above, a new Master can be elected. This feature is quite new, old versions on HBase were not able to handle Master failures. Thus the whole system became unavailable until a new Master was bootstrapped by hand. Now by using the automatic failover nearly no downtime is involved.

- **Failure of Region Servers**: If a Region Server fails, the Master will assign a new node for the region. This new machine will read the data image and the commit logs from HDFS and can then take over.

3.4.3 Scaling Mechanisms

In HBase there is no such thing like a replication count. Each row of a table is stored on exactly one Region Server. The only scaling mechanisms is to add more nodes to the cluster: This makes the regions smaller and so takes load from the servers or provides additional space. It is advantageous if each Region Server can store its entire region in the main memory. Adding further nodes makes this possible and so speeds up both read and write operations on the cluster.
The performance of the HDFS has immediate influence on the performance and availability of HBase. For each HBase cluster the replication count of the commit log and the data images can be set. Increasing this value also increases the time a write requests needs because the commit-log has to be saved on more nodes. On the other hand, a larger replication-count for the commit-logs makes the log (and so the data) more robust against node failures.
It is possible to install the HDFS DataNode and the HBase Region Server on the same machine. HBase can then be instructed to save its logs on the local DataServer which increases the throughput because of the fast communication. This leads into problems if the machine fails: Then both, the Region Server and the DataNode, are unavailable on the same time.

3.4.4 Consistency Guarantees

Using its single Master setup, HBase can guarantee strong consistency: Every read and write operation is immediately visible to all clients, no outdated data is returned. The tradeoff is, that both, the Region Servers and the Master, are single points of failures: If one of them fails, some or all of the data is not available for a short period of time until a replacement node has been established.

3.5 Redis

The previously described systems are designed to run on large clusters from the beginning on and it makes no sense to deploy them on a single node. In contrast to this, the following systems can be used on a single system and later on additional nodes can be added to improve performance or availability.
One of those systems is *Redis*[12]. It was released as a small open source project in 2008 written in C, but has gained a huge momentum since the involvement of VMWare[13] who sponsor two full time developers. Additionally a large community of developers exists so Riak can be regarded a living project.

[12]`http://redis.io/`
[13]Developer of virtualization solutions, see `http://www.vmware.com`

The central idea of Redis is to provide a key value store: Using a simple interface, data of various types can be assigned to a key. Later on, the data can be requested again using the key. It tries to keep all the data in the main memory of the system for performance reasons. If there is not enough space, some parts can be swapped to disk using a virtual memory system.

3.5.1 Scaling Techniques

As mentioned previously, Redis is designed as a single node system with some extensions to make the system scalable.

3.5.1.1 Replication

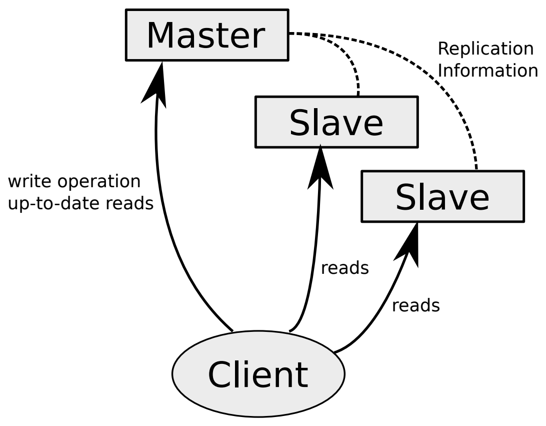

Figure 3.4: A possible replication setup: The data from the master gets replicated onto two slaves. Read requests can be issued on the slaves whereas write requests must always go to the master due to the read only property of the slaves.

Redis supports unidirectional replication: If you have a running Redis-node (the master), a second node (the slave) can replicate all the data from the master. This is done in a asynchronous way which makes it possible that the slave is outdated for some time. Although write requests to the slave are possible, they make no sense as they are silently overwritten by the replication. In further versions writing to the slave will be prohibited. This makes the slave a read-only node. Multiple slaves can be set up and even a multi-level replication can be set up if replicated nodes get replicated again. Replication always copies all data, so there is no selective replication on a subset of the data.
Because of the read only property of the replication slaves, this can only be used to scale read requests: They can be distributed over all replicas whereas write requests always have to go to the master. Also, if a client needs up to date data, it can query the master for the data.

3.5.1.2 Sharding

To reduce load from a Redis setup or to store more data than a single node can handle, sharding[14] can be used. Sharding is release as a separate project, independent from

[14]https://github.com/jzawodn/redis-sharding

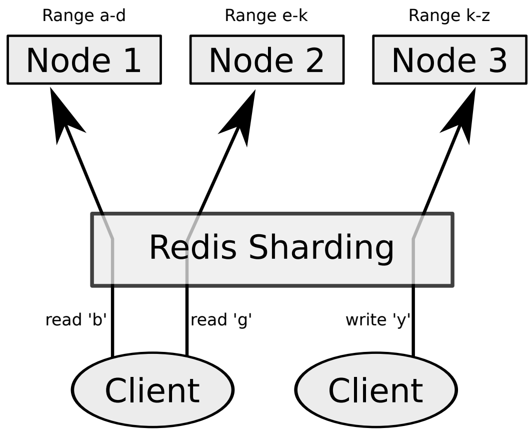

Figure 3.5: Sharding in Redis: The data is distributed on three nodes. During setup, the key ranges have been defined and assigned to the nodes. A Redis sharding server serves as a front end for the clients: They only communicate with the sharding server and transparently get the data from the appropriate nodes.

Redis. To setup a sharded Redis cluster, first the keyspace is divided into to same number of parts as there are nodes on the cluster. This can be done for example by using a hash function on the keys and mapping the results of the hash function to the servers. This distribution is fixed and can not be changed after the cluster has been started. This means that no new nodes can be added after the initial setup. Each node runs a normal Redis setup. A additional Redis sharding server distributes the requests to the matching nodes. The clients only see one big cluster whereas the nodes don't know that other nodes exist. The only component which manages the cluster is the sharding server. This simple setup allows it to increase the performance of Redis.
To guarantee availability, additional nodes can be deployed and set up as replicas of the primary nodes. So if the keyspace is divided into three parts and so three primary nodes exist additional three nodes can be used to act as replicas.

3.5.1.3 Redis Cluster

Because of the amount of nodes in a cluster is fixed when using sharding, a new front end for Redis is currently written: *Redis Cluster* [Sanfilippo, 2010]. The cluster should consist of various nodes, some of them store the actual data, others keep information about where which key ranges are stored. Sharding can be regarded as a temporal substitution until Redis Cluster is ready. Because of the experimental nature, the nonexistence of a stable source and documentation no further work will be done here on Redis Cluster.

3.6 CouchDB

Apache CouchDB[15] [Anderson et al., 2010] was developed by a IBM employee in 2005, so it was written before the NoSQL movement got momentum. In 2008 it became a Apache project and is since then maintained by a large community of developers which

[15] `http://couchdb.apache.org/`

continuing involvement of IBM. It is written in Erlang because this functional programming language was found ideal for concurrent distributed systems. It runs on Linux as well as several mobile devices like iPhones and Android based phones.

3.6.1 Data Model

CouchDB features a document-oriented data model: It stores documents which can be addressed by a unique key. The document itself is stored in JSON [Anderson et al., 2010, pp. 231ff] which features a hierarchical structure within each document and enables the developer to store objects in a data store while preserving their structure. For document retrieval, only the document identifier and ranges of these identifies can be used, so the documents can't be searched by their contents directly. Nevertheless, queries can be issued by using the built in map reduce function which enables aggregation and selection using JavaScript. Furthermore, JavaScript can be used to validate the data which should be stored in the documents. Each time a document is modified, a piece of code is run to check whether the changes meet the requirements specified.

3.6.2 System Design

A CouchDB instance can be accessed using a simple REST protocol over HTTP. This makes debugging easy due to the fact that HTTP is a well understood and widely accepted protocol.
CouchDB is designed to store versioned copies of the documents. This means that every update request on a document does not overwrite the old data but instead creates a new instance. These versions are identified by a revision identifier (`_rev`). If a read request is issued without explicitly specifying a revision, the latest fully written version is returned. The developer can also retrieve specific revisions of a document. This schema is called "Multi-Version Concurrency Control" or "MVCC".
Using MVCC, CouchDB can implement read and write accesses without any locking: Read requests don't have to wait (or lock) for write requests to finish but instead return the latest completely written version. This makes read and write requests very fast, even under high load because all requests can be issued and processed in parallel.

3.6.3 Replication

To distribute data over several machines CouchDB supports replication. In contrast to Redis (see section 3.5.1.1), which only supports simple unidirectional replication, CouchDB contains advanced replication features:

3.6.3.1 Bidirectionality

Multiple CouchDB servers can synchronize their data using bidirectional replication. This means that write requests can still be issued on all nodes and changes are replicated onto all remaining nodes. This can even be used if there is no stable connection between

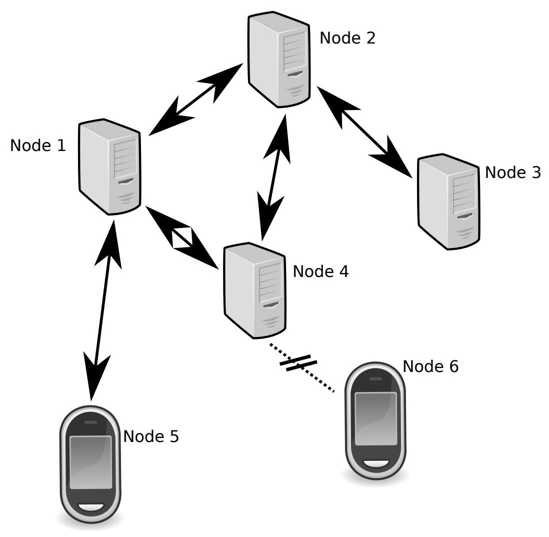

Figure 3.6: A CouchDB cluster consisting of six devices: Four servers replicate their data. A mobile device (node 5) synchronizes with node 1. A second mobile device (node 6) also running CouchDB has currently no connection to the cluster. Nevertheless each device can query their local data stores and also update the documents stored there. If the connection of node 6 reestablishes, all changes are sent from node 4 to node 6 and vice versa.

the nodes: If a connection can be established, all documents which have changed are synchronized between both nodes. This makes CouchDB an ideal choice on mobile devices where data can be kept in a local CouchDB instance and synchronized with a node on the Internet if connectivity is present.
Every time replication should take place, the two databases are compared to find out which documents have been added, which have changed and which have been deleted. Then these changes are transmitted and applied on both sides so that afterwards each node has the same data. Replication can be either continuously or done once. Replication can be activated and configured separately for each database. It is even possible to write a so-called replication filter function in JavaScript which specifies which documents should be synchronized and which should only stay local.
Using this replication scheme data is always local and can be accessed in a fast and efficient way. The replication is completely asynchronous, so there is no guarantee that a update has been sent to all nodes. Of course this guarantee would be impossible due to the existence of currently disconnected peers.
Replication in CouchDB should be regarded as a Master-Master replication because the nodes are all equal and exchangeable. This is advantage over a master-slave setup like HBase.

3.6.3.2 Consistency

Because of the asynchronous nature of replication in CouchDB, conflicts can occur and the data store is only eventually consistent. This can happen if a document is currently synchronized between two nodes and the connection fails. If both nodes now change their local documents and the connection comes back again, two conflicting versions exist. CouchDB solves this by storing all versions which exist. This fits into the MVCC schema and is integrated seamlessly. A request for a document returns the most recent version stored locally. The developer can query CouchDB for all versions and then merge them using application logic and store them back into the storage. This is called semantic reconciliation. CouchDB informs the developer that there are conflicting versions of a document upon read queries by returning the additional `_conflicts:true` flag.

3.6.4 Scaling Mechanisms

There are two possibilities to use the previously described properties of CouchDB to build a scalable cluster.

3.6.4.1 Load Balancing

By adding nodes and setting up a continuous replication between them, the load of read and write requests is distributed between those nodes. Still, the data has to fit into each nodes memory because each nodes holds all data. The system must tolerate that some of the data might be outdated for a short time and also must pay attention to conflicting versions.
In contrast to the unidirectional replication in Redis, where write requests have to go to a single master, in a clustered CouchDB setup each node can handle write requests. This improves CouchDB's scalability.

3.6.4.2 Sharding

If more data should be stored than a single node could save or the traffic generated by replication is to much for a machine, the data must be split over several nodes. As with Redis, CouchDB does not support this out of the box. There exists a own application called *CouchDB Lounge*[16] which acts as a proxy in front of a cluster of CouchDB nodes. Lounge was developed for Meebo[17] because they wanted to handle a huge amount of data using CouchDB.
Lounge features two configuration steps: The first one, also called initial setup, defines, how many so-called shards should be available. The whole key range is split into shards. Now, using the second configuration, the shards are assigned to the actual nodes. This second setup can be changed during run time while the amount of shards and the splitting must stay static. This is why typically more shards are defined than nodes are available and will be ever in the planed future. A shard can be assigned to multiple node for availability reasons. Typically if the initial cluster size is ten nodes, 1000 shards are created. Now for the initial setup, 100 shards are assigned to each node in the cluster. If the load increases, 10 new nodes can be deployed and each of the old nodes gives away half of their shards to a new one so that after the upsizing, each node handles 50 shards.

[16] `http://tilgovi.github.com/couchdb-lounge/`
[17] A web based instant messenger, see `http://www.meebo.com`

3.7 MongoDB

MongoDB[18] [Dirolf and Chodorow, 2010] is a distributed data store written in C++ by some developers, who worked previously for DoubleClick[19]. They created a startup with the goal to create a software stack that was very scalable. One component of this stack was the database system MongoDB which was later released as open source. Today prominent users of MongoDB include GitHub[20] and SourceForce[21]. MongoDB compares to CouchDB but adds additional features for scalable clusters, like dynamic sharding, which CouchDB does not support out of the box.

3.7.1 Data Design

MongoDB stores data as documents. Instead of addressing these documents using a unique key like CouchDB does, MongoDB allows the developer to query for documents by every field. This is made possible using indexes on every field in the documents. CouchDB's behaviour can be emulated by simply adding a `id` field to every document. Due to the more flexible approach in MongoDB it is also possible to query for multiple fields and use multiple criteria. The queries for read access are encoded in a JSON document.
In contrast to CouchDB, which features a JSON protocol over HTTP, MongoDB uses a compressed binary protocol. This saves traffic and increases the throughput and so makes data transfers faster.

3.7.2 System Architecture

MongoDB can be run in two modes: Either as stand-alone version, where only the `mongod` daemon is running. This mode is intended to be run on a single node without any distribution and can be used until a database gets to big for a single node. MongoDB tries to provide a good performance even on single node in contrast to systems like Cassandra, which are not optimized for single machine clusters. The other mode is the so-called "sharded mode". This mode uses various services to distribute the MongoDB instance over several nodes. Whereas on CouchDB true distribution is only supported using external tools, MongoDB supports this out of the box using several components which are all included in the application package.

3.7.2.1 Sharding

MongoDB divides the keyspace into parts, the so-called "shards". Each of these shards is small enough to be handled by a single server, running the `mongod` process. If a shard gets to big or gets to much traffic, the cluster can decided to give parts of the keyspace to another shard and so balance the traffic. This resharding happens in background

[18]`http://www.mongodb.org`
[19]An ad serving service, today owned by Google, see `http://en.wikipedia.org/wiki/DoubleClick`
[20]A source code hosting `http://www.github.com`
[21]A open source project hosting, see `http://www.sf.net`

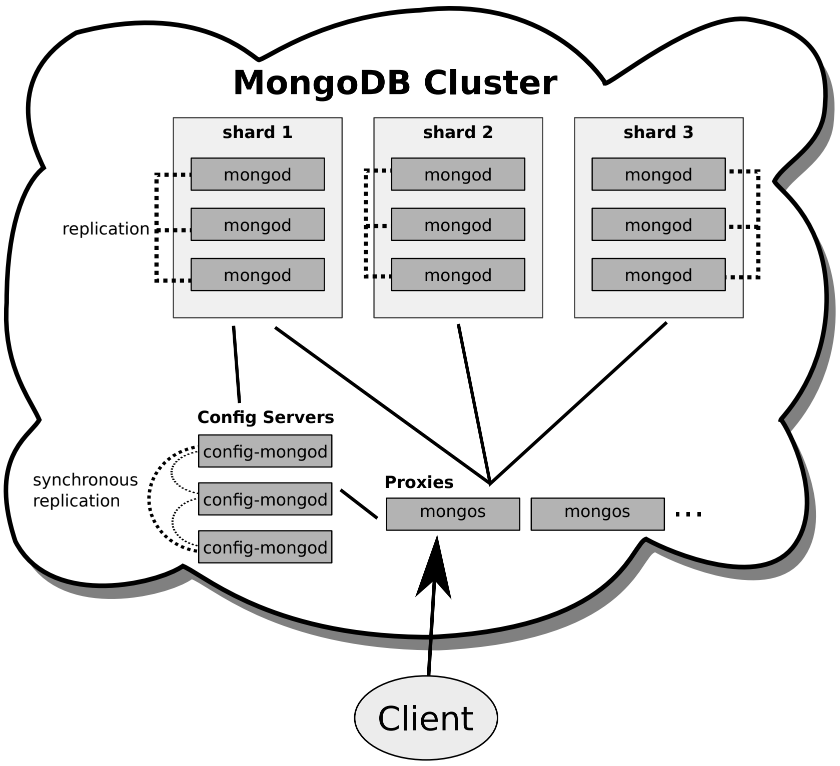

Figure 3.7: The keyspace is divided into three shards: Each shard is handled by three servers, where each server holds the whole shard. The config servers store the location and size of each shard. Proxies forward requests to the right nodes in each shard and return the answers to the clients. Clients only communicate with the proxies using the same protocol as a stand alone MongoDB. *According to Chodorow [2011].*

and is handled by the shard-nodes it self. For availability-reasons, a shard is not only stored on one node but instead replicated onto multiple nodes. Still each node stores the complete shard. For data exchange between the copies, a replication algorithm is used. Depending on the replication setting involved (see 3.7.3), MongoDB can only guarantee eventual consistency.

3.7.2.2 Config Servers

Config Servers, which run the `config-mongod` process, are used to store the key ranges which are stored in each shard. They also hold the information where each shard is stored. Because of the importance of correct information, there is not only a sole config server but this information is stored on multiple nodes. They synchronize using a special two phase commit protocol which ensures, that the data is always in sync between the config servers. If a config server fails, the configuration switches over to read only mode until all configuration nodes are available again. This only affects the configuration, the cluster itself can still operate as normal except that no resharding can happen until the config servers are up again.

3.7.2.3 Proxies

Multiple proxies can be installed which provide the interface to the cluster for the client. These `mongos` processes are completely stateless: Upon start-up they retrieve the shard information from the config servers. They forward each request to the appropriate node in a shard. If a node does not respond, the proxy marks the node as dead in the configuration and asks another replica in the shard. The interface which is provided by the proxies is the same as the one provided by a single `mongod` process. Because of this, a stand-alone MongoDB and a cluster can be used in the same way. Multiple proxies can be run, although due to the lightweight nature of the design, there is no need for a huge amount of proxies. If a read query is issued which needs results from more than one shard, the proxy aggregates the results and provides the client with a unified response.

3.7.2.4 Server Layout

The architecture above is made at process level. Many of these processes can be run on the same physical hardware node. For example, the configuration servers and shard servers could be run on the same node. Additionally, the proxies could be installed directly on the shard machines. Using this aggregation the amount of hardware nodes needed can be reduced.

3.7.3 User Access

Whenever the user accesses a MongoDB cluster, he can specify how the cluster should handle write operations. This is similar to the consistency levels provided by Cassandra (see section 3.1.5.3) and is called `WriteConcerns`. Using this setting, the developer can tell for how many successful writes of replicas the proxy should wait, before it returns a successful requests to the client. Possible settings provided by MongoDB are:

- `NORMAL`: This is the default mode: The proxy immediately returns after successfully forwarding the request to all nodes. It does not matter if one or even all servers fail to write the update to their data store.
- `SAFE`: Using this mode, the proxy waits for at least one node to successfully retrieve and store the update in its main memory.
- `FSYNC_SAFE`: Like `SAFE` but also waits for a successful store on durable memory or the hard disk.
- `REPLICAS_SAFE`: Waits for at least two nodes to write the request.
- `NONE`: Does not wait at all, the proxy returns before forwarding the requests.

3.7.4 Scaling Mechanisms

There are multiple possibilities to scale a MongoDB cluster, most of them involve adding servers:

- **Adding sharding servers**: By adding new sharding servers, two things can be accomplished: If the shard count is not changed, the replication count can be increased. This allows more nodes to fail before a shard gets unavailable. Alternatively the shard count can be increased. This reduces load from each shard and so each other sharding server and increases throughput.
- **Adding config servers**: Adding config servers does not increase performance. In fact, due to the protocol involved, the performance decreases if more configuration servers are added. As the configuration data is very important for the operation of a MongoDB cluster, it still makes sense to add further config server to increase availability.
- **Adding proxies**: Depending on the workload of the cluster, the proxies might get a bottleneck. This can especially happen if many range queries are issued which involve a lot of aggregation work by the proxies. This bottleneck can be avoided if more proxies are added to the cluster.
- **Tuning consistency**: Using the WriteConcerns, the developer can choose between performance and consistency.

3.8 Membase

The *Membase*[22] [Couchbase, 2010] project was started as a joined venture between NorthScale and Zynga. Zynga is a developer of browser games, especially for social networks. In this context they needed a fast and reliable data store which was able to handle thousands of requests. As these games are typically very write heavy, focus was made to optimize the data store for write throughput. NorthScale on the other hand already wrote a cache software named "memcached". This software was in wide use and the simple protocol was implemented in many programming languages and is well understood. The project was started in 2009 and a memcached compatible system was created, that added many features to the simple cache which made a true data store out of memcached. It is written in a mix of C, C++ and Erlang with some parts written in Python.

3.8.1 Data Model

Due to its memcached origins, Membase supports a simple key-value data model: Data can be saved to the system by using a unique key and any data, binary or string, can be assigned to that key. When writing to a key a second time, the value is updated. It is up to the developer or a additional library to structure the data which should be

[22] `http://www.membase.org`

saved in the database. Membase supports multiple pluggable storage engines which are optimized for different workloads.

3.8.2 System Architecture

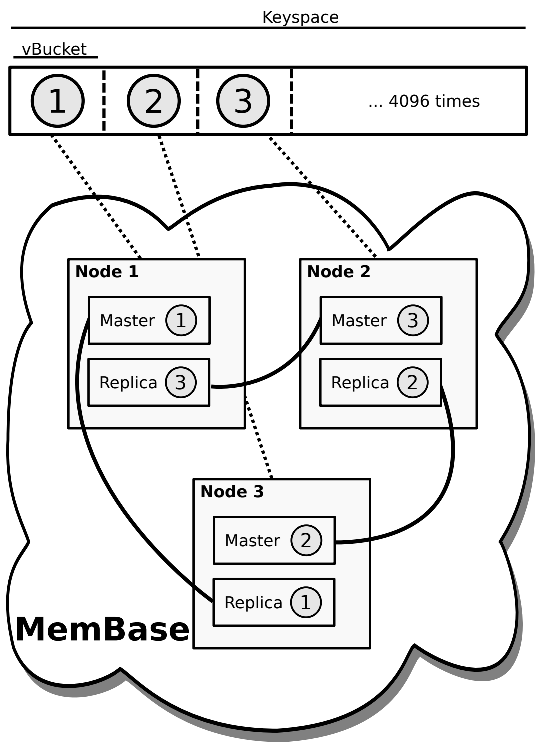

Figure 3.8: Distribution in Membase: The keyspace is partitioned into 4096 vBuckets. Each of these buckets has a master server and may have multiple replicas. The master of each vBucket forwards updates to all replicas in a synchronous way (filled lines). A vBucket map stores which node stores which vBuckets (dotted lines). This map is saved on all nodes and must be retrieved by the client to know which node must be contacted.

The keyspace is divided into 4096 so-called vBuckets. This means that each key is assigned to exactly one of these vBuckets. The assignment is static and done using a consistent hashing approach [Karger et al., 1997], so each client can calculate which vBucket holds the key it requests.
A Membase cluster consists of many equal nodes. None of them is a designated master although a master slave protocol is involved which is described later on. Each vBucket always has one master server. This node is responsible for accepting requests. Additional copies will be stored on other nodes. The amount of copies which exist of each vBucket can be specified. The assignment, which nodes stores which vBuckets is called the "vBucket Map". Updates to the map are done using a synchronous two phase commit protocol.
A client will try to retrieve a vBucket map prior to any request. This map can be cached in the client application. The client then tries to contact the master server of the vBucket it wants to write to. If the contacted server is not the master server anymore (perhaps due to reorganisation), the client simply pulls a new up to date copy of the vBucket map and then contacts the correct server. The client then read or writes to the master server. If the request is a read request, the answer is provided from the local memory of

the master server. A write request is forwarded to all server which hold a replica of the vBucket modified. This is done in a synchronous way, so that the operation can only be successful if all replicas are modified. Because of this replication method, Membase is always consistent, no out dated copies can exist.
Membase supports rebalancing out of the box: vBuckets can be moved from one node to another without influence on the cluster operation. The rebalancing can even move the master node of a vBucket.
Each Membase node runs two processes, the "Data Manager", which handles the read and write requests issued by the clients and the "Cluster Manager", which manages the replication and detects and handles the failures of other nodes.

3.8.3 Handling Failures

All nodes storing a vBucket monitor each other using a watch dog protocol. If a server detects, that the master of the vBucket has failed, the failover mechanism comes into play. This mechanism first chooses a new master. This new master is one of the servers which hold a replica of the vBucket, so this node can take over the master functionality at once without any resynchronization. The new master is saved in the vBucket map on all other nodes. Additionally it is saved that the old master has failed. If the old master should start again, it has to resynchronize all data before it can return to the cluster and serve as a replica there. If a client would contact the node which previously failed and still contains out dated data the access would be denied and the client would try to find out the new master of the vBucket. While the new master is elected and broadcast using the vBucket map, all access to the vBucket will fail. The remaining buckets still stay accessible.
This architecture was chosen, because the developers think, that while inconsistencies is a problem all the time and should be avoided, node failures are only a temporal problems which can be handled. The value of the approach is dependent upon the requirements in consistency and availability.

3.9 Additional systems

In addition to the systems and implementation explained above, two other interesting system have been published which might get further attention in future. Due to the lack of additional information no detailed analysis can currently be made.

3.9.1 PNUTS

Like many Internet companies, Yahoo[23] needed a data store for their growing amount of data. They wanted a system which could be used as a back end for each of their services so that application developers didn't have to think about database issues and trained people can administrate these servers instead. Out of this need Yahoo created *PNUTS*

[23] A web search engine and web portal

and published a papers about it [Cooper et al., 2008, 2009]. Except for this released publication, no further information is available about this system as it was never released to the public but is only used internaly at Yahoo.
PNUTS chooses the same consistency approach as Dynamo: Data is only eventually consistent, but the developer can query for the most recent version if he is willing to give up performance and availability. For each read call, the client can specify which version it will accept as result: Either any version, even an outdated version, or only the most recent version, even if this should mean that the client might get no result at all.
PNUTS assigns each key to exactly one node. This node is called the master for that row. The master can be migrated to adapt to load changes. Every request is redirected by each node to the master. The master node accepts the request and distributes it to all replica nodes in asynchronous way. A so-called "Tablet Controller" saves which nodes is the master of which row. Using this information, the information can be forwarded to the right node. Yahoo optimized the system for geographic distribution. This means that data access must be fast and reliable although the data is distributed over big distances in different countries.
To implement the replication, a separate service is used. This message broker, called Yahoo Message Broker (YMB), stores replication data and guarantees the updates will be delivered whenever a node is online. It also guarantees that the order is preserved which is important to keep data consistent. Because of the missing additional information, no further analysis on the replication technologies involved can be done.

3.9.2 Google Megastore

In addition to BigTable (see section 2.2) Google has recently published a second data store called *Megastore* [Baker et al., 2011]. It was published because Google thought that there are to many difficulties for the developer to implement a application on top of a limited API and loose consistency models which current NoSQL data stores provide.
It can be regarded as a mix between traditional RDMS and NoSQL: For example, Megastore needs a defined schema for each table. Nevertheless it supports advanced column types like arrays which make make it easier to model relationships between rows. Megastore also guarantees strong consistency using a Paxos like replication protocol. It guarantees ACID within so-called entity groups. These could be a data center for example. Between these entity groups only loose consistency is provided by Megastore. This enables a performant and consistent view on the database for local operations.
Google has used Megastore for some years in various products where the main use is Google AppEngine[24], a platform for hosting web applications. Currently, no further specifications of the replication technologies of Megastore are available.

[24]`https://code.google.com/appengine/`

4 Comparison

4.1 General Overview

	Programming Language	**License**	**First Release**
Apache Cassandra	Java	Apache License 2	July 2008
Basho Riak	Erlang, JavaScript	Apache License 2	2009
Project Voldermort	Java	Apache License 2	Feburary 2009
Apache HBase	Java	Apache License 2	2007
Redis	C	New BSD	March 2009
CouchDB	Erlang, JavaScript	Apache License 2	2005
MongoDB	C++	GNU AGPL v3.0	Summer 2008
Membase	C++, Erlang	Apache License 2	2009

Table 4.1: Overview of the system which were examined

In the following sections, the technologies involved in the systems will be compared and discussed. Prior to this, table 4.1 gives a quick overview of the systems which were explained previously.

	Replication	**Fragmentation**	**Category**
Cassandra	✓	✓	
Riak	✓	✓	Ring (4.2)
Project Voldemort	✓	✓	
HBase	*by underlying DFS*	✓	
MongoDB	✓	✓	Master-Slave (4.3)
Membase	✓	✓	
Redis	✓ *(unidirectional)*	*using external tools*	Replication based (4.4)
CouchDB	✓ *(bidirectional)*	*using external tools*	

Table 4.2: Categorization and features of the systems

The systems discussed above use very different technologies to provide scalability to the users. The implementations can roughly be groups into three categories which share the same ideas and properties in regard to scaling techniques, consistency and system architecture. As the data model of the systems was not considered when making this categorization, the systems do not share the same data model. The question of data models is discussed later in chapter 5. Table 4.2 assigns each system to one of these three categories and also shows which distribution technologies are implemented. Each of these categories will be explained further in the following sections.

4.2 Ring Systems

All of the systems in this category borrow the ring like design from the Dynamo Paper (see section 2.1). The whole system is decentralized, each node can answer every request by eventually forwarding the request to the appropriate node. Because each node can potentially handle a write request, these systems are always available for write requests. If the most recent version of a document needs to be read, it might be possible that this request can't be answered in the presence of failures. So no read availability is guaranteed.

4.2.1 Node Types Involved

Due to the completely decentralized approach, all nodes are the same, so only one type of nodes exists.

4.2.2 Implementations

Cassandra (see section 3.1), Riak (see section 3.2) and Project Voldemort (see section 3.3) all borrow their distribution ideas from the Dynamo paper. Apart from this there are differences in the data model and the actual implementation.

4.2.3 Failure Situation

There is no single point of failure with Dynamo systems. If a node fails, other nodes still accept write requests and as long as there is at least one replica remaining read requests can also still be answered. The client must not handle data unavailability, if a node does not answer within a time frame, the node can simply ask another node for the data given that enough replicas exist and the client is willing to accept outdated answers. If the client needs the most recent versions of a row, it might have to wait for the system to restore. This is why ring systems are always available for writes but not for reads. If the network partitions, conflicting versions of rows can exist. Upon reading the developer has to decide how to handle this.

4.2.4 Consistency Model

The ring systems feature tunable consistency. This means that the developer can choose whether he wants strict consistency or whether eventual consistency is enough. This makes it possible to configure how much consistency should be given up for an increase in performance. This can be chosen for every request what makes ring systems very flexible.

4.2.5 Scaling Methods

To scale a system up a simple addition of nodes increases throughput and capacity. The system rebalances the data to fill the newly added nodes and the overall load is reduced.

4.2.6 Data Access

Each node can be queried for all rows, if the node can't answer a read request on its own it gets forwarded. To save this round trip, the client library can cache the information which node might store a specific row and ask this node.
Ring systems can only support fast range queries if the data is saved ordered on the ring. This ordering leads into problems because the ring gets unbalanced and so some nodes get higher load than others. If the rows are randomly distributed on the ring, then range queries are either slow or impossible.

4.3 Master-Slave Systems

These system have one similarity: Each row is assigned to exactly one slave. This slave is responsible for answering read and write requests for all rows he is responsible for. So there is a 1 : 1 mapping between rows and slaves. A master server stores this mapping so that the clients can query the correct slave. In this systems read and write requests are handled in the same way. This means that they are not specifically optimized for write requests like the ring systems mentioned above.

4.3.1 Node Types Involved

- A single **master** keeps the information which row is stored on which node.
- Multiple **slaves** are responsible for answering requests for specific rows.
- **Replicas** can be created for each slave: These contain the same information as the slaves but operate in a standby mode and can take over if a slave fails.

4.3.2 Implementations

- **HBase** (see section 3.4) supports synchronous replication between replicas using the underlying DFS.
- **MongoDB** (see section 3.7) features synchronous and asynchronous replication. The developer can choose for every request which one to use.
- **Membase** (see section 3.8) uses synchronous replication for all requests.

4.3.3 Failure Situation

- **Master**: If the master fails, the system can still operate because the clients can cache the mapping information. A new client connecting can not retrieve the mapping information for the master and so can't operate. Also rows can't be moved from one client to another until a new master has been started. Some implementations support multiple copies of the master, but as all existing masters synchronize using a synchronous protocol, adding more masters makes write operations to the mapping (not to the rows) slower.
  - **HBase**: Only a single master can operate at the same time, but since version 0.20 multiple standby masters can exist which elect a new master using ZooKeeper in the case of a failure. Until this master election has been done, no client can retrieve the mapping information.
  - **MongoDB**: Multiple config servers can be created to avoid availability problems.
  - **Membase**: Each node holds the mapping information, so there is actually no single point of failure for the master at Membase. The mapping information is always available.
- **Slave**: Replicas can take over after a short downtime. During this time, the mapping in the master is changed and the clients must update this information. The downtime must be handled by the clients by delaying their requests until a new slave has taken over operation.

Network partitions do not have direct effects on master-slave systems: Because each row only is served by one slave, network partitions simply make some rows unreachable for clients and so preventing every access to these rows.

4.3.4 Consistency Model

Master-slave systems feature strict consistency: Because one specific node is responsible for all requests, no inconsistencies can occur. For the replicas, synchronous replication is used, so no inconsistencies can occur here either. The developer has no choice on this consistency model, as a result he must design his application so that it can handle node failures.

4.3.5 Scaling Methods

Adding additional nodes reduces the load on each slave because the slaves have to handle less data. Although the overall throughput increases, it is not possible to raise the throughput for operations on a specific node because there is always only one server responsible for serving this row. Node additions also help if the existing nodes can not handle the data because of its amount.

4.3.6 Data Access

Clients first query the master for the slave which handles the row the client wants to access. In a second step, the client directly communicates with this slave. For faster access the clients can cache the information which slave is responsible.
Depending on the implementation range queries tend to be very fast because each slaves handles a large ordered set of nodes. Within these sets ordered queries are very fast.

4.4 Replication Based Systems

All these systems use replication between their nodes. They also do not support fragmentation, consequently there is no need to store mapping between rows and nodes. All nodes involved in the system store all rows and exchange the updates using a asynchronous replication protocol. Depending on the replication scheme involved only a single node can be written or all nodes accept write requests.

4.4.1 Node Types Involved

The type of nodes involved depends on the replication scheme used: If all nodes can replicate from each other (bidirectional replication) then all nodes can be the same. If only unidirectional replication is supported, a tree like structure must the built with a single master at the root where all nodes at least indirectly replicate from.

4.4.2 Implementations

- **CouchDB** (see section 3.6) supports bidirectional replication which makes it possible that all nodes accept writes.
- **Redis** (see section 3.5) only supports unidirectional replication so that only a single master can be written, all other nodes are read only.

4.4.3 Failure Situation

If bidirectional replication is supported, all nodes can be queried for each row. This means that a node outage does not affect the availability of the system, as long as the clients can ask another node for the information. If a master exists due to the unidirectional replication, the system gets unavailable for write requests if this node goes down. By using a external system, a new master could be elected although this is not supported by Redis directly.

4.4.4 Consistency Model

Because of the asynchronous nature of replication, some nodes can contain old replicas. This can even happen without any failures simply due to the network and replication latency. Because of this fact, these systems can only guarantee eventual consistency. If a specific master exists where all write requests go to, then this node can be queried for up-to-date information.

4.4.5 Scaling Methods

Adding nodes to a replication based system can not increase the capacity it can handle because each node holds all the data. Nevertheless the addition of nodes makes sense because it increases the throughput of read requests as they can be distributed onto more nodes. In case of bidirectional replicating systems, the write performance can be increased as well by adding more nodes.

4.4.6 Data Access

As described above, depending on the implementation, the client can only issue write requests to one node. Reads operations are handled by every node.
Range queries are always fast because each node stores all information.

4.5 Summary of the Categorization

To summarize the categorization and comparison, table 4.3 gives a short overview of the differences between the categories which were discussed. In regards to the consistency models involved, figure 4.1 puts the categories on a scale between strict and eventual consistency. Nevertheless it should be clear that there could exist other tunable consistent systems because this concept could be implemented in other systems and other structures than a ring.

	Ring	**Master-Slave**	**Replication Based**
Single Point Of Failure	none	master and slaves	unidirectional: the master otherwise: none
Consistency Model	tunable consistency	strict consistency	eventual consistency
Availability	write: always available read: maybe unavailable	maybe unavailable	unidirectional: Maybe unavailble bidirectional: always available
Data Access	range scans make no sense (if randomly placed)	range scans fast (same slave)	range scans fast (all data available)

Table 4.3: Differences of the categories which have been discussed above

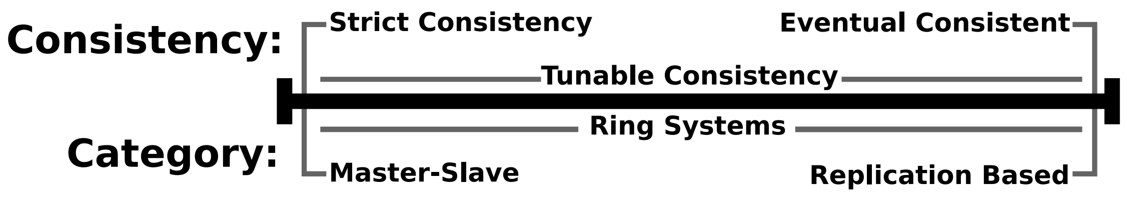

Figure 4.1: The categories introduced directly correlate with the consistency models they implement if put on a scale between strict consistency on the one side and eventual consistency on the other.

5 Use-Cases

When it comes to the decision which NoSQL Database System to choose for a specific use case, multiple factors have to be taken into account. Like on every other system decision the knowledge of the developers is important. In this section some points of the real world use of the database systems presented above will be discussed.

5.1 Simplest Setup

When starting with a data store, it might be important how simple it is to set up the cluster and get a running version. There is a huge difference in the data stores discussed above.
Some of them, namely MongoDB, Redis and CouchDB can be run on a single node. Setting up simply involves starting a process without any configuration which is related to distribution. These systems are developed to be used on single node setups. If during the later development process distribution features are needed, they can be switched on or added then. This initial scaling might involve code changes to add fragmentation algorithms.
Other systems like HBase are much more difficult to set up because distribution features need to be configured from the beginning and many components need to work together and thus be setup up correctly. Because these systems are not designed to be run on a single node, the initial starting is much more difficult.
All ring based systems like Cassandra can started on a single node and do not need a extensive setup up. Because every node in the cluster will behave the same, a later addition of nodes for deployment will not involve a change in the system which was developed. Nevertheless these implementations are not optimized for single node setups so performance will suffer.

5.2 Data Model

Although this thesis focuses on the distribution technologies, the data model plays an important role for the developer. Table 5.1 helps to decide the distribution category and the data model.

5.3 Consistency

Before deciding which NoSQL system should be used, it should be evaluated which consistency model fits best. It should be clear that higher consistency levels like strict

	Column-Oriented	**Document-Oriented**	**Schemaless**
Ring	Cassandra		Riak, Project Voldemort
Master-Slave	HBase	MongoDB	Membase
Replication based		CouchDB	Redis

Table 5.1: Systems compared in regard to data models and distribution categories

consistency imply a lower tolerance to failures like discussed above. If strict consistency is needed for most or every request, systems optimized for this consistency setup like HBase work better than using ring systems and setting their consistency levels to strict consistency. This would involve setting the write threshold to n while keeping the read quorum at 1. This setting leads to the fact that every write request has to be forwarded and answered by every replica which involves heavy network traffic and thus increases the latency. In Master-slave systems on the other side, the write is handled by one single node, the network traffic is reduced and a greater performance can be achieved.
On the other hand, if lower consistency levels are acceptable or desired, ring or replication based systems are well suited. The big advantage here is that these systems can even work if failures occur.

5.4 Concurrent Access

There is a fundamental difference in the handling of concurrent accesses from multiple clients on the same row in the NoSQL systems. Although all systems support this access pattern, the performance is different:
Ring systems run into problems if at the same time many clients write to the same row and the row should be returned in a consistent state. Because the updates can be distributed over the network, even in different data centers, the time needed to reconcile all the conflicting updates is very high.
On the other hand, master-slave systems do not have these problems because all clients access the same slave and the data stored there is always consistent. This advantage is also a disadvantage because this slave can get a bottleneck.
Because of this difference, ring systems are better suited if only a small number of clients concurrently write to the same row. This scenario is typical for private user data in web systems: For example the cart in a online shop is only modified by the client the cart belongs to. Master-slave systems have advantages if it comes to many clients writing to the same row like it is the case for bulk processing systems.

6 Summary

This thesis showed that there a many systems which can help developers to handle large amounts of information while still retaining performant access, durability and availability. To achieve this, the users of those systems must be willing to give up some of the feature traditional database systems have.
The categorization of the systems in the three categories helps to make a decision in two steps: First the developer can select one of these categories and then further examine the systems in this category and their differences. Nevertheless, it is not clear whether other systems not discussed above fit into this categorization or new categories must be created. As the market of NoSQL systems is in a static flux due to its innovative nature, it is likely that new products will be created in future with new ideas and solutions to the problems of large scale databases.
If a shift from traditional systems to NoSQL systems is discussed, not only the technology side must be evaluated but the availability of people with knowledge is also important. Long established traditional systems might have a larger user base and might be tested heavier.

6.1 Further Work

Because of the limited focus of this work, many points have been left out or were not discussed fully:

- A performance evaluation of the systems and categories could help to decide which systems perform better under which workloads. The categorization above can help to decide if a comparison between systems makes sense.
- Replication technologies are not the only distinguishing features of NoSQL database systems: A comparison of the data models (see section 5.2) would help to evaluate the advantages and disadvantages in regards to special use cases.
- Because of the huge market and the limited time some systems were not discussed in this thesis. Adding more systems could show if the categorization is good enough to handle these implementations.

References

J.C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*. Oreilly & Associates Inc, 2010. ISBN 0596155891.

Jason Baker, Chris Bondc, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In Gerhard Weikum, Joseph Hellerstein, and Michael Stonebraker, editors, *Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.

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

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: 10.1145/1281100.1281103.

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

Kristina Chodorow. Sharding Introduction, January 2011. URL `http://www.mongodb.org/display/DOCS/Sharding+Introduction`.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008. doi: 10.1145/1454159.1454167.

Brian F. Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J. Kistler, P. P. S. Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. Building a Cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

Couchbase. Membase unleashed. Technical report, Couchbase, 2010. URL `http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Membase-Technical-Whitepaper.pdf`.

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

M. Dirolf and K. Chodorow. *MongoDB: The Definitive Guide.* O'Reilly Media, Inc., 2010. ISBN 1449381561.

Eric Evans. NOSQL 2009, May 2009. URL `http://blog.sym-link.com/2009/05/12/nosql_2009.html`.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450.

Eben Hewitt. *Cassandra: The Definitive Guide.* O'Reilly Media, Inc., 2010. ISBN 1449390412.

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855840.1855851`.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258660.

A. Khetrapal and V. Ganesh. HBase and Hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 2008.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922.

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

Leslie Lamport. Time clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563.

Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, 1988. Springer-Verlag. ISBN 3-540-18796-0. URL `http://portal.acm.org/citation.cfm?id=646752.704751`.

B. Clifford Neuman and B. Cli Ord Neuman. Scale in Distributed Systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

Salvatore Sanfilippo. Redis Cluster, 2010. URL `http://redis.io/presentation/Redis_Cluster.pdf`.

Michael Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010. doi: 10.1145/1721654.1721659.

Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432.

Tom White. *Hadoop: the definitive guide : [storage and analysis at internet scale]*. OReilly, Beijing, 2. ed., [rev. and updated] edition, 2011. ISBN 978-1-449-38973-4 ; 1-449-38973-2.