NoSQL Databases

Christof Strauch
(cs134@hdm-stuttgart.de)

Lecture
Selected Topics on Software-Technology
Ultra-Large Scale Sites

Lecturer
Prof. Walter Kriha

Course of Studies
Computer Science and Media (CSM)

University
Hochschule der Medien, Stuttgart
(Stuttgart Media University)

Contents

List of Figures

List of Tables

1. Introduction

1.1. Introduction and Overview

Relational database management systems (RDMBSs) today are the predominant technology for storing structured data in web and business applications. Since Codds paper “A relational model of data for large shared data banks“ [Cod70] from 1970 these datastores relying on the relational calculus and providing comprehensive ad hoc querying facilities by SQL (cf. [CB74]) have been widely adopted and are often thought of as the only alternative for data storage accessible by multiple clients in a consistent way. Although there have been different approaches over the years such as object databases or XML stores these technologies have never gained the same adoption and market share as RDBMSs. Rather, these alternatives have either been absorbed by relational database management systems that e. g. allow to store XML and use it for purposes like text indexing or they have become niche products for e. g. OLAP or stream processing.

In the past few years, the ”one size fits all“-thinking concerning datastores has been questioned by both, science and web affine companies, which has lead to the emergence of a great variety of alternative databases. The movement as well as the new datastores are commonly subsumed under the term *NoSQL*, “used to describe the increasing usage of non-relational databases among Web developers” (cf. [Oba09a]).

This paper’s aims at giving a systematic overview of the motives and rationales directing this movement (chapter 2), common concepts, techniques and patterns (chapter 3) as well as several classes of NoSQL databases (key-/value-stores, document databases, column-oriented databases) and individual products (chapters 4–6).

1.2. Uncovered Topics

This paper excludes the discussion of datastores existing before and are not referred to as part of the NoSQL movement, such as object-databases, pure XML databases and DBMSs for special purposes (such as analytics or stream processing). The class of graph databases is also left out of this paper but some resources are provided in the appendix A. It is out of the scope of this paper to suggest individual NoSQL datastores in general as this would be a total misunderstanding of both, the movement and the approach of the NoSQL datastores, as not being “one size fits all”-solutions. In-depth comparisons between all available NoSQL databases would also exceed the scope of this paper.

2. The NoSQL-Movement

In this chapter, motives and main drivers of the NoSQL movement will be discussed along with remarks passed by critics and reactions from NoSQL advocates. The chapter will conclude by different attempts to classify and characterize NoSQL databases. One of them will be treated in the subsequent chapters.

2.1. Motives and Main Drivers

The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL (see [Str10]). The term was picked up again in 2009 and used for conferences of advocates of non-relational databases such as Last.fm developer Jon Oskarsson, who organized the NoSQL meetup in San Francisco (cf. [Eva09a]). A blogger, often referred to as having made the term popular is Rackspace employee Eric Evans who later described the ambition of the NoSQL movement as "the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for" (cf. [Eva09b]).

This section will discuss rationales of practitioners for developing and using nonrelational databases and display theoretical work in this field. Furthermore, it will treat the origins and main drivers of the NoSQL movement.

2.1.1. Motives of NoSQL practioners

The Computerworld magazine reports in an article about the NoSQL meet-up in San Francisco that "NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data." (cf. [Com09a]). It states that especially Web 2.0 startups have begun their business without Oracle and even without MySQL which formerly was popular among startups. Instead, they built their own datastores influenced by Amazon's Dynamo ([DHJ$^+$07]) and Google's Bigtable ([CDG$^+$06]) in order to store and process huge amounts of data like they appear e. g. in social community or cloud computing applications; meanwhile, most of these datastores became open source software. For example, Cassandra originally developed for a new search feature by Facebook is now part of the Apache Software Project. According to engineer Avinash Lakshman, it is able to write 2500 times faster into a 50 gigabytes large database than MySQL (cf. [LM09]).

The Computerworld article summarizes reasons commonly given to develop and use NoSQL datastores:

Avoidance of Unneeded Complexity Relational databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases.

As an example, Adobe's ConnectNow holds three copies of user session data; these replicas do not neither have to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory (cf. [Com09b]).

High Throughput Some NoSQL databases provide a significantly higher data throughput than traditional RDBMSs. For instance, the column-store Hypertable which pursues Google's Bigtable approach allows the local search engine Zvent to store one billion data cells per day [Jud09]. To give another example, Google is able to process 20 petabyte a day stored in Bigtable via it's MapReduce approach [Com09b].

Horizontal Scalability and Running on Commodity Hardware "Definitely, the volume of data is getting so huge that people are looking at other technologies", says Jon Travis, an engineer at SpringSource (cited in [Com09a]). Blogger Jonathan Ellis agrees with this notion by mentioning three problem areas of current relational databases that NoSQL databases are trying to address (cf. [Ell09a]):

1. Scale out data (e. g. 3 TB for the *green badges* feature at Digg, 50 GB for the inbox search at Facebook or 2 PB in total at eBay)
2. Performance of single servers
3. Rigid schema design

In contrast to relational database management systems most NoSQL databases are designed to scale well in the horizontal direction and not rely on highly available hardware. Machines can be added and removed (or crash) without causing the same operational efforts to perform sharding in RDBMS cluster-solutions; some NoSQL datastores even provide automatic sharding (such as MongoDB as of March 2010, cf. [Mer10g]). Javier Soltero, CTO of SpringSource puts it this way: "Oracle would tell you that with the right degree of hardware and the right configuration of Oracle RAC (Real Application Clusters) and other associated magic software, you can achieve the same scalability. But at what cost?" (cited in [Com09a]). Especially for Web 2.0 companies the scalability aspect is considered crucial for their business, as Johan Oskarsson of Last.fm states: "Web 2.0 companies can take chances and they need scalability. When you have these two things in combination, it makes [NoSQL] very compelling." (Johan Oskarsson, Organizer of the meet-up and web developer at Last.fm, cf. [Com09a]). Blogger Nati Shalom agrees with that: "cost pressure also forced many organizations to look at more cost-effective alternatives, and with that came research that showed that distributed storage based on commodity hardware can be even more reliable then[sic!] many of the existing high end databases" (cf. [Sha09a] and for further reading [Sha09c]). He concludes: "All of this led to a demand for a cost effective "scale-first database"".

Avoidance of Expensive Object-Relational Mapping Most of the NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures. They do not make expensive object-relational mapping necessary (such as Key/Value-Stores or Document-Stores). This is particularly important for applications with data structures of low complexity that can hardly benefit from the features of a relational database. Dare Obasanjo claims a little provokingly that "all you really need [as a web developer] is a key<->value or tuple store that supports some level of query functionality and has decent persistence semantics." (cf. [Oba09a]). The blogger and database-analyst Curt Monash iterates on this aspect: "SQL is an awkward fit for procedural code, and almost all code is procedural. [For data upon which users expect to do heavy, repeated manipulations, the cost of mapping data into SQL is] well worth paying [. . .] But when your database structure is very, very simple, SQL may not seem that beneficial." Jon Travis, an engineer at SpringSource agrees with that: "Relational databases give you too much. They force you to twist your object data to fit a RDBMS." (cited in [Com09a]).

In a blog post on the Computerworld article Nati Shalom, CTO and founder of GigaSpaces, identifies the following further drivers of the NoSQL movement (cf. [Sha09b]):

Complexity and Cost of Setting up Database Clusters He states that NoSQL databases are designed in a way that "PC clusters can be easily and cheaply expanded without the complexity and cost of 'sharding,' which involves cutting up databases into multiple tables to run on large clusters or grids".

Compromising Reliability for Better Performance Shalom argues that there are "different scenarios where applications would be willing to compromise reliability for better performance." As an example of such a scenario favoring performance over reliability, he mentions HTTP session data which "needs to be shared between various web servers but since the data is transient in nature (it goes away when the user logs off) there is no need to store it in persistent storage."

The Current "One size fit's it all" Databases Thinking Was and Is Wrong Shalom states that "a growing number of application scenarios cannot be addressed with a traditional database approach". He argues that "this realization is actually not that new" as the studies of Michael Stonebraker (see below) have been around for years but the old 'news' has spread to a larger community in the last years. Shalom thinks that this realization and the search for alternatives towards traditional RDBMSs can be explained by two major trends:

1. The continuous growth of data volumes (to be stored)
2. The growing need to process larger amounts of data in shorter time

Some companies, especially web-affine ones have already adopted NoSQL databases and Shalom expects that they will find their way into mainstream development as these datastores mature. Blogger Dennis Forbes agrees with this issue by underlining that the requirements of a bank are not universal and especially social media sites have different characteristics: "unrelated islands of data", a "very low [. . .] user/transaction value" and no strong need for data integrity. Considering these characteristics he states the following with regard to social media sites and big web applications:

> "The truth is that you don't need ACID for Facebook status updates or tweets or Slashdots comments. So long as your business and presentation layers can robustly deal with inconsistent data, it doesn't really matter. It isn't ideal, obviously, and preferrably [sic!] you see zero data loss, inconsistency, or service interruption, however accepting data loss or inconsistency (even just temporary) as a possibility, breaking free of by far the biggest scaling "hindrance" of the RDBMS world, can yield dramatic flexibility. [. . .]
>
> This is the case for many social media sites: data integrity is largely optional, and the expense to guarantee it is an unnecessary expenditure. When you yield pennies for ad clicks after thousands of users and hundreds of thousands of transactions, you start to look to optimize." (cf. [For10])

Shalom suggests caution when moving towards NoSQL solutions and to get familiar with their specific strengths and weaknesses (e. g. the ability of the business logic to deal with inconsistency). Others, like David Merriman of 10gen (the company behind MongoDB) also stress that there is no single tool or technology for the purpose of data storage but that there is a segmentation currently underway in the database field bringing forth new and different data stores for e. g. business intelligence vs. online transaction processing vs. persisting large amounts of binary data (cf. [Tec09]).

The Myth of Effortless Distribution and Partitioning of Centralized Data Models Shalom further addresses the myth surrounding the perception that data models originally designed with a single database in mind (centralized datamodels, as he puts it) often cannot easily be partitioned and distributed among database servers. This signifies that without further effort, the application will neither necessarily scale and nor work correct any longer. The professionals of Ajatus agree with this in a blog post stating that if a database grows, at first, replication is configured. In addition, as the amount of data grows further, the database is sharded by expensive system admins requiring large financial sums or a fortune worth of money for commercial DBMS-vendors are needed to operate the sharded database (cf. [Aja09]). Shalom reports from an architecture summit at eBay in the summer of 2009. Participants agreed on the fact that although typically, abstractions involved trying to hide distribution and partitioning issues away from applications (e. g. by proxy layers routing requests to certain servers) "this abstraction cannot insulate the application from the reality that [. . .] partitioning and distribution is involved. The spectrum of failures within a network is entirely different from failures within a single machine. The application needs to be made aware of latency, distributed failures, etc., so that it has enough information to make the correct context-specific decision about what to do. The fact that the system is distributed leaks through the abstraction." ([Sha09b]). Therefore he suggests designing datamodels to fit into a partioned environment even if there will be only one centralized database server initially. This approach offers the advantage to avoid exceedingly late and expensive changes of application code.

Shalom concludes that in his opinion relational database management systems will not disappear soon. However, there is definitely a place for more specialized solutions as a "one size fits all" thinking was and is wrong with regards to databases.

Movements in Programming Languages and Development Frameworks The blogger David Intersimone additionally observes movements in programming languages and development frameworks that provide abstractions for database access trying to hide the use of SQL (cf. []) and relational databases ([Int10]). Examples for this trend in the last couple of years include:

- Object-relational mappers in the Java and .NET world like the Java Persistence API (JPA, part of the EJB 3 specification, cf. [DKE06], [BO06],), implemented by e. g. Hibernate ([JBo10a], or the LINQ-Framework (cf. [BH05]) with its code generator SQLMetal and the ADO.NET Entity Framework (cf. [Mic10]) since .NET version 4.
- Likewise, the popular Ruby on Rails (RoR, [HR10]) framework and others try to hide away the usage of a relational database (e. g. by implementing the active record pattern as of RoR).
- NoSQL datastores as well as some databases offered by cloud computing providers completely omit a relational database. One example of such a cloud datastore is Amazon's SimpleDB, a schema-free, Erlang-based eventually consistent datastore which is characterized as an Entity-Attribute-Value (EAV). It can store large collections of items which themselves are hashtables containing attributes that consist of key-value-pairs (cf. [Nor09]).

The NoSQL databases react on this trend and try to provide data structures in their APIs that are closer to the ones of programming languages (e. g. key/value-structures, documents, graphs).

Requirements of Cloud Computing In an interview Dwight Merriman of 10gen (the company behind MongoDB) mentions two major requirements of datastores in cloud computing environments ([Tec09]):

1. High until almost ultimate scalability—especially in the horizontal direction
2. Low administration overhead

In his view, the following classes of databases work well in the cloud:

- Data warehousing specific databases for batch data processing and map/reduce operations.
- Simple, scalable and fast key/value-stores.
- Databases containing a richer feature set than key/value-stores fitting the gap with traditional RDBMSs while offering good performance and scalability properties (such as document databases).

Blogger Nati Shalom agrees with Merriman in the fact that application areas like cloud-computing boosted NoSQL databases: "what used to be a niche problem that only a few fairly high-end organizations faced, became much more common with the introduction of social networking and cloud computing" (cf. [Sha09a]).

The RDBMS plus Caching-Layer Pattern/Workaround vs. Systems Built from Scratch with Scalability in Mind In his article "MySQL and memcached: End of an era?" Todd Hoff states that in a "pre-cloud, relational database dominated world" scalability was an issue of "leveraging MySQL and memcached":

> "Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow[sic!] this pattern today, largely because with enough elbow grease, it works." (cf. [Hof10c])

But as scalability requirements grow and these technologies are less and less capable to suit with them. In addition, as NoSQL datastores are arising Hoff comes to the conclusion that "[with] a little perspective, it's clear the MySQL + memcached era is passing. It will stick around for a while. Old technologies seldom fade away completely." (cf. [Hof10c]). As examples, he cites big websites and players that have moved towards non-relational datastores including LinkedIn, Amazon, Digg and Twitter. Hoff mentions the following reasons for using NoSQL solutions which have been explained earlier in this paper:

- Relational databases place computation on reads, which is considered wrong for large-scale web applications such as Digg. NoSQL databases therefore do not offer or avoid complex read operations.
- The serial nature of applications[1] often waiting for I/O from the data store which does no good to scalability and low response times.
- Huge amounts of data and a high growth factor lead Twitter towards facilitating Cassandra, which is designed to operate with large scale data.
- Furthermore, operational costs of running and maintaining systems like Twitter escalate. Web applications of this size therefore "need a system that can grow in a more automated fashion and be highly available." (cited in [Hof10c]).

For these reasons and the "clunkiness" of the MySQL and memcached era (as Hoff calls it) large scale (web) applications nowadays can utilize systems built from scratch with scalability, non-blocking and asynchronous database I/O, handling of huge amounts of data and automation of maintainance and operational tasks in mind. He regards these systems to be far better alternatives compared to relational DBMSs with additional object-caching. Amazon's James Hamilton agrees with this by stating that for many large-scale web sites scalability from scratch is crucial and even outweighs the lack of features compared to traditional RDBMSs:

[1] Hoff does not give more detail on the types of applications meant here, but—in the authors opinion—a synchronous mindset and implementation of database I/O can be seen in database connectivity APIs (such as ODBC or JDBC) as well as in object-relational mappers and it spreads into many applications from these base technologies.

> “Scale-first applications are those that absolutely must scale without bound and being able to do this without restriction is much more important than more features. These applications are exemplified by very high scale web sites such as Facebook, MySpace, Gmail, Yahoo, and Amazon.com. Some of these sites actually do make use of relational databases but many do not. The common theme across all of these services is that scale is more important than features and none of them could possibly run on a single RDBMS.” (cf. [Ham09] cited in [Sha09a])

Yesterday’s vs. Today’s Needs In a discussion on CouchDB Lehnardt and Lang point out that needs regarding data storage have considerably changed over time (cf. [PLL09]; this argument is iterated further by Stonebraker, see below). In the 1960s and 1970s databases have been designed for single, large high-end machines. In contrast to this, today, many large (web) companies use commodity hardware which will predictably fail. Applications are consequently designed to handle such failures which are considered the “standard mode of operation”, as Amazon refers to it (cf. [DHJ$^+$07, p. 205]). Furthermore, relational databases fit well for data that is rigidly structured with relations and allows for dynamic queries expressed in a sophisticated language. Lehnhardt and Lang point out that today, particularly in the web sector, data is neither rigidly structured nor are dynamic quieries needed as most applications already use prepared statements or stored procedures. Therefore, it is sufficient to predefine queries within the database and assign values to their variables dynamically (cf. [PLL09]).

Furthermore, relational databases were initially designed for centralized deployments and not for distribution. Although enhancements for clustering have been added on top of them it still leaks through that traditional were not designed having distribution concepts in mind at the beginning (like the issues adverted by the “fallaclies of network computing” quoted below). As an example, synchronization is often not implemented efficiently but requires expensive protocols like two or three phase commit. Another difficulty Lehnhardt and Lang see is that clusters of relational databases try to be “transparent” towards applications. This means that the application should not contain any notion if talking to a singly machine or a cluster since all distribution aspects are tried to be hidden from the application. They question this approach to keep the application unaware of all consequences of distribution that are e. g. stated in the famous eight fallacies of distributed computing (cf. [Gos07][2]:

> “Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.
>
> 1. The network is reliable
> 2. Latency is zero
> 3. Bandwidth is infinite
> 4. The network is secure
> 5. Topology doesn’t change
> 6. There is one administrator
> 7. Transport cost is zero

[2] The fallacies are cited according to James Gosling’s here. There is some discussion about who came up with the list: the first seven fallacies are commonly credited to Peter Deutsch, a Sun Fellow, having published them in 1994. Fallacy number eight was added by James Gosling around 1997. Though—according to the English Wikipedia—“Bill Joy and Tom Lyon had already identified the first four as "The Fallacies of Networked Computing"” (cf. [Wik10]). More details on the eight fallacies can be found in an article of Rotem-Gal-Oz (cf. [RGO06])

8. The network is homogeneous"

While typical business application using relational database management systems try, in most cases, to hide distribution aspects from the application (e. g. by clusters, persistence layers doing object-relational mapping) many large web companies as well as most of the NoSQL databases do not pursue this approach. Instead, they let the application know and leverage them. This is considered a paradigm shift in the eyes of Lehnardt and Lang (cf. [PLL09]).

Further Motives In addition to the aspects mentioned above David Intersimone sees the following three goals of the NoSQL movement (cf. [Int10]):

- Reach less overhead and memory-footprint of relational databases
- Usage of Web technologies and RPC calls for access
- Optional forms of data query

2.1.2. Theoretical work

In their widely adopted paper "The End of an Architectural Era" (cf. [SMA+07]) Michael Stonebraker[3] et al. come to the conclusion "that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing". *Nothing* in this context means that they can neither compete with "specialized engines in the data warehouse, stream processing, text, and scientific database markets" which outperform them "by 1–2 orders of magnitude" (as shown in previous papers, cf. [Sc05], [SBc+07]) nor do they perform well in their their home market of business data processing / online transaction processing (OLTP), where a prototype named H-Store developed at the M.I.T. beats up RDBMSs by nearly two orders of magnitude in the TPC-C benchmark. Because of these results they conclude that RDBMSs" are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not continue to push code lines and architectures designed for yesterday's needs". But how do Stonebraker et al. come to this conclusion? Which inherent flaws do they find in relational database management systems and which suggestions do they provide for the "complete rewrite" they are requiring?

At first, Stonebraker et al. argue that RDBMSs have been architected more than 25 years ago when the hardware characteristics, user requirements and database markets where different from those today. They point out that "popular relational DBMSs all trace their roots to System R from the 1970s": IBM's DB2 is a direct descendant of System R, Microsoft's SQL Server has evolved from Sybase System 5 (another direct System R descendant) and Oracle implemented System R's user interface in its first release. Now, the architechture of System R has been influenced by the hardware characteristics of the 1970s. Since then, processor speed, memory and disk sizes have increased enormously and today do not limit programs in the way they did formerly. However, the bandwidth between hard disks and memory has not increased as fast as the CPU speed, memory and disk size. Stonebraker et al. criticize that this development in the field of hardware has not impacted the architecture of relational DBMSs. They especially see the following architectural characteristics of System R shine through in today's RDBMSs:

- "Disk oriented storage and indexing structures"
- "Multithreading to hide latency"

[3] When reading and evaluating Stonebraker's writings it has to be in mind that he is commercially involved into multiple DBMS products such Vertica, a column-store providing data warehousing and business analytics.

- “Locking-based concurrency control mechanisms”
- “Log-based recovery”

In this regard, they underline that although “there have been some extensions over the years, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators [. . .] no system has had a complete redesign since its inception”.

Secondly, Stonebraker et al. point out that new markets and use cases have evolved since the 1970s when there was only business data processing. Examples of these new markets include “data warehouses, text management, and stream processing” which “have very different requirements than business data processing”. In a previous paper (cf. [Sc05]) they have shown that RDBMSs “could be beaten by specialized architectures by an order of magnitude or more in several application areas, including:

- Text (specialized engines from Google, Yahoo, etc.)
- Data Warehouses (column stores such as Vertica, Monet, etc.)
- Stream Processing (stream processing engines such as StreamBase and Coral8)
- Scientific and intelligence databases (array storage engines such as MATLAB and ASAP)”

They go on noticing that user interfaces and usage model also changed over the past decades from terminals where “operators [were] inputting queries” to rich client and web applications today where interactive transactions and direct SQL interfaces are rare.

Stonebraker et al. now present “evidence that the current architecture of RDBMSs is not even appropriate for business data processing”. They have designed a DBMS engine for OLTP called H-Store that is functionally equipped to run the TPC-C benchmark and does so 82 times faster than a popular commercial DBMS. Based on this evidence, they conclude that “there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step”.

Design Considerations

In a section about design considerations Stonebraker et al. explain why relational DBMSs can be outperformed even in their home market of business data processing and how their own DMBS prototype H-Store can “achieve dramatically better performance than current RDBMSs”. Their considerations especially reflect the hardware development over the past decades and how it could or should have changed the architecture of RDBMSs in order to gain benefit from faster and bigger hardware, which also gives hints for “the complete rewrite” they insist on.

Stonebraker et al. see five particularly significant areas in database design:

Main Memory As—compared to the 1970s—enormous amounts of main memory have become cheap and available and as “The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing [. . .] quite slowly” they conclude that such databases are “capable of main memory deployment now or in near future”. Stonebraker et al. therefore consider the OLTP market a main memory market even today or in near future. They criticize therefore that “the current RDBMS vendors have disk-oriented solutions for a main memory problem. In summary, 30 years of Moore’s law has antiquated the disk-oriented relational architecture for OLTP applications”. Although there are relational databases operating in memory (e. g. TimesTen, SolidDB) these systems also inherit “baggage”—as Stonebraker et al. call it—from System R, e. g. disk-based recovery logs or dynamic locking, which have a negative impact on the performance of these systems.

Multi-Threading and Resource Control As discussed before, Stonebraker et al. consider databases a main-memory market. They now argue that transactions typically affect only a few data sets that have to be read and/or written (at most 200 read in the TPC-C benchmark for example) which is very cheap if all of this data is kept in memory and no disk I/O or user stalls are present. As a consequence, the do not see any need for multithreaded execution models in such main-memory databases which makes a considerable amount of "elaborate code" of conventional relational databases irrelevant, namely multi-threading systems to maximize CPU- and disk-usage, resource governors limiting load to avoid resource exhausting and multi-threaded datastructures like concurrent B-trees. "This results in a more reliable system, and one with higher performance", they argue. To avoid long running transactions in such a single-threaded system they require either the application to break up such transactions into smaller ones or—in the case of analytical purposes—to run these transactions in data warehouses optimized for this job.

Grid Computing and Fork-Lift Upgrades Stonebraker et al. furthermore outline the development from shared-memory architectures of the 1970s over shared-disk architectures of the 1980s towards shared-nothing approaches of today and the near future, which are "often called grid computing or blade computing". As a consequence, Stonebraker et al. insist that databases have to reflect this development, for example (and most obviously) by horizontal partitioning of data over several nodes of a DBMS grid. Furthermore, they advocate for incremental horizontal expansion of such grids without the need to reload any or all data by an administrator and also without downtimes. They point out that these requirements have significant impact on the architecture of DBMSs—e. g. the ability to transfer parts of the data between nodes without impacting running transactions—which probably cannot be easily added to existing RDBMSs but can be considered in the design of new systems (as younger databases like Vertica show).

High Availability The next topics Stonebraker et al. address are high availability and failover. Again, they outline the historical development of these issues from log-tapes that have been sent off site by organizations and were run on newly delivered hardware in the case of disaster over disaster recovery services installing log-tapes on remote hardware towards hot standby or multiple-site solutions that are common today. Stonebraker et al. regard high availability and built-in disaster recovery as a crucial feature for DBMSs which—like the other design issues they mention—has to be considered in the architecture and design of these systems. They particularly require DBMSs in the OLTP field to

1. "keep multiple replicas consistent, requiring the ability to run seamlessly on a grid of geographically dispersed systems"
2. "start with shared-nothing support at the bottom of the system" instead of gluing "multi-machine support onto [. . .] SMP architectures."
3. support a shared-nothing architecture in the best way by using "multiple machines in a peer-to-peer configuration" so that "load can be dispersed across multiple machines, and inter-machine replication can be utilized for fault tolerance". In such a configuration all machine resources can be utilized during normal operation and failures only cause a degraded operation as fewer resources are available. In contrast, todays HA solutions having a hot standby only utilize part of the hardware resources in normal operation as standby machines only wait for the live machines to go down. The conclude that "[these] points argue for a complete redesign of RDBMS engines so they can implement peer-to-peer HA in the guts of a new architecture" they conclude this aspect.

In such a highly available system that Stonebraker et al. require they do not see any need for a redo log as in the case of failure a dead site resuming activity "can be refreshed from the data on an operational site". Thus, there is only a need for an undo log allowing to rollback transactions. Such an undo log does not have to be persisted beyond a transaction and therefore "can be a main memory data structure that is discarded on transaction commit". As "In an HA world, one is led to having no persistent redo log, just a transient undo one" Stonebraker et al. see another potential to remove complex code that is needed for recovery from a redo log; but they also admit that the recovery logic only changes "to new functionality to bring failed sites up to date from operational sites when they resume operation".

No Knobs Finally, Stonebraker et al. point out that current RDBMSs were designed in an "era, [when] computers were expensive and people were cheap. Today we have the reverse. Personnel costs are the dominant expense in an IT shop". They especially criticize that "RDMBSs have a vast array of complex tuning knobs, which are legacy features from a bygone era" but still used as automatic tuning aids of RDBMSs "do not produce systems with anywhere near the performance that a skilled DBA can produce". Instead of providing such features that only try to figure out a better configuration for a number of knobs Stonebraker et al. require a database to have no such knobs at all but to be ""self-everything" (self-healing, self-maintaining, self- tuning, etc.)".

Considerations Concerning Transactions, Processing and Environment

Having discussed the historical development of the IT business since the 1970s when RDBMSs were designed and the consequences this development should have had on their architecture Stonebraker et al. now turn towards other issues that impact the performance of these systems negatively:

- Persistent redo-logs have to be avoided since they are "almost guaranteed to be a significant performance bottleneck". In the HA/failover system discussed above they can be omitted totally.
- Communication between client and DBMS-server via JDBC/ODBC-like interfaces is the next performance-degrading issue they address. Instead of such an interface they "advocate running application logic – in the form of stored procedures – "in process" inside the database system" to avoid "the inter-process overheads implied by the traditional database client / server model."
- They suggest furthermore to eliminate an undo-log "wherever practical, since it will also be a significant bottleneck".
- The next performance bottleneck addressed is dynamic locking to allow concurrent access. The cost of dynamic locking should also be reduced or eliminated.
- Multi-threaded datastructures lead to latching of transactions. If transaction runtimes are short, a single-threaded execution model can eliminate this latching and the overhead associated with multi-threaded data structures "at little loss in performance".
- Finally, two-phase-commit (2PC) transactions should be avoided whenever possible as the network round trips caused by this protocol degrade performance since they "often take the order of milliseconds".

If these suggestions can be pursued depends on characteristics of OLTP transactions and schemes as Stonbraker et al. point out subsequently.

Transaction and Schema Characteristics

In addition to hardware characteristics, threading model, distribution or availability requirements discussed above, Stonebraker et al. also point out that the characteristics of database schemes as well as transaction properties also significantly influence the performance of a DBMS. Regarding database schemes and transactions they state that the following characteristics should be exploited by a DBMS:

Tree Schemes are database schemes in which “every table except a single one called *root*, has exactly one join term which is a 1-n relationship with its ancestor. Hence, the schema is a tree of 1-n relationships”. Schemes with this property are especially easy to distribute among nodes of a grid “such that all equi-joins in the tree span only a single site”. The root table of such a schema may be typically partitioned by it’s primary key and moved to the nodes of a grid, so that on each node has the partition of the root table together with the data of all other tables referencing the primary keys in that root table partition.

Constrained Tree Application (CTA) in the notion of Stonebraker et al. is an applications that has a tree schema and only runs transactions with the following characteristics:

1. “every command in every transaction class has equality predicates on the primary key(s) of the root node”
2. “every SQL command in every transaction class is local to one site”

Transaction classes are collections of “the same SQL statements and program logic, differing in the run-time constants used by individual transactions” which Stonebraker et al. require to be defined in advance in their prototype H-Store. They furthermore argue that current OLTP applications are often designed to be CTAs or that it is at least possible to decompose them in that way and suggest schema transformations to be applied systematically in order to make an application CTA (cf. [SMA+07, page 1153]. The profit of these efforts is that “CTAs [. . .] can be executed very efficiently”.

Single-Sited Transactions can be executed to completion on only one node without having to communicate with other sites of a DBMS grid. Constrained tree application e. g. have that property.

One-Shot Applications entirely consist of “transactions that can be executed in parallel without requiring intermediate results to be communicated among sites”. In addition, queries in one-shot applications never use results of earlier queries. These properties allow the DBMS to decompose transactions “into a collection of single-site plans which can be dispatched to the appropriate sites for execution”. A common technique to make applications one-shot is to partition tables vertically among sites.

Two-Phase Transactions are transactions that contain a first phase of read operations, which can—depending on its results—lead to an abortion of the transaction, and a second phase of write operations which are guaranteed to cause no integrity violations. Stonebraker et al. argue that a lot of OLTP transactions have that property and therefore exploit it in their H-Store prototype to get rid of the undo-log.

Strongly Two-Phase Transactions in addition to two-phase transactions have the property that in the second phase all sites either rollback or complete the transaction.

Transaction Commutativity is defined by Stonebraker et al. as follows: “Two concurrent transactions from the same or different classes *commute* when any interleaving of their single-site sub-plans produces the same final database state as any other interleaving (assuming both transactions commit)”.

Sterile Transactions Classes are those that commute “with all transaction classes (including itself)”.

H-Store Overview

Having discussed the parameters that should be considered when designing a database management system today, Stonebraker et al. sketch their protoype H-Store that performs significantly better than a commercial DBMS in the TPC-C benchmark. Their system sketch shall not need to be repeated in this paper (details can be found in [SMA+07, p. 154ff], but a few properties shall be mentioned:

- H-Store runs on a grid

- On each rows of tables are placed contiguously in main memory
- B-tree indexing is used
- Sites are partitioned into logical sites which are dedicated to one CPU core
- Logical sites are completely independent having their own indexes, tuple storage and partition of main memory of the machine they run on
- H-Store works single-threaded and runs transactions uninterrupted
- H-Store allows to run only predefined transactions implemented as stored procedures
- H-Store omits the redo-log and tries to avoid writing an undo-log whenever possible; if an undo-log cannot be avoided it is discarded on transaction commit
- If possible, query execution plans exploit the single-sited and one-shot properties discussed above
- H-Store tries to achieve the no-knobs and high availability requirements as well as transformation of transactions to be single-sited by "an automatical physical database designer which will specify horizontal partitioning, replication locations, indexed fields"
- Since H-Store keeps replicas of each table these have to be updated transactionally. Read commands can go to any copy of a table while updates are directed to all replicas.
- H-Store leverages the above mentioned schema and transaction characteristics for optimizations, e. g. omitting the undo-log in two-phase transactions.

TPC-C Benchmark

When comparing their H-Store prototype with a commercial relational DBMS Stonebraker et al. apply some important tricks to the benchmarks implementation. First, they partition the database scheme and replicate parts of it in such a way that "the schema is decomposed such that each site has a subset of the records rooted at a distinct partition of the warehouses". Secondly, they discuss how to profit of transaction characteristics discussed above. If the benchmark would run "on a single core, single CPU machine" then "every transaction class would be single-sited, and each transaction can be run to completion in a single-threaded environment". In a "paired-HA site [. . .]all transaction classes can be made strongly two-phase, meaning that all transactions will either succeed or abort at both sites. Hence, on a single site with a paired HA site, ACID properties are achieved with no overhead whatsoever." By applying some further tricks, they achieve that "with the basic strategy [of schema partitioning and replication (the author of this paper)] augmented with the tricks described above, all transaction classes become one-shot and strongly two-phase. As long as we add a short delay [. . .], ACID properties are achieved with no concurrency control overhead whatsoever."

Based on this setting, they achieved a 82 times better performance compared to a commercial DBMS. They also analyzed the overhead of performance in the commercial DBMS and examined that it was mainly caused by logging and concurrency control.

It has to be said that Stonebraker et al. implemented only part of the TPC-C benchmark and do not seem to have adapted the TPC-C benchmark to perfectly fit with the commercial DBMS as they did for their own H-Store prototype although they hired a professional DBA to tune the DBMS they compared to H-Store and also tried to optimize the logging of this system to allow it to perform better.

Consequences

As a result of their analysis Stonebraker et al. conclude that “we are heading toward a world with at least 5 (and probably more) specialized engines and the death of the “one size fits all” legacy systems”. This applies to the relational model as well as its query language SQL.

Stonebraker et al. state that in contrast to the 1970s when “the DBMS world contained only business data processing applications” nowadays there are at least the following markets which need specialized DBMSs[4]:

1. **Data warehouses** which typically have star or snowflake schemes, i. e. “a central fact table with 1-n joins to surrounding dimension tables, which may in turn participate in further 1-n joins to second level dimension tables, and so forth”. These datastructures could be easily modeled using the relational model but Stonebraker et al. suggest an entity-relationship model in this case which would be simpler and more natural to model and to query.
2. The **stream processing** market has different requirements, namely to “Process streams of messages at high speed [and to] Correlate such streams with stored data”. An SQL generalization called StreamSQL which allows to mix streams and relational data in SQL FROM-clauses has caused some enthusiasm and been suggested for standardization in this field. Stonebraker et al. also mention a problem in stream processing often requiring stream data to be flat (as some news agencies deliver it) but there is also a need for hierarchically structured data. Therefore, they “expect the stream processing vendors to move aggressively to hierarchical data models” and that “they will assuredly deviate from Ted Codd’s principles”.
3. **Text processing** is a field where relational databases have never been used.
4. **Scientific-oriented databases** will likely supply arrays rather than tables as their basic data structure.
5. **Semi-structured data** is a field where useful data models are still being discussed. Suggestions include e. g. XML schema (fiercely debated because of its complexity) and RDF.

While “the relational model was developed for a “one size fits all” world, the various specialized systems which we envision can each rethink what data model would work best for their particular needs“ Stonebraker et al. conclude.

Regarding query languages for DBMSs they argue against a “one size fits all language” like SQL as, in their opinion, it has no use case at all: in the OLTP market ad-hoc queries are seldom or not needed, applications query in a prepared statement fashion or the query logic is deployed into the DBMS as stored procedures. In opposition to this, other DBMS markets like data warehouses need abilities for complex ad-hoc queries which are not fulfilled by SQL. For this reason, Stonebraker et al. do not see a need for this language any more.

For specialized DBMSs in the above mentioned markets they furthermore discuss how these languages should integrate with programming languages and argue against data sublanguages “interfaced to any programming language” as it is done in JDBC and ODBC. “[this] has led to high overhead interfaces”. Stonebraker et al. therefore suggest an “embedding of database capabilities in programming languages”. Examples of such language embeddings include Pascal R and Rigel in the 1970s or Microsoft’s LINQ-approach in the .NET platform nowadays. Regarding the languages to integrate such capabilities they are in favor of what they call “little languages” like Python, Perl, Ruby and PHP which are “open source, and can be altered by the community” and are additionally “less daunting to modify than the current general

[4] Further details on these DBMS markets can be found in the earlier released paper *“One Size Fits All”: An Idea Whose Time Has Come and Gone* (cf. [Sc05])

purpose languages" (that appear to them as a "one size fits all approach in the programming languages world"). For their H-Store prototype they plan to move from C++ to Ruby for stored procedures.

2.1.3. Main Drivers

The NoSQL movement has attracted a great number of companies and projects over the last couple of years. Especially big web companies or businesses running large and highly-frequented web sites have switched from relational databases (often with a caching layer typically realized with memcached[5], cf. [Hof10c], [Hof10b]) towards non-relational datastores. Examples include Cassandra ([Apa10d]) originally developed at Facebook and also used by Twitter and Digg today (cf. [Pop10a], [Eur09]), Project Voldemort developed and used at LinkedIn, cloud services like the NoSQL store Amazon SimpleDB (cf. [Ama10b]) as well as Ubuntu One, a cloud storage and synchronization service based on CouchDB (cf. [Can10c], [Hof10c]). These users of NoSQL datastores are naturally highly interested in the further development of the non-relational solutions they use. However, most of the popular NoSQL datastores have adopted ideas of either Google's Bigtable (cf. [CDG$^+$06]) or Amazon's Dynamo (cf. [DHJ$^+$07]); Bigtable-inspired NoSQL stores are commonly referred to as column-stores (e.g. HyperTable, HBase) whereas the Dynamo influenced most of the key-/values-stores (e.g. Cassandra [Apa10d], Redis [S$^+$10], Project Voldemort [K$^+$10a]). Other projects pursue different attempts like graph-databases that form an own class of data stores and document-stores which can be seen as key/value-stores with additional features (at least allowing different, often hierarchical namespaces for key/value-pairs which provides the abstraction of "documents"); at least one of the latter (CouchDB) is a reimplementation of existing document stores such as Lotus Notes, which has been around since the 1990s and is consequently not designed with the awareness of current web technologies (which is criticized by CouchDB-developers who consequently implemented their document-store from scratch, cf. [Apa10c], [PLL09]). In summary, it can be concluded that the pioneers of the NoSQL movement are mainly big web companies or companies running large-scale web sites like Facebook, Google and Amazon (cf. [Oba09a]) and others in this field have adopted their ideas and modified them to meet their own requirements and needs.

2.2. Criticism

2.2.1. Scepticism on the Business Side

In an article about the NoSQL meet-up in San Francisco Computerworld mentions some business related issues concerning NoSQL databases. As most of them are open-source software they are well appreciated by developers who do not have to care about licensing and commercial support issues. However, this can scare business people in particular in the case of failures with nobody to blame for. Even at Adobe the developers of ConnectNow which uses a Terracotta cluster instead of a relational database were only able to convince their managers when they saw the system up and running (cf. [Com09a]).

2.2.2. NoSQL as a Hype

Some businesses appear to be cautious towards NoSQL as the movement seems like a hype potentially lacking the fulfillment of its promises. This is a general skepticism towards new technologies provoking considerable enthusiasm and has been expressed e.g. by James Bezdek in an IEEE editorial as follows:

[5] Examples include Friendfeed, Wikipedia, XING, StudiVz/SchülerVz/MeinVz

> “Every new technology begins with naïve euphoria – its inventor(s) are usually submersed in the ideas themselves; it is their immediate colleagues that experience most of the wild enthusiasm. Most technologies are overpromised, more often than not simply to generate funds to continue the work, for funding is an integral part of scientific development; without it, only the most imaginative and revolutionary ideas make it beyond the embryonic stage. Hype is a natural handmaiden to overpromise, and most technologies build rapidly to a peak of hype. Following this, there is almost always an overreaction to ideas that are not fully developed, and this inevitably leads to a crash of sorts, followed by a period of wallowing in the depths of cynicism. Many new technologies evolve to this point, and then fade away. The ones that survive do so because someone finds a good use (= true user benefit) for the basic ideas.” (cf. [Bez93] cited in [For10])

The participants of the NoSQL meet-up in San Francisco gave pragmatic advice for such remarks: companies do not miss anything if they do not switch to NoSQL databases and if a relational DBMS does its job, there is no reason to replace it. Even the organizer of the meet-up, Johan Oskarsson of Last.fm, admitted that Last.fm did not yet use a NoSQL database in production as of June 2009. He furthermore states that NoSQL databases “aren’t relevant right now to mainstream enterprises, but that might change one to two years down the line” (Johan Oskarsson, Last.fm, cf. [Com09a]). Nonetheless, the participants of the NoSQL meet-up suggest to take a look at NoSQL alternatives if it is possible and it makes sense (e. g. in the development of new software) (cf. [Com09a]). Blogger Dennis Forbes does not see any overenthusiasm among the inventors and developers of non-relational datastores (“most of them are quite brilliant, pragmatic devs”) but rather among developers using these technologies and hoping that “this movement invalidates their weaknesses”. He—coming from traditional relational database development for the financial, insurance, telecommunication and power generation industry—however states that “there is indisputably a lot of fantastic work happening among the NoSQL camp, with a very strong focus on scalability”. On the other hand, he criticizes in a postnote to his blog post that “the discussion is, by nature of the venue, hijacked by people building or hoping to build very large scale web properties (all hoping to be the next Facebook), and the values and judgments of that arena are then cast across the entire database industry—which comprises a set of solutions that absolutely dwarf the edge cases of social media—which is really...extraordinary” (cf. [For10]).

2.2.3. NoSQL as Being Nothing New

Similar to the hype argument are common remarks by NoSQL critics that NoSQL databases are nothing new since other attempts like object databases have been around for decades. As an example for this argument blogger David Intersimone mentions Lotus Notes which can be subsumed as an early document store supporting distribution and replication while favoring performance over concurrency control (unless otherwise indicated, [Int10]). This example is especially interesting as the main developer of CouchDB, Damien Katz, worked for Lotus Notes for several years. NoSQL advocates comment that CouchDB is 'Notes done right’ as Notes’ distribution features were not aware of current web technologies and the software also got bloated with business relevant features instead of beeing just a slim datastore ([PLL09]).

Examples like Lotus Notes, business analytic or stream processing oriented datastores show that these alternatives to relational databases have existed for a long time and blogger Dennis Forbes criticizes therefore that the “one size fitting it all” argument is nothing more but a strawman since few people ever held “Rdbms’ as the only tool for all of your structured and unstructured data storage needs” (cf. [For10]).

2.2.4. NoSQL Meant as a Total “No to SQL”

At first, many NoSQL advocates especially in the blogosphere understood the term and the movement as a total denial of RDBMSs and proclaimed the death of these systems. Eric Evans to whom the term “NoSQL” is often credited though he was not the first who used it (see section 2.1 and e. g. [Ell09a]) suggested in a blog post of 2009 that the term now should mean “Not only SQL” instead of “No to SQL” (cf. [Eva09b]). This term has been adopted by many bloggers as it stresses that persistence in databases does not automatically mean to use a relational DBMS but that alternatives exist. Blogger Nati shalom comments this shift in the following way: “I think that what we are seeing is more of a realization that existing SQL database alternatives are probably not going away any time soon, but at the same time they can’t solve all the problems of the world. Interestingly enough the term NOSQL has now been changed to Not Only SQL, to represent that line of thought” (cf. [Sha09a]). Some bloggers stress that the term is imprecise such as Dare Obasanjo saying that “there is a[sic!] yet to be a solid technical definition of what it means for a product to be a "NoSQL" database aside from the fact that it isn’t a relational database” (cf. [Oba09a]). Others, such as Michael Stonebraker claim that it has nothing to do with SQL at all[6] and should be named something like “NoACID“ as suggested by Dennis Forbes, who refers to Stonebraker in his suggestion ([For10]). Still others such as Adam Keys criticize the term “NoSQL” as defining the movement by what it does not stand for ([Key09]): “The problem with that name is that it only defines what it is not. That makes it confrontational and not amazingly particular to what it includes or excludes. [. . .] What we’re seeing its [sic!] the end of the assumption that valuable data should go in some kind of relational database. The end of the assumption that SQL and ACID are the only tools for solving our problems. The end of the viability of master/slave scaling. The end of weaving the relational model through our application code”. He suggests to subsume the movement and the datastores under the term “post-relational” instead of “NoSQL”: “We’re seeing an explosion in the ideas about how one should store important data. We’re looking at data to see if it’s even worth persisting. We’re experimenting with new semantics around structure, consistency and concurrency. [. . .] In the same way that post-modernism is about reconsidering the ways of the past in art and architecture, post-relational is a chance for software developers to reconsider our own ways. Just as post-modernism didn’t invalidate the entire history of art, post-relational won’t invalidate the usefulness of relational databases.” (cf. [Key09]). However, as the readers of his blog post comment this term is not that much better than “NoSQL” as it still defines the movement and the databases by what they do not reflect (or better: by what they have omitted) instead of what they stand for.

The irritation about the term and its first notion as a total neglect of relational databases has lead to many provoking statements by NoSQL advocates[7] and caused a number of unfruitful discussions and some flamewars (see e. g. [Dzi10] and as a response to it [Sch10]).

2.2.5. Stonebraker’s Critical Reception of NoSQL Databases

In his blog post “The “NoSQL” Discussion has Nothing to Do With SQL” ([Sto09]) Michael Stonebraker states that there has been a lot of buzz around NoSQL databases lately. In his reception the main drivers behind NoSQL conferences in the US are advocates of document stores and key/values-stores which provide a “a low-level record-at-a-time DBMS interface, instead of SQL” in his sight. Stonebraker sees two reasons for moving towards non-relational datastores—flexibility and performance.

[6] See the next section on Stonebraker’s reception of the NoSQL-movement which goes far beyond criticizing only the term “NoSQL“.

[7] E. g. “The ACIDy, Transactional, RDBMS doesn’t scale, and it needs to be relegated to the proper dustbin before it does any more damage to engineers trying to write scalable software”. The opinion expressed here has been softened by the author in a postnote to his blog post: “This isn’t about a complete death of the RDBMS. Just the death of the idea that it’s a tool meant for all your structured data storage needs.” (cf. [Ste09] cited in [For10])

The Flexibility Argument is not further examined by Stonebraker but it contains the following view: there might be data that does not fit into a rigid relational model and which is bound too much by the structure of a RDBMS. For this kind of data something more flexible is needed.

The Performance Argument is described as follows Stonebraker: one starts with MySQL to store data and performance drops over the time. This leads to the following options: either to shard/partition data among several sites causing "a serious headache managing distributed data" in the application; or to move from MySQL towards a commercial RDMBS which can provoke large licensing fees; or to even totally abandon a relational DBMS.

Stonebraker subsequently examines the latter argument in his blog post. He focuses on "workloads for which NoSQL databases are most often considered: update- and lookup-intensive OLTP workloads, not query-intensive data warehousing workloads" or specialized workflows like document-repositories.

Stonebraker sees two options to improve the performance of OLTP transactions:

1. Horizontal scaling achieved by automatic sharding "over a shared-nothing processing environment". In this scenario performance gets improved by adding new nodes. In his point of view, RDMBSs written in the last ten years provide such a "shared nothing architecture" and "nobody should ever run a DBMS that does not provide" this.
2. Improvement of the OLTP performance of a single node.

Stonebraker focuses on the second option and analyzes the sources of overhead which decrease performance of OLTP transactions on a single node; as indicated by the title of his blog post, these have no relation to the the query language (SQL). He states only a small percentage of total transaction cost is incurred by useful work and sees five sources of performance overhead that have to be addressed when single node performance should be optimized:

Communication between the application and the DBMS via ODBC or JDBC. This is considered the main source of overhead in OLTP transactions which is typically addressed as follows: "Essentially **all** applications that are performance sensitive use a stored-procedure interface to run application logic inside the DBMS and avoid the crippling overhead of back-and-forth communication between the application and the DBMS". The other option to reduce communication overhead is using an embeddable DBMS which means that the application and the DBMS run in the same address space; because of strong coupling, security and access control issues this is no viable alternative "for mainstream OLTP, where security is a big deal" in Stonebrakers sight.

Logging is done by traditional DBMSs in addition to modifications of relational data on each transaction. As log files are persisted to disk to ensure durability logging is expensive and decreases transaction performance.

Locking of datasets to be manipulated causes overhead as write operations in the lock-table have to occur before and after the modifications of the transaction.

Latching because of shared data structures (e. g. B-trees, the lock-table, resource-tables) inside an RDBMS incurs further transaction costs. As these datastructures have to be accessed by multiple threads short-term locks (aka latches) are often used to provide parallel but careful access to them.

Buffer Management finally also plays its part when it comes to transaction overhead. As data in traditional RDBMSs is organized in fixed pages work has to be done to manage the disk-pages cached in memory (done by the buffer-pool) and also to resolve database entries to disk pages (and back) and identify field boundaries.

As stated before, communication is considered the main source of overhead according to Stonebraker and by far outweighs the other ones (take from this survey: [HAMS08]) which almost equally increase total transaction costs. Besides avoiding communication between the application and the database all four other sources of performance overhead have to be eliminated in order to considerably improve single node performance.

Now, datastores whether relational or not have specific themes in common and NoSQL databases also have to address the components of performance overhead mentioned above. In this context, Stonebraker raises the following examples:

- Distribution of data among multiple sites and a shared-nothing approach is provided by relational as well as non-relational datastores. “Obviously, a well-designed multi-site system, whether based on SQL or something else, is way more scalable than a single-site system” according to Stonebraker.
- Many NoSQL databases are disk-based, implement a buffer pool and are multi-threaded. When providing these features and properties, two of the four sources of performance overhead still remain (Locking, Buffer management) and cannot be eliminated.
- Transaction-wise many NoSQL datastores provide only single-record transactions with BASE properties (see chapter 3 on that). In contrast to relational DBMSs ACID properties are sacrificed in favor of performance.

Stonebraker consequently summarizes his considerations as follows: “However, the net-net is that the single-node performance of a NoSQL, disk-based, non-ACID, multithreaded system is limited to be a modest factor faster than a well-designed stored-procedure SQL OLTP engine. In essence, ACID transactions are jettisoned for a modest performance boost, and this performance boost has nothing to do with SQL”. In his point of view, the real tasks to speed up a DBMS focus on the elimination of locking, latching, logging and buffer management as well as support for stored procedures which compile a high level language (such as SQL) into low level code. How such a system can look like is described in the paper “The end of an architectural era: (it’s time for a complete rewrite)” ([SMA$^+$07]) that has already been discussed above.

Stonebraker also does not expect SQL datastores to die but rather states: “I fully expect very high speed, open-source SQL engines in the near future that provide automatic sharding. [. . .] Moreover, they will continue to provide ACID transactions along with the increased programmer productivity, lower maintenance, and better data independence afforded by SQL.” Hence “high performance does not require jettisoning either SQL or ACID transactions”. It rather “depends on removing overhead” caused by traditional implementations of ACID transactions, multi-threading and disk management. The removal of these sources of overhead “is possible in either a SQL context or some other context”, Stonebraker concludes.

2.2.6. Requirements of Administrators and Operators

In his blog post “The dark side of NoSQL” (cf. [Sch09]) Stephan Schmidt argues that the NoSQL debate is dominated by a developer’s view on the topic which usually iterates on properties and capabilities developers like (e. g. performance, ease of use, schemalessness, nice APIs) whereas the needs of operations people and system administrators are often forgotten in his sight. He reports that companies[8] encounter difficulties especially in the following fields:

Ad Hoc Data Fixing To allow for ad hoc data fixing there first has to be some kind of query and manipulation language. Secondly, it is more difficult to fix data in distributed databases (like Project Voldemort or Cassandra) compared to datastores that run on a single node or have dedicated shards.

[8] He cites an Infinispan director and the vice president of engineering at a company called Loop.

Ad Hoc Data Querying Similarly to data fixing a query and manipulation for the particular datastore is required when it comes to ad hoc queries and querying distributed datastores is harder than querying centralized ones. Schmidt states that for some reporting tasks the MapReduce approach (cf. [DG04]) is the right one, but not for every ad hoc query. Furthermore, he sees the rather cultural than technical problem that customers have become trained and "addicted" to ad hoc reporting and therefore dislike the absence of these means. For exhaustive reporting requirements Schmidt suggests to use a relational database that mirrors the data of live databases for which a NoSQL store might be used due to performance and scalability requirements.

Data Export Schmidt states that there are huge differences among the NoSQL databases regarding this aspect. Some provide a useful API to access all data and in some it is absent. He also points out that it is more easy to export data from non-distributed NoSQL stores like CouchDB, MongoDB or Tokyo Tyrant as from distributed ones like Projekt Voldemort or Cassandra.

Schmidt's points are humorously and extensively iterated in the talk "Your Guide to NoSQL" (cf. [Ake09]) which especially parodies the NoSQL advocates' argument of treating every querying need in a MapReduce fashion.

2.2.7. Performance vs. Scalability

BJ Clark presents an examination of various NoSQL databases and MySQL regarding performance and scalability in his blog post "NoSQL: If only it was that easy". At first, he defines scalability as "to change the size while maintaining proportions and in CS this usually means to increase throughput." Blogger Dennis Forbes agrees with this notion of scalabilty as "pragmatically the measure of a solution's ability to grow to the highest realistic level of usage in an achievable fashion, while maintaining acceptable service levels" (cf. [For10]). BJ Clark continues: "What scaling isn't: performance. [. . .] In reality, scaling doesn't have anything to do with being fast. It has only to do with size. [. . .] Now, scaling and performance do relate in that typically, if something is performant, it may not actually need to scale." (cf. [Cla09]).

Regarding relational databases Clark states that "The problem with RDBMS isn't that they don't scale, it's that they are incredibly hard to scale. Ssharding[sic!] is the most obvious way to scale things, and sharding multiple tables which can be accessed by any column pretty quickly gets insane." Blogger Dennis Forbes agrees with him that "There are some real scalability concerns with old school relational database systems" (cf. [For10]) but that it is still possible to make them scale using e. g. the techniques described by Adam Wiggins (cf. [Wig09]).

Besides sharding of relational databases and the avoidance of typical mistakes (as expensive joins caused by rigid normalization or poor indexing) Forbes sees vertical scaling as still an option that can be easy, computationally effective and which can lead far with "armies of powerful cores, hundreds of GBs of memory, operating against SAN arrays with ranks and ranks of SSDs". On the downside, vertical scaling can relatively costly as Forbes also admits. But Forbes also argues for horizontal scaling of relational databases by partitioning data and adding each machine to a failover cluster in order to achieve redundancy and availability. Having deployments in large companies in mind where constraints are few and money is often not that critical he states from his own experience[9] that "This sort of scaling that is at the heart of virtually every bank, trading system, energy platform, retailing system, and so on. [. . .] To claim that SQL systems don't scale, in defiance of such *obvious* and *overwhelming* evidence, defies all reason" (cf. [For10]). Forbes argues for the use of own servers as he sees some artificial limits in cloud computing environments such as limitations of IO and relatively high expenditures for single instances in Amazon's EC2; he states that "These financial and artificial limits explain the strong interest in technologies that allows you to spin up and cycle down as needed" (cf. [For10]).

[9] Forbes has worked in the financial, assurance, telecommunication and power supply industry.

BJ Clark continues his blog post by an evaluation of some NoSQL datastores with a focus on automatic scalability (such as via auto-sharding) as he updates millions of objects (primary objects as he calls them) on which other objects depend in 1:1 and 1:n relationships (he calls them secondary objects); secondary objects are mostly inserted at the end of tables.

His evaluation can be summarized in the following way:

- The key/value-stores **Tokyo Tyrant/Cabinet** and **Redis** do not provide means for automatic, horizontal scalability. If a machine is added—similar to memcached—it has to be made known to the application which then (re-)hashes the identifiers of database entries against the collection of database servers to benefit from the additional ressources. On the other hand he states that Tokyo Tyrant/Cabinet and Redis perform extremely well so that the need for scaling horizontally will not appear very early.
- The distributed key/value-store **Project Voldemort** utilizes additional servers added to a cluster automatically and also provides for fault tolerance[10]. As it concentrates on sharding and fault-tolerance and has a pluggable storage architecture, Tokyo Tyrant/Cabinet or Redis can be used as a storage backend for Voldemort. This might also be a migration path from these systems if they need to be scaled.
- The document-database **MongoDB** showed good performance characteristics but did not scale automatically at the time of the evaluation as it did not provide automatic sharding (which has changed in version 1.6 released in August 2010 cf. [Mon10] and [MHC+10b]).
- Regarding the column-database **Cassandra** Clark thinks that it is "definitely supposed to scale, and probably does at Facebook, by simply adding another machine (they will hook up with each other using a gossip protocol), but the OSS version doesn't seem to support some key things, like loosing a machine all together". This conclusion reflects the development state of Cassandra at the time of the evaluation.
- The key/value store **S3** of Amazon scaled very well, although, due to Clark, it is not as performant as other candidates.
- **MySQL**, in comparison, does not provide automatic horizontal scalability by e. g. automatic sharding, but Clark states that for most applications (and even web applications like Friendfeed, cf. [Tay09]) MySQL is fast enough and—in addition—is "familiar and ubiquitous". In comparison to Tokyo Tyrant/Cabinet and Redis Clark concludes that "It can do everything that Tokyo and Redis can do, and it really isn't that much slower. In fact, for some data sets, I've seen MySQL perform ALOT[sic!] faster than Tokyo Tyrant" and "it's just as easy or easier to shard MySQL as it is Tokyo or Redis, and it's hard to argue that they can win on many other points."

The systems mentioned in the above evaluation will discussed in more detail later on in this paper. Besides, it has to be mentioned that the evaluation took place in summer of 2009 (the blog post is of August), therefore results reflect the development state of the evaluated systems at that time.

Clark summarizes his results by indicating that RDBMSs are not harder to scale than "lots of other things" in his perspective, and that only a couple of NoSQL databases provide means for automatic, horizontal scalability allowing to add machines while not requiring operators to interact. Therefore, it could be even argued that "it's just as easy to scale mysql (with sharding via mysql proxy) as it is to shard some of these NoSQL dbs." Therefore he does not proclaim an early death of RDBMSs and reminds that MySQL is still in use at big web sites like Facebook, Wikipedia and Friendfeed. For new applications he suggests to use the tool fitting the job best, reflecting that non-relational databases—just like relational ones—are no "one size fits all" solutions either:

[10] Redis in particular does not offer fault-tolerance and as data is held in memory it will be lost if a server crashes.

> “If I need reporting, I won’t be using any NoSQL. If I need caching, I’ll probably use Tokyo Tyrant. If I need ACIDity, I won’t use NoSQL. If I need a ton of counters, I’ll use Redis. If I need transactions, I’ll use Postgres. If I have a ton of a single type of documents, I’ll probably use Mongo. If I need to write 1 billion objects a day, I’d probably use Voldemort. If I need full text search, I’d probably use Solr. If I need full text search of volatile data, I’d probably use Sphinx.”

2.2.8. Not All RDBMSs Behave like MySQL

An argument often found in the NoSQL debate is that RDBMSs do not scale very well and are difficult to shard. This is often pointed out by the example of MySQL. Furthermore, NoSQL databases are sometimes seen as the successor of a MySQL plus memcached solution (cf. e. g. [Hof10c]) where the latter is taking load away from the database to decrease and defer the need to distribute it. Concerning these typical arguments blogger Dennis Forbes reminds that RDBMSs in general cannot be easily identified with MySQL but others may be easier or better scalable: “MySQL isn’t the vanguard of the RDBMS world. Issues and concerns with it on high load sites have remarkably little relevance to other database systems” (cf. [For10]).

2.2.9. Misconceptions of Critics

NoSQL advocate Ben Scofield responds to some of the criticism mentioned above which he perceives to be misconceived. He does so by expressing incisive arguments from the NoSQL debate responding to them (cf. [Sco09]):

“NoSQL is just about scalability and/or performance.” Scofield argues that this could be an attractive claim for those “traditionalists” (as he calls them) who think that NoSQL data stores can be made obsolete by making RDBMSs faster and more scalable. He claims that “there’s a lot more to NoSQL than just performance and scaling” and that for example “NoSQL DBs often provide better substrates for modeling business domains”.

“NoSQL is just document databases, or key-value stores, or . . . ” Scofield notes that many NoSQL-articles address only document-oriented databases ore key-value stores, sometimes column-stores. He states that “Good arguments can be made against each of those solutions for specific circumstances, but those are arguments against a specific type of storage engine.” Scofield therefore criticizes that narrowing the discussion to only one sort of NoSQL databases allows traditionalists to argue easily against the whole movement or whole set of different NoSQL approaches.

“I can do NoSQL just as well in a relational database.” With the frequent argument of Friendfeed’s usage of MySQL (cf. [Tay09]) in mind, Scofield notes that although it is possible to tweak and tune a relational database this does not make sense for all types of data. “Different applications are good for different things; relational databases are great for relational data, but why would you want to use them for non-relational data?” he asks.

“NoSQL is a wholesale rejection of relational databases.” Some time ago this claim was often heard in the NoSQL community but it became less common which Scofield appreciates that: “It seems that we’re moving towards a pluralistic approach to storing our data, and that’s a good thing. I’ve suggested ’polyglot persistence’ for this approach (though I didn’t coin the term), but I also like Ezra Zygmuntowicz’s ’LessSQL’ as a label, too.”

2.3. Classifications and Comparisons of NoSQL Databases

In the last years a variety of NoSQL databases has been developed mainly by practitioners and web companies to fit their specific requirements regarding scalability performance, maintainance and feature-set. As it has been revealed some of these databases have taken up ideas from either Amazon's Dynamo (cf. [DHJ+07]) or Google's Bigtable (cf. [CDG+06]) or a combination of both. Others have ported ideas in existing databases towards modern web technologies such as CouchDB. Still others have pursued totally different approaches like Neo4j or HypergraphDB.

Because of the variety of these approaches and overlappings regarding the nonfunctional requirements and the feature-set it could be difficult to get and maintain an overview of the nonrelational database scene. So there have been various approaches to classify and subsume NoSQL databases, each with different categories and subcategories. Some classification approaches shall be presented here out of which one possibility to classify NoSQL datastores will be pursued in the subsequent chapters.

2.3.1. Taxonomies by Data Model

Concerning the classification of NoSQL stores Highscalability author Todd Hoff cites a presentation by Stephen Yen in his blog post "A yes for a NoSQL taxonomy" (cf. [Hof09c]). In the presentation "NoSQL is a Horseless Carriage" (cf. [Yen09]) Yen suggests a taxononmy that can be found in table 2.1.

Term	**Matching Databases**
Key-Value-Cache	Memcached Repcached Coherence Infinispan EXtreme Scale Jboss Cache Velocity Terracoqa
Key-Value-Store	keyspace Flare Schema Free RAMCloud
Eventually-Consistent Key-Value-Store	Dynamo Voldemort Dynomite SubRecord Mo8onDb Dovetaildb
Ordered-Key-Value-Store	Tokyo Tyrant Lightcloud NMDB Luxio MemcacheDB Actord

Term	**Matching Databases**
Data-Structures Server	Redis
Tuple Store	Gigaspaces Coord Apache River
Object Database	ZopeDB DB4O Shoal
Document Store	CouchDB Mongo Jackrabbit XML Databases ThruDB CloudKit Perservere Riak Basho Scalaris
Wide Columnar Store	Bigtable Hbase Cassandra Hypertable KAI OpenNeptune Qbase KDI

Table 2.1.: Classifications – NoSQL Taxonomy by Stephen Yen (cf. [Yen09])

A similar taxonomy which is less fine-grained and comprehensive than the classification above can be found in the article "Databases in the cloud" by Ken North (cf. [Nor09]). Table 2.2 summarizes his classification of datastores that additionally include some datastores available in cloud-computing environments only.

Category	**Matching databases**
Distributed Hash Table, Key-Value Data Stores	memcached MemcacheDB Project Voldemort Scalaris Tokyo Cabinet
Entity-Attribute-Value Datastores	Amazon SimpleDB Google AppEngine datastore Microsoft SQL Data Services Google Bigtable Hadoop HyperTable HBase

Category	Matching databases
Amazon Platform	Amazon SimpleDB
Document Stores, Column Stores	Sybase IQ Vertica Analytic Database Apache CouchDB

Table 2.2.: Classifications – Categorization by Ken North (cf. [Nor09]

Similarly to the classifications mentioned above Rick Cattel subsumes different NoSQL databases primarily by their data model (cf. [Cat10]) as shown in table 2.3.

Category	Matching databases
Key-value Stores	Redis Scalaris Tokyo Tyrant Voldemort Riak
Document Stores	SimpleDB CouchDB MongoDB Terrastore
Extensible Record Stores	Bigtable HBase HyperTable Cassandra

Table 2.3.: Classifications – Categorization by Rick Cattell (cf. [Cat10]

2.3.2. Categorization by Ben Scofield

Blogger Alex Popescu summarizes a presentation by Ben Scofield who gave a generic introduction to NoSQL databases along with a categorization and some ruby examples of different NoSQL databases (cf. [Sco10]). The categorization is in fact a short comparison of classes of NoSQL databases by some nonfunctional categories (“(il)ities“) plus a rating of their feature coverage. Popescu summarizes Scofield’s ideas as presented in table 2.4.

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

Table 2.4.: Classifications – Categorization and Comparison by Scofield and Popescu (cf. [Pop10b], [Sco10])

2.3.3. Comparison by Scalability, Data and Query Model, Persistence-Design

In his blog post "NoSQL ecosystem" Jonathan Ellis discusses NoSQL datastores by three important aspects:

1. Scalability
2. Data and query model
3. Persistence design

Scalability

Ellis argues that it is easy to scale read operations by replication of data and load distribution among those replicas. Therefore, he only investigates the scaling of write operations in databases that are really distributed and offer automatic data partitioning. When data size exceeds the capabilities of a single machine the latter systems seem to be the only option to him if provided no will to partition data manually (which is not a good idea according to [Oba09b]). For the relevant distributed systems which provide auto-sharding Ellis sees two important features:

- Support for multiple datacenters
- Possibility to add machines live to an existing cluster, transparent for applications using this cluster

By these features Ellis compares a selection of NoSQL datastores fulfilling the requirements of real distribution and auto-sharding (cf. table 2.5).

Datastore	Add Machines Live	Multi-Datacenter Support
Cassandra	x	x
HBase	x	
Riak	x	
Scalaris	x	
Voldemort		Some code required

Table 2.5.: Classifications – Comparison of Scalability Features (cf. [Ell09a])

The following NoSQL stores have been excluded in this comparison as they are not distributed in the way Ellis requires (at the time of his blog post in November 2009): CouchDB, MongoDB, Neo4j, Redis and Tokyo Cabinet. Nonetheless, these systems can find use as a persistence layer for distributed systems, according to him. The document databases MongoDB and CouchDB supported limited support for auto-sharding at the time of his investigation (MongoDB out of the box, CouchDB by partitioning/clustering framework Lounge). Regarding Tokyo Cabinet, Ellis notices that it can be used as a storage backend for Project Voldemort.

Data and Query Model

The second area Ellis examines is the data model and the query API offered by different NoSQL stores. Table 2.6 shows the results of his investigation pointing out a great variety of data models and query APIs.

Datastore	**Data Model**	**Query API**
Cassandra	Columnfamily	Thrift
CouchDB	Document	map/reduce views
HBase	Columnfamily	Thrift, REST
MongoDB	Document	Cursor
Neo4j	Graph	Graph
Redis	Collection	Collection
Riak	Document	Nested hashes
Scalaris	Key/value	get/put
Tokyo Cabinet	Key/value	get/put
Voldemort	Key/value	get/put

Table 2.6.: Classifications – Comparison of Data Model and Query API (cf. [Ell09a])

Ellis makes the following remarks concerning the data models and query APIs of these systems:

- The **columnfamily model**, implemented by Cassandra and HBase is inspired by the corresponding paragraph in the second section of Google's Bigtable paper (cf. [CDG$^+$06, Page 2]). Cassandra omits historical versions in contrast to Bigtable and introduces the further concept of supercolumns. In Cassandra as well as HBase rows are sparse[11], which means that they may have different numbers of cells and columns do not have to be defined in advance.
- The **key/value model** is easiest to implement but may be inefficient if one is only interested in requesting or updating part of the value associated with a certain key. Furthermore, it is difficult to build complex data structures on top of key/value stores (as Ellis describes in another blog post, cf. [Ell09b]).

[11] See e. g. the Wikipedia-article on Sparse arrays

- Ellis sees **document databases** being the next step from key/value stores as they allow nested values. They permit to query data structures more efficiently than key/value stores as they do not necessarily reply whole BLOBs when requesting a key.
- The **graph database** Neo4j has a unique data model in Ellis' selection as objects and their relationships are modelled and persisted as nodes and edges of a graph. Queries fitting this model well might be three orders faster than corresponding queries in the other stores discussed here (due to Emil Eifrem, CEO of the company behind Neo4j, cf. [Eif09]).
- **Scalaris** is unique among the selected key/value stores as it allows distributed transactions over multiple keys.

Persistence Design

The third aspect by which Ellis compares his selection of NoSQL datastores is the way they store data (see table 2.7).

Datastore	**Persistence Design**
Cassandra	Memtable / SSTable
CouchDB	Append-only B-tree
HBase	Memtable / SSTable on HDFS
MongoDB	B-tree
Neo4j	On-disk linked lists
Redis	In-memory with background snapshots
Riak	?
Scalaris	In-memory only
Tokyo Cabinet	Hash or B-tree
Voldemort	Pluggable (primarily BDB MySQL)

Table 2.7.: Classifications – Comparison of Persistence Design (cf. [Ell09a])

Ellis considers persistence design as particularly important to estimate under which workloads these databases will perform well:

In-Memory Databases are very fast (e. g. Redis can reach over 100.000 operations per second on a single machine) but the data size is inherently limited by the size of RAM. Another downside is that durability may become a problem as the amount of data which can get lost between subsequent disk flushes (e. g. due to server crashes or by losing power supply) is potentially large. Scalaris addresses this issue by replication but—as it does not support multiple datacenters—threats like power supply failures remain.

Memtables and SSTables work in the following way: write operations are buffered in memory (in a Memtable) after they have been written to an append-only commit log to ensure durability. After a certain amount of writes the Memtable gets flushed to disk as a whole (and is called SSTable then; these ideas are taken from Google's Bigtable paper, cf. [CDG+06, Sections 5.3 and 5.4]). This persistence strategy has performance characteristics comparable to those of in-memory-databases (as—compared to disk-based strategies—disk seeks are reduced due to the append-only log and the flushing of whole Memtables to disk) but avoids the durability difficulties of pure in-memory-databases.

B-trees have been used in databases since their beginning to provide a robust indexing-support. Their performance characteristics on rotating disks are not very positive due to the amount of disk seeks needed for read and write operations. The document-database CouchDB uses B-trees internally but tries to avoid the overhead of disk seeks by only appending to B-trees, which implies the downside that only one write operation at a time can happen as concurrent reads to different sections of a B-tree are not allowed in this case.

Categorization Based on Customer Needs

Todd Hoff (cf. [Hof09b]) cites blogger James Hamilton who presents a different approach to classify NoSQL datastores (cf. [Ham09]) subsuming databases by customer requirements:

Features-First This class of databases provides a (large) number of high level features that make the programmer's job easier. On the downside, they are difficult to scale. Examples include: *Oracle*, *Microsoft SQL Server*, *IBM DB2*, *MySQL*, *PostgreSQL*, *Amazon RDS*[12].

Scale-First This sort of databases has to scale from the start. On the downside, they lack particular features and put responsibility back to the programmer. Examples incude: *Project Voldemort*, *Ringo*, *Amazon SimpleDB*, *Kai*, *Dynomite*, *Yahoo PNUTS*, *ThruDB*, *Hypertable*, *CouchDB*, *Cassandra*, *MemcacheDB*.

Simple Structure Storage This class subsumes key/value-stores with an emphasis on storing and retrieving sets of arbitrary structure. The downside according to Hamilton is that "they generally don't have the features or the scalability of other systems". Examples include: *file systems*, *Cassandra*, *BerkelyDB*, *Amazon SimpleDB*.

Purpose-Optimized Storage These are databases which are designed and built to be good at one thing, e. g. data warehousing or stream processing. Examples of such databases are: *StreamBase*, *Vertica*, *VoltDB*, *Aster Data*, *Netezza*, *Greenplum*.

Hamilton and Hoff consider this categorization useful to match a given use case to a class of databases. Though, the categorization is not complete as e. g. graph-databases are missing and it is not clear how they would fit into the four categories mentioned above. Furthermore, it is important to notice that some databases can be found in different classes (Cassandra, SimpleDB) so the categorization does not provide a sharp distinction where each database only fits into one class (which is—in the authors opinion—at least difficult if not impossible in the field of NoSQL databases).

2.3.4. Classification in this Paper

After considering basic concepts and techniques of NoSQL databases in the next chapter, the following chapters will study various classes of nonrelational datastores and also take a look on particular products. These products are classified by their datastructure, following the taxonomies of Yen, North and Cattell mentioned above (see 2.3.1 on page 23) which is also shared by most other sources of this paper. The classes used in this writing will be: key/value stores, document databases, and column-oriented databases.

[12] Relational Database Service providing cocooned MySQL instances in Amazon's cloud services.

3. Basic Concepts, Techniques and Patterns

This chapter outlines some fundamental concepts, techniques and patterns that are common among NoSQL datastores and not unique to only one class of nonrelational databases or a single NoSQL store. Specific concepts and techniques of the various classes of NoSQL datastores and individual products will be discussed in the subsequent chapters of this paper.

3.1. Consistency

3.1.1. The CAP-Theorem

In a keynote titled "Towards Robust Distributed Systems" at ACM's PODC[1] symposium in 2000 Eric Brewer came up with the so called CAP-theorem (cf. [Bre00]) which is widely adopted today by large web companies (e. g. Amazon, cf. [Vog07], [Vog08]) as well as in the NoSQL community. The CAP acronym stands for (as summarized by Gray in [Gra09]):

Consistency meaning if and how a system is in a consistent state after the execution of an operation. A distributed system is typically considered to be consistent if after an update operation of some writer all readers see his updates in some shared data source. (Nevertheless there are several alternatives towards this strict notion of consistency as we will see below.)

Availability and especially high availability meaning that a system is designed and implemented in a way that allows it to continue operation (i. e. allowing read and write operations) if e. g. nodes in a cluster crash or some hardware or software parts are down due to upgrades.

Partition Tolerance understood as the ability of the system to continue operation in the presence of network partitions. These occur if two or more "islands" of network nodes arise which (temporarily or permanently) cannot connect to each other. Some people also understand partition tolerance as the ability of a system to cope with the dynamic addition and removal of nodes (e. g. for maintainance purposes; removed and again added nodes are considered an own network partition in this notion; cf. [Ipp09]).

Now, Brewer alleges that one can at most choose two of these three characteristics in a "shared-data system" (cf. [Bre00, slide 14]). In his talk, he referred to trade-offs between ACID and BASE systems (see next subsection) and proposed as a decision criteria to select one or the other for individual use-cases: if a system or parts of a system have to be consistent and partition-tolerant, ACID properties are required and if availability and partition-tolerance are favored over consistency, the resulting system can be characterized by the BASE properties. The latter is the case for Amazon's Dynamo (cf. [DHJ$^+$07]), which is available and partition-tolerant but not strictly consistent, i. e. writes of one client are not seen immediately after being committed to all readers. Google's Bigtable chooses neither ACID nor BASE but the third CAP-alternative being a consistent and available system and consequently not able to fully operate in the presence of network partitions. In his keynote Brewer points out traits and examples of the three different choices that can be made according to his CAP-theorem (see table 3.1).

[1] Principles of Distributed Computing

Choice	Traits	Examples
Consistence + Availability (Forfeit Partitions)	2-phase-commit cache-validation protocols	Single-site databases Cluster databases LDAP xFS file system
Consistency + Partition tolerance (Forfeit Availability)	Pessimistic locking Make minority partitions unavailable	Distributed databases Distributed locking Majority protocols
Availability + Partition tolerance (Forfeit Consistency)	expirations/leases conflict resolution optimistic	Coda Web cachinge[sic!] DNS

Table 3.1.: CAP-Theorem – Alternatives, Traits, Examples (cf. [Bre00, slides 14–16])

With regards to databases, Brewer concludes that current "Databases [are] better at C[onsistency] than Availability" and that "Wide-area databases can't have both" (cf. [Bre00, slide 17])—a notion that is widely adopted in the NoSQL community and has influenced the design of nonrelational datastores.

3.1.2. ACID vs. BASE

The internet with its wikis, blogs, social networks etc. creates an enormous and constantly growing amount of data needing to be processed, analyzed and delivered. Companies, organizations and individuals offering applications or services in this field have to determine their individual requirements regarding performance, reliability, availability, consistency and durability (cf. [Gra09]). As discussed above, the CAP-theorem states that a choice can only be made for two options out of consistency, availability and partition tolerance. For a growing number of applications and use-cases (including web applications, especially in large and ultra-large scale, and even in the e-commerce sector, see [Vog07], [Vog08]) availability and partition tolerance are more important than strict consistency. These applications have to be reliable which implicates availability and redundancy (consequently distribution among two or more nodes, which is necessary as many systems run on "cheap, commoditized and unreliable" machines [Ho09a] and also provides scalability). These properties are difficult to achieve with ACID properties therefore approaches like BASE are applied (cf. [Ipp09]).

The BASE approach according to Brewer forfeits the ACID properties of consistency and isolation in favor of "availability, graceful degradation, and performance" (cf. [Bre00, slide 12]). The acronym BASE is composed of the following characteristics:

- **B**asically **a**vailable
- **S**oft-state
- **E**ventual consistency

Brewer contrasts ACID with BASE as illustrated in table 3.2, nonetheless considering the two concepts as a spectrum instead of alternatives excluding each other. Ippolito summarizes the BASE properties in the following way: an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known-state state eventually (eventual consistency, cf. [Ipp09]).

ACID	BASE
Strong consistency	Weak consistency – stale data OK
Isolation	Availability first
Focus on “commit”	Best effort
Nested transactions	Approximate answers OK
Availability?	Aggressive (optimistic)
Conservative (pessimistic)	Simpler!
Difficult evolution (e. g. schema)	Faster
	Easier evolution

Table 3.2.: ACID vs. BASE (cf. [Bre00, slide 13])

In his talk “Design Patterns for Distributed Non-Relational Databases” Todd Lipcon contrasts strict and eventual consistency. He defines that a consistency model—such as strict or eventual consistency—“determines rules for **visibility** and **apparent order** of updates”. Lipcon, like Brewer, refers to consistency as “a continuum with tradeoffs” and describes strict and eventual consistency as follows (cf. [Lip09, slides 14–16]):

Strict Consistency according to Lipcon means that “All read operations must return data from the latest completed write operation, regardless of which replica the operations went to”. This implies that either read and write operations for a given dataset have to be executed on the same node[2] or that strict consistency is assured by a distributed transaction protocol (like two-phase-commit or Paxos). As we have seen above, such a strict consistency cannot be achieved together with availability and partition tolerance according to the CAP-theorem.

Eventual Consistency means that readers will see writes, as time goes on: “In a steady state, the system will eventually return the last written value”. Clients therefore may face an inconsistent state of data as updates are in progress. For instance, in a replicated database updates may go to one node which replicates the latest version to all other nodes that contain a replica of the modified dataset so that the replica nodes eventually will have the latest version.

Lipcon and Ho point out that an eventually consistent system may provide more differentiated, additional guarantees to its clients (cf. [Lip09, slide 16], [Ho09a]):

Read Your Own Writes (RYOW) Consistency signifies that a client sees his updates immediately after they have been issued and completed, regardless if he wrote to one server and in the following reads from different servers. Updates by other clients are not visible to him instantly.

Session Consistency means read your own writes consistency which is limited to a session scope (usually bound to one server), so a client sees his updates immediately only if read requests after an update are issued in the same session scope.

Casual Consistency expresses that if one client reads version x and subsequently writes version y, any client reading version y will also see version x.

Monotonic Read Consistency provides the time monotonicity guarantee that clients will only see more updated versions of the data in future requests.

[2] In such a scenario there can be further (slave) nodes to which datasets are replicated for availability purposes. But this replications cannot be done asynchronously as data on the replica nodes has to be up to date instantaneously.

Ho comments that eventual consistency is useful if concurrent updates of the same partitions of data are unlikely and if clients do not immediately depend on reading updates issued by themselves or by other clients (cf. [Ho09a]). He furthermore notes that the consistency model chosen for a system (or parts of the system) implicates how client requests are dispatched to replicas as well as how replicas propagate and apply updates.

3.1.3. Versioning of Datasets in Distributed Scenarios

If datasets are distributed among nodes, they can be read and altered on each node and no strict consistency is ensured by distributed transaction protocols, questions arise on how "concurrent" modifications and versions are processed and to which values a dataset will eventually converge to. There are several options to handle these issues:

Timestamps seem to be an obvious solution for developing a chronological order. However, timestamps "rely on synchronized clocks and don't capture causality" as Lipcon points out (cf. [Lip09, slide 17]; for a more fundamental and thorough discussion on these issues cf. [Mat89]).

Optimistic Locking implies that a unique counter or clock value is saved for each piece of data. When a client tries to update a dataset it has to provide the counter/clock-value of the revision it likes to update (cf. [K+10b]). As a downside of this procedure, the Project Voldemort development team notes that it does not work well in a distributed and dynamic scenario where servers show up and go away often and without prior notice. To allow casuality reasoning on versions (e. g. which revision is considered the most recent by a node on which an update was issued) a lot of history has to be saved, maintained and processed as the optimistic locking scheme needs a total order of version numbers to make casuality reasoning possible. Such a total order easily gets disrupted in a distributed and dynamic setting of nodes, the Project Voldemort team argues.

Vector Clocks are an alternative approach to capture order and allow reasoning between updates in a distributed system. They are explained in more detail below.

Multiversion Storage means to store a timestamp for each table cell. These timestamps "don't necessarily need to correspond to real life", but can also be some artificial values that can be brought into a definite order. For a given row multiple versions can exist concurrently. Besides the most recent version a reader may also request the "most recent before T" version. This provides "optimistic concurrency control with compare-and-swap on timestamps" and also allows to take snapshots of datasets (cf. [Lip09, slide 20]).

Vector Clocks

A vector clock is defined as a tuple $V[0], V[1], ..., V[n]$ of clock values from each node (cf. [Lip09, slide 18]). In a distributed scenario node i maintains such a tuple of clock values, which represent the state of itself and the other (replica) nodes' state as it is aware about at a given time ($V_i[0]$ for the clock value of the first node, $V_i[1]$ for the clock value of the second node, ... $V_i[i]$ for itself, ... $V_i[n]$ for the clock value of the last node). Clock values may be real timestamps derived from a node's local clock, version/revision numbers or some other ordinal values.

As an example, the vector clock on node number 2 may take on the following values:

$$V_2[0] = 45, V_2[1] = 3, V_2[2] = 55$$

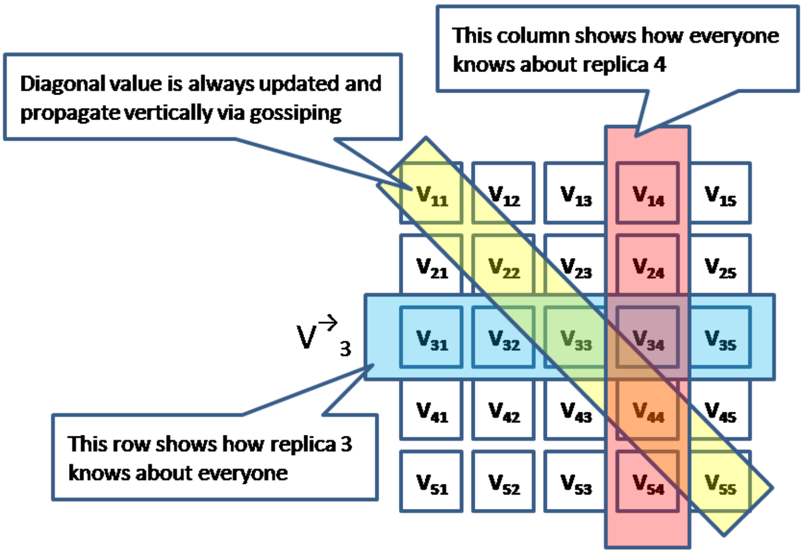

Diagonal value is always updated and propagate vertically via gossiping
This column shows how everyone knows about replica 4
This row shows how replica 3 knows about everyone
$V^{\rightarrow}_3$
V11
V12
V13
V14
V15
V21
V22
V23
V24
V25
V31
V32
V33
V34
V35
V41
V42
V43
V44
V45
V51
V52
V53
V54
V55

Figure 3.1.: Vector Clocks (taken from [Ho09a])

This reflects that from the perspective of the second node, the following updates occurred to the dataset the vector clock refers to: an update on node 1 produced revision 3, an update on node 0 lead to revision 45 and the most recent update is encountered on node 2 itself which produced revision 55.

Vector clocks are updated in a way defined by the following rules (cf. [Ho09a], [K+10b]):

- If an internal operation happens at node i, this node will increment its clock $V_i[i]$. This means that internal updates are seen immediately by the executing node.
- If node i sends a message to node k, it first advances its own clock value $V_i[i]$ and attaches the vector clock V_i to the message to node k. Thereby, he tells the receiving node about his internal state and his view of the other nodes at the time the message is sent.
- If node i receives a message from node j, it first advances its vector clock $V_i[i]$ and then merges its own vector clock with the vector clock $V_{message}$ attached to the message from node j so that:

$$V_i = max(V_i, V_{message})$$

To compare two vector clocks V_i and V_j in order to derive a partial ordering, the following rule is applied:

$$V_i > V_j, if\ \forall k\ V_i[k] > V_j[k]$$

If neither $V_i > V_j$ nor $V_i < V_j$ applies, a conflict caused by concurrent updates has occurred and needs to be resolved by e. g. a client application.

As seen, vector clocks can be utilized "to resolve consistency between writes on multiple replicas" (cf. [Lip09, slide 18]) because they allow casual reasoning between updates. Replica nodes do typically not maintain a vector clock for clients but clients participate in the vector clock scenario in such a way that they keep a vector clock of the last replica node they have talked to and use this vector clock depending on the client consistency model that is required; e. g. for monotonic read consistency a client attaches this last vector clock it received to requests and the contacted replica node makes sure that the vector clock of its response is greater than the vector clock the client submitted. This means that the client can be sure to see only newer versions of some piece of data ("newer" compared to the versions it has already seen; cf. [Ho09a]).

Compared to the alternative approaches mentioned above (timestamps, optimistic locking with revision numbers, multiversion storage) the advantages of vector clocks are:

- No dependence on synchronized clocks
- No total ordering of revision numbers required for casual reasoning
- No need to store and maintain multiple revisions of a piece of data on all nodes

Vector Clocks Utilized to Propagate State via Gossip

Blogger Ricky Ho gives an overview about how vector clocks can be utilized to handle consistency, conflicting versions and also transfer state between resplicas in a partitioned database (as it will be discussed in the next section). He describes the transfer of vector clocks between clients and database nodes as well as among the latter over the Gossip protocol which can be operated in either a *state* or an *operation transfer model* to handle read and update operations as well as replication among database nodes (cf. [Ho09a]). Regarding the internode propagation of state via Gossip Todd Lipcon points out that this method provides scalability and avoids a single point of failure (SPOF). However, as the information about state in a cluster of n nodes needs $O(log\ n)$ rounds to spread, only eventual consistency can be achieved (cf. [Lip09, slide 34]).

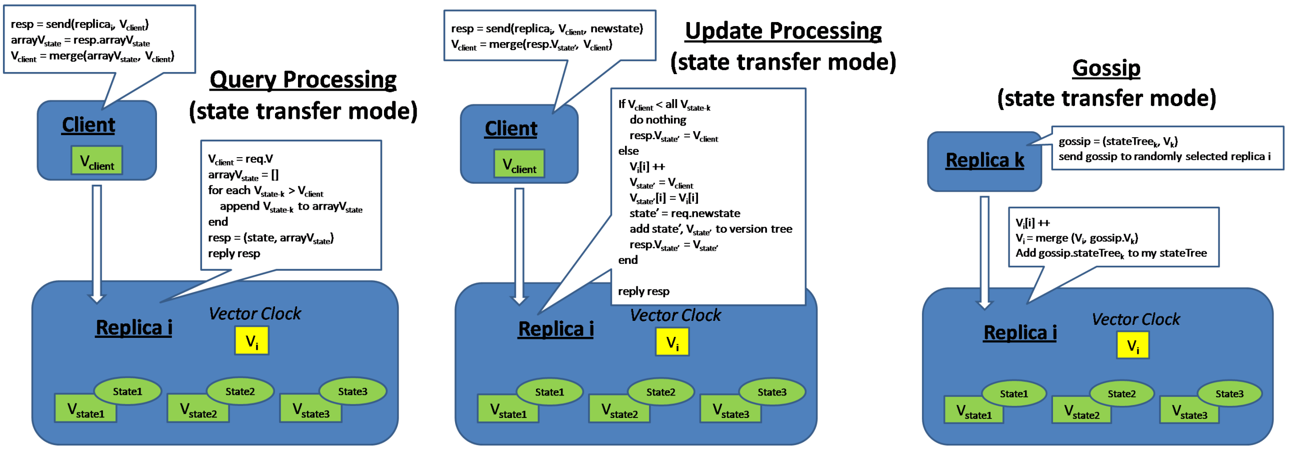
State Transfer Model In a state transfer model data or deltas of data are exchanged between clients and servers as well as among servers. In this model database server nodes maintain vector clocks for their data and also state version trees for conflicting versions (i. e. versions where the corresponding vector clocks cannot be brought into a $V_A < V_B$ or $V_A > V_B$ relation); clients also maintain vector clocks for pieces of data they have already requested or updated. These are exchanged and processed in the following manner:

Query Processing When a client queries for data it sends its vector clock of the requested data along with the request. The database server node responds with part of his state tree for the piece of data that *precedes* the vector clock attached to the client request (in order to guarantee monotonic read consistency) and the server's vector clock. Next, the client advances his vector clock by merging it with the server node's vector clock that had been attached to the servers response. For this step, the client also has to resolve potential version conflicts; Ho notes that this is necessary because if the client did not resolve conflicts at read time, it may be the case that he operates on and maybe submits updates for an outdated revision of data compared to the revision the contacted replica node maintains.

Update Processing Like in the case of read requests clients also have to attach their vector clock of the data to be updated along with the update request. The contacted replica server then checks, if the client's state according to the transmitted vector clock precedes the current server state and if so omits the update request (as the client has to acquire the latest version and also fix version conflicts at read time—as described in the former paragraph); if the client's transmitted vector clock is greater than its own the server executes the update request.

Internode Gossiping Replicas responsible for the same partitions of data exchange their vector clocks and version trees in the background and try to merge them in order to keep synchronized.

Figure 3.2 depicts the messages exchanged via Gossip in the state transfer model and how they are processed by receivers in the cases just described.



```
resp = send(replica$_i$, V$_{client}$)
arrayV$_{state}$ = resp.arrayV$_{state}$
V$_{client}$ = merge(arrayV$_{state}$, V$_{client}$)
```

```
V$_{client}$ = req.V
arrayV$_{state}$ = []
for each V$_{state-k}$ > V$_{client}$
   append V$_{state-k}$ to arrayV$_{state}$
end
resp = (state, arrayV$_{state}$)
reply resp
```

```
resp = send(replica$_i$, V$_{client}$, newstate)
V$_{client}$ = merge(resp.V$_{state'}$, V$_{client}$)
```

```
If V$_{client}$ < all V$_{state-k}$
   do nothing
   resp.V$_{state'}$ = V$_{client}$
else
   V$_i$[i] ++
   V$_{state'}$ = V$_{client}$
   V$_{state'}$[i] = V$_i$[i]
   state' = req.newstate
   add state', V$_{state'}$ to version tree
   resp.V$_{state'}$ = V$_{state'}$
end

reply resp
```

```
gossip = (stateTree$_k$, V$_k$)
send gossip to randomly selected replica i
```

```
V$_i$[i] ++
V$_i$ = merge (V$_i$, gossip.V$_k$)
Add gossip.stateTree$_k$ to my stateTree
```

Figure 3.2.: Vector Clocks – Exchange via Gossip in State Transfer Mode (taken from [Ho09a])

Operation Transfer Model In contrast to the state transfer model discussed above, operations applicable to locally maintained data are communicated among nodes in the operation transfer model. An obvious advantage of this procedure is that lesser bandwidth is consumed to interchange operations in contrast to actual data or deltas of data. Within the operation transfer model, it is of particular importance to apply the operations in correct order on each node. Hence, a replica node first has to determine a casual relationship between operations (by vector clock comparison) before applying them to its data. Secondly, it has to defer the application of operations until all preceding operations have been executed; this implies to maintain a queue of deferred operations which has to be exchanged and consolidated with those of other replicas (as we will see below). In this model a replica node maintains the following vector clocks (cf. [Ho09a]):

- V_{state}: The vector clock corresponding to the last updated state of its data.
- V_i: A vector clock for itself where—compared to V_{state}—merges with received vector clocks may already have happened.
- V_j: A vector clock received by the last gossip message of replica node j (for each of those replica nodes)

In the operation transfer model the exchange and processing vector clocks in read-, update- and internode-messages is as follows:

Query Processing In a read request a client again attaches his vector clock. The contacted replica node determines whether it has a view causally following the one submitted in the client request. It responds with the latest view of the state i. e. either the state corresponding to the vector clock attached by the client or the one corresponding to a casually succeeding vector clock of the node itself.

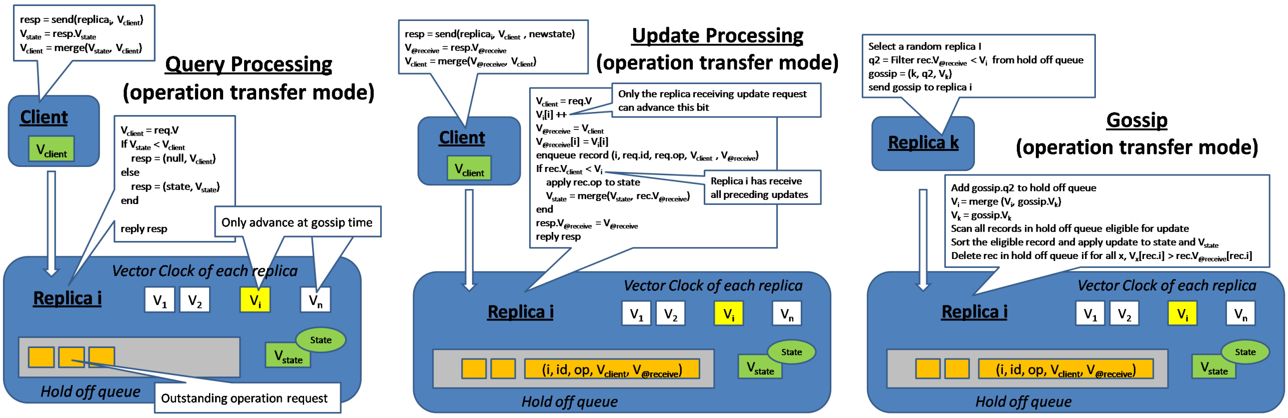
Update Processing When a client submits an update request the contacted replica node buffers this update operation until it can be applied to its local state taking into account the three vector clocks it maintains (see above) and the queue of already buffered operations. The buffered update operation gets tagged with two vector clocks: V_{client} representing the clients view when submitting the update, and $V_{received}$ representing the replica node's view when it received the update (i. e. the vector clock V_i mentioned above). When all other operations casually preceding the received update operation have arrived and been applied the update requested by the client can be executed.

Internode Gossiping In the operation transfer model replica nodes exchange their queues of pending operations and update each other's vector clocks. Whenever nodes have exchanged their operation-queues they decide whether certain operations in these queues can be applied (as all dependent operations have been received). If more than one operation can be executed at a time, the replica node brings these operations into the correct (casual) order reasoned by the vector clocks attached to these operations and applies them to its local state of data. In case of concurrent updates on different replicas—resulting in multiple sequences of operations valid at a time—there has to be a distinction between the following cases:

1. The concurrent updates are commutative, so their order of application does not matter.
2. The concurrent updates are not commutative. In this case partial ordering reasoned from vector clock comparison is not sufficient but a total order has to be determined. To achieve this, Ho mentions the introcuction of a global sequence number obtained from a central counter: "whoever do[sic!] the update first acquire a monotonic sequence number and late comers follow the sequence" (cf. [Ho09a]).

When operations have been applied locally on a replica they can only be removed from the node's operation queue after all other replica nodes have been informed about the execution of the update.

Figure 3.3 shows the messages exchanged and processed in operational transfer model via Gossip in the case of queries, updates and internode communication.


Query Processing
(operation transfer mode)
Client
V_client
Only advance at gossip time
Vector Clock of each replica
Replica i
V_1
V_2
V_i
V_n
State
V_state
Hold off queue
Outstanding operation request
Update Processing
(operation transfer mode)
Client
V_client
Only the replica receiving update request can advance this bit
Replica i has receive all preceding updates
Vector Clock of each replica
Replica i
V_1
V_2
V_i
V_n
(i, id, op, V_client, V_@receive)
State
V_state
Hold off queue
Gossip
(operation transfer mode)
Replica k
Vector Clock of each replica
Replica i
V_1
V_2
V_i
V_n
(i, id, op, V_client, V_@receive)
State
V_state
Hold off queue

```
resp = send(replica_i, V_client)
V_state = resp.V_state
V_client = merge(V_state, V_client)
```

```
V_client = req.V
If V_state < V_client
    resp = (null, V_client)
else
    resp = (state, V_state)
end

reply resp
```

```
resp = send(replica_i, V_client , newstate)
V_@receive = resp.V_@receive
V_client = merge(V_@receive, V_client)
```

```
V_client = req.V
V_i[i] ++
V_@receive = V_client
V_@receive[i] = V_i[i]
enqueue record (i, req.id, req.op, V_client , V_@receive)
If rec.V_client < V_i
    apply rec.op to state
    V_state = merge(V_state, rec.V_@receive)
end
resp.V_@receive = V_@receive
reply resp
```

```
Select a random replica I
q2 = Filter rec.V_@receive < V_i  from hold off queue
gossip = (k, q2, V_k)
send gossip to replica i
```

```
Add gossip.q2 to hold off queue
V_i = merge (V_i, gossip.V_k)
V_k = gossip.V_k
Scan all records in hold off queue eligible for update
Sort the eligible record and apply update to state and V_state
Delete rec in hold off queue if for all x, V_x[rec.i] > rec.V_@receive[rec.i]
```

Figure 3.3.: Vector Clocks – Exchange via Gossip in Operation Transfer Mode (taken from [Ho09a])

3.2. Partitioning

Assuming that data in large scale systems exceeds the capacity of a single machine and should also be replicated to ensure reliability and allow scaling measures such as load-balancing, ways of partitioning the data of such a system have to be thought about. Depending on the size of the system and the other factors like dynamism (e. g. how often and dynamically storage nodes may join and leave) there are different approaches to this issue:

Memory Caches like memcached (cf. [F$^+$10a], [F$^+$10b]) can be seen as partitioned—though transient—in-memory databases as they replicate most frequently requested parts of a database to main memory, rapidly deliver this data to clients and therefore disburden database servers significantly. In the case of memcached the memory cache consists of an array of processes with an assigned amount of memory that can be launched on several machines in a network and are made known to an application via configuration. The memcached protocol (cf. [F$^+$10c]) whose implementation is available in different programming languages (cf. [F+09]) to be used in client applications provides a simple key-/value-store API[3]. It stores objects placed under a key into the cache by hashing that key against the configured memcached-instances. If a memcached process does not respond, most API-implementations ignore the non-answering node and use the responding nodes instead which leads to an implicit rehashing of cache-objects as part of them gets hashed to a different memcached-server after a cache-miss; when the formerly non-answering node joins the memcached server array again, keys for part of the data get hashed to it again after a cache miss and objects now dedicated to that node will implicitly leave the memory of some other memcached server they have been hashed to while the node was down (as memcached applies a LRU[4] strategy for cache cleaning and additionally allows it to specify timeouts for cache-objects). Other implementations of memory caches are available e. g. for application servers like JBoss (cf. [JBo10b]).

Clustering of database servers is another approach to partition data which strives for transparency towards clients who should not notice talking to a cluster of database servers instead of a single server. While this approach can help to scale the persistence layer of a system to a certain degree many criticize that clustering features have only been added on top of DBMSs that were not originally designed for distribution (cf. the comments of Stonebraker et al. in subsection 2.1.2).

Separating Reads from Writes means to specify one or more dedicated servers, write-operations for all or parts of the data are routed to (master(s)), as well as a number of replica-servers satisfying read-requests (slaves). If the master replicates to its clients asynchronously there are no write lags but if the master crashes before completing replication to at least one client the write-operation is lost; if the master replicates writes synchronously to one slave lags the update does not get lost, but read request cannot go to any slave if strict consistency is required and furthermore write lags cannot be avoided (cf. [Ho09a] and regarding write lags StudiVz's Dennis Bemmann's comments [Bem10]). In the latter case, if the master crashes the slave with the most recent version of data can be elected as the new master. The master-/slave-model works well if the read/write ratio is high. The replication of data can happen either by transfer of state (i. e. copying of the recent version of data or delta towards the former version) or by transfer of operations which are applied to the state on the slaves nodes and have to arrive in the correct order (cf. [Ho09a]).

Sharding means to partition the data in such a way that data typically requested and updated together resides on the same node and that load and storage volume is roughly even distributed among the servers (in relation to their storage volume and processing power; as an example confer the experiences of Bemmann with a large German social-network on that topic [Bem10]). Data shards may also be replicated for reasons of reliability and load-balancing and it may be either allowed to write to a dedicated replica only or to all replicas maintaining a partition of the data. To allow such a sharding scenario there has to be a mapping between data partitions (shards) and storage nodes that are responsible for these shards. This mapping can be static or dynamic, determined by a client application, by some dedicated "mapping-service/component" or by some network infrastructure between the client application and the storage nodes. The downside of sharding scenarios is that joins between data shards are not possible, so that the client application or proxy layer inside or outside the database has to issue several requests and postprocess (e. g. filter, aggregate) results instead. Lipcon therefore comments that with sharding "you lose all the features that make a RDBMS useful" and that sharding "is operationally obnoxious" (cf. [Lip09]). This valuation refers to the fact that sharding originally was not designed within current RDBMSs but rather added on top. In contrast many NoSQL databases have embraced sharding as a key feature and some even provide automatic

[3] The memcached API provides the operations `get(key)`, `put(key, value)` and—for reasons of completeness—`remove(key)`.

[4] Least recently used

partitioning and balancing of data among nodes—as e. g. MongoDB of version 1.6 (cf. [MHC+10b], [Mon10]).

In a partitioned scenario knowing how to map database objects to servers is key. An obvious approach may be a simple hashing of database-object primary keys against the set of available database nodes in the following manner:

$$partition \; = \; hash(o) \; mod \; n \qquad with \; o = object \; to \; hash, \; n = number \; of \; nodes$$

As mentioned above the downside of this procedure is that at least parts of the data have to be redistributed whenever nodes leave and join. In a memory caching scenario data redistribution may happen implicitly by observing cache misses, reading data again from a database or backend system, hashing it against the currently available cache servers and let stale cache data be purged from the cache servers by some cleaning policy like LRU. But for persistent data stores this implicit redistribution process is not acceptable as data not present on the available nodes cannot be reconstructed (cf. [Ho09a], [Whi07]). Hence, in a setting where nodes may join and leave at runtime (e. g. due to node crashes, temporal unattainability, maintenance work) a different approach such as consistent hashing has to be found which shall be discussed hereafter.

3.2.1. Consistent Hashing

The idea of consistent hashing was introduced by David Karger et al. in 1997 (cf. [KLL+97]) in the context of a paper about "a family of caching protocols for distrib-uted[sic!] networks that can be used to decrease or eliminate the occurrence of hot spots in the networks". This family of caching protocols grounds on consistent hashing which has been adopted in other fields than caching network protocols since 1997, e. g. in distributed hash-table implementations like Chord ([Par09]) and some memcached clients as well as in the NoSQL scene where it has been integrated into databases like Amazon's Dynamo and Project Voldemort.

"The basic idea behind the consistent hashing algorithm is to hash both objects and caches using the same hash function" blogger Tom White points out. Not only hashing objects but also machines has the advantage, that machines get an interval of the hash-function's range and adjacent machines can take over parts of the interval of their neighbors if those leave and can give parts of their own interval away if a new node joins and gets mapped to an adjacent interval. The consistent hashing approach furthermore has the advantage that client applications can calculate which node to contact in order to request or write a piece of data and there is no metadata server necessary as in systems like e. g. the Google File System (GFS) which has such a central (though clustered) metadata server that contains the mappings between storage servers and data partitions (called *chunks* in GFS; cf. [GL03, p. 2f]).

Figures 3.4 and 3.5 illustrate the idea behind the consistent hashing approach. In figure 3.4 there are three red colored nodes A, B and C and four blue colored objects 1–4 that are mapped to a hash-function's result range which is imagined and pictured as a ring. Which object is mapped to which node is determined by moving clockwise around the ring. So, objects 4 and 1 are mapped to node A, object 2 to node B and object 3 to node C. When a node leaves the system, cache objects will get mapped to their adjacent node (in clockwise direction) and when a node enters the system it will get hashed onto the ring and will overtake objects. An example is depicted in figure 3.5 where compared to figure 3.4 node C left and node D entered the system, so that now objects 3 and 4 will get mapped to node D. This shows that by changing the number of nodes not all objects have to be remapped to the new set of nodes but only part of the objects.

In this raw form there are still issues with this procedure: at first, the distribution of nodes on the ring is actually random as their positions are determined by a hash function and the intervals between nodes may

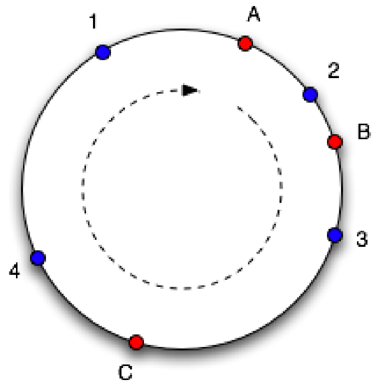


Figure 3.4.: Consistent Hashing – Initial Situation (taken from [Whi07])

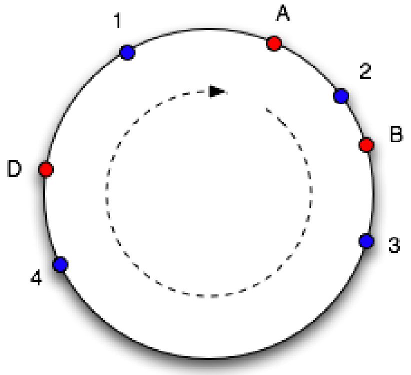


Figure 3.5.: Consistent Hashing – Situation after Node Joining and Departure (taken from [Whi07])

be “unbalanced” which in turn results in an unbalanced distribution of cache objects on these nodes (as it can already be seen in the small scenario of figure 3.5 where node D has to take cache objects from a greater interval than node A and especially node B). An approach to solve this issue is to hash a number of representatives/replicas—also called virtual nodes—for each physical node onto the ring (cf. [Whi07], as an example see figure 3.6). The number of virtual nodes for a physical can be defined individually according to its hardware capacity (cpu, memory, disk capacity) and does not have to be the same for all physical nodes. By appending e. g. a replica counter to a node’s id which then gets hashed, these virtual nodes should distribute points for this node all over the ring.

In his blog post on consistent hashing Tom White has simulated the effect of adding virtual nodes in a setting where he distributes 10,000 objects across ten physical nodes. As a result, the standard deviation of the objects distribution can be dropped from 100% without virtual nodes to 50% with only 2–5 virtual nodes and to 5–10% with 500 virtual nodes per physical node (cf. [Whi07]).

When applied to persistent storages, further issues arise: if a node has left the scene, data stored on this node becomes unavailable, unless it has been replicated to other nodes before; in the opposite case of a new node joining the others, adjacent nodes are no longer responsible for some pieces of data which they still store but not get asked for anymore as the corresponding objects are no longer hashed to them by requesting clients. In order to address this issue, a replication factor (r) can be introduced. By doing so, not only the next node but the next r (physical!) nodes in clockwise direction become responsible for an object (cf. [K$^+$10b], [Ho09a]). Figure 3.7 depicts such a scenario with replicated data: the uppercase letters again represent storage nodes that are–according to the idea of virtual nodes–mapped multiple times onto

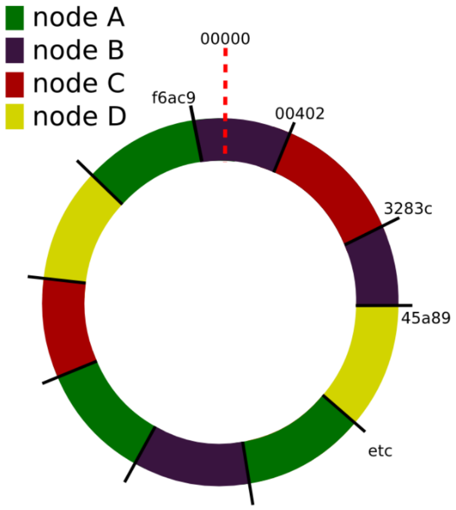


Figure 3.6.: Consistent Hashing – Virtual Nodes Example (taken from [Lip09, slide 12])

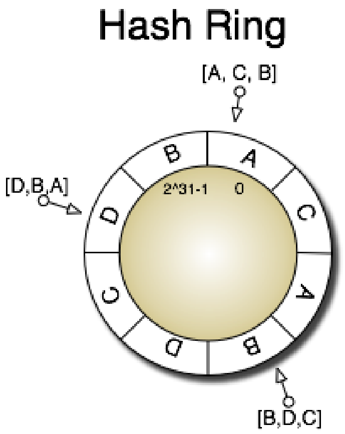


Figure 3.7.: Consistent Hashing – Example with Virtual Nodes and Replicated Data (taken from [K$^+$10b])

the ring, and the circles with arrows represent data objects which are mapped onto the ring at the depicted positions. In this example, the replication factor is three, so for every data object three physical nodes are responsible which are listed in square brackets in the figure. Introducing replicas implicates read and write operations on data partitions which will be considered in the next subsection. The above mentioned issues causing membership changes shall be discussed thereafter.

3.2.2. Read- and Write Operations on Partitioned Data

Introducing replicas in a partitioning scheme—besides reliability benefits—also makes it possible to spread workload for read requests that can go to any physical node responsible for a requested piece of data. In qualification, it should be stated that the possibility of load-balancing read operations does not apply to scenarios in which clients have to decide between multiple versions of a dataset and therefore have to read from a quorum of servers which in turn reduces the ability to load-balance read requests. The Project Voldemort team points out that three parameters are specifically important regarding read as well as write operations (cf. [K$^+$10b]):

N The number of replicas for the data or the piece of data to be read or written.

R The number of machines contacted in read operations.

W The number of machines that have to be blocked in write operations[5].

In the interest to provide e. g. the read-your-own-writes consistency model the following relation between the above parameters becomes necessary:

$$R + W > N$$

Regarding write operations clients have to be clear that these are neither immediately consistent nor isolated (cf. [K+10b]):

- If a write operation completes without errors or exceptions a client can be sure that at least W nodes have executed the operation.
- If the write operation fails as e. g. less than W nodes have executed it, the state of the dataset is unspecified. If at least one node has successfully written the value it will eventually become the new value on all replica nodes, given that the replicas exchange values and agree on the most recent versions of their datasets in background. If no server has been capable to write the new value, it gets lost. Therefore, if the write operation fails, the client can only achieve a consistent state by re-issuing the write operation.

3.2.3. Membership Changes

In a partitioned database where nodes may join and leave the system at any time without impacting its operation all nodes have to communicate with each other, especially when membership changes.

When a new node joins the system the following actions have to happen (cf. [Ho09a]):

1. The newly arriving node announces its presence and its identifier to adjacent nodes or to all nodes via broadcast.
2. The neighbors of the joining node react by adjusting their object and replica ownerships.
3. The joining node copies datasets it is now responsible for from its neighbours. This can be done in bulk and also asynchronously.
4. If, in step 1, the membership change has not been broadcasted to all nodes, the joining node is now announcing its arrival.

In figure 3.8, this process is illustrated. Node X joins a system for which a replication factor of three is configured. It is hashed between A and B, so that the nodes H, A and B transfer data to the new node X and after that the nodes B, C and D can drop parts of their data for which node X is now responsible as a third replica (in addition to nodes H, A and B).

Ho notes that while data transfer and range adjustments happen in steps 2 and 3 the adjacent nodes of the new node may still be requested and can forward them to the new node. This is the case if it e. g. has already received the requested piece of data or can send vector clocks to clients to let them determine the most recent version of a dataset after having contacted multiple replicas.

When a node leaves the system the following actions have to occur (cf. [Ho09a]):

[5] W can be set to zero if a client wishes to write in a non-blocking fashion which does not assure the client of the operation's success.

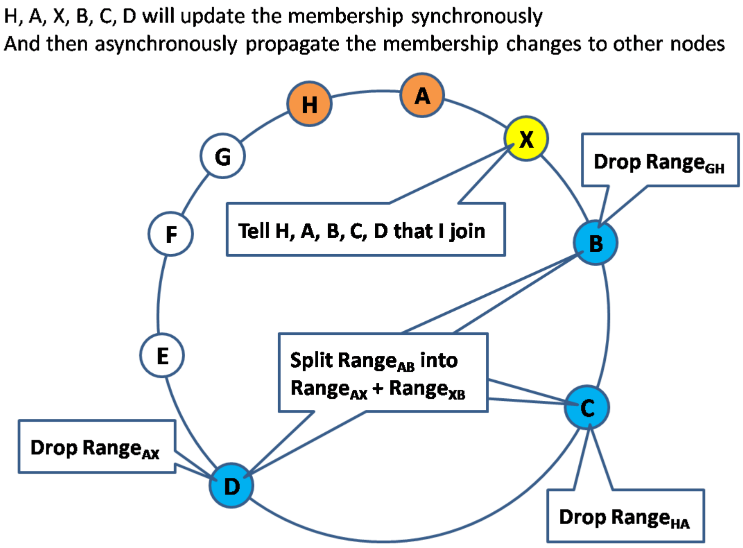

H, A, X, B, C, D will update the membership synchronously
And then asynchronously propagate the membership changes to other nodes
H
A
X
G
Drop Range$_{GH}$
Tell H, A, B, C, D that I join
F
B
E
Split Range$_{AB}$ into Range$_{AX}$ + Range$_{XB}$
C
Drop Range$_{AX}$
D
Drop Range$_{HA}$

Figure 3.8.: Membership Changes – Node X joins the System (taken from [Ho09a])

1. Nodes within the system need to detect whether a node has left as it might have crashed and not been able to notify the other nodes of its departure. It is also common in many systems that no notifications get exchanged when a node leaves. If the nodes of the system communicate regularly e. g. via the Gossip protocol they are able to detect a node's departure because it no longer responds.
2. If a node's departure has been detected, the neighbors of the node have to react by exchanging data with each other and adjusting their object and replica ownerships.

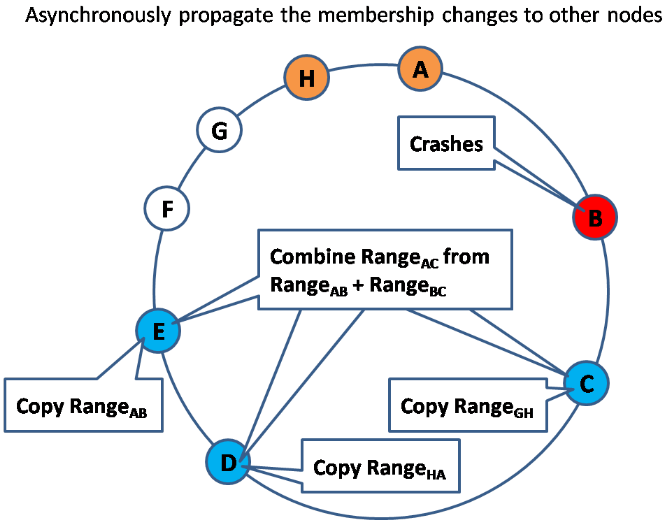

Asynchronously propagate the membership changes to other nodes
H
A
G
Crashes
F
B
Combine Range$_{AC}$ from Range$_{AB}$ + Range$_{BC}$
E
C
Copy Range$_{AB}$
Copy Range$_{GH}$
D
Copy Range$_{HA}$

Figure 3.9.: Membership Changes – Node B leaves the System (taken from [Ho09a])

Figure 3.9 shows the actions to be taken when node B—due to a crash—leaves the system. Nodes C, D and E become responsible for new intervals of hashed objects and therefore have to copy data from nodes in counterclockwise direction and also reorganize their internal representation of the intervals as the $Range_{AB}$ and $Range_{BC}$ now have collapsed to $Range_{AC}$.

3.3. Storage Layout

In his talk on design patterns for nonrelational databases Todd Lipcon presents an overview of storage layouts, which determine how the disk is accessed and therefore directly implicate performance. Furthermore, the storage layout defines which kind of data (e. g. whole rows, whole columns, subset of columns) can be read en bloque (cf. [Lip09, slide 21–31]).

Row-Based Storage Layout means that a table of a relational model gets serialized as its lines are appended and flushed to disk (see figure 3.10a). The advantages of this storage layout are that at first whole datasets can be read and written in a single IO operation and that seconldy one has a "[g]ood locality of access (on disk and in cache) of different columns". On the downside, operating on columns is expensive as a considerable amount data (in a naïve implementation all of the data) has to be read.

Columnar Storage Layout serializes tables by appending their columns and flushing them to disk (see figure 3.10b). Therefore operations on columns are fast and cheap while operations on rows are costly and can lead to seeks in a lot or all of the columns. A typical application field for this type of storage layout is analytics where an efficient examination of columns for statistical purposes is important.

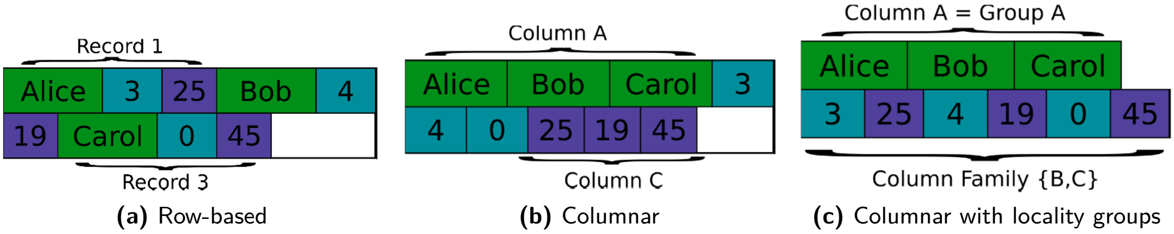
Columnar Storage Layout with Locality Groups is similar to column-based storage but adds the feature of defining so called locality groups that are groups of columns expected to be accessed together by clients. The columns of such a group may therefore be stored together and physically separated from other columns and column groups (see figure 3.10c). The idea of locality groups was introduced in Google's Bigtable paper. It firstly describes the logical model of column-families for semantically affine or interrelated columns (cf. [CDG+06, section 2]) and later on presents the locality groups idea as a refinement where column-families can be grouped or segregated for physical storage (cf. [CDG+06, section 6]).

Log Structured Merge Trees (LSM-trees) in contrast to the storage layouts explained before do not describe how to serialize logical datastructures (like tables, documents etc.) but how to efficiently use memory and disk storage in order to satisfy read and write requests in an efficient, performant and still safely manner. The idea, brought up by O'Neil et al. in 1996 (cf. [OCGO96]), is to hold chunks of data in memory (in so called Memtables), maintaining on-disk commit-logs for these in-memory data structures and flushing the memtables to disk from time to time into so called SSTables (see figure 3.11a and 3.11e). These are immutable and get compacted over time by copying the compacted SSTable to another area of the disk while preserving the original SSTable and removing the latter after the compactation process has happened (see figure 3.11f). The compactation is necessary as data stored in these SSTable may have changed or been deleted by clients. These data modifications are first reflected in a Memtable that later gets flushed to disk as a whole into a SSTable which may be compacted together with other SSTables already on disk. Read-requests go to the Memtable as well as the SSTables containing the requested data and return a merged view of it (see figure 3.11b). To optimize read-requests and only read the relevant SSTables bloom filters[6] can be used (see figure 3.11c). Write requests go to the Memtable as well as an on-disk commit-log synchronously (see figure 3.11d).

The advantages of log structured merge trees are that memory can be utilized to satisfy read requests quickly and disk I/O gets faster as SSTables can be read sequentially because their data is not randomly distributed over the disk. LSM-trees also tolerate machine crashes as write operations not only go to memory but also (synchronously) to a commit-log by which a machine can recover from a

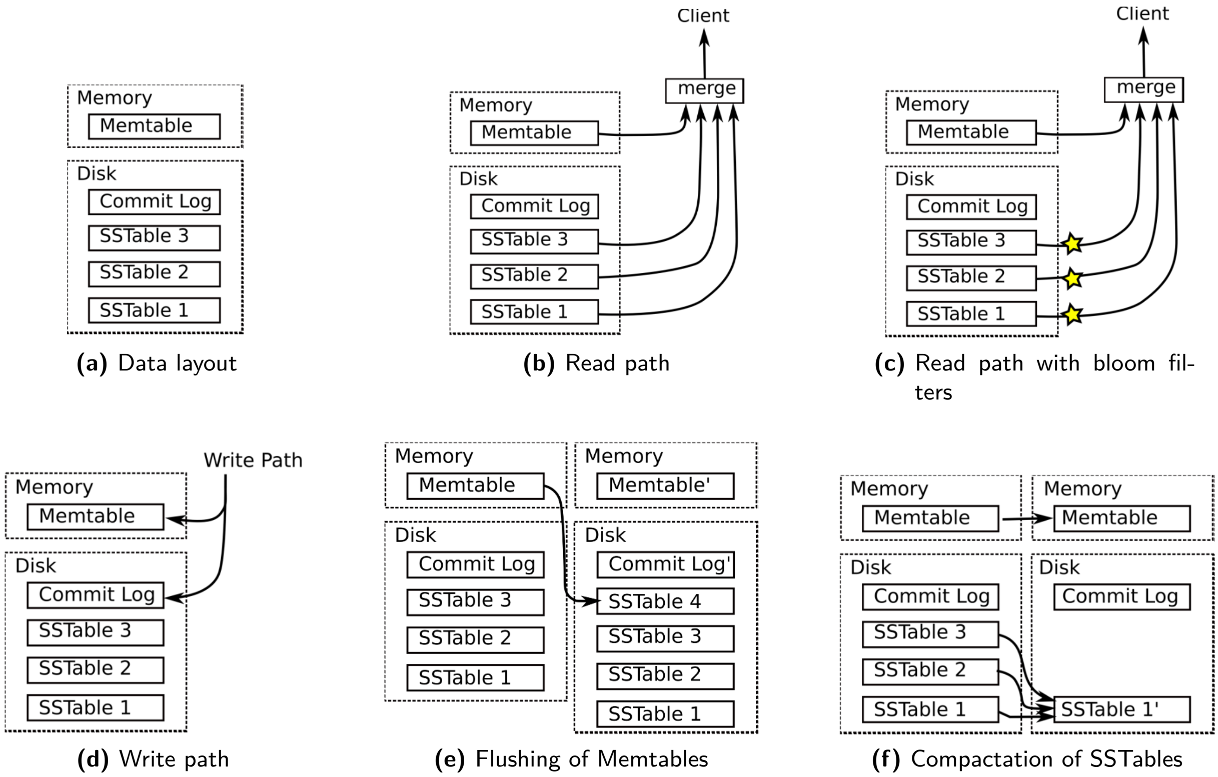
[6] Bloom filters, as described by Burton Bloom in 1970 (cf. [Blo70]), are probabalistic data structures used to efficiently determine if elements are member of a set in a way that avoids false-negatives (while false-positives are possible).

crash. Log structured merge trees are implemented in e. g. Google's Bigtable (cf. [CDG+06, sections 4, 5.3, 5.4, 6]) as blogger Ricky Ho summarizes graphically in figure 3.12.



(a) Row-based **(b)** Columnar **(c)** Columnar with locality groups

Figure 3.10.: Storage Layout – Row-based, Columnar with/out Locality Groups (taken from [Lip09, slides 22–24])



(a) Data layout **(b)** Read path **(c)** Read path with bloom filters

(d) Write path **(e)** Flushing of Memtables **(f)** Compactation of SSTables

Figure 3.11.: Storage Layout – Log Structured Merge Trees (taken from [Lip09, slides 26–31])

Regarding the discussion of on-disk or in-memory storage or a combination of both (as in LSM-trees) blogger Nati Shalom notes that it "comes down mostly to cost/GB of data and read/write performance". He quotes from an analysis by the Stanford University (entitled "The Case for RAMClouds", cf. [OAE+10]) which comes to the conclusion that "cost is also a function of performance". He cites from the analysis that "if an application needs to store a large amount of data inexpensively and has a relatively low access rate, RAMCloud is not the best solution" but "for systems with high throughput requirements a RAM-Cloud[sic!] can provide not just high performance but also energy efficiency" (cf. [Sha09a]).

Blogger Ricky Ho adds to the categorization of storage layouts above that some NoSQL databases leave the storage implementation open and allow to plug-in different kinds of it—such as Project Voldemort which allows to use e. g. a relational database, the key/value database BerkleyDB, the filesystem or a memory

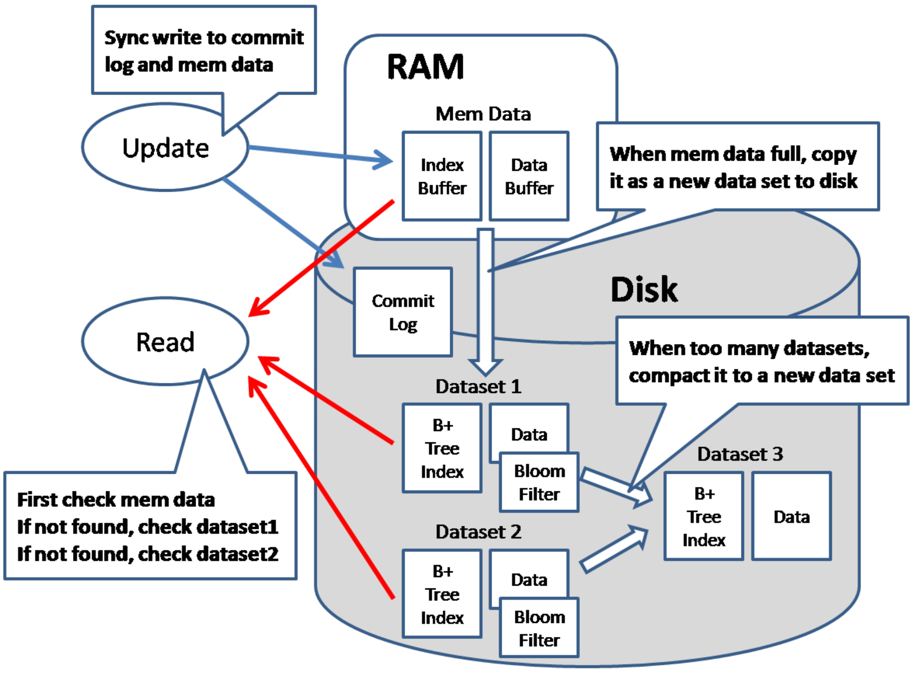

Sync write to commit log and mem data
Update
RAM
Mem Data
Index Buffer
Data Buffer
When mem data full, copy it as a new data set to disk
Disk
Commit Log
Read
When too many datasets, compact it to a new data set
Dataset 1
B+ Tree Index
Data
Bloom Filter
Dataset 3
B+ Tree Index
Data
First check mem data
If not found, check dataset1
If not found, check dataset2
Dataset 2
B+ Tree Index
Data
Bloom Filter

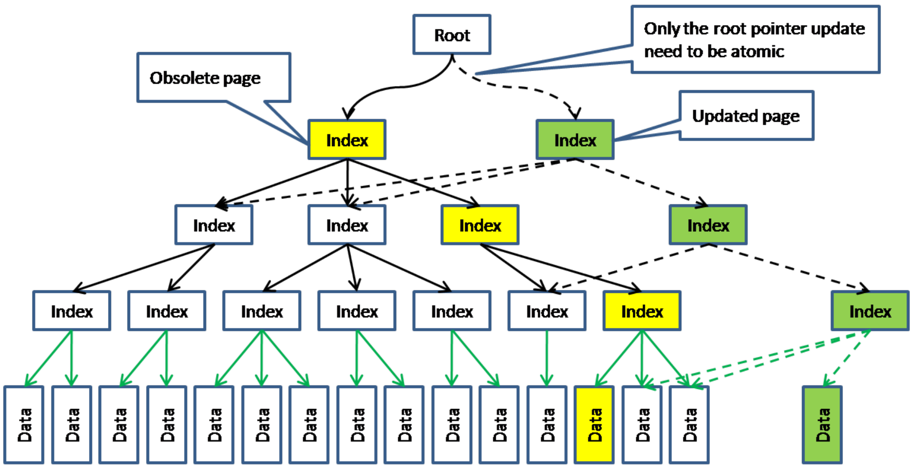
Figure 3.12.: Storage Layout – MemTables and SSTables in Bigtable (taken from [Ho09a])

hashtable as storage implementations. In contrast, most NoSQL databases implement a storage system optimized to their specific characteristics.

In addition to these general storage layouts and the distinction between pluggable and proprietary storage implementations most databases carry out optimizations according to their specific data model, query processing means, indexing structures etc..

As an example, the document database CouchDB provides a copy-on-modified semantic in combination with an append only file for database contents (cf. [Apa10b, Section “ACID Properties”]). The copy-on-modified semantic means that a private copy for the user issuing the modification request is made from the data as well as the index (a B-Tree in case of CouchDB); this allows CouchDB to provide read-your-own-writes consistency for the issuer while modifications are only eventually seen by other clients. The private copy is propagated to replication nodes and—as CouchDB also supports multi-version concurrency control (MVCC)—maybe clients have to merge conflicting versions before a new version becomes valid for all clients (which is done by swapping the root pointer of storage metadata that is organized in a B-Tree as shown in figure 3.13). CouchDB synchronously persists all updates to disk in an append-only fashion. A garbage-collection compacts the persisted data from time to time by copying the compacted file contents into a new file and not touching the old file until the new one is written correctly. This treatment allows continuous operation of the system while the garbage-collection is executed.

Ho depicts how the CouchDB index and files are affected by updates as shown in figure 3.13.



Copy on modify. Everyone sees his own copy of update
Finally the root pointer is swapped and everyone's view is updated
Yellow page becomes garbage over time.
File will be compacted periodically by copying to a different file.

Figure 3.13.: Storage Layout – Copy-on-modfify in CouchDB (taken from [Ho09a])

3.4. Query Models

As blogger Nati Shalom notes there are substantial differences in the querying capabilities the different NoSQL datastores offer (cf. [Sha09a]): whereas key/value stores by design often only provide a lookup by primary key or some id field and lack capabilities to query any further fields, other datastores like the document databases CouchDB and MongoDB allow for complex queries—at least static ones predefined on the database nodes (as in CouchDB). This is not surprising as in the design of many NoSQL databases rich dynamic querying features have been omitted in favor of performance and scalability. On the other hand, also when using NoSQL databases, there are use-cases requiring at least some querying features for non-primary key attributes. In his blog post "Query processing for NOSQL DB" blogger Ricky Ho addresses this issue and presents several ways to implement query features that do not come out of the box with some NoSQL datastores:

Companion SQL-database is an approach in which searchable attributes are copied to a SQL or text database. The querying capabilities of this database are used to retrieve the primary keys of matching datasets by which the NoSQL database will subsequently be accessed (see figure 3.14).

Scatter/Gather Local Search can be used if the NoSQL store allows querying and indexing within database server nodes. If this is the case a query processor can dispatch queries to the database nodes where the query is executed locally. The results from all database servers are sent back to the query processor postprocessing them to e. g. do some aggregation and returning the results to a client that issued the query (see figure 3.15).

Distributed B+Trees are another alternative to implement querying features ((see figure 3.16)). The basic idea is to hash the searchable attribute to locate the root node of a distributed B+tree (further information on scalable, distributed B+Trees can be found in a paper by Microsoft, HP and the University of Toronto, cf. [AGS08]). The "value" of this root node then contains an id for a child node in the B+tree which can again be looked up. This process is repeated until a leaf node is reached which contains the primary-key or id of a NoSQL database entry matching search criteria.

Ho notes that node-updates in distributed B+trees (due to splits and merges) have to be handled cautiously and should be handled in an atomic fashion.

Prefix Hash Table (aka Distributed Trie) is a tree-datastructure where every path from the root-node to the leafs contains the prefix of the key and every node in the trie contains all the data whose key is prefixed by it (for further information cf. a Berkley-paper on this datastructure [RRHS04]). Besides an illustration (see figure 3.17) Ho provides some code-snippets in his blog post that describe how to operate on prefix hash tables / distributed tries and how to use them for querying purposes (cf. [Ho09b]).

Ho furthermore points out that junctions in the search criteria have to be addressed explicitely in querying approaches involving distribution (scatter/gather local search, distributed B+trees):

OR-Junctions are simple since the search results from different database nodes can be put together by a union-operation.

AND-Junctions are more difficult as the intersection of individually matching criteria is wanted and therefore efficient ways to intersect potentially large sets are needed. A naïve implementation might send all matching objects to a server that performs set intersection (e. g. a query processor initiating the distributed query as in figures 3.14–3.16). However, this involves large bandwith consumption, if some or all datasets are large. A number of more efficient approaches are described in a paper by Reynolds and Vahdat (cf. [RV03]):

- Bloom filters can be used to test if elements are definitely not contained in a (result) set, which can help in intersecting sets as the number of elements having to be transferred over the network and/or compared by more expensive computation than bloom filters can be drastically reduced.
- Result sets of certain popular searches or bloom filters of popular search criteria can be cached.
- If client applications do not require the full result set at once an incremental fetch strategy can be applied where result set data is streamed to the client using a cursor mode. This moves computation into the client application which could perhaps also become responsible for filtering operations like (sub-)set intersection.

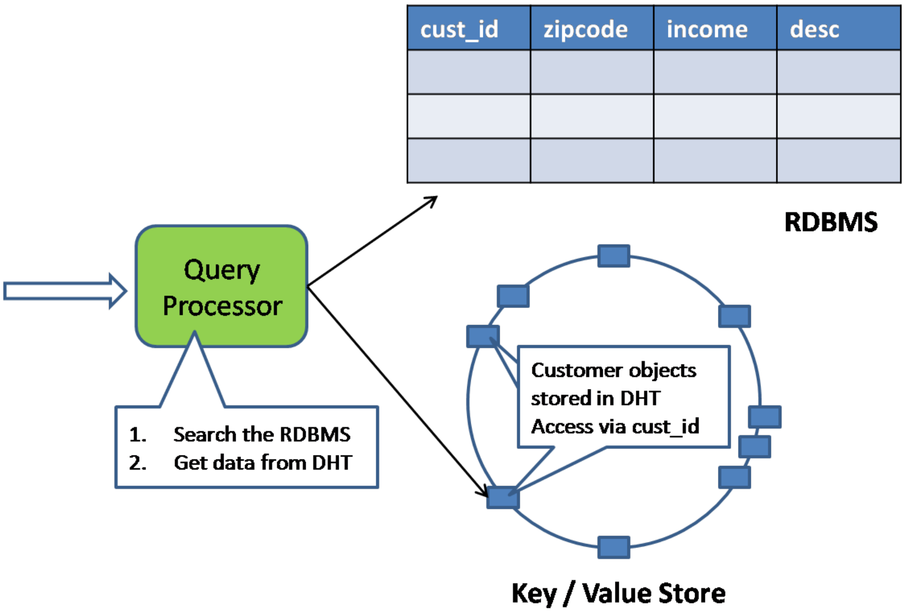


Figure 3.14.: Query Models – Companion SQL-Database (taken from [Ho09b])

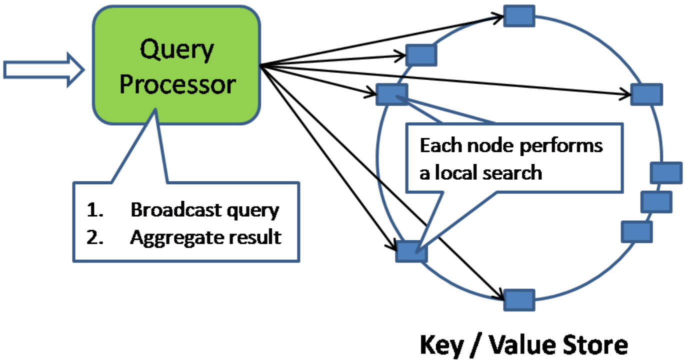


Figure 3.15.: Query Models – Scatter/Gather Local Search (taken from [Ho09b])

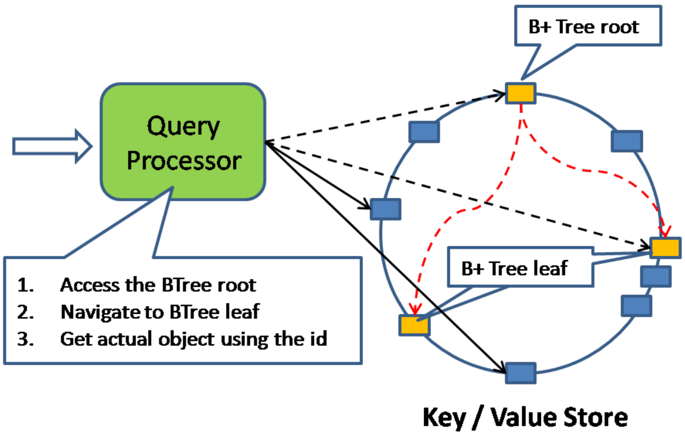


Figure 3.16.: Query Models – Distributed B+Tree (taken from [Ho09b])

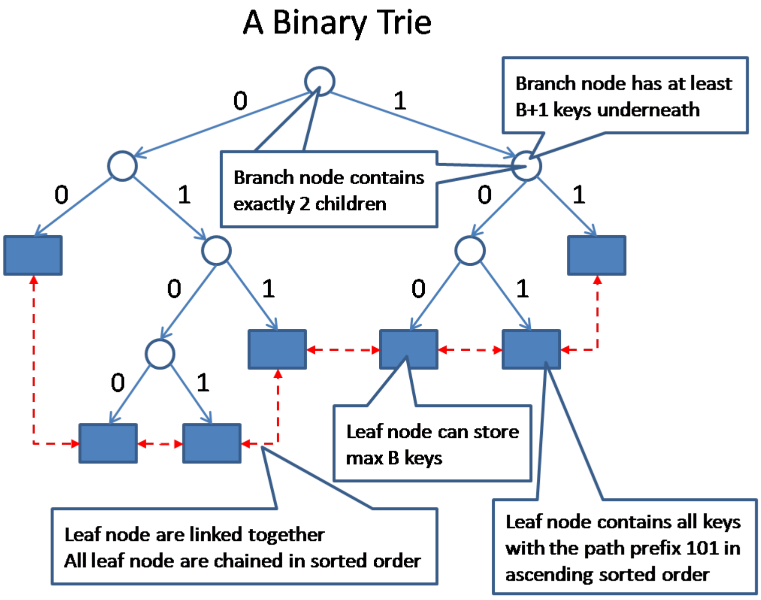

A Binary Trie
0
1
Branch node has at least B+1 keys underneath
Branch node contains exactly 2 children
0
1
0
1
0
1
0
1
0
1
Leaf node can store max B keys
Leaf node are linked together
All leaf node are chained in sorted order
Leaf node contains all keys with the path prefix 101 in ascending sorted order

Figure 3.17.: Query Models – Prefix Hash Table / Distributed Trie (taken from [Ho09b])

3.5. Distributed Data Processing via MapReduce

The last section focused on how to operate on distributed environments by concentrating on dynamic querying mechanisms. Although no substitution for lacking query and maintenance capabilites (as Brian Aker correctly humorously argues in his talk "Your Guide To NoSQL"), but somehow related is the possibility for operating on distributed database nodes in a MapReduce fashion. This approach, brought up by Google employees in 2004 (cf. [DG04]), splits a task into two stages, described by the functions `map` and `reduce`. In the first stage a coordinater designates pieces of data to process a number of nodes which execute a given `map` function and produce intermediate output. Next, the intermediate output is processed by a number of machines executing a given `reduce` function whose purpose it is to create the final output from the intermediate results, e. g. by some aggregation. Both the `map` and `reduce` functions have to be understood in a real functional manner, so they do not depend on some state on the machine they are executed on and therefore produce identical output on each execution environment given the identical input data. The partitioning of the original data as well as the assignment of intermediate data is done by the coordinator according to Google's original paper.

Figure 3.18 depicts and summarzies the MapReduce fashion of distributed data processing.

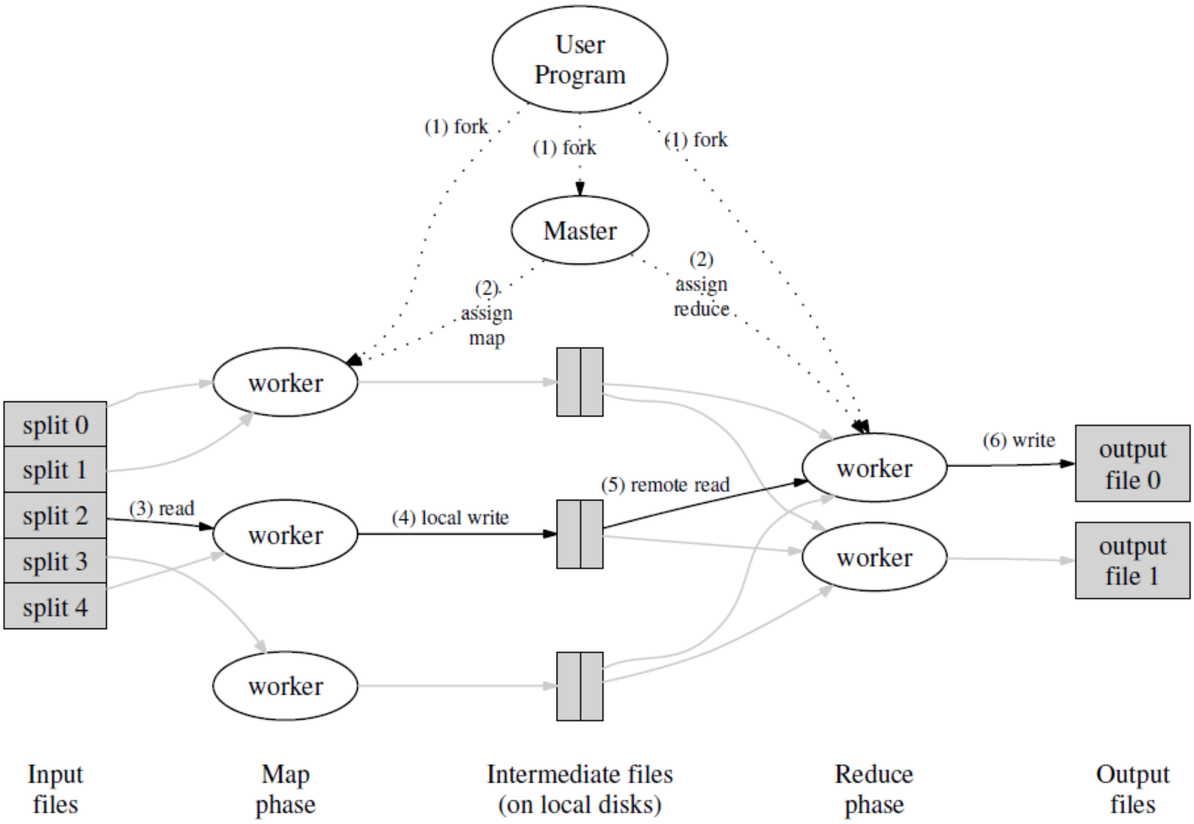


Figure 3.18.: MapReduce – Execution Overview (taken from [DG04, p. 3])

The MapReduce paradigm has been adopted by many programming languages (e. g. Python), frameworks (e. g. Apache Hadoop), even JavaScript toolkits (e. g. Dojo) and also NoSQL databases (e. g. CouchDB). This is the case because it fits well into distributed processing as blogger Ricky Ho notes (cf. [Ho09a]), especially for analytical purposes or precalculation tasks (e. g. in CouchDB to generate views of the data cf. [Apa10b]). When applied to databases, MapReduce means to process a set of keys by submitting the process logic (`map`- and `reduce`-function code) to the storage nodes which locally apply the `map` function to keys that should be processed and that they own. The intermediate results can be consistently hashed just as regular data and processed by the following nodes in clockwise direction, which apply the `reduce` function to the intermediate results and produce the final results. It should be noted that by consistently hashing the intermediate results there is no coordinator needed to tell processing nodes where to find them.

The idea of applying MapReduce to a distributed datastore is illustrated in figure 3.19.

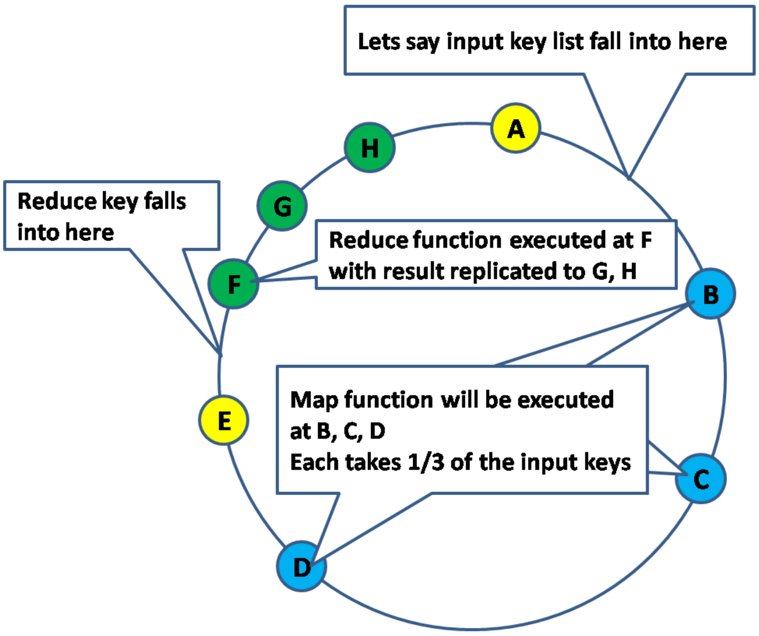

Lets say input key list fall into here
A
H
G
Reduce key falls into here
Reduce function executed at F with result replicated to G, H
F
B
Map function will be executed at B, C, D
Each takes 1/3 of the input keys
E
C
D

Figure 3.19.: MapReduce – Execution on Distributed Storage Nodes (taken from [Ho09a]

4. Key-/Value-Stores

Having discussed common concepts, techniques and patterns the first category of NoSQL datastores will be investigated in this chapter. Key-/value-stores have a simple data model in common: a map/dictionary, allowing clients to put and request values per key. Besides the data-model and the API, modern key-value stores favor high scalability over consistency and therefore most of them also omit rich ad-hoc querying and analytics features (especially joins and aggregate operations are set aside). Often, the length of keys to be stored is limited to a certain number of bytes while there is less limitation on values (cf. [Ipp09], [Nor09]).

Key-/value-stores have existed for a long time (e. g. Berkeley DB [Ora10d]) but a large number of this class of NoSQL stores that has emerged in the last couple of years has been heavily influenced by Amazon's Dynamo which will be investigated thoroughly in this chapter. Among the great variety of free and open-source key-/value-stores, Project Voldemort will be further examined. At the end of the chapter some other notable key-/value-stores will be briefly looked at.

4.1. Amazon's Dynamo

Amazon Dynamo is one of several databases used at Amazon for different purposes (others are e. g. SimpleDB or S3, the Simple Storage Service, cf. [Ama10b], [Ama10a]). Because of its influence on a number of NoSQL databases, especially key-/value-stores, the following section will investigate in more detail Dynamo's influencing factors, applied concepts, system design and implementation.

4.1.1. Context and Requirements at Amazon

The technologial context these storage services operate upon shall be outlined as follows (according to Amazon's Dynamo Paper by DeCandia et al. cf. [DHJ+07, p. 205]):

- The infrastructure is made up by tens of thousands of servers and network components located in many datacenters around the world.
- Commodity hardware is used.
- Component failure is the "standard mode of operation".
- "Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services."

Apart from these technological factors, the design of Dynamo is also influenced by business considerations (cf. [DHJ+07, p. 205]):

- Strict, internal service level agreements (SLAs) regarding "performance, reliability and efficiency" have to be met in the 99.9th percentile of the distribution[1]. DeCandia et al. consider "state management" as offered by Dynamo and the other databases as being crucial to meet these SLAs in a service whose business logic is in most cases rather lightweight at Amazon (cf. [DHJ+07, p. 207–208]).
- One of the most important requirements at Amazon is reliability "because even the slightest outage has significant financial consequences and impacts customer trust".
- "[To] support continuous growth, the platform needs to be highly scalable".

At Amazon "Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance". DeCandia et al. furthermore argue that a lot of services only need access via primary key (such as "best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog") and that the usage of a common relational database "would lead to inefficiencies and limit scale and availability" (cf. [DHJ+07, p. 205]).

4.1.2. Concepts Applied in Dynamo

Out of the concepts presented in chapter 3, Dynamo uses consistent hashing along with replication as a partitioning scheme. Objects stored in partitions among nodes are versioned (multi-version storage). To maintain consistency during updates Dynamo uses a quorum-like technique and a (not further specified) protocol for decentralized replica synchronization. To manage membership and detect machine failures it employs a gossip-based protocol which allows to add and remove servers with "a minimal need for manual administration" (cf. [DHJ+07, p. 205–206]).

DeCandia et al. note their contribution to the research community is that Dynamo as an "eventually-consistent storage system can be used in production with demanding applications".

4.1.3. System Design

Cosiderations and Overview

DeCandia et al. militate against RDMBSs at Amazon as most services there "only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS". Furthermore they consider the "available replication technologies" for RDBMSs as "limited and typically choose[ing] consistency over availability". They admit that advances have been made to scale and partition RDBMSs but state that such setups remain difficult to configure and operate (cf. [DHJ+07, p. 206]).

Therefore, they took the decision to build Dynamo with a simple key/value interface storing values as BLOBs. Operations are limited to one key/value-pair at a time, so update operations are limited to single keys allowing no cross-references to other key-/value-pairs or operations spanning more than one key-/value-pair. In addition, hierarchichal namespaces (like in directory services or filesystems) are not supported by Dynamo (cf. [DHJ+07, p. 206 and 209]).

Dynamo is implemented as a partitioned system with replication and defined consistency windows. Therefore, Dynamo "targets applications that operate with weaker consistency [. . .] if this results in high availability". It does not provide any isolation guarantees (cf. [DHJ+07, p. 206]). DeCandia et al. argue

[1] To meet requirements in the average or median case plus some variance is good enough to satisfy more than the just majority of users according to experience made at Amazon.

Problem	Technique	Advantages
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information

Table 4.1.: Amazon's Dynamo – Summary of Techniques (taken from [DHJ+07, p. 209])

that a synchronous replication scheme is not achievable given the context and requirements at Amazon (especially high-availability and scalability). Instead, they have selected an optimistic replication scheme. This features background replication and the possibility for write operations even in the presence of disconnected replicas (due to e. g. network or machine failures). So, as Dynamo is designed to be "always writeable (i. e. a datastore that is highly available for writes)" conflict resolution has to happen during reads. Furthermore DeCandia et al. argue that a datastore can only perform simple policies of conflict resolution and therefore a client application is better equipped for this task as it is "aware of the data schema" and "can decide on a conflict resolution method that is best suited for the client's experience". If application developers do not want to implement such a business logic specific reconciliation strategy Dynamo also provides simple strategies they can just use, such as "last write wins", a timestamp-based reconciliation (cf. [DHJ+07, p. 207–208 and 214]).

To provide simple scalability Dynamo features "a simple scale-out scheme to address growth in data set size or request rates" which allows adding of "one storage host [. . .] at a time" (cf. [DHJ+07, p. 206 and 208]).

In Dynamo, all nodes have equal responsibilities; there are no distinguished nodes having special roles. In addition, its design favors "decentralized peer-to-peer techniques over centralized control" as the latter has "resulted in outages" in the past at Amazon. Storage hosts added to the system can have heterogeneous hardware which Dynamo has to consider to distribute work proportionally "to the capabilities of the individual servers" (cf. [DHJ+07, p. 208]).

As Dynamo is operated in Amazon's own administrative domain, the environment and all nodes are considered non-hostile and therefore no security related features such as authorization and authentication are implemented in Dynamo (cf. [DHJ+07, p. 206 and 209]).

Table 4.1 summarizes difficulties faced in the design of Dynamo along with techniques applied to address them and respective advantages.

System Interface

The interface Dynamo provides to client applications consists of only two operations:

- `get(key)`, returning a list of objects and a context
- `put(key, context, object)`, with no return value

The get-operation may return more than one object if there are version conflicts for objects stored under the given key. It also returns a context, in which system metadata such as the object version is stored, and clients have to provide this context object as a parameter in addition to key and object in the put-operation.

Key and object values are not interpreted by Dynamo but handled as "an opaque array of bytes". The key is hashed by the MD5 algorithm to determine the storage nodes responsible for this key-/value-pair.

Partitioning Algorithm

To provide incremental scalability, Dynamo uses consistent hashing to dynamically partition data across the storage hosts that are present in the system at a given time. DeCandia et al. argue that using consistent hashing in its raw form has two downsides: first, the random mapping of storage hosts and data items onto the ring leads to unbalanced distribution of data and load; second, the consistent hashing algorithm treats each storage node equally and does not take into account its hardware resources. To overcome both difficulties, Dynamo applies the concepts of virtual nodes, i. e. for each storage host multiple virtual nodes get hashed onto the ring (as described in subsection 3.2.1) and the number of virtual nodes per physical node is used to take the hardware capabilities of storage nodes into account (i. e. the more resources the more virtual nodes per physical node, cf. [DHJ+07, p. 209–210]).

Replication

To ensure availability and durability in an infrastructure where machine-crashes are the "standard mode of operation" Dynamo uses replication of data among nodes. Each data item is replicated N-times where N can be configured "per-instance" of Dynamo (a typical for N at Amazon is 3, cf. [DHJ+07, p. 214]). The storage node in charge of storing a tuple with key k[2] also becomes responsible for replicating updated versions of the tuple with key k to its N-1 successors in clockwise direction. There is a list of nodes – called *preference list* – determined for each key k that has to be stored in Dynamo. This list consists of more than N nodes as N successive virtual nodes may map to less than N distinct physical nodes (as also discussed in subsection 3.2.1 on consistent hashing).

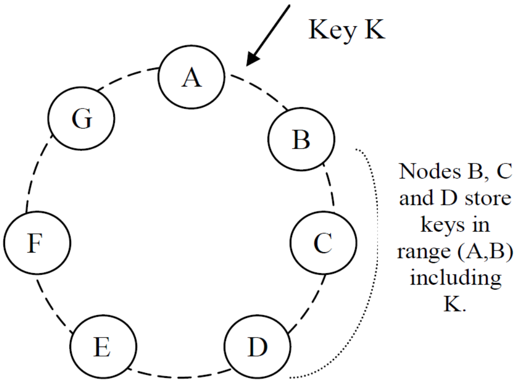


Figure 4.1.: Amazon's Dynamo – Consistent Hashing with Replication (taken from [DHJ+07, p. 209])

[2] i. e. the next node on the ring in clockwise direction from hash(k)

Data Versioning

Dynamo is designed to be an eventually consistent system. This means that update operations return before all replica nodes have received and applied the update. Subsequent read operations therefore may return different versions from different replica nodes. The update propagation time between replicas is limited in Amazon's platform if no errors are present; under certain failure scenarios however "updates may not arrive at all replicas for an extend period of time" (cf. [DHJ+07, p. 210]).

Such inconsistencies need to be taken into consideration by applications. As an example, the shopping cart application never rejects add-to-cart-operations. Even when evident that the replica does not feature the latest version of a shopping cart (indicated by a vector clock delivered with update requests, see below), it applies the add-operation to its local shopping cart.

As a consequence of an update operation, Dynamo always creates a new and immutable version of the updated data item. In Amazon's production systems most of these versions subsume one another linearly and the system can determine the latest version by syntactic reconciliation. However, because of failures (like network partitions) and concurrent updates multiple, conflicting versions of the same data item may be present in the system at the same time. As the datastore cannot reconcile these concurrent versions only the client application that contains knowledge about its data structures and semantics is able to resolve version conflicts and conciliate a valid version out of two or more conflicting versions (semantic reconciliation). So, client applications using Dynamo have to be aware of this and must "explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates)"; [DHJ+07, p. 210]).

To determine conflicting versions, perform syntactic reconciliation and support client application to resolve conflicting versions Dynamo uses the concept of vector clocks (introduced in subsection 3.1.3). As mentioned in the subsection on Dynamo's system interface (see 4.1.3) clients have to deliver a context when issuing update requests. This context includes a vector clock of the data they have read earlier, so that Dynamo knows which version to update and can set the vector clock of the updated data item properly. If concurrent versions of a data item occur, Dynamo will deliver them to clients within the context-information replied by the read-request. Before client issues an update to such a data item, it has to reconcile the concurrent versions into one valid version.

Figure 4.2 illustrates the usage of vector clocks as well as the detection and reconciliation of concurrent versions.

In this illustration a client first creates a data item and the update request is handled by a storage host Sx, so that the vector clock ([Sx,1]) will be associated with it. Next, a client updates this data item and this update request leading to version 2 is also executed by node Sx. The resulting vector clock will be ([Sx,2]), as a casual ordering between [Sx,1] and [Sx,2] can be determined as the latter clearly succeeds the former (same storage host, ascending version numbers). As [Sx,1] has only one successor ([Sx,2]) it is no longer needed for casual reasoning and can be removed from the vector clock.

Next, one client issues an update for version D2 that gets handled by storage host Sy and results in the vector clock ([Sx,2],[Sy,1]). Here, the element [Sx,2] of the vector clock cannot get omitted. This is because the example assumes[3] that a client issuing it has read version D2 from a node (say Sx) while storage host Sy has not yet received this version from its companion nodes. So the client wants to update a more recent version than node Sy knows at that time which is accepted due to the "always writeable"-property of Dynamo. The same happens with another client that has read D2 and issued an update handled by storage host Sz which is unaware of the version D2 at that time. This results in version D4 with the vector clock ([Sx,2],[Sz,1]).

[3] These assumptions cannot be seen in the illustration of figure 4.2 but are only contained in the textual description by DeCandia et al. (cf. [DHJ+07, p. 211]).

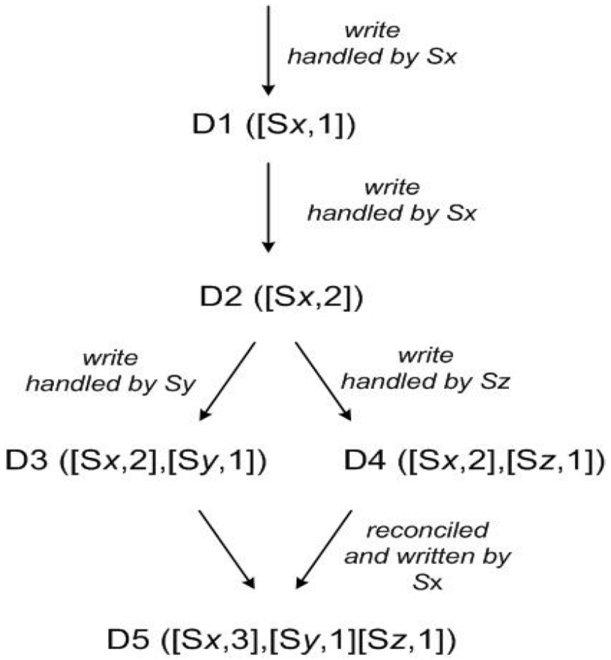


Figure 4.2.: Amazon's Dynamo – Concurrent Updates on a Data Item (taken from [DHJ^{+}07, p. 211])

In the next read request both D3 and D4 get delivered to a client along with a summary of their vector clocks—in particular: ([Sx,2],[Sy,1],[Sz,1]). The client can detect that versions D3 and D4 are in conflict, as the combined vector clock submitted in the read context does not reflect a linear, subsequent ordering. Before a client can issue another update to the system it has to reconcile a version D5 from the concurrent versions D3 and D4. If this update request is handled by node Sx again, it will advance it's version number in the vector clock resulting in ([Sx,3],[Sy,1],[Sz,1]) as depicted in figure 4.2.

Execution of get() and put() Operations

Dynamo allows any storage node of a Dynamo instance to receive get and put requests for any key. In order to contact a node, clients applications may use either a routing through a generic load balancer or a client-library that reflects Dynamo's partitioning scheme and can determine the storage host to contact by calculation. The advantage of the first node-selection approach is that it can remain unaware of Dynamo specific code. The second approach reduces latency as the load balancer does not have to be contacted, demanding one less network hop (cf. [DHJ^{+}07, p. 211]).

As any node can receive requests for any key in the ring the requests may have to be forwarded between storage hosts. This is taking place based on the preference list containing prioritized storage hosts to contact for a given key. When a node receives a client request, it will forward it to storage hosts according to their order in the preference list (if the node itself is placed first in the preference list, forwarding will naturally become dispensable). Once a read or update request occurs, the first N healthy nodes will be taken into account (inaccessible and down nodes are just skipped over; cf. [DHJ^{+}07, p. 211]).

To provide a consistent view to clients, Dynamo applies a quorum-like consistency protocol containing two configurable values: R and W serving as the minimum number of storage hosts having to take part in successful read or write operations, respectively. To get a quorum-like system, these values have to be set to $R + W > N$. DeCandia et al. remark that the slowest replica dictates the latency of an operation and therefore R and W are often chosen so that $R + W < N$ (in order to reduce the probability of slow hosts getting involved into read and write requests; cf. [DHJ^{+}07, p. 211]). In other use cases such as Dynamo setups that "function as the authoritative persistence cache for data stored in more heavy weight backing stores" R is typically set to 1 and W to N to allow high performance for read operations. On the other hand "low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas". In addition, durability remains vulnerable for a certain amount of time when the write request has been completed but the updated version has been propagated to a few nodes only. DeCandia et al. comment that "[the] main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability" (cf. [DHJ+07, p. 214]). (A typical configuration that fits Amazon's SLAs concerning performance, durability, consistency and availability is $N = 3, R = 2, W = 2$, cf. [DHJ+07, p. 215]).

Upon write requests (such as the put operation of Dynamo's interface), the first answering node on the preference list (called *coordinator* in Dynamo) creates a new vector clock for the new version and writes it locally. Following this procedure, the same node replicates the new version to other storage hosts responsible for the written data item (the next top N storage hosts on the preference list). The write request is considered successful if at least $W - 1$ of these hosts respond to this update (cf. [DHJ+07, p. 211–212]).

Once a storage host receives a read request it will ask the N top storage hosts on the preference list for the requested data item and waits for $R - 1$ to respond. If these nodes reply with different versions that are in conflict (observable by vector-clock reasoning), it will return concurrent versions to the requesting client (cf. [DHJ+07, p. 212]).

Membership

Dynamo implements an explicit mechanism of adding and removing nodes to the system. There is no implicit membership detection implemented in Dynamo as temporary outages or flapping servers would cause the system to recalculate ranges on the consistent hash ring and associated data structures like Merkle-trees (see below). Additionally, Dynamo already provides means to handle temporary unavailable nodes, as will be discussed in the next subsection. Therefore, Dynamo administrators have to explicitly add and remove nodes via a command-line or a browser interface. This results in a membership-change request for a randomly chosen node which persists the change along with a timestamp (to keep a membership history) and propagates it via a Gossip-based protocol to the other nodes of the system. Every second, the membership protocol requires the storage hosts to randomly contact a peer in order to bilaterally reconcile the "persisted membership change histories" (cf. [DHJ+07, p. 212]). Doing so, membership changes are spread and an eventually consistent membership view is being established.

In the process described above logical partitions can occur temporarily when adding nodes to a Dynamo system. To prevent these, certain nodes can take the special role of seeds which can be discovered by an external mechanism such as static configuration or some configuration service. Seeds are consequently known to all nodes and will thus be contacted in the process of membership change gossiping so that all nodes will eventually reconcile their view of membership and "logical partitions are highly unlikely" (cf. [DHJ+07, p. 213]).

Once a node is joining the system, the number of virtual nodes getting mapped to the consistent hash ring has to be determined. For this purpose, a token is chosen for each virtual node, i. e. a value that determines the virtual node's position on the consistent hash ring. The set of tokens is persisted to disk on the physical node and spread via the same Gossip-based protocol that is used to spread to membership changes, as seen in the previous paragraph (cf. [DHJ+07, p. 212f]). The number of virtual nodes per physical node is chosen proportionally to the hardware resources of the physical node (cf. [DHJ+07, p. 210]).

By adding nodes to the system the ownership of key ranges on the consistent hash ring changes. When some node gets to know a recently added node via the membership propagation protocol and determines that it is no longer in charge of some portion of keys it transfers these to the node added. When a node is being removed, keys are being reallocated in a reverse process (cf. [DHJ+07, p. 213]).

With regards to the mapping of nodes on the consistent hash ring and its impact on load distribution, partition, balance, data placement, archival and bootstrapping, DeCandia et al. discuss three strategies implemented in Dynamo since its introduction:

1. T random tokens per node and partition by token value (the initial strategy)
2. T random tokens per node and equal sized partitions (an interim strategy as a migration path from strategy 1 to 3)
3. Q/S tokens per node and equal sized partitions (the current strategy; Q is the number of equal sized partitions on the consistent hash ring, S is the number of storage nodes)

For details on these strategies see [DHJ+07, p. 215–217]. Experiments and practice at Amazon show that "strategy 3 is advantageous and simpler to deploy" as well as more efficient and requiring the least amount of space to maintain membership information. According to DeCandia et al., the major benefits of this strategy are (cf. [DHJ+07, p. 217]):

- "Faster bootstrapping/recovery: Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery."
- "Ease of archival: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow."

However, a disadvantage of this strategy is that "changing the node membership requires coordination in order to preserve the properties required of the assignment" (cf. [DHJ+07, p. 217]).

Handling of Failures

For tolerating failures provoked by temporary unavailability storage hosts, Dynamo is not employing any strict quorum approach but a *sloopy* one. This approach implies that in the execution of read and write operations the first N *healthy* nodes of a data item's preference list are taken into account. These are not necessarily the first N nodes walking clockwise around the consistent hashing ring (cf. [DHJ+07, 212]).

A second measure to handle temporary unavailable storage hosts are so called *hinted handoffs*. They come into play if a node is note accessible during a write operation of a data item it is responsible for. In this case, the write coordinator will replicate the update to a different node, usually carrying no responsibility for this data item (to ensure durability on N nodes). In this replication request, the identifier of the node the update request was originally destined to is contained as a hint. As this node is recovering and becoming available again, it will receive the update; the node having received the update as a substitute can then delete it from its local database (cf. [DHJ+07, 212]).

DeCandia et al. remark that it is sufficient to address temporary node unavailability as permanent removal and addition of nodes is done explicitly as discussed in the prior subsection. In earlier versions of Dynamo a global view of node failures was established by an external failure detector that informed all nodes of a Dynamo ring. "Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests)", DeCandia et al. conclude (cf. [DHJ$^+$07, p. 213]).

In order to address the issue of entire datacenter outage, Dynamo is configured in a way to ensure storage in more than one data center. "In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers", DeCandia et al. remark (cf. [DHJ$^+$07, 212]). Data centers of Amazon's infrastructure are connected via high speed network links, so that latency due to replication between datacenters appears to be no issue.

In addition to the failure scenarios described above there may be threats to durability. These are addressed by implementing an anti-entropy protocol for replica synchronization. Dynamo uses Merkle-trees to efficiently detect inconsistencies between replicas and determine data to be synchronized. "A Merkle-tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children", DeCandia et al. explicate. This allows efficient checking for inconsistencies, as two nodes determine differences by hierarchically comparing hash-values of their Merkle-trees: firstly, by examining the tree's root node, secondly—if inconsistencies have been detected—by inspecting its child nodes, and so forth. The replica synchronization protocol is requiring little network bandwidth as only hash-values have to be transferred over the network. It can also operate fast as data is being compared in a divide-and conquer-manner by tree traversal. In Dynamo a storage node maintains a Merkle-tree for each key-range (i. e. range between two storage nodes on the consistent hash ring) it is responsible for and can compare such a Merkle-tree with those of other replicas responsible for the same key-range. DeCandia et al. consider a downside of this approach that whenever a node joins or leaves the system some Merkle-tree will get invalid due to changed key-ranges (cf. [DHJ$^+$07, p. 212]).

4.1.4. Implementation and Optimizations

In addition to the system design, DeCandia et al. make concrete remarks on the implementation of Dynamo. Based on experience at Amazon, they also provide suggestions for its optimization (cf. [DHJ$^+$07, p. 213f]):

- The code running on a Dynamo node consists of a request coordination, a membership and a failure detection component—each of them implemented in Java.
- Dynamo provides pluggable persistence components (e. g. Berkeley Database Transactional Data Store and BDB Java Edition [Ora10d], MySQL [Ora10c], an in-memory buffer with a persistent backing store). Experience from Amazon shows that "[applications] choose Dynamo's local persistence engine based on their object size distribution".
- The component in charge of coordinating requests is implemented "on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages". Internode communication employs Java NIO channels (see [Ora10b]).
- When a client issues a read or write request, the contacted node will become its coordinator. It then creates a state machine that "contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request."
- If a request coordinator receives a stale version of data from a node, it will update it with the latest version. This is called read-repair "because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it".

- To achieve even load-distribution write requests can address to "any of the top N nodes in the preference list".
- As an optimization reducing latency for both, read and write requests, Dynamo allows client applications to be coordinator of these operations. In this case, the state machine created for a request is held locally at a client. To gain information on the current state of storage host membership the client periodically contacts a random node and downloads its view on membership. This allows clients to determine which nodes are in charge of any given key so that it can coordinate read requests. A client can forward write requests to nodes in the preference list of the key to become written. Alternatively, clients can coordinate write requests locally if the versioning of the Dynamo instance is based on physical timestamps (and not on vector clocks). Both, the 99.9th percentile as well as the average latency can be dropped significantly by using client-controlled request coordination. This "improvement is because the client.driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node", DeCandia et al. conclude (cf. [DHJ+07, p. 217f]).
- As write requests are usually succeeding read requests, Dynamo is pursuing a further optimization: in a read response, the storage node replying the fastest to the read coordinator is transmitted to the client; in a subsequent write request, the client will contact this node. "This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency".
- A further optimization to reach Amazon's desired performance in the 99.9th percentile is the introduction of an object buffer in the main memory of storage nodes. In this buffer write requests are queued and persisted to disk periodically by a writer thread. In read operations both the object buffer and the persisted data of storage nodes have to be examined. Using an in-memory buffer that is being persisted asynchronously can result in data loss when a server crashes. To reduce this risk, the coordinator of the write request will choose one particular replica node to perform a *durable write* for the data item. This durable write will not impact the performance of the request as the coordinator only waits for $W - 1$ nodes to answer before responding to the client (cf. [DHJ+07, p. 215]).
- Dynamo nodes do not only serve client requests but also perform a number of background tasks. For resource division between request processing and background tasks a component named *admission controller* is in charge that "constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation". Monitoring includes disk I/O latencies, lock-contention and transaction timeouts resulting in failed database access as well as waiting periods in the request queue. Via monitoring feedback, the admission controller will decide on the number of time-slices for resource access or consumption to be given to background tasks. It also coordinates the execution of background tasks which have to explicitly apply for resources with the admission controller (cf. [DHJ+07, p. 218]).

4.1.5. Evaluation

To conclude the discussions on Amazon's Dynamo, a brief evaluation of Bob Ippolito's talk "Drop ACID and think about Data" shall be presented in table 4.2.

Advantages	Disadvantages
• "No master" • "Highly available for write" operations • "Knobs for tuning reads" (as well as writes) • "Simple"	• "Proprietary to Amazon" • "Clients need to be smart" (support vector clocks and conflict resolution, balance clusters) • "No compression" (client applications may compress values themselves, keys cannot be compressed) • "Not suitable for column-like workloads" • "Just a Key/Value store" (e. g. range queries or batch operations are not possible)

Table 4.2.: Amazon's Dynamo – Evaluation by Ippolito (cf. [Ipp09])

4.2. Project Voldemort

Project Voldemort is a key-/value-store initially developed for and still used at LinkedIn. It provides an API consisting of the following functions: (cf. [K+10b][4]):

- `get(key)`, returning a value object
- `put(key, value)`
- `delete(key)`

Both, keys and values can be complex, compound objects as well consisting of lists and maps. In the Project Voldemort design documentation it is discussed that—compared to relational databases—the simple data structure and API of a key-value store does not provide complex querying capabilities: joins have to be implemented in client applications while constraints on foreign-keys are impossible; besides, no triggers and views may be set up. Nevertheless, a simple concept like the key-/value store offers a number of advantages (cf. [K+10b]):

- Only efficient queries are allowed.
- The performance of queries can be predicted quite well.
- Data can be easily distributed to a cluster or a collection of nodes.
- In service oriented architectures it is not uncommon to have no foreign key constraints and to do joins in the application code as data is retrieved and stored in more than one service or datasource.
- Gaining performance in a relational database often leads to denormalized datastructures or storing more complex objects as BLOBs or XML-documents.

[4] Unless indicated otherwise, the information provided on Project Voldemort within this section has been taken from its official design documentation (cf. [K+10b]).

- Application logic and storage can be separated nicely (in contrast to relational databases where application developers might get encouraged to mix business logic with storage operation or to implement business logic in the database as stored procedures to optimize performance).
- There is no such impedance mismatch between the object-oriented paradigm in applications and paradigm of the datastore as it is present with relational databases.

4.2.1. System Architecture

Project Voldemort specifies a logical architecture consisting of a number of layers as depicted in figure 4.3.

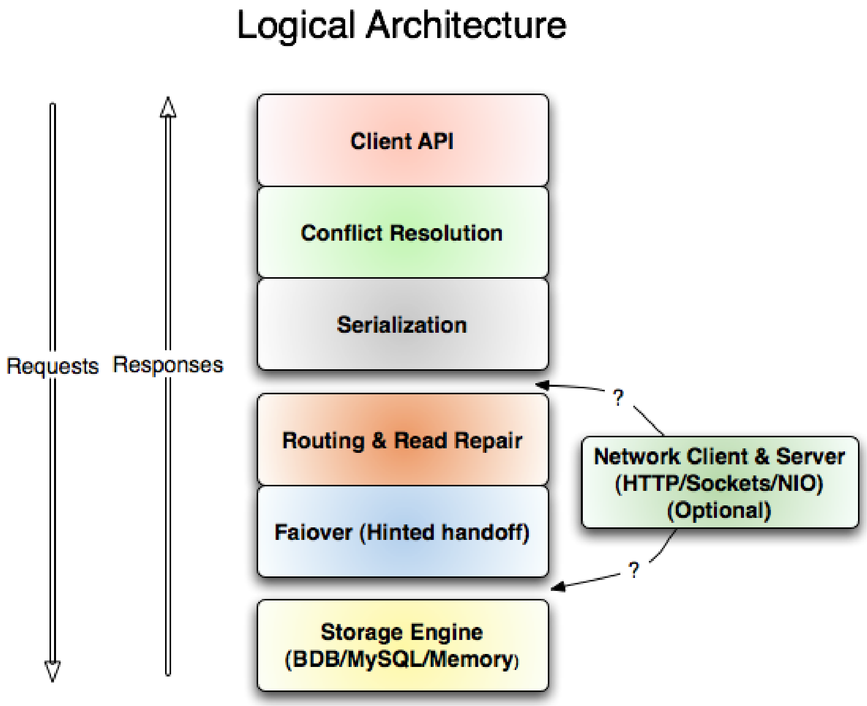


Figure 4.3.: Project Voldemort – Logical Architecture (taken from [K+10b])

Each layer of the logical architecture has its own responsibility (e. g. TCP/IP network communication, serialization, version recovery, routing between nodes) and also implements an interface consisting of the operations `get`, `put` and `delete`. These also operations exposed by a Project Voldemort instance as a whole. If e. g. the `put` operation is invoked on the routing layer it is responsible for distributing this operation to all nodes in parallel and for handling possible errors.

The layered logical architecture provides certain flexibility for deployments of Project Voldemort as layers can be mixed and matched to meet the requirements of an application. For example, a compression layer may be introduced beneath to the serialization layer in order to compress all exchanged data. Likewise, intelligent routing (i. e. determining the node which manages the partition containing the requested data) can be provided transparently by the datastore if the network layer is placed on top of the routing layer; if these layers are twisted, the application can do the routing itself reducing latency caused by network hops.

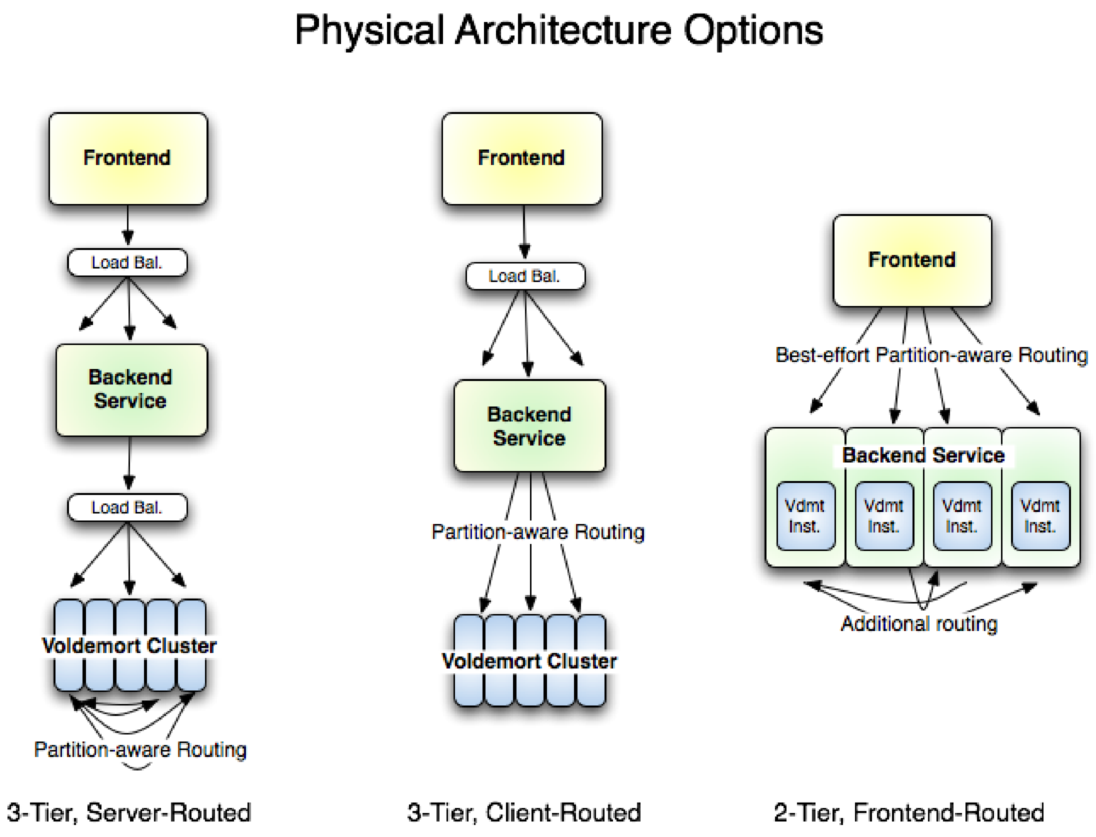


Figure 4.4.: Project Voldemort – Physical Architecture Options (taken from [K+10b])

Figure 4.4 depicts options for the physical deployment of Project Voldemort with a focus on routing and load-balancing.

- The left side illustrates a common three-tier architecture with two load-balancing components (either realized in hardware or software, e. g. in a round-robin process).
- The centre illustration avoids the load-balancer between the backend-service and the Voldemort cluster. While reducing latency, this approach tightens coupling. This is the case because the backend service has to determine the partitions of data and the node(s) they are residing on in order to route requests from the frontend ("partition-aware routing").
- In the right subfigure backend services even wrap Voldemort instances. The frontend is already trying to carry out partition-aware routing to enable high performant setups. This is the case as frontend requests are immediately directed to a backend-service containing a partition of the datastore (which might go wrong but then gets corrected by routing between the backend-services).

The trade-off depicted in figure 4.4 is reducing latency by minimizing network hops vs. strong coupling by routing-logic that moves up the software-stack towards the frontend[5].

Further examinations could be conducted about reducing the impact of disk I/O, the biggest performance killer in storage systems. The Project Voldemort design documentation suggests data partitioning as well as heavy caching to reduce performance bottlenecks caused by disk I/O, especially disk seek times and a low disk-cache efficiency. Therefore Project Voldemort—like Dynamo—uses consistent hashing with replication of data to tolerate downtimes of storage nodes.

[5] Voldemort provides a Java-library that can be used to do routing.

4.2.2. Data Format and Queries

Project Voldemort allows namespaces for key-/value-pairs called “stores“, in which keys are unique. While each key is associated with exactly one value, values are allowed to contain lists and maps as well as scalar values.

Operations in Project Voldemort are atomic to exactly one key-/value-pair. Once a get operation is executed, the value is streamed from the server via a cursor. Documentation of Project Voldemort considers this approach to not work very well in combination with values consisting of large lists “which must be kept on the server and streamed lazily via a cursor”; in this case, breaking the query into subqueries is seen as more efficient.

4.2.3. Versioning and Consistency

Like Amazon’s Dynamo Project Voldemort is designed to be highly available for write operations, allows concurrent modifications of data and uses vector clocks to allow casual reasoning about different versions (see sections 3.1.3 and 4.1.3). If the datastore itself cannot resolve version conflicts, client applications are requested for conflict resolution at read time. This read reconciliation approach is being favored over the strongly consistent but inefficient two-phase commit (2PC) approach as well as Paxos-style consensus protocols. This is the case because it requires little coordination and provides high availability and efficiency[6] as well as failure tolerance. On the downside, client applications have to implement conflict resolution logic that is not necessary in 2PC and Paxos-style consensus protocols.

4.2.4. Persistence Layer and Storage Engines

As indicated in figure 4.3, page 63, Project Voldemort provides pluggable persistency as the lowest layer of the logical architecture allows for different storage engines. Out of the box Berkeley DB (default storage engine), MySQL as well as in-memory storage are delivered with Project Voldemort. To use another existing or self-written storage engine, the operations `get`, `put` and `delete` besides an iterator for values will have to be implemented.

4.2.5. Data Model and Serialization

On the lowest level keys and values in Project Voldemort are simply byte-arrays. In order to allow applications a more sophisticated notion of keys and values, data models can be configured for each Voldemort store. These data models define serializers that are responsible to convert byte-arrays into the desired data structures and formats. Project Voldemort already contains serializers for the following data structures and formats:

JSON (JavaScript Object Notation) is a binary and typed data model which supports the data types list, map, date, boolean as well as numbers of different precision (cf. [Cro06]). JSON can be serialized to and deserialized from bytes as well as strings. By using the JSON data type it is possible to communicate with Project Voldemort instances in a human-readable format via administration tools like the Voldemort command line client.

[6] As an example, the Project Voldemort design documentation indicates the number of roundtrips needed for write-operations: W in case of read-repair compared to $2 * N$ for 2PC (with N as the number of nodes and W as the number of nodes that are required to successfully write an update); the number of roundtrips in Paxos-style consensus protocols varies for different implementations but is considered to be in the magnitude of 2PC.

String to store uninterpreted strings, which can also be used for XML blobs.

Java Serialization provided by Java classes implementing the `java.io.Serializable` interface (cf. [Ora10a], [Blo01, p. 213ff]).

Protocol Buffers are "Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data" which also contains an interface description language to generate code for custom data interchange. Protocol Buffers are widely used at Google "for almost all of its internal RPC protocols and file formats" (cf. [Goo10a], [Goo10b]).

Identity does no serialization or deserialization at all but simply hands over byte-arrays.

Project Voldemort can be extended by further custom serializers. In order to allow correct data interchange, they have to be made known to both, the data store logic and client applications.

The Project Voldemort design documentation makes further remarks with regards to the JSON format. It is considered to map well with numerous programming languages as it provides common data types (strings, numbers, lists, maps, objects) and does not have the object relational mapping problem of impedance mismatch. On the downside, JSON is schema-less internally which causes some issues for applications processing JSON data (e. g. data stores wishing to pursue fundamental checks of JSON-values). Each JSON document could contain an individual schema definition but this would be very wasteful considering that in a datastore a large amount of data shares the same structure. Project Voldemort therefore offers the possibility to specify data formats for keys and values, as listed in table 4.3.

Type	**Storable Subtypes**	**Bytes used**	**Java-Type**	**JSON Example**	**Definition Example**
number	int8, int16, int32, int64, float32, float64, date	8, 16, 32, 64, 32, 64, 32	Byte, Short, Integer, Long Float, Double, Date	1	"int32"
string	string, bytes	2 + length of string or bytes	String, byte[]	"hello"	"string"
boolean	boolean	1	Boolean	true	"boolean"
object	object	1 + size of contents	Map<String, Object>	{"key1":1, "key2":"2", "key3":false}	{"name":"string", "height":"int16"}
array	array	size * sizeof(type)	List<?>	[1, 2, 3]	["int32"]

Table 4.3.: Project Voldemort – JSON Serialization Format Data Types (taken from [K+10b])

The data type definition for the JSON serialization format allows Project Voldemort to check values and store them efficiently, albeit the data types for values cannot be leveraged for data queries and requests. To prevent invalidated data caused by redefinition of value data types, Project Voldemort is storing a version along with the data allowing schema migrations.

4.2.6. Index Precalculation

Project Voldemort allows to prebuild indexes offline (e. g. on Hadoop), upload them to the datastore and transparently swap to them. This is especially useful for batch operations inserting or updating large amounts of data causing index rebuilds if the data is uploaded as a whole or index fragmentation if it is inefficiently inserted in small portions. In Project Voldemort large amounts of data can be inserted as a whole and the offline index building feature disburdens live systems from full index rebuilding or index fragmentation.

4.3. Other Key-/Value-Stores

4.3.1. Tokyo Cabinet and Tokyo Tyrant

This datastore builds on a collection of software components out of which the Tokyo Cabinet and Tokyo Tyrant are the most important in this context. Tokyo Cabinet ([FAL10a]) is the core library of this datastore persisting data and exposing a key-/value-interface to clients and thereby abstracting from internal data structures such as hash-tables or B+tree-indexes. Tokyo Tyrant ([FAL10b]) provides access to this database library via network which is possible via a proprietary binary protocol, HTTP as well as through the memcached protocol. Tokyo Cabinet manages data storage on disk and in memory in a fashion similar to paging / swapping. Memory pages are flushed to disk periodically "which leaves an open data loss hole", as Hoff comments (cf. [Hof09a]). On the other side, Tokyo Cabinet allows to compress pages by the LZW-algorithm which can achieve good compression ratios (see e. g. [Lin09]). Tokyo Cabinet does partition data automatically and therefore has to be replicated with a strategy similar to MySQL. In addition to lookups by key it can match prefixes and ranges if keys are ordered. Regarding transactional features worth mentioning Tokyo Cabinet provides write-ahead logging and shadow paging. The Toyko suite is developed actively, well documented and and widely regarded as high-performant: 1 million records can be stored in 0.7 seconds using the hash-table engine and in 1.6 seconds using the b-tree according to North (cf. [Nor09], [Ipp09], [See09], [Lin09], [Hof09a]).

4.3.2. Redis

Redis is a relatively new datastore which its developers unspecifically refer to as a "data structure store"; it is commonly subsumed under key-/value-stores because of its map/dictionary-API. Special about Redis is that it allows matching for key-ranges e. g. matching of numeric ranges or regular expressions. In contrast to other key-/value-stores, Redis does not only store bytes as values but also allows lists and sets in values by supporting them directly. A major disadvantage about Redis is that the amount of main memory limits the amount of data that is possible to store. This cannot be expanded by the usage of hard-disks. Ippolito comments that for this reason Redis is probably a good fit for a caching-layer (cf. [Ipp09]).

4.3.3. Memcached and MemcacheDB

Memcached, the popular and fast memory-caching solution widely used among large and ultra-large scale web sites to reduce database-load, has already been mentioned in section 3.2 on partitioning. Memcached servers—like the key-/value-stores discussed in this chapter—provide a map/dictionary API consisting of the operations `get`, `put` and `remove`. Though not intended for persistent storage (cf. [F^+10b]) there is an existing solution named MemcacheDB (cf. [Chu09]) that conforms to the memcached protocol (cf. [F^+10c]) and adds persistence based on Berkeley DB to it (cf. [Nor09], [Ipp09]). As memcached does not provide any replication between nodes and is also not tolerant towards machine failures, simply adding a persistent storage to it is not enough, as blogger Richard Jones of Last.fm remarks. He notes that solutions like repcached can replicate whole memcached servers (in a master slave setup) but without fault-tolerant partitioning they will cause management and maintenance efforts (cf. [Jon09]).

4.3.4. Scalaris

Scalaris is a key-/value-store written in Erlang taking profit of this language's approach e. g. in implementing a non-blocking commit protocol for atomic transactions. Scalaris uses an adapted version of the chord service (cf. [SMK$^+$01]) to expose a distributed hash table to clients. As it stores keys in lexicographic order, range queries on prefixes are possible. In contrast to other key-/value-stores Scalaris has a strict consistency model, provides symmetric replication and allows for complex queries (via programming language libraries). It guarantees ACID properties also for concurrent transactions by implementing an adapted version of the Paxos consensus protocol (cf. [Lam98]). Like memcached, Scalaris is a pure in-memory key-/value-store (cf. [Nor09], [Jon09]).

5. Document Databases

In this chapter another class of NoSQL databases will be discussed. Document databases are considered by many as the next logical step from simple key-/value-stores to slightly more complex and meaningful data structures as they at least allow to encapsulate key-/value-pairs in documents. On the other hand there is no strict schema documents have to conform to which eliminates the need schema migration efforts (cf. [Ipp09]). In this chapter Apache CouchDB and MongoDB as the two major representatives for the class of document databases will be investigated.

5.1. Apache CouchDB

5.1.1. Overview

CouchDB is a document database written in Erlang. The name CouchDB is nowadays sometimes referred to as "Cluster of unreliable commodity hardware" database, which is in fact a backronym according to one of it's main developers (cf. [PLL09]).

CouchDB can be regarded as a descendant of Lotus Notes for which CouchDB's main developer Damien Katz worked at IBM before he later initiated the CouchDB project on his own[1]. A lot of concepts from Lotus Notes can be found in CouchDB: documents, views, distribution, and replication between servers and clients. The approach of CouchDB is to build such a document database from scratch with technologies of the web area like Representational State Transfer (REST; cf. [Fie00]), JavaScript Object Notation (JSON) as a data interchange format, and the ability to integrate with infrastructure components such as load balancers and caching proxies etc. (cf. [PLL09]).

CouchDB can be briefly characterized as a document database which is accessible via a RESTful HTTP-interface, containing schema-free documents in a flat address space. For these documents JavaScript functions select and aggregate documents and representations of them in a MapReduce manner to build views of the database which also get indexed. CouchDB is distributed and able to replicate between server nodes as well as clients and servers incrementally. Multiple concurrent versions of the same document (MVCC) are allowed in CouchDB and the database is able to detect conflicts and manage their resolution which is delegated to client applications (cf. [Apa10c], [Apa10a]).

The most notable use of CouchDB in production is *ubuntu one* ([Can10a]) the cloud storage and replication service for Ubuntu Linux ([Can10b]). CouchDB is also part of the BBC's new web application platform (cf. [Far09]). Furthermore some (less prominent) blogs, wikis, social networks, Facebook apps and smaller web sites use CouchDB as their datastore (cf. [C$^+$10]).

[1] Advocates therefore call it "Notes done right" sometimes.

5.1.2. Data Model and Key Abstractions

Documents

The main abstraction and data structure in CouchDB is a document. Documents consist of named fields that have a key/name and a value. A fieldname has to be unique within a document and its assigned value may a string (of arbitrary length), number, boolean, date, an ordered list or an associative map (cf. [Apa10a]). Documents may contain references to other documents (URIs, URLs) but these do not get checked or held consistent by the database (cf. [PLL09]). A further limitation is that documents in CouchDB cannot be nested (cf. [Ipp09]).

A wiki article may be an example of such a document:

```
"Title" : "CouchDB",
"Last editor" : "172.5.123.91",
"Last modified": "9/23/2010",
"Categories": ["Database", "NoSQL", "Document Database"],
"Body": "CouchDB is a ...",
"Reviewed": false
```

Besides fields, documents may also have attachments and CouchDB maintains some metadata such as a unique identifier and a sequence id[2]) for each document (cf. [Apa10b]). The document id is a 128 bit value (so a CouchDB database can store 3.4^{38} different documents) ; the revision number is a 32 bit value determined by a hash-function[3].

CouchDB considers itself as a semi-structured database. While relational databases are designed for structured and interdependent data and key-/value-stores operate on uninterpreted, isolated key-/value-pairs document databases like CouchDB pursue a third path: data is contained in documents which do not correspond to a fixed schema (schema-free) but have some inner structure known to applications as well as the database itself. The advantages of this approach are that first there is no need for schema migrations which cause a lot of effort in the relational databases world; secondly compared to key-/value-stores data can be evaluated more sophisticatedly (e. g. in the calculation of views). In the web application field there are a lot of document-oriented applications which CouchDB addresses as its data model fits this class of applications and the possibility to iteratively extend or change documents can be done with a lot less effort compared to a relational database (cf. [Apa10a]).

Each CouchDB database consists of exactly one flat/non-hierarchical namespace that contains all the documents which have a unique identifier (consisting of a document id and a revision number aka sequence id) calculated by CouchDB. A CouchDB server can host more than one of these databases (cf. [Apa10c], [Apa10b]). Documents were formerly stored as XML documents but today they are serialized in a JSON-like format to disk (cf. [PLL09]).

Document indexing is done in B-Trees which are indexing the document's id and revision number (sequence id; cf. [Apa10b]).

[2] The sequence id is a hash value that represents a revision number of the document and it will be called revision number in the discussion of CouchDB here.

[3] CouchDB assumes that 32 bit are enough to not let hash values of revision numbers collide and therefore does in fact not implement any means to prevent or handle such collisions (cf. [PLL09]).

Views

CouchDBs way to query, present, aggregate and report the semi-structured document data are views (cf. [Apa10a], [Apa10b]). A typical example for views is to separate different types of documents (such as blog posts, comments, authors in a blog system) which are not distinguished by the database itself as all of them are just documents to it ([PLL09]).

Views are defined by JavaScript functions which neither change nor save or cache the underlying documents but only present them to the requesting user or client application. Therefore documents as well as views (which are in fact special documents, called *design-documents*) can be replicated and views do not interfere with replication. Views and are calculated on demand. There is no limitation regarding the number of views in one database or the number of representations of documents by views.

The JavaScript functions defining a view are called `map` and `reduce` which have similar responsibilities as in Google's MapReduce approach (cf. [DG04]). The `map` function gets a document as a parameter, can do any calculation and may emit arbitrary data for it if it matches the view's criteria; if the given document does not match these criteria the `map` function emits nothing. Examples of emitted data for a document are the document itself, extracts from it, references to or contents of other documents (e. g. semantically related ones like the comments of a user in a forum, blog or wiki).

The data structure emitted by the `map` function is a triple consisting of the document id, a key and a value which can be chosen by the `map` function. Documents get sorted by the key which does not have to be unique but can occur for more than one document; the key as a sorting criteria can be used to e. g. define a view that sorts blog posts descending by date for a blog's home page. The value emitted by the `map` function is optional and may contain arbitrary data. The document id is set by CouchDB implicitly and represents the document that was given to the emitting `map` function as an argument (cf. [PLL09]).

After the `map` function has been executed it's results get passed to an optional `reduce` function which is optional but can do some aggregation on the view (cf. [PLL09]).

As all documents of the database are processed by a view's functions this can be time consuming and resource intensive for large databases. Therefore a view is not created and indexed when write operations occur[4] but on demand (at the first request directed to it) and updated incrementally when it is requested again[5]. To provide incremental view updates CouchDB holds indexes for views. As mentioned before views are defined and stored in special documents. These design documents can contain functions for more than one view if they are named uniquely. View indexes are maintained on based on these design documents and not single views contained in them. Hence, if a user requests a view it's index and the indexes of all views defined in the same design document get updated (cf. [Apa10b], [PLL09]). Incremental view updates furthermore have the precondition that the `map` function is required to be referentially transparent which means that for the same document it has to emit the same key and value each time it is invoked (cf. [Apa10e]).

To update a view, the component responsible for it (called view-builder) compares the sequence id of the whole database and checks if it has changed since the last refresh of the view. If not, the view-builder determines the documents changed, deleted or created since that time; it passes new and updated documents to the view's `map` and `reduce` functions and removes deleted documents from the view. As changes to the database are written in an append-only fashion to disk (see subsection 5.1.7), the incremental updates of views can occur efficiently as the number of disk head seeks is minimal. A further advantage of the append-only index persistence is that system crashes during the update of indexes the previous state

[4] This means in CouchDB there is no "write-penalty", as some call it (cf. [PLL09]).

[5] One might want to precalculate an index if e. g. bulk-updates or inserts have occurred in order not to punish the next client requesting it. In this case, one can simply request the view to be its next reader (cf. [PLL09]).

remains consistent, CouchDB omits the incompletely appended data when it starts up and can update an index when it is requested the next time.

While the view-builder is updating a view data from the view's old state can be read by clients. It is also possible to present the old state of the view to one client and the new one to another client as view indexes are also written in an append-only manner and the compactation of view data does not omit an old index state while a client is still reading from it (more on that in subsection 5.1.7).

5.1.3. Versioning

Documents are updated optimistically and update operations do not imply any locks (cf. [Apa10b]). If an update is issued by some client the contacted server creates a new document revisions in a copy-on-modify manner (see section 3.3) and a history of recent revisions is stored in CouchDB until the database gets compacted the next time. A document therefore is identified by a document id/key that sticks to it until it gets deleted and a revision number created by CouchDB when the document is created and each time it is updated (cf. [Apa10b]). If a document is updated, not only the current revision number is stored but also a list of revision numbers preceding it to allow the database (when replicating with another node or processing read requests) as well as client applications to reason on the revision history in the presence of conflicting versions (cf. [PLL09]).

CouchDB does not consider version conflicts as an exception but rather a normal case. They can not only occur by different clients operating on the same CouchDB node but also due to clients operating on different replicas of the same database. It is not prohibited by the database to have an unlimited number of concurrent versions. A CouchDB database can deterministically detect which versions of document succeed each other and which are in conflict and have to be resolved by the client application. Conflict resolution may occur on any replica node of a database as the node which receiving the resolved version transmits it to all replicas which have to accept this version as valid. It may occur that conflict resolution is issued on different nodes concurrently; the locally resolved versions on both nodes then are detected to be in conflict and get resolved just like all other version conflicts (cf. [Apa10b]).

Version conflicts are detected at read time and the conflicting versions are returned to the client which is responsible for conflict resolution. (cf. [Apa10b]).

A document which's most recent versions are in conflict is excluded from views (cf. [Apa10b]).

5.1.4. Distribution and Replication

CouchDB is designed for distributed setups that follows a peer-approach where each server has the same set of responsibilities and there are no distinguished roles (like in master/slave-setups, standby-clusters etc.). Different database nodes can by design operate completely independent and process read and write requests. Two database nodes can replicate databases (documents, document attachments, views) bilaterally if they reach each other via network. The replication process works incrementally and can detect conflicting versions in simple manner as each update of a document causes CouchDB to create a new revision of the updated document and a list of outdated revision numbers is stored. By the current revision number as well as the list of outdated revision number CouchDB can determine if are conflicting or not; if there are version conflicts both nodes have a notion of them and can escalate the conflicting versions to clients for conflict resolution; if there are no version conflicts the node not having the most recent version of the document updates it (cf. [Apa10a], [Apa10b], [PLL09])

Distribution scenarios for CouchDB include clusters, offline-usage on a notebook (e. g. for employees visiting customers) or at company locations distributed over the world where live-access to a company's or organization's local network is slow or unstable. In the latter two scenarios one can work on a disconnected CouchDB instance and is not limited in its usage. If the network connection to replica nodes is established again the database nodes can synchronize their state again (cf. [Apa10b]).

The replication process operates incrementally and document-wise. Incrementally means that only data changed since the last replication gets transmitted to another node and that not even whole documents are transferred but only changed fields and attachment-blobs; document-wise means that each document successfully replicated does not have to be replicated again if a replication process crashes (cf. [Apa10b]).

Besides replicating whole databases CouchDB also allows for partial replicas. For these a JavaScript filter function can be defined which passes through the data for replication and rejects the rest of the database (cf. [Apa10b]).

This partial replication mechanism can be used to shard data manually by defining different filters for each CouchDB node. If this is not used and no extension like Lounge (cf. [FRLL10]) CouchDB replicates all data to all nodes and does no sharding automatically and therefore by default behaves just like MySQLs replication, as Ippolito remarks (cf. [Ipp09]).

According to the CouchDB documentation the replication mechanisms are designed to distribute and replicate databases with little effort. On the other hand they should also be able to handle extended and more elaborated distribution scenarios e. g. partitioned databases or databases with full revision history. To achieve this "[the] CouchDB replication model can be modified for other distributed update models". As an example the storage engine can be "enhanced to allow multi-document update transactions" to make it possible "to perform Subversion-like "all or nothing" atomic commits when replicating with an upstream server, such that any single document conflict or validation failure will cause the entire update to fail" (cf. [Apa10b]; more information on validation can be found below).

5.1.5. Interface

CouchDB databases are addressed via a RESTful HTTP interface that allows to read and update documents (cf. [Apa10b]). The CouchDB project also provides libraries providing convenient access from a number of programming languages as well as a web administration interface (cf. [Apa10c]).

CouchDB documents are requested by their URL according to the RESTful HTTP paradigm (read via HTTP GET, created and updated via HTTP PUT and deleted via HTTP DELETE method). A read operation has to go before an update to a document as for the update operation the revision number of the document that has been read and should be updated has to be provided as a parameter. To retrieve document urls—and maybe already their data needed in an application[6]—views can be requested by client applications (via HTTP GET). The values of view entries can be retrieved by requesting their document id and key (cf. [PLL09]). Documents as well as views are exchanged via the JSON format (cf. [Ipp09]).

Lehnhardt and Lang point out that that by providing RESTful HTTP interface many standard web infrastructure components like load-balancers, caches, SSL proxies and authentication proxies can be easily integrated with CouchDB deployments. An example of that is to use the revision number of a document as an ETag header (cf. [FGM+99, section 14.19]) on caching servers. Web infrastructure components may also hide distribution aspects from client applications—if this is required or chosen to keep applications simpler though offering performance and the possibility to leverage from a notion of the distributed database (cf. [PLL09]).

[6] As mentioned in the subsection on views they may contain arbitrary key-/value-pairs associated to a document and therefore can be designed to contain all data the client application needs for a certain use case.

5.1.6. ACID Properties

According to the technical documentation ACID properties can be attributed to CouchDB because of its commitment system and the way it operates on files (cf. [Apa10b]).

Atomicity is provided regarding single update operations which can either be executed to completion or fail and are rolled back so that the database never contains partly saved or updated documents (cf. [Apa10b]).

Consistency can be questioned as it cannot mean strong consistency in a distributed CouchDB setup as all replicas are always writable and do not replicate with each other by themselves. This leads to a MVCC system in which version conflicts have to be resolved at read time by client applications if no syntactic reconciliation is possible.

Consistency of databases on single CouchDB nodes as well as durability is ensured by the way CouchDB operates on database files (see below; cf. [Apa10b]).

5.1.7. Storage Implementation

Considering the storage implementation CouchDB applies some methods guaranteeing consistency and durability as well as optimizing performance (cf. [Apa10b]):

- CouchDB never overwrites committed data or associated structures so that a database file is in a consistent state at each time. Hence, the database also does not need a shutdown to terminate correctly but its process can simply be killed.
- Document updates are executed in two stages to provide transactions while maintaining consistency and durability of the database:
    1. The updated documents are serialized to disk synchronously. An exception to this rule is BLOB-data in document which gets written concurrently.
    2. The updated database header is written in two identical and subsequent chunks to disk.

  Now, if the system crashes while step 1 is executed, the incompletely written data will be ignored when CouchDB restarts. If the system goes down in step 2 there is chance that one of the two identical database headers is already written; if this is not the case, the inconsistency between database headers and database contents is discovered as CouchDB checks the database headers for consistency when it starts up. Besides these check of database headers there are no further checks or purges needed.
- Read requests are never blocked, never have to wait for a reader or writer and are never interrupted by CouchDB. It is guaranteed for a reading client to see a consistent snapshot of the database from the beginning to the end of a read operation.
- As mentioned above documents of CouchDB databases are indexed in B-Trees. On update operations on documents new revision numbers (sequence-ids) for the updated documents are generated, indexed and the updated index is written in an append-only way to disk.
- To store documents efficiently a document and its metadata is combined in a so called buffer first then written to disk sequentially. Hence, documents can be read by clients and the database (for e. g. indexing purposes, view-calculation) efficiently in one go.
- As mentioned before view indexes are written in an append-only manner just like document indexes.

- CouchDB needs to compact databases from time to time to gain disk space back that is no longer needed. This is due to the append-only database and index-files as well as the document revision history. Compactation can either be scheduled or is implicitly executed when the "wasted" space exceeds a certain threshold. The compactation process clones all database contents that are still needed and copies it to a new database file. During the copying process the old database file remains so that copy and even update operations can still be executed. If the system crashes during copying the old database file is still present and integer. The copying process is considered successful as soon as all data has been transferred to the new database file and all requests have been redirected to it; then CouchDB omits the old database file.

5.1.8. Security

Access Control

CouchDB implements a simple access control model: a document may have a list of authorized roles (called "readers") allowed to read it. A user may be associated to zero, one or more of these roles and it is determined which roles he has when he accesses the database. If documents have a reader list associated they can only be read if an accessing user owns the role of one of these readers. Documents with reader lists that are returned by views also get filtered dynamically when the view's contents are returned to a user (cf. [Apa10b]).

Administrator Privileges

There is a special role of an administrator in CouchDB who is allowed to create and administer user accounts as well as to manipulate design documents (e. g. views; cf. [Apa10b]).

Update Validation

Documents can be validated dynamically before they get written to disk to ensure security and data validity. For this purpose JavaScript functions can be defined that take a document and the credentials of the logged in user as parameters and can return a negative value if they do not consider the document should be written to disk. This leads CouchDB to an abort of the update request and an error message gets returned to the client that issued the update. As the validation function can be considered quite generic custom security models that go beyond data validation can be implemented this way.

The update validation functions are executed for updates occurring in live-usage of a CouchDB instance as well as for updates due to replication with another node (cf. [Apa10b]).

5.1.9. Implementation

CouchDB was originally implemented in C++ but for concurrency reasons later ported to the Erlang OTP (Open Telecommunication Platform; cf. [Eri10]), a functional, concurrent language and platform originally developed with a focus on availability and reliability in the telecommunications sector (by Ericsson). The peculiarities of the Erlang language are lightweight processes, an asynchronous and message-based approach to concurrency, no shared-state threading as well as immutability of data. The CouchDB implementation profits from these characteristics as Erlang was chosen to implement completely lock-free concurrency for read and write requests as well as replication in order to reduce bottlenecks and keep the database working predictably under high load (cf. [Apa10a], [Apa10b]).

To furthermore provide for high availability and allow large scale concurrent access CouchDB uses a shared-noting clustering approach which lets all replica nodes of a database work independently even if they are disconnected (see the subsection on views above and [Apa10b]).

Although the database itself is implemented in Erlang two libraries written in C are used by CouchDB the *IBM Components for Unicode* as well as Mozilla's *Spidermonkey* JavaScript Engine (cf. [Apa10a]).

5.1.10. Futher Notes

CouchDB allows to create whole applications in the database and document attachments may consist of HTML, CSS and JavaScript (cf. [Apa10b], [PLL09]).

CouchDB has a mechanism to react on server side events. For that purpose one can register for events on the server and provide JavaScript code that processes these events (cf. [PLL09]).

CouchDB allows to provide JavaScript transformation functions for documents and views that can be used for example to create non-JSON representations of them (e.g. XML documents; cf. [PLL09]).

Some projects have been emerged around CouchDB which extend it with additional functionality, most notably a fulltext search and indexing integration with Apache Lucene (cf. [N+10], [Apa09]) and the clustering framework Lounge (cf. [FRLL10]).

5.2. MongoDB

5.2.1. Overview

MongoDB is a schema-free document database written in C++ and developed in an open-source project which is mainly driven by the company 10gen Inc that also offers professional services around MongoDB. According to its developers the main goal of MongoDB is to close the gap between the fast and highly scalable key-/value-stores and feature-rich traditional RDBMSs relational database management systems. MongoDBs name is derived from the adjective *humongous* (cf. [10g10]). Prominent users of MongoDB include SourceForge.net, foursquare, the New York Times, the URL-shortener bit.ly and the distributed social network DIASPORA* (cf. [Cop10], [Hey10], [Mah10], [Rid10], [Ric10]).

5.2.2. Databases and Collections

MongoDB databases reside on a MongoDB server that can host more than one of such databases which are independent and stored separately by the MongoDB server. A database contains one or more collections consisting of documents. In order to control access to the database a set of security credentials may be defined for databases (cf. [CB10]).

Collections inside databases are referred to by the MongoDB manual as "named groupings of documents" (cf. [CBH10a]). As MongoDB is schema-free the documents within a collection may be heterogeneous although the MongoDB manual suggests to create "one database collection for each of your top level objects" (cf. [MMM+10b]). Once the first document is inserted into a database, a collection is created automatically and the inserted document is added to this collection. Such an implicitly created collection gets configured with default parameters by MongoDB—if individual values for options such as auto-indexing, preallocated disk space or size-limits (cf. [MDM+10] and see the subsection on *capped collections* below) are demanded, collections may also be created explicitly by the createCollection-command[7]:

```
db.createCollection(<name>, {<configuration parameters>})
```

As an example, the following command creates a collection named mycoll with 10,000,000 bytes of preallocated disk space and no automatically generated and indexed document-field _id:

```
db.createCollection("mycoll", {size:10000000, autoIndexId:false});
```

MongoDB allows to organize collections in hierarchical namespaces using a dot-notation, e. g. the collections wiki.articles, wiki.categories and wiki.authors residing under the namespace wiki. The MongoDB manual notes that "this is simply an organizational mechanism for the user -- the collection namespace is flat from the database's perspective" (cf. [CBH10a]).

5.2.3. Documents

The abstraction and unit of data storable in MongoDB is a document, a data structure comparable to an XML document, a Python dictionary, a Ruby hash or a JSON document. In fact, MongoDB persists documents by a format called BSON which is very similar to JSON but in a binary representation for reasons of efficiency and because of additional datatypes compared to JSON[8]. Nonetheless, "BSON maps readily to and from JSON and also to various data structures in many programming languages" (cf. [CBH+10b]).

As an example, a document representing a wiki article[9] may look like the following in JSON notation:

```
{
  title: "MongoDB",
  last_editor: "172.5.123.91",
  last_modified: new Date("9/23/2010"),
  body: "MongoDB is a...",
  categories: ["Database", "NoSQL", "Document Database"] ,
  reviewed: false
}
```

To add such a document into a MongoDB collection the insert function is used:

```
db.<collection>.insert( { title: "MongoDB", last_editor: ... } );
```

Once a document is inserted it can be retrieved by matching queries issued by the find operation and updated via the save operation:

```
db.<collection>.find( { categories: [ "NoSQL", "Document Databases" ] } );
db.<collection>.save( { ... } );
```

[7] All syntax examples in this section are given in JavaScript which is used by the interactive MongoDB shell.

[8] As documents are represented as BSON objects when transmitted to and persisted by a MongoDB server, the MongoDB manual also uses the term *object* for them.

[9] This example contains the same information as the wiki article exemplified in the CouchDB section.

Documents in MongoDB are limited in size by 4 megabytes (cf. [SM10]).

Datatypes for Document Fields

MongoDB provides the following datatypes for document fields (cf. [CHM10a], [MC09], [MMM+10a]):

- scalar types: **boolean**, **integer**, **double**
- character sequence types: **string** (for character sequences encoded in UTF-8), **regular expression**, **code** (JavaScript)
- **object** (for BSON-objects)
- **object id** is a data type for 12 byte long binary values used by MongoDB and all officially supported programming language drivers for a field named `_id` that uniquely identifies documents within collections[10]. The object id datatype is composed of the following components:
  - timestamp in seconds since epoch (first 4 bytes)
  - id of the machine assigning the object id value (next 3 bytes)
  - id of the MongoDB process (next 2 bytes)
  - counter (last 3 bytes)

  For documents whose `_id` field has an object id value the timestamp of their creation can be extracted by all officially supported programming drivers.
- **null**
- **array**
- **date**

References between Documents

MongoDB does not provide a foreign key mechanism so that references between documents have to be resolved by additional queries issued from client applications. References may be set manually by assigning some reference field the value of the `_id` field of the referenced document. In addition, MongoDB provides a more formal way to specify references called DBRef ("Database Reference"). The advantages of using DBRefs are that documents in other collections can be referenced by them and that some programming language drivers dereference them automatically (cf. [MDC+10], [MMM+10d, Database References]). The syntax for a DBRef is:

```
{ $ref : <collectionname>, $id : <documentid>[, $db : <dbname>] }
```

[10] The `_id`-field is not required to be an object id as all other data types are also allowed for object id values—provided that the `_id`-value is unique within a collection. However, MongoDB itself and all officially supported programming language bindings generate and assign an instance of object id if no id value is explicitly defined on document creation.

The MongoDB manual points out that the field `$db` is optional and not supported by many programming language drivers at the moment (cf. [MDC+10]).

The MongoDB points out that although references between documents are possible there is the alternative to nest documents within documents. The embedding of documents is "much more efficient" according to the MongoDB manual as "[data] is then colocated on disk; client-server turnarounds to the database are eliminated". Instead when using references, "each reference traversal is a query to the database" which results at least in an addition of latency between the web or application server and the database but typically more as the referenced data is typically not cached in RAM but has to be loaded from disk.

The MongoDB manual gives some guidance when to reference an object and when to embed it as presented in table 5.1 (cf. [MMM+10b]).

Criteria for Object References	**Criteria for Object Embeddings**
• First-class domain objects (typically residing in a separate collection) • Many-to-many reference between objects • Objects of this type are often queried in large numbers (request all / the first n objects of a certain type) • The object is large (multiple megabytes)	• Objects with "line-item detail" characteristic • Aggregation relationship between object and host object • Object is not referenced by another object (DBRefs for embedded objects are not possible as of MongoDB, version 1.7) • Performance to request and operate on the object and its host-object is crucial

Table 5.1.: MongoDB – Referencing vs. Embedding Objects (cf. [MMM+10b])

5.2.4. Database Operations

Queries

Selection Queries in MongoDB are specified as *query objects*, BSON documents containing selection criteria[11], and passed as a parameter to the `find` operation which is executed on the collection to be queried (cf. [CMH+10]):

```
db.<collection>.find( { title: "MongoDB" );
```

The selection criteria given to the `find` operation can be seen as an equivalent to the `WHERE` clause in SQL statements[12] (cf. [Mer10f]). If the query object is empty, all documents of a collection are returned.

[11] The examples in this section are in given JavaScript as it can be used in the interactive MongoDB shell. Thus, the query objects are represented in JSON and transformed by the JavaScript language driver to BSON when sent to the database.

[12] The MongoDB manual provides an "SQL to Mongo Mapping Chart" which helps to get familiar with the document database approach and MongoDB's query syntax, cf. [MKB+10].

In the selection criteria passed to the `find` operation a lot of operators are allowed—besides equality comparisons as in the example above. These have the following general form:

```
<fieldname>: {$<operator>: <value>}
<fieldname>: {$<operator>: <value>, $<operator>: value} // AND-junction
```

The following comparison operators are allowed (cf. [MMM+10c]):

- Non-equality: `$ne`
- Numerical Relations: `$gt`, `$gte`, `$lt`, `$lte` (representing $>$, $\geq$, $<$, $\leq$)
- Modulo with divisor and the modulo compare value in a two-element array, e. g.

  ```
  { age: {$mod: [2, 1]} } // to retrieve documents with an uneven age
  ```

- Equality-comparison to (at least) one element of an array: `$in` with an array of values as comparison operand, e. g.

  ```
  { categories: {$in: ["NoSQL", "Document Databases"]} }
  ```

- Non-equality-comparison to all elements of an array: `$nin` with an array of values as comparison operand
- Equality-comparison to all elements of an array: `$all`, e. g.

  ```
  { categories: {$all: ["NoSQL", "Document Databases"]} }
  ```

- Size of array comparison: `$size`, e. g.

  ```
  { categories: {$size: 2} }
  ```

- (Non-)Existence of a field: `$exists` with the parameter `true` or `false`, e. g.

  ```
  { categories: { $exists: false}, body: {$exists: true} }
  ```

- Field type: `$type` with a numerical value for the BSON data typ (as specified in the MongoDB manual, cf. [MMM+10c, Conditional Operators – $type])

Logical junctions can be specified in query objects as follows:

- Comparison expressions that are separated by comma specify an **AND-junction**.
- **OR-junctions** can be defined by the special `$or` operator that is assigned to an array of booleans or expressions, each of which can satisfy the query. As an example, the following query object matches documents that either are reviewed or have exactly two categories assigned:

  ```
  { $or: [ {reviewed: {$exists: true} }, {categories: {$size: 2} } ] }
  ```

- To express **NOR-junctions** the `$nor` operator is supported by MongoDB. Like `$or` it is assigned to an array of expressions or boolean values.
- Via the `$not` operator a term can be **negated**, e. g.

```
{ $not: {categories: {$in: {"NoSQL"}}} } // category does not contain "NoSQL"
{ $not: {title: /^Mongo/i} }             // title does not start with "Mongo"
```

The following remarks are made by the MongoDB manual regarding certain field types (cf. [MMM+10c]):

- Besides the abovementioned comparison operators it is also possible to do (PCRE[13]) regular expression matching for string fields. If the regular expression only consists a prefix check (`/^prefix/ [modifiers]` equivalent to SQL's `LIKE ’prefix%’`) MongoDB uses indexes that are possibly defined on that field.
- To search for a single value inside an array the array field can simply be assigned to the desired value in the query object, e. g.

```
{ categories : "NoSQL" }
```

  It is also possible to specify the position of an element inside an array:
  `{<field>.<index>.<field>: <value>}`

- If multiple selection criteria are specified for an array field, it has to be distinguished whether documents have to fulfill all or one of these criteria. If the criteria are separated by comma, each criterion can be matched by a different clause. In contrast, if each return document has to satisfy all of that criteria the special `$elemMatch` operator has to be used (see also [MMM+10c, Value in an Array – $elemMatch]):

```
{ x: {$elemMatch: {a: 1, b: {$gt: 2}} } } // documents with x.a==1 and x.b>2
{ "x.a": 1, "x.b": {$gt: 2} }             // documents with x.a==1 or x.b>2
```

- Fields inside objects are referenced by separating the fieldnames with a dot and placing the whole term in double quotes: `"field.subfield"`. If all fields of matching documents in their precise order are relevant as selection criteria the following syntax has to be used: `{ <field>: { <subfield_1>: <value>, <subfield_2>: <value>} }` (cf. [CMH+10], [Mer10f], [MHC+10a]).

Selection criteria may be specified independent of programming language bindings used applications via the `$where` operator that is assigned to a string containing the selection criteria in JavaScript syntax (cf. [MMC+10b]),e. g.

```
db.<collection>.find( { $where : "this.a==1" } );
```

As noted in the code snippet, the object for which a comparison happens is referenced by the `this` keyword. The `$where` operator may be specified in addition to other operators. The MongoDB recommends to prefer standard operators as mentioned above over the `$where` operator with a JavaScript criteria expression. This is because the first kind of statements can directly be used by the query optimizer and therefore evaluate

[13] Perl-compatible Regular Expressions (cf. [Haz10])

faster. To optimize query performance MongoDB first evaluates all other criteria before it interprets the JavaScript expression assigned to `$where` (cf. [MMC+10b]).

Projection A second parameter can be given to the `find` operation to limit the fields that shall be retrieved—analogous to the projection clause of a SQL statement (i. e. the field specification between the keywords `SELECT` and `FROM`). These fields are again specified by a BSON object consisting of their names assigned to the value `1` (cf. [MS10], [CMB+10, Optimizing A Simple Example]):

```
db.<collection>.find( {<selection criteria>}, {<field_1>:1, ...} );
```

The field specification is also applicable to fields of embedded objects using a dot-notation (`field.subfield`) and to ranges within arrays[14].

If only certain fields shall be excluded from the documents returned by the `find` operation, they are assigned to the value `0`:

```
db.<collection>.find( {<selection criteria>}, {<field_1>:0, <field_2>:0, ...} );
```

In both cases only partial documents are returned that cannot be used to update the documents they have been derived from. The MongoDB manual furthermore remarks that the primary key field `_id` is always returned in result documents and cannot be excluded.

Result Processing The results of the `find` operation may be processed further by arranging them using the `sort` operation, restricting the number of results by the `limit` operation and ignoring the first n results by the `skip` operation (cf. [CMH+10, Sorting, Skip and Limit], [MMM+10c, Cursor Methods], [CMB+10, Optimizing A Simple Example]):

```
db.<collection>.find( ... ).sort({<field>:<1|-1>}).limit(<number>).skip(<number>);
```

The `sort` operation—equivalently to the `ORDER BY` clause in SQL—takes a document consisting of pairs of field names and their desired order (`1`: ascending, `-1`: descending) as a parameter. It is recommended to specify an index on fields that are often used as a sort criteria or to limit the number of results in order to avoid in memory sorting of large result sets. If no `sort` operation is applied to selection the documents of the result set are returned in their *natural order* which is which is "is not particularly useful because, although the order is often close to insertion order, it is not *guaranteed* to be" for standard tables according to the MongoDB manual. However, for tables in so called *capped collections* (see below), the natural order of a result set can be leveraged and used to efficiently "store and retrieve data in insertion order". On the contrary, using the `sort` operation for selections on non-capped collections is highly advisable ((cf. [MMD+10])).

If the `limit` operation is not used, MongoDB returns all matching documents which are returned to clients in groups of documents called *chunks* (i. e. not as a whole); the number of documents in such a chunk is

[14] The array ranges to be returned are specified by the special `$slice` operator:

```
db.<collection>.find( {...}, {<field>: {$slice: n}});       // first n elements
db.<collection>.find( {...}, {<field>: {$slice: -n}});      // last n elements
db.<collection>.find( {...}, {<field>: {$slice: [m, n]}}); // skip m from beginning, limit n
db.<collection>.find( {...}, {<field>: {$slice: [-m, n]}}); // skip m from end, limit n
```

not fixed and may vary from query to query. The values for the `limit` and `skip` operation can also be given to the `find` operation as its third and fourth argument:

```
db.<collection>.find( {<selection-criteria>}, {<field-inclusion/exclusion>},
                      <limit-number>, <skip-number> );
```

To retrieve the number of query results fast and efficiently, the `count` operation can be invoked on result sets:

```
db.<collection>.find( ... ).count();
```

To aggregate results by predefined or arbitrary custom functions, the `group` operation is provided which can be seen as an equivalent to SQL's `GROUP BY`.

To limit the amount of documents to be evaluated for a query it is possible to specify minimum and / or maximum values for fields (cf. [MCB10a]):

```
db.<collection>.find( {...} ).min( {<field>: <value>, <...>: <...>, ....} ).
                              max( {<field>: <value>, <...>: <...>, ....} );
db.<collection>.find( $min: {...}, $max: {...}, $query: {...} );
```

The fields used to specify the upper and lower bounds have to be indexed. The values passed to `min` are inclusive while the values for `max` are exclusive when MongoDB evaluates ranges. For single fields this can be expressed also by the `$gte` and `$lt` operators which is strongly recommended by the MongoDB manual. On the other hand, it is difficult to specify ranges for compound fields—in these cases the minimum/maximum specification is preferred.

Besides these operations on queries or result sets, the query execution and its results can be configured by special operators that e. g. limit the number of documents to be scanned or explain the query (for further details cf. [MMM+10c, Special Operators]).

Cursors The return value of the `find` operation is a cursor by which the result documents of a query can be processed. The following JavaScript example shows how the result set of a query can be iterated (cf. [MMM+10c, Cursor Methods]):

```
var cursor = db.<collection>.find( {...});
cursor.forEach( function(result) { ... } );
```

To request a single document—typically identified by its default primary key field `_id`—the `findOne` operation should be used instead of `find` (cf. [MMM+10h]):

```
db.<collection>.findOne({_id: 921394});
```

Query Optimizer As seen above, MongoDB—unlike many NoSQL databases—supports ad-hoc queries. To answer these queries efficiently query plans are created by a MongoDB component called *query optimizer*. In contrast to similar components in relational databases it is not based on statistics and does not model the costs of multiple possible query plans. Instead, it just executes different query plans in parallel and stops all of them as soon as the first has returned thereby learning which query plan worked the best for a certain query. The MongoDB manual states that this approach "works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins)" (cf. [MC10b]).

Inserts

Documents are inserted into a MongoDB collection by executing the `insert` operation which simply takes the document to insert as an argument (cf. [CBH+10b, Document Orientation]):

```
db.<collection>.insert( <document> );
```

MongoDB appends the primary key field `_id` to the document passed to insert.

Alternatively, documents may also be inserted into a collection using the `save` operation (cf. [MMM+10d]):

```
db.<collection>.save( <document> );
```

The `save` operation comprises inserts as well as updates: if the `_id` field is not present in the document given to `save` it will be inserted; otherwise it updates the document with that `_id` value in the collection.

Updates

As discussed in the last paragraph, the `save` operation can be used to update documents. However, there is also an explicit `update` operation with additional parameters and the following syntax (cf. [CHD+10]):

```
db.<collection>.update( <criteria>, <new document>, <upsert>, <multi> );
```

The first argument specifies the selection criteria by which the document(s) to update shall be selected; it has to be provided in the same syntax as for the `find` operation. A document by which the matching documents shall be replaced is given as a second argument. If the third argument is set to `true`, the document in the second argument is inserted even if no document of the collection matches the criteria in the first argument (*upsert* is short for *update or insert*). If the last argument is set to `true`, all documents matching the criteria are replaced; otherwise only the first matching document is updated. The last two arguments for `update` are optional and set to `false` by default.

In its strive for good update performance, MongoDB collects statistics and infers which documents tend to grow. For these documents, some padding is reserved to allow them to grow. The rationale behind this approach is that updates can be processed most efficiently if the modified documents do not grow in size (cf. [CHD+10, Notes, Object Padding]).

Modifier Operations If only certain fields of a document shall be modified, MongoDB provides so called *modifier operations* that can be used instead of a complete document as a second parameter for `update` (cf. [CHD+10, Modifier Operations]). Modifier operations include incrementation of numerical values (`$inc`), setting and removing of fields (`$set`, `$unset`), adding of values to an array (`$push`, `$pushall`, `$addToSet`), removing values from an array (`$pop`, `$pull`, `$pullAll`), replacing of array values (`$`[15]) and renaming of fields (`$rename`).

As an example, the following command increments by `1` the revision number of the first document whose title is “MongoDB”:

```
db.<collection>.update( { title: "MongoDB"}, {$inc: {revision: 1}} );
```

The advantage of modifier operations in comparison to the replacement by whole documents is that they can be efficiently executed as the “latency involved in querying and returning the object” is avoided. In addition, modifier operations feature “operation atomicity and very little network data transfer” according to the MongoDB manual (cf. [CHD+10, Modifier Operations]).

If modifier operations are used with `upsert` set to `true`, new documents containing the fields defined in modifier operation expressions will be inserted.

Atomic Updates By default, update operations are non-blocking since MongoDB version 1.5.2. This is especially relevant as updates for multiple documents that are issued by the same command may be interleaved by other read and also write operations which can lead to undesired results. Therefore, if atomicity is required the `$atomic` flag has to be set to `true` and added to the selection criteria in `update` and also `remove` operations (cf. [CHD+10, Notes, Blocking]).

That said, the following operations and approaches already provide or achieve atomicity without using the `$atomic` flag explicitely:

- Updates using modifier operations are atomic by default (cf. [CHD+10, Modifier Operations], [MCS+10, Modifier operations]).
- Another way of executing updates atomically is called *Update if Current* by the MongoDB manual and comparable to the *Compare and Swap* strategy employed in operating systems (cf. [MCS+10, Update if Current]). It means to select an object (here: document), modify it locally and request an update for it only if the object has not changed in the datastore since it has been selected.
  As an example for the *Update if Current* the following operations increment the revision number of a wiki article:

  ```
  wikiArticle = db.wiki.findOne( {title: "MongoDB"} );      // select document
  oldRevision = wikiArticle.revision; // save old revision number
  wikiArticle.revision++;             // increment revision number in document
  db.wiki.update( { title: "MongoDB",
                    revision = oldRevision}, wikiArticle ); // Update if Current
  ```

  The last statement will only update the document if the revision number is the same as at the time the first statement was issued. If it has changed while the first tree lines have been executed, the selection criteria in first argument of the fourth line will no longer match any document[16].

[15] $ is a positional operator representing an array element matched by a selection.

[16] The MongoDB furthermore argues that the selection criteria from the first selection (in the example `title: "MongoDB"`) might have changed as well before the `update` statement is processed. This can be avoided by using e. g. an object-id field

- Single documents can be updated atomically by the special operation `findAndModify` which selects, updates and returns a document (cf. [MSB+10]). It's syntax is as follows:

  ```
  db.<collection>.findAndModify( query: {...},
                 sort: {...},
                 remove: <true|false>,
                 update: {...},
                 new: <true|false>,
                 fields: {...},
                 upsert: <true|false>);
  ```

  The `query` argument is used to select documents of which the first one is modified by the operation. To change the order of the result set, it can be sorted by fields specified in the `sort` argument. If the returned document shall be removed before it is returne the `remove` argument has to be set to `true`. In the `update` argument it is specified how the selected document shall be modified—either by giving a whole document replacing it or by using modifier expressions. The `new` argument has to be set to `true` if the modified document shall be returned instead of the original. To restrict the number of fields returned as a result document, the `fields` argument has to be used which contains field names assigned to 1 or 0 to select or deselect them. The last argument `upsert` specifies if a document shall be created if the result set of the `query` is empty.
  The `findAndModify` operation is also applicable in sharded setups. If the collection affected by the modification is sharded, the `query` must contain the shard key.

Deletes

To delete documents from a collection, the `remove` operation has to be used which takes a document containing selection criteria as a parameter (cf. [CNM+10]):

```
db.<collection>.remove( { <criteria> } );
```

Selection criteria has to be specified in the same manner as for the `find` operation. If it is empty, all documents of the collection are removed. The `remove` operation does not eliminate references to the documents it deletes[17]. To remove a single document from a collection it is best to pass its `_id` field to the `remove` operation as using the whole document is inefficient according to the MongoDB manual.

Since MongoDB version 1.3 concurrent operations are allowed by default while a `remove` operation executes. This may result in documents being not removed although matching the criteria passed to `remove` if a concurrent update operation grows a document. If such a behaviour is not desired, the `remove` operation may be executed atomically, not allowing any concurrent operations, via the following syntax:

```
db.<collection>.remove( { <criteria> , $atomic: true} );
```

(by default `_id`) or the entire object as selection criteria, or by using either the modifier expression `$set` or the positional operator `$` to modify the desired fields of the document.

[17] However, such "orphaned" references in the database can be detected easily as they return `null` when being evaluated.

Transaction Properties

Regarding transactions, MongoDB only provides atomicity for `update` and `delete` operations by setting the `$atomic` flag to `true` and adding it to the selection criteria. Transactional locking and complex transactions are not supported for the following reasons discussed in the MongoDB manual (cf. [MCS+10]):

- Performance as "in sharded environments, distributed locks could be expensive and slow". MongoDB is intended to be "leightweight and fast".
- Avoidance of deadlocks
- Keeping database operations simple and predictable
- MongoDB shall "work well for realtime problems". Therefore, locking of large amounts of data which "might stop some small light queries for an extended period of time [. . .] would make it even harder" to achieve this goal.

Server-Side Code Execution

MongoDB—like relational databases with their stored procedures—allows to execute code locally on database nodes. Server-side code execution comprises three different approaches in MongoDB:

1. Execution arbitrary code on a *single* database node via the `eval` operation (cf. [MMC+10b])
2. Aggregation via the operations `count`, `group` and `distinct` (cf. [MMM+10g])
3. MapReduce-fashioned code execution on multiple database nodes (cf. [HMC+10])

Each of these approaches shall be briefly discussed in the next paragraphs.

The `eval`-operation To execute arbitrary blocks of code locally on a database server, the code has to be enclosed by a anonymous JavaScript function and passed to MongoDB's generic `eval` operation (cf. [MMC+10b]):

```
db.eval( function(<formal parameters>) { ... }, <actual parameters>);
```

If function passed to `eval` has formal parameters, these have to be bound to actual parameters by passing them as arguments to `eval`. Although `eval` may be helpful to process large amounts of data locally on a server, it has to be used carefully a write lock is held during execution. A another important downside the `eval` operation is not supported in sharded setups, so that the MapReduce-approach has to be used in these scenarios as described below.

Functions containing arbitrary code may also be saved under a key (with the fieldname `_id`) on the database server in a special collection named `system.js` and later be invoked via their key, e. g.

```
system.js.save( {_id: "myfunction", value: function(...) {...}} );
db.<collection>.eval(myfunction, ...); // invoke the formerly saved function
```

Aggregation To accomplish the aggregation of query results MongoDB provides the `count`, `distinct` and `group` operation that may be invoked via programming language libraries but executed on the database servers (cf. [MMM+10g]).

The **`count`** operation returning the number of documents matching a query is invoked on a collection and takes selection criteria that is specified in the same way as for the `find` operation:

```
db.<collection>.count( <criteria> );
```

If `count` is invoked with empty criteria, the number of documents in the collection is returned. If selection criteria is used, the document fields used in it should be indexed to accelerate the execution of `count` (and likewise `find`).

To retrieve distinct values for certain document fields, the **`distinct`** operation is provided. It is invoked by passing a document in the following syntax to the generic `runCommand` operation or—if provided by a programming language driver–using the `distinct` operation directly:

```
db.runCommand( {distinct: <collection>, key: <field> [, query: <criteria> ]} );
db.<collection>.distinct( <document field> [, { <criteria> } ]);
```

The `query` part of the document is optional but may be useful to consider only the distinct field values of relevant documents of a collection. Document fields specified as `key` may also be nested fields using a dot notation (`field.subfield`).

As an equivalent to the `GROUP BY` clause in SQL MongoDB provides the aggregation operation **`group`**. The `group` operation returns an array of grouped items and is parameterized by a document consisting of the following fields:

```
db.<collection>.group( {
                        key: { <document field to group by> },
                        reduce: function(doc, aggrcounter) { <aggregation logic> },
                        initial: { <initialization of aggregation variable(s)> },
                        keyf: { <for grouping by key-objects or nestes fields> }
                        cond: { <selection criteria> },
                        finalize: function(value){ <return value calculation> }
                       });
```

The fields of the document passed to the `group` operation are explained in table 5.2.

Field	**Description**	**Mandatory?**
`key`	The document fields by which the grouping shall happen.	Yes (if `keyf` is not specified)
`reduce`	A function that "aggregates (reduces) the objects iterated. Typical operations of a reduce function include summing and counting. `reduce` takes two arguments: the current document being iterated over and the aggregation counter object."	Yes
`initial`	Initial values for the aggregation variable(s).	No

Field	Description	Mandatory?
keyf	If the grouping key is a calculated document/BSON object a function returning the key object has to be specified here; in this case key has to be omitted and keyf has to be used instead.	Yes (if key is not specified)
cond	Selection criteria for documents that shall be considered by the group operation. The criteria syntax is the same as for the find and count operation. If no criteria is specified, all documents of the collection are processed by group.	No
finalize	“An optional function to be run on each item in the result set just before the item is returned. Can either modify the item (e. g., add an average field given a count and a total) or return a replacement object (returning a new object with just _id and average fields).”	No

Table 5.2.: MongoDB - Parameters of the group operation (cf. [MMM+10g, Group])

The MongoDB manual provides an example how to use the group operation (cf. [MMM+10g, Group]):

```
db.<collection>.group(
                    {key: { a:true, b:true },
                     cond: { active:1 },
                     reduce: function(doc,acc) { acc.csum += obj.c; },
                     initial: { csum: 0 }
                    }
                   );
```

A corresponding SQL statement looks like this:

```
select a, b, sum(c) csum from <collection> where active=1 group by a, b;
```

The SQL function sum is reconstructed in the group function example by the fields initial and reduce of the parameter document given to group. The value of initial defines the variable csum and initializes it with 0; this code-block is executed once before the documents are selected and processed. For each document matching the criteria defined via cond the function assigned to the reduce key is executed. This function adds the attribute value c of the selected document (doc) to the field csum of the accumulator variable (acc) which is given to the function as a second argument (cf. [MMM+10g]).

A limitation of the group operation is that it cannot be used in sharded setups. In these cases the MapReduce approach discussed in the next paragraph has to be taken. MapReduce also allows to implement custom aggregation operations in addition to the predefined count, distinct and group operations discussed above.

MapReduce A third approach to server-side code execution especially suited for batch manipulation and aggregation of data in sharded setups is MapReduce (cf. [HMC+10]). MongoDB’s MapReduce implementation is similar to the concepts described in Google’s MapReduce paper (cf. [DG04]) and its open-source implementation Hadoop: there are two phases—map and the reduce—in which code written in JavaScript is executed on the database servers and results in a temporary or permanent collection containing the outcome. The MongoDB shell—as well as most programming language drivers—provides a syntax to launch such a MapReduce-fashioned data processing:

```
db.<collection>.mapreduce( map: <map-function>,
         reduce: <reduce-function>,
         query: <selection criteria>,
         sort: <sorting specificiation>,
         limit: <number of objects to process>,
         out: <output-collection name>,
         outType: <"normal"|"merge"|"reduce">,
         keeptemp: <true|false>,
         finalize: <finalize function>,
         scope: <object with variables to put in global namespace>,
         verbose: <true|false>
);
```

The MongoDB manual makes the following remarks concerning the arguments of `mapreduce`:

- The mandatory fields of the `mapreduce` operation are `map` and `reduce` which have to be assigned to JavaScript functions with a signature discussed below.
- If `out` is specified or `keeptemp` is set to `true` is specified the outcome of the MapReduce data processing is saved in a permanent collection; otherwise the collection containing the resuls is temporary and removed if the client disconnects or explicitely drops it.
- The `outType` parameter controls how the collection specified by `out` is populated: if set to `"normal"` the collection will be cleared before the new results are written into it; if set to `"merge"` old and new results are merged so that values from the latest MapReduce job overwrite those of former jobs for keys present in the old and new results (i. e. keys not present in the latest result remain untouched); if set to `"reduce"` the `reduce` operation of the latest MapReduce job is executed for keys that have old and new values.
- The `finalize` function is applied to all results when the `map` and `reduce` functions have been executed. In contrast to `reduce` it is only invoked once for a given key and value while `reduce` is invoked iteratively and may be called multiple times for a given key (see constraint for the `reduce` function below).
- The `scope` argument is assigned to a javascript object that is put into the global namespace. Therefore, its variables are accessible in the functions `map`, `reduce` and `finalize`.
- If `verbose` is set to `true` statistics on the execution time of the MapReduce job are provided.

The functions `map`, `reduce` and `finalize` are specified as follows:

- In the `map` function a single document whose reference is accessible via the keyword `this` is considered. `map` is required to call the function `emit(key, value)` at least once in its implementation to add a key and a single value to the intermediate results which are processed by the `reduce` function in the second phase of the MapReduce job execution. The `map` function has the following signature:

  ```
  function map(void) --> void
  ```

- The `reduce` is responsible to calculate a single value for a given key out of an array of values emitted by the `map` function. Therefore, its signature is as follows:

```
function reduce(key, value_array) --> value
```

The MongoDB manual notes that "[the] MapReduce engine may invoke reduce functions iteratively; thus, these functions must be idempotent". This constraint for `reduce` can be formalized by the following predicate:

$$\forall\ k, vals : reduce(k, [reduce(k, vals)]) \equiv reduce(k, vals)$$

The value returned by `reduce` is not allowed to be an array as of MongoDB version 1.6. The MongoDB furthermore suggests, that the values emitted by `map` and `reduce` "should be the same format to make iterative reduce possible" and avoid "weird bugs that are hard to debug".

- The optional `finalize` function is executed after the reduce-phase and processes a key and a value:

```
function finalize(key, value) --> final_value
```

In contrast to `reduce`, the `finalize` function is only called once per `key`.

The MongoDB manual points out that—in contrast to CouchDB—MapReduce is neither used for basic queries nor indexing in MongoDB; it is only be used if explicitly invoked by the `mapreduce` operation. As mentioned in the sections before, the MapReduce approach is especially useful or even has to be used (e. g. for server-side aggregation) in sharded setups. Only in these scenarios MapReduce jobs can be executed in parallel as " jobs on a single mongod process are single threaded [. . .] due to a design limitation in current JavaScript engines" (cf. [MMM+10g, Map/Reduce], [HMC+10]).

Commands for Maintainance and Administration

MongoDB features commands which are sent to the database to gain information about its operational status or to perform maintainance or administration tasks. A command is executed via the special namespace `$cmd` and has the following general syntax:

```
db.$cmd.findOne( { <commandname>: <value> [, options] } );
```

When such a command is sent to a MongoDB server it will answer with a single document containing the command results (cf. [MMM+10e]).

Examples of MongoDB commands include cloning of databases, flushing of pending writes to datafiles on disk, locking and unlocking of datafiles to block and unblock write operations (for backup purposes), creating and dropping of indexes, obtaining information about recent errors, viewing and terminating running operations, validating collections and retrieving statistics as well as database system information (cf. [CHM+10b]).

5.2.5. Indexing

Like relational database systems, MongoDB allows to specify indexes on document fields of a collection. The information gathered about these fields is stored in B-Trees and utilized by the query optimizing component to "to quickly sort through and order the documents in a collection" thereby enhancing read performance.

As in relational databases, indexes accelerate select as well as update operations as documents can be found faster by the index than via a full collection scan; on the other side indexes add overhead to insert and delete operations as the B-tree index has to be updated in addition to the collection itself. Therefore the MongoDB manual concludes that “indexes are best for collections where the number of reads is much greater than the number of writes. For collections which are write-intensive, indexes, in some cases, may be counterproductive” (cf. [MMH+10]). As a general rule, the MongoDB manual suggests to index “fields upon which keys are looked up” as well as sort fields; fields that should get indexed can also be determined using the profiling facility provided by MongoDB or by retrieving an execution plan by invoking the `explain` operation on a query[18] (cf. [MMM+10b, Index Selection], [CMB+10, Optimizing A Simple Example]).

Indexes are created by an operation named `ensureIndex`:

```
db.<collection>.ensureIndex({<field1>:<sorting>, <field2>:<sorting>, ...});
```

Indexes may be defined on fields of any type—even on nested documents[19]. If only certain fields of nested documents shall get indexed, they can be specified using a dot-notation (i. e. `fieldname.subfieldname`). If an index is specified for an array field, each element of the array gets indexed by MongoDB. The sorting flag may be set to 1 for an ascending and -1 for a descending order. The sort order is relevant for sorts and range queries on compound indexes (i. e. indexes on multiple fields) according to the MongoDB manual (cf. [MMH+10, Compound Keys Indexes], [MMM+10f]). To query an array or object field for multiple values the `$all` operator has to be used:

```
db.<collection>.find( <field>: { $all: [ <value_1>, <value_2> ... ] } );
```

By default, MongoDB creates an index on the `_id` field as it uniquely identifies documents in a collection and is expected to be chosen as a selection criterion often; however, when creating a collection manually, the automatic indexing of this field can be neglected. Another important option regarding indexes is background index building which has to be explicitly chosen as by default the index building blocks all other database operations. When using background indexing, the index is built incrementally and which is slower than indexing as a foreground job. Indexes built in background are taken into account for queries only when the background indexing job has finished. The MongoDB manual mentions two limitations with regards to (background) indexing: first, only one index per collection can be built a time; second, certain administrative operations (like `repairDatabase`) are disabled while the index is built (cf. [MH10b]).

MongoDB also supports unique indexes specifying that no two documents of a collections have the same value for such a unique index field. To specify unique indexes, another parameter is passed to the aforementioned `ensureIndex` operation:

```
db.<collection>.ensureIndex({<field1>:<sorting>, <field2>:<sorting>, ...}, {unique:
     true});
```

[18] For example, the execution plan of a query can be retrieved by:
`db.<collection>.find( ... ).explain();`

[19] As discussed in the MongoDB manual the indexing of nested documents may be helpful if the set of fields to index is not known in advance and shall be easily extensible. In this case, a field containing a nested document can be defined and indexed as a whole. As there is no strict schema for documents, fields to be indexed are put into the nested document and thereby get indexed. The downside of this approach is that it limits index parameters like the sort order and the uniqueness as the index is defined on the whole nested document and therefore all contained fields of this documents get indexed in the same manner.

The MongoDB manual warns that index fields have to be present in all documents of a collection. As MongoDB automatically inserts all missing indexed fields with null values, no two documents can be inserted missing the same unique indexed key.

MongoDB can be forced to use a certain index, by the `hint` operation which takes the names of index fields assigned to 1 as a parameter:

```
<db.<collection>.find( ... ).hint( {<indexfield_1>:1, <indexfield_2>:1, ...} );
```

The MongoDB points out that the query optimizer component in MongoDB usually leverages indexes but may fail in doing so e. g. if a query involves indexed and non indexed fields. It is also possible to force the query optimizer to ignore all indexes and do a full collection scan by passing `$natural:1` as a parameter to the `hint` operation (cf. [CMB+10, Hint]).

To view all indexes specified for a certain collection, the `getIndexes` operation is to be called:

```
db.<collection>.getIndexes();
```

To remove all indexes or a specific index of a collection the following operations have to be used:

```
db.<collection>.dropIndexes(); // drops all indexes
db.<collection>.dropIndex({<fieldname>:<sortorder>, <fieldname>:<sortorder>, ...});
```

Indexes can also be rebuilt by invoking the `reIndex` operation:

```
db.<collection>.reIndex();
```

The MongoDB manual considers this manually executed index rebuilding only “unnecessary” in general but useful for administrators in situations when the “size of your collection has changed dramatically or the disk space used by indexes seems oddly large”.

5.2.6. Programming Language Libraries

As of December 2010 the MongoDB project provides client libraries for the following programming languages: C, C#, C++, Haskell, Java, JavaScript, Perl, PHP, Python, and Ruby. Besides these officially delivered libraries, there are a lot of additional, community ones for further languages, such as Clojure, F#, Groovy, Lua, Objective C, Scala, Schema, and Smalltalk. These libraries—called *drivers* by the MongoDB project—provide data conversion between the programming language’s datatypes and the BSON format as well an API to interact with MongoDB servers (cf. [HCS+10]).

5.2.7. Distribution Aspects

Concurrency and Locking

The MongoDB architecture is described to be “concurrency friendly” although “some work with respect to granular locking and latching is not yet done. This means that some operations can block others.”

MongoDB uses read/write locks for many operations with "[any] number of concurrent read operations allowed, but typically only one write operation". The acquisition of write locks is greedy and, if pending, prevents subsequent read lock acquisitions (cf. [MHH10]).

Replication

For redundancy and the failover of a MongoDB cluster in the presence of unavailable database nodes, MongoDB provides asynchronous replication. In such a setup only one database node is in charge of write operations at any given time (called *primary server/node*). Read operations may go to this same server for strong consistency semantics or to any of its replica peers if eventual consistency is sufficient (cf. [MMG+10]).

The MongoDB documentation discusses two approaches to replication—Master-Slave Replication and Replica Sets—that are depicted in figure 5.1.

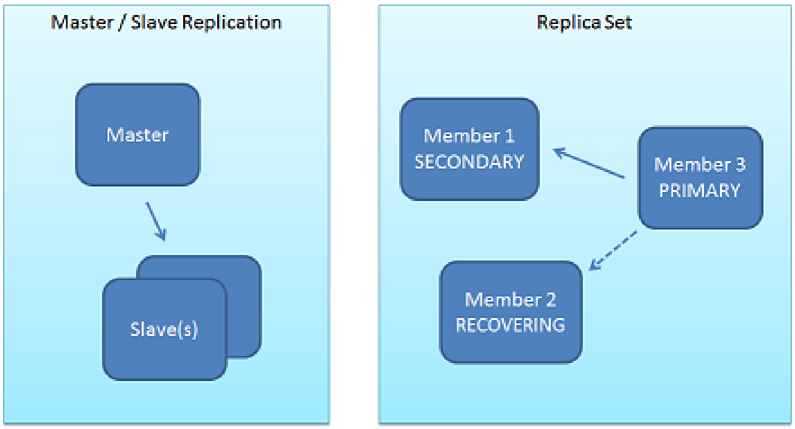


Figure 5.1.: MongoDB – Replication Approaches (taken from [MMG+10])

Master-Slave is a setup consisting of two servers out of one which takes the role of a master handling write requests and replicating those operations to the second server, the slave (cf. [MJS+10]).

Starting up a master-slave setup is done via starting up a MongoDB process as a master and a second process in slave mode with a further parameter identifying its master:

```
mongod --master <further parameters>
mongod --slave --source <master hostname>[:<port>] <further parameters>
```

An important note about slave servers is that replication will stop if they get too far behind the write operations from their master or if they are restarted and the update operations during their downtime can no longer be retrieved completely from the master. In these cases "replication will terminate and operator intervention is required by default if replication is to be restarted" (cf. [MJS+10], also [BMH10]). As an alternative, slaves can started with the -autoresync parameter which causes them to restart the replication if they become out of sync.

A permanent failover from a master to a slave or an inversion of their roles is only possible with a shutdown and restart of both servers (cf. [MJS+10, Administrative Tasks]).

Replica Sets are groups of MongoDB nodes "that work together to provide automated failover" (cf. [BCM+10]). They are described as an "an elaboration on the existing master/slave replication, adding automatic failover and automatic recovery of member nodes" (cf. [MCD+10]). To set up a replica set, the following steps are required (cf. [BCM+10]):

1. Start up a number of n MongoDB nodes with the -replSet parameter:

```
mongod --replSet <name> --port <port> --dbpath <data directory>
```

2. Initialize the replica set by connecting to one of the started mongod processes and passing a configuration document[20] for the replica set to the rs.initiate operation via the MongoDB shell:

```
mongo <host>:<port>
> config = {_id: '<name>',
            members: [
              {_id: 0: host: '<host>:<port>'},
              {_id: 1: host: '<host>:<port>'},
              {_id: 2: host: '<host>:<port>'}
             ]}
> rs.initiate(config);
```

The MongoDB node receiving the rs.initiate command propagates the configuration object to its peers and a primary node is elected among the set members. The status of the replica set can be retrieved via the rs.status operation or via a administrative web interface if one of the replica set nodes has been started with the -rest option (cf. [Mer10c]).

To add further nodes to the replica set they are started up with the -replSet parameter and the name of the replica set the new node shall become part of (cf. [DBC10]):

```
mongo --replSet <name> <further parameters>
```

Nodes added to a replica set either have to have an empty data directory or a recent copy of the data directory from another node (to accelerate synchronization). This is due to the fact that MongoDB does not feature multi-version storage or any conflict resolution support (see the section on limitations below).

If a replica set is up and running correctly write operations to the primary node get replicated to the secondary nodes by propagating operations to them which are applied to the locally stored data (cf. [Mer10e]). This approach is similar to the *operation transfer model* described in section 3.1.3 in the context of state propagation via vector clocks. In MongoDB operations received by peer nodes of a replica set are written into a capped collection that has a limited size which should not be chosen to low. However, no data gets lost if operations are omitted as new operations arrive faster than the node can apply them to its local data store; it is still possible to fully resynchronize such a node [Mer10d]. The write operations propagated among the replica set nodes are identified by the id of the server receiving them (the primary server at that time) and a monolithically increasing ordinal number (cf. [MHD+10]). The MongoDB points out that an "initial replication is essential for failover; **the system won't fail over to a new master until an initial sync between nodes is complete**".

[20] In the example only the mandatory configuration parameters are set – a full reference of configuration options–such as priorities for servers to get priority, the number votes in consensus protocols or different roles of servers—can be found in the MongoDB documentation ([MCB+10b, The Replica Set Config Object]).

Replica Sets can consist of up to seven servers which can take the role of a standard server (stores data, can become primary server), passive server (stores data, cannot become primary server) or arbiter (does not store data but participates in the consensus process to elect new primary servers; cf. [MHD+10]). Replica sets take into account data center configuration which—according to the MongoDB documentation—is implemented rudimentary as of version 1.6 but already features options like primary and disaster recovery sites as well as local reads (cf. [Mer10b]). Replica sets have the following consistency and durability implications (cf. [MD10]):

- Write operations are committed only if they have been replicated to a quorum (majority) of database nodes.
- While write operations are propagated to the replica nodes they are already visible at the primary node so that a newer version not committed at all nodes can already be read from the master. This *read uncommitted* semantic[21] is employed as—in theory—a higher performance and availability can be achieved this way.
- In case of a failover due to a failure of the primary database node all data that has not been replicated to the other nodes is dropped. If the primary node gets available again, its data is available as a backup but not recovered automatically as manual intervention is required to merge its database contents with those of the replica nodes.
- During a failover, i. e. the window of time in which a primary node is declared down by the other nodes and a new primary node is elected, write and strongly consistent read operations are not possible. However, eventually consistent read operations can still be executed (cf. [MH10a, How long does failover take?]).
- To detect network partitions each node monitors the other nodes via heartbeats. If a primary server does not receives heartbeats from at least half of the nodes (including itself), it leaves its primary state and does not handle write operations any longer. “Otherwise in a network partition, a server might think it is still primary when it is not” (cf. [MHD+10, Design, Heartbeat Monitoring]).
- If a new primary server is elected some serious implications regarding consistency may result: as the new primary is assumed to have the latest state all newer data on the new secondary nodes will be discarded. This situation is detected by the secondary nodes via their operation logs. Operations that have already been executed are rolled back until their state corresponds to the state of the new primary. The data changed during this rollback is saved to a file and cannot be applied automatically to the replica set. Situations in which secondary nodes may have a later state than a newly elected primary can result from operations already committed at a secondary node but not at the new primary (which was secondary at the time they were issued; cf. [MHD+10, Design, Assumption of Primary; Design, Resync (Connecting to a New Primary)]).

In MongoDB version 1.6 authentication is not available for replica sets. A further limitation is that MapReduce jobs may only run on the primary node as they create new collections (cf. [MC10a]).

Excluding Data from Replication can be achieved by putting the data that shall not become subject to replication into the special database `local`. Besides user data MongoDB also stores administrative data such as replication configuration documents in this database (cf. [Mer10a]).

[21] This read semantic is comparable to the Read Your Own Writes (RYOW) and Session Consistency model discussed in 3.1.2 if the client issuing write operations at the primary node also reads from this node subsequently.

Sharding

Since Version 1.6 MongoDB supports horizontal scaling via an automatic sharding architecture to distribute data across "thousands of nodes" with automatic balancing of load and data as well as automatic failover (cf. [MHC+10b], [MDH+10]).

Sharding is understood to be "the partitioning of data among multiple machines in an order-preserving manner" by the MongoDB documentation. The MongoDB mentions Yahoo!'s PNUTS[22] ([CRS+08]) and Google's Bigtable ([CDG+06]) as important influences for the partitioning scheme implemented in MongoDB (cf. [MDH+10, MongoDB's Auto-Sharding, Scaling Model]). Sharding in MongoDB "occurs on a per-collection basis, not on the database as a whole". In a setup configured for sharding, MongoDB automatically detects which collections grow much faster than the average so that they become subject to sharding while the other collections may still reside on single nodes. MongoDB also detects imbalances in the load different shards have to handle and can automatically rebalance data to reduce disproportionate load distribution (cf. [MDH+10, MongoDB's Auto-Sharding]).

Sharding in MongoDB is built on top of *replica sets* which have been discussed above. This means that for each partition of data a number of nodes forming a replica set is in charge: at any time one of these servers is primary and handles all write requests which then get propagated to the secondary servers to replicate changes and keep the set in-sync; if the primary node fails the remaining nodes elect a new primary via consensus so that the operation of the replica set is continued. This way, automated failover is provided for each shard (cf. [MDH+10, MongoDB's Auto-Sharding, Balancing and Failover]).

Sharding Architecture A MongoDB shard cluster is built up by three components as depicted in figure 5.2 (cf. [MDH+10, Architectural Overview]):

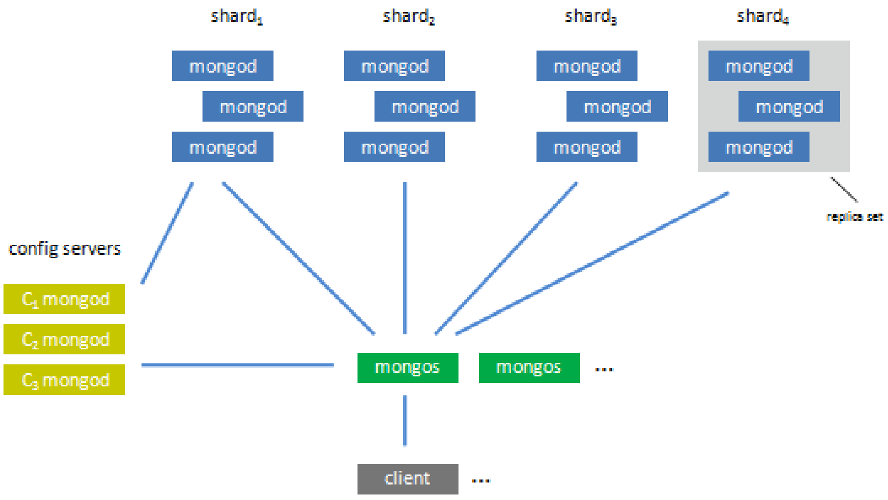


Figure 5.2.: MongoDB – Sharding Components (taken from [MDH+10, Architectural Overview])

Shards consisting of servers that run `mongod` processes and store data. To ensure availability and automated failover in production systems, each shard typically consists of multiple servers comprising a replica set.

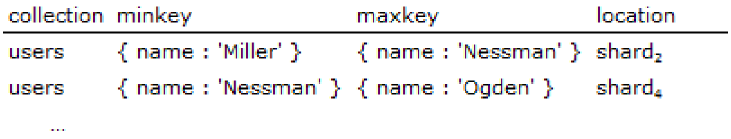
[22] Platform for Nimble Universal Table Storage

Config Servers "store the cluster's metadata, which includes basic information on each shard server and the chunks contained therein". Chunks are contiguous ranges of data from collections which are ordered by the sharding key and stored on the shards (sharding keys and chunks are discussed in more detail below). Each config server stores the complete chunk metadata and is able to derive on which shards a particular document resides. The data on config servers is kept consistent via a two-phase commit protocol and a special replication scheme (i. e. Master Slave Replication or Replica Sets are not used for this purpose). The metadata stored on config servers becomes read only if any of these servers is unreachable; in this state, the database is still writable, but it becomes impossible to redistribute data among the shards (cf. also [MHB+10b]).

Routing Services are server-side mongos-processes executing read and write requests on behalf of client applications. They are in charge of looking up the shards to be read from or written to via the config servers, connecting to these shards, executing the requested operation and returning the results to client applications, also merging results of the request execution on different shards. This makes a distributed MongoDB setup look like a single server towards client applications which do not have to be aware of sharding; even the programming language libraries do not have to take sharding into account. The mongos processes do not persist any state and do not coordinate with one another within a sharded setup. In addition, they are described to be lightweight by the MongoDB documentation, so that it is uncritical to run any number of mongos processes. The mongos processes may be executed on any type of server—shards, config servers as well as application servers.

Shards, config and routing servers can be organized in different ways on physical or virtual servers as described by the MongoDB documentation (cf. [MDH+10, Server Layout]). A minimal sharding setup requires at least two shards, one config and one routing server (cf. [MHB+10a, Introduction]).

Sharding Scheme As discussed above, partitioning in MongoDB occurs based on collections. For each collection, a number of fields can be configured by which the documents shall get partitioned. If MongoDB then detects imbalances in load and data size for a collection it partitions its documents by the configured keys while preserving their order. If e. g. the document field name is configured as a shard key for the collection users, the metadata about this collection held on the config servers may look like figure 5.3.



collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	shard$_2$
users	{ name : 'Nessman' }	{ name : 'Ogden' }	shard$_4$
...			

Figure 5.3.: MongoDB – Sharding Metadata Example (taken from [MDH+10, Architectural Overview, Shards, Shard Keys])

Sharding Persistence Documents of a collection are stored in so called *chunks*, contiguous ranges of data. They are identified by the triple (collection, minkey, maxkey) with minkey and maxkey as the minimum and maximum value for the document field(s) chosen as sharding keys. If chunks grow to a configured size they are split into two new chunks (cf. [MDH+10, Architectural Overview]). As of version 1.6, chunk "splits happen as a side effect of inserting (and are transparent)" towards client applications. Chunks can get migrated in the background among the shard servers to evenly distribute data as well as load; this is done by a "sub-system called Balancer, which constantly monitors shards loads and [. . .] moves chunks around if it finds an imbalance" (cf. [MSL10]).

The MongoDB manual points out that it is important to choose sharding keys that are "*granular* enough to ensure an even distribution of data". This means that the there should be a high number of distinct values in the document fields chosen as sharding keys, so that it is always possible to split chunks. A counterexample might be the `name` field mentioned above: if one chunk only contains documents with popular names like "Smith" it cannot be split into smaller chunks. This is due to the fact that a chunk is identified by the abovementioned triple of `(collection, minkey, maxkey)`: if `minkey` and `maxkey` already have the same value and sharding shall preserve the order of documents, such a chunk cannot be split up. To avoid such issues it is also possible to specify compound sharding keys (such as `lastname`, `firstname`; cf. [MDH+10, Architectural Overview]).

Setting up a Sharded MongoDB Cluster is done via the following steps (cf. [MHB+10a, Configuring the Shard Cluster], [MCH+10]):

1. Start up a number shards via the `mongod` command with the `-shardsvr` flag:

```
./mongod --shardsrv <further parameters>
```

2. Start up a number of config servers via the `mongod` command with the `-configsvr` flag:

```
./mongod --configsvr <further parameters>
```

3. Start up a number of routing servers via the `mongos` command which are pointed to one config server via the `-configdb` flag:

```
./mongos --configdb <config server host>:<port> <further parameters>
```

4. Connect to a routing server, switch to its admin database and add the shards:

```
./mongo <routig server host>:<routing server port>/admin
> db.runCommand( { addshard : "<shard hostname>:<port>" } )
```

5. Enable sharding for a database and therein contained collection(s):

```
./mongo <routig server host>:<routing server port>/admin
> db.runCommand( { enablesharding : "<db>" } )
> db.runCommand( { shardcollection : "<db>.<collection>",
                   key : {<sharding key(s)>} } )
```

It is also possible to add database nodes from a non-sharded environment to a sharded one without downtime. To achieve this, they can be added as shards in step 4 like any other shard server (cf. [HL10]).

Failures in a sharded environment have the following effects, depending on the type of failure (cf. [MHB+10b]):

- The **failure of a mongos routing process** is uncritical as there may—and in production environments: should—be more than one routing process. In case of a failure of one of these processes, it can simple be launched again or requests from client applications may be redirected to another routing process.
- The **failure of a single mondod process within a shard** do not affect the availability of this shard if it is distributed on different servers comprising a replica set. The aforementioned notes on failures of replica set nodes apply here again.
- The **failure of all mongodb processes comprising a shard** makes the shard unavailable for read and write operations. However, operations concerning other shards are not affected in this failure scenario.
- The **failure of a config server** does not affect read and write operations for shards but makes it impossible to split up and redistribute chunks.

Implications of Sharding are that it “must be ran in trusted security mode, without explicit security”, that shard keys cannot be altered as of version 1.6, and that write operations (update, insert, upsert) must include the shard key (cf. [MDN+10]).

Limitations and Renounced Features Regarding Distribution

MongoDB does not provide multiversion-storage, multi-master replication setups or any support for version conflict reconciliation. The rationale for taking a different approach than many other NoSQL stores and not providing these features is explained by the MongoDB documentation as follows:

> “Merging back old operations later, after another node has accepted writes, is a hard problem. One then has multi-master replication, with potential for conflicting writes. Typically that is handled in other products by manual version reconciliation code by developers. We think that is too much work : we want MongoDB usage to be less developer work, not more. Multi-master also can make atomic operation semantics problematic. It is possible (as mentioned above) to manually recover these events, via manual DBA effort, but we believe in large system with many, many nodes that such efforts become impractical.” (cf. [MD10, Rationale])

5.2.8. Security and Authentication

According to its documentation, as of version 1.6 “Mongo supports only very basic security”. It is possible to create user accounts for databases and require them to authenticate via username and password. However, access control for authenticated users only distinguishes read and write access. A special user for a MongoDB database server process (i. e. a mongod process) is the admin user which has full read and write access to all databases provided by that process including the admin database containing administrative data. An admin user is also privileged to execute operations declared as administrative (cf. [MMC+10a, Mongo Security]).

To require authentication for a database server process, the mongod process has to be launched with the -auth parameter. Before that, an administrator has to be created in the admin database. A logged in administrator then can create accounts for read- and write-priviledged users in the system.users collection by the addUser operation (cf. [MMC+10a, Configuring Authentication and Security]):

```
./mongo <parameters>               // log on as administrator
> use <database>
> db.addUser(<username>, <password> [, <true|false>])
```

The optional third parameter of the addUser operation tells whether the user shall only have read-access; if it is omitted, the user being created will have read and write access for the database.

Authentication and access control is not available for replicated and sharded setups. They can only be run securely in an environment that ensures that "only trusted machines can access database TCP ports" (cf. [MMC+10a, Running Without Security (Trusted Environment)]).

5.2.9. Special Features

Capped Collections

MongoDB allows to specify collections of a fixed size that are called *capped collections* and can be processed highly performant by MongoDB according to its manual. The size of such a capped collection is preallocated and kept constant by dropping documents in a FIFO[23]-manner according to the order of their insertion. If a selection is issued on a capped collection the documents of the result set are returned in the order of their insertion if they are not sorted explicitly, i. e. the `sort` operation is not applied to order the results by document fields[24].

Capped collections have some restrictions regarding operations on documents: while insertions of documents are always possible, update operations only succeed if the modified document does not grow in size and the delete operation is not available for single documents but only for dropping all documents of the capped collection. The size of a capped collection is limited to 1^9 bytes on a 32-bit machine while there is no such limitation on 64-bit machine. A further limitation of capped collections is the fact that they cannot be subject to sharding. If no ordering is specified when selecting documents, they are returned in the order of their insertion. In addition to limiting the size of a capped collection it is possible to also specify the maximum number of objects to be stored (cf. [MDM+10]).

The MongoDB mentions the following use cases for capped collections (cf. [MDM+10, Applications]):

Logging is an application for capped collections as inserts of documents are processed at a speed comparable to file system writes while the space required for such a log is held constant.

Caching of small or limited numbers of precalculated objects in a LRU-manner can be done via capped collections conveniently.

Auto Archiving is understood by the MongoDB manual as a situation in which aged data needs to be dropped from the database. While for other databases and regular MongoDB collections scripts have to be written and scheduled to remove data by age capped collections already provide such behavior and no further effort is needed.

[23] First in, first out

[24] To be precise, it has to be stated that the sort operation may also be invoked with the `$natural` parameter to enforce natural ordering. This can be leveraged to achieve e. g. a descending natural order as follows (cf. [MMD+10]):
`db.<collection>.find( ... ).sort( $natural: -1 )`

GridFS

BSON objects that allow to store binary data in MongoDB are limited in size by 4MB. Therefore, MongoDB implements a specification named *GridFS* to store large binary data and provide operations on large data objects—such as videos or audios—like retrieving a limited number bytes of such an object. The GridFS specification is implemented by MongoDB transparent to client applications by dividing large data objects among multiple documents maintained in a collection and maintaining a collection containing metadata on these documents. Most programming language drivers also support the GridFS specification (cf. [CDM+10]).

The MongoDB manual considers GridFS also as an alternative to file system storage when large numbers of files have to be maintained, replicated and backed up as well as if files are expected to change often. In contrast, if a large number of small and static files has to be stored and processed, storage in MongoDB by GridFS is not regarded reasonable (cf. [Hor10]).

Geospatial Indexes

To support location based queries like “find the closest n items to a specific location” MongoDB provides two-dimensional geospatial indexing (cf. [HMS+10]). The latitude and longitude values have to be saved in a document field that is either an object or an array with the first two elements representing coordinates, e. g.

```
{ loc : { 50, 30 } }            // coordinates in an array field
{ loc : { x : 50, y : 30} }     // coordinates in an object field
```

If an object field is used, the object’s fields do not have to have certain names and also do not have to appear in a certain order (though ordering has to be consistent among all documents).

An index on the geospatial coordinates is created using the special value 2d instead of an order:

```
db.<collection>.createIndex( { <field> : "2d" } );
```

Without additional parameters for index creation, MongoDB suggests that latitude and longitude values shall be indexed and therefore aligns the index range to the interval $]-180\dots180[$. If other values are indexed the index range should be given at index creation as follows:

```
db.<collection>.createIndex( { <field> : "2d" } , { min : <min-value,
                                                    max : <max-value });
```

The range boundaries are exclusive which means that index values cannot take these values.

To query based on a geospatial index the query document can contain special criteria like `$near` and $maxDistance:

```
db.<collection>.find( { <field> : [<coordinate>, <coordinate>]} ); // exact match
db.<collection>.find( { <field> : { $near : [<coordinate>, <coordinate>] } } );
db.<collection>.find( { <field> : { $near : [<coordinate>, <coordinate>],
                                    $maxDistance : <value>} } );
```

In addition to limiting the distance in which matches shall be found it is also possible to define a shape (box or circle) covering a geospatial region as a selection criterion.

Besides the generic `find` operation MongoDB also provides a specialized way to query by geospatial criterions using the `geoNear` operation. The advantages of this operation are that it returns a distance value for each matching document and allows for diagnostics and troubleshooting. The `geoNear` operation has to be invoked using the `db.runCommand` syntax:

```
db.runCommand({geoNear: <collection>, near= [<coordinate>, <coordinate>], ...});
```

As shown in the syntax example, no field name has to be provided to the `geoNear` operation as it automatically determines the geospatial index of the collection to query.

When using the aforementioned `$near` criterion MongoDB does its calculations based on an idealized model of a flat earth where an arcdegree of latitude and longitude is the same distance at each location. Since MongoDB 1.7 a spherical model of the earth is provided so that selections can use correct spherical distances by applying the `$sphereNear` criterion (instead of `$near`), the `$centerSphere` criterion (to match circular shapes) or adding the option `spherical:true` to the parameters of the `geoNear` operation. When spherical distances are used, coordinates have to be given as decimal values in the order longitude, latitude using the radians measurement.

Geospatial indexing has some limitations in the MongoDB versions 1.6.5 (stable) and 1.7 (unstable as of December 2010). First, it is limited to indexing squares without wrapping at their outer boundaries. Second, only one geospatial index is allowed per collection. Third, MongoDB does not implement wrapping at the poles or at the -180° to 180° boundary. Lastly, geospatial indexes cannot be sharded.

6. Column-Oriented Databases

In this chapter a third class of NoSQL datastores is investigated: column-oriented databases. The approach to store and process data by column instead of row has its origin in analytics and business intelligence where column-stores operating in a shared-nothing massively parallel processing architecture can be used to build high-performance applications. Notable products in this field are Sybase IQ and Vertica (cf. [Nor09]). However, in this chapter the class of column-oriented stores is seen less puristic, also subsuming datastores that integrate column- and row-orientation. They are also described as "[sparse], distributed, persistent multidimensional sorted [maps]" (cf. e. g. [Int10]). The main inspiration for column-oriented datastores is Google's Bigtable which will be discussed first in this chapter. After that there will be a brief overview of two datastores influenced by Bigtable: Hypertable and HBase. The chapter concludes with an investigation of Cassandra, which is inspired by Bigtable as well as Amazon's Dynamo. As seen in section 2.3 on classifications of NoSQL datastores, Bigtable is subsumed differently by various authors, e. g. as a "wide columnar store" by Yen (cf. [Yen09]), as an "extensible record store" by Cattel (cf. [Cat10]) or as an entity-attribute-value[1] datastore by North (cf. [Nor09]). In this paper, Cassandra is discussed along with the column-oriented databases as most authors subsume it in this category and because one of its main inspirations, Google's Bigtable, has to be introduced yet.

6.1. Google's Bigtable

Bigtable is described as "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" (cf. [CDG+06, p. 1]). It is used by over sixty projects at Google as of 2006, including web indexing, Google Earth, Google Analytics, Orkut, and Google Docs (formerly named *Writely*)[2]. These projects have very different data size, infrastructure and latency requirements: "from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data" (cf. [CDG+06, p. 1]). According to Chang et al. experience at Google shows that "Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability" (cf. [CDG+06, p. 1]). Its users "like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time". For Google as a company the design and implementation of Bigtable has shown to be advantageous as it has "gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise" (cf. [CDG+06, p. 13]).

[1] The entity-attribute-value (EAV) datastores predate relational databases and do not provide the full feature set of RDBMSs (e. g. no comprehensive querying capabilities based on a declarative language like SQL) but their data model is richer than that of a simple key-/value-store. EAV based on modern technologies currently have a revival as they are offered by major cloud computing providers such as Amazon (SimpleDB), Google (App Engine datastore) and Microsoft (SQL Data Services; cf. [Nor09]).

[2] As of 2011 a number of further notable projects such as the Google App Engine and Google fusion tables also use Bigtable.

Bigtable is described as a database by Google as “it shared many implementation strategies with databases”, e. g. parallel and main-memory databases. However, it distinguishes itself from relational databases as it “does not support a full relational data model”, but a simpler one that can be dynamically controlled by clients. Bigtable furthermore allows “clients to reason about the locality properties of the data” which are reflected “in the underlying storage” (cf. [CDG+06, p. 1]). In contrast to RDBMSs, data can be indexed by Bigtable in more than one dimension—not only row- but also column-wise. A further distinguishing proposition is that Bigtable allows data to be delivered out of memory or from disk—which can be specified via configuration.

6.1.1. Data Model

Chang et al. state that they “believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers.” Therefore the data model they designed for Bigtable should be “richer than simple key-value pairs, and [support] sparse semi-structured data”. On the other hand, it should remain “simple enough that it lends itself to a very efficient flat-file representation, and [. . .] transparent enough [. . .] to allow our users to tune important behaviors of the system” (cf. [CDG+06, p. 12]).

The data structure provided and processed by Google’s Bigtable is described as “a sparse, distributed, persistent multidimensional sorted map”. Values are stored as arrays of bytes which do not get interpreted by the data store. They are addressed by the triple (`row-key, column-key, timestamp`) (cf. [CDG+06, p. 1]).

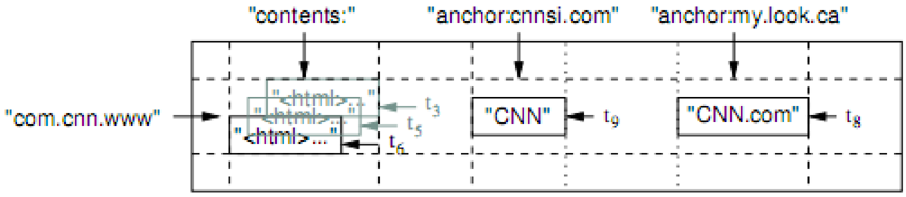


Figure 6.1.: Google’s Bigtable – Example of Web Crawler Results (taken from [CDG+06, p. 2])

Figure 6.1 shows a simplified example of a Bigtable[3] storing information a web crawler might emit. The map contains a non-fixed number of rows representing domains read by the crawler as well as a non-fixed number of columns: the first of these columns (`contents:`) contains the page contents whereas the others (`anchor:<domain-name>`) store link texts from referring domains—each of which is represented by one dedicated column. Every value also has an associated timestamp (t_3, t_5, t_6 for the page contents, t_9 for the link text from *CNN Sports Illustrated*, t_8 for the link text from *MY-look*). Hence, a value is addressed by the triple (`domain-name, column-name, timestamp`) in this example (cf. [CDG+06, p. 2]).

Row keys in Bigtable are strings of up to 64KB size. Rows are kept in lexicographic order and are dynamically partitioned by the datastore into so called ***tablets***, “the the unit of distribution and load balancing” in Bigtable. Client applications can exploit these properties by wisely choosing row keys: as the ordering of row-keys directly influences the partitioning of rows into tablets, row ranges with a small lexicographic distance are probably split into only a few tablets, so that read operations will have only a small number of servers delivering these tablets (cf. [CDG+06, p. 2]). In the aforementioned example of figure 6.1 the domain names used as row keys are stored hierarchically descending (from a DNS point of view), so that subdomains have a smaller lexicographic distance than if the domain names were stored reversely (e. g. `com.cnn.blogs`, `com.cnn.www` in contrast to `blogs.cnn.com`, `www.cnn.com`).

[3] Chang et al. also refer to the datastructure itself as *a Bigtable*.

The number of **columns** per table is not limited. Columns are grouped by their key prefix into sets called *column families*. Column families are an important concept in Bigtable as they have specific properties and implications (cf. [CDG+06, p. 2]):

- They "form the basic unit of access control", discerning privileges to list, read, modify, and add column-families.
- They are expected to store the same or a similar type of data.
- Their data gets compressed together by Bigtable.
- They have to be specified before data can be stored into a column contained in a column family.
- Their name has to be printable. In contrast, column qualifiers "may be arbitrary strings".
- Chang et al. suggest that "that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation".

The example of figure 6.1 shows two column families: `content` and `anchor`. The `content` column family consists of only one column whose name does not have to be qualified further. In contrast, the `anchor` column family contains two columns qualified by the domain name of the referring site.

Timestamps, represented as 64-bit integers, are used in Bigtable to discriminate different reversion of a cell value. The value of a timestamp is either assigned by the datastore (i. e. the actual timestamp of saving the cell value) or chosen by client applications (and required to be unique). Bigtable orders the cell values in decreasing order of their timestamp value "so that the most recent version can be read first". In order to disburden client applications from deleting old or irrelevant revisions of cell values, an automatic garbage-collection is provided and can be parameterized per column-family by either specifying the number of revisions to keep or their maximum age (cf. [CDG+06, p. 2f]).

6.1.2. API

The Bigtable datastore exposes the following classes of operations to client applications (cf. [CDG+06, p. 3]):

Read Operations include the lookup and selection of rows by their key, the limitation of column families as well as timestamps (comparable to projections in relational databases) as well as iterators for columns.

Write Operations for Rows cover the creation, update and deletion of values for a columns of the particular row. Bigtable also supports "batching writes across row keys".

Write Operations for Tables and Column Families include their creation and deletion.

Administrative Operations allow to change "cluster, table, and column family metadata, such as access control rights".

Server-Side Code Execution is provided for scripts written in Google's data processing language Sawzall (cf. [PDG+05], [Gri08]). As of 2006, such scripts are not allowed to write or modify data stored in Bigtables but "various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators".

MapReduce Operations may use contents of Bigtable maps as their input source as well as output target.

Transactions are provided on a single-row basis: "Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row" (cf. [CDG+06, p.2]).

6.1.3. Infrastructure Dependencies

Bigtable depends on a number of technologies and services of Google's infrastructure (cf. [CDG+06, p.3f]):

- The distributed **Google File System (GFS)** (cf. [GL03]) is used by Bigtable to persist its data and log files.
- As Bigtable typically shared machines with "wide variety of other distributed applications" it depends on a **cluster management system** "for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status".
- Bigtable data is stored in Google's **SSTable** file format (see section 3.3 on page 44). A SSTable is a "persistent, ordered immutable map" whose keys and values "are arbitrary byte strings". SSTables allow applications to look up values via their keys and "iterate over all key/value pairs in a specified key range". Internally an SSTable is represented as a sequence of blocks with a configurable, fixed size. When an SSTable is opened, a block index located at the end of the SSTable is loaded into memory. If a block from the SSTable is requested, only one disk seek is necessary to read the block after a binary search in the in-memory block index has occurred. To further enhance read performance SSTables can be loaded completely into memory.
- The "highly-available and persistent distributed lock service" **Chubby** (cf. [Bur06]) is employed by Bigtable for several tasks:
  - "[Ensure] that there is at most one active master at any time" in a Bigtable cluster
  - Store of location information for Bigtable data required to bootstrap a Bigtable instance
  - Discover tablet servers[4]
  - Finalize of tablet server deaths
  - Store schema information, i. e. the column-families for each table of a Bigtable instance
  - Store of access control lists

  Each instance of a Chubby service consists a cluster of "five active replicas, one of which is elected to be the master and actively serve requests". To keep the potentially fallible replicas consistent, an implementation of the Paxos consensus algorithm (cf. [Lam98]) is used [5]. Chubby provides a namespace for directories and (small) files and exposes a simplified file-system interfaces towards clients. Each file and directory "can act as a reader-writer lock" ([Bur06, p. 338]). Client application initiate sessions with a Chubby instance, request a lease and refresh it from time to time. If a Chubby cluster becomes unavailable, its clients—including Bigtable—instances are unable to refresh their session lease within a predefined amount of time so that their sessions expire; in case of Bigtable this results in an unavailability of Bigtable itself "after an extended period of time" (i. e. Chubby unavailability; cf. [CDG+06, p. 4]).

[4] Tablet servers will be introduced and explained in the next section.

[5] Details on Chubby as well difficulties met when implementing the theoretically well described Paxos consensus algorithm are described in Google's "Paxos made live" paper (cf. [CGR07]).

6.1.4. Implementation

Components

Bigtable's implementation consists of three major components per Bigtable instance (cf. [CDG+06, p. 4]):

- Multiple **tablet servers** each of which is responsible for a number of tablets. This implies the handling of read and write requests for tablets as well as the splitting of tablets "that have grown too large"[6]. Tablet servers can be added and removed at runtime.
- A **client library** provided for applications to interact with Bigtable instances. The library responsible for looking up tablet servers that are in charge of data that shall be read or written, directing requests to them and providing their responses to client applications.
- One **master server** with a number of responsibilities. Firstly, it manages the tablets and tablet servers: it assigns tablets to tablet servers, detects added and removed tablet servers, and distributes workload across them. Secondly, it is responsible to process changes of a Bigtable schema, like the creation of tables and column families. Lastly, it has to garbage-collect deleted or expired files stored in GFS for the particular Bigtable instance. Despite these responsibilities the load on master servers is expected to be low as client libraries lookup tablet location information themselves and therefore "most clients never communicate with the master". As the master server is a single point of failure for a Bigtable instances it is backed up by a second machine according to Ippolito (cf. [Ipp09]).

Tablet Location Information

As it has been stated in the last sections, tables are dynamically split into tablets and these tablets are distributed among a multiple tablet servers which can dynamically enter and leave a Bigtable instance at runtime. Hence, Bigtable has to provide means for managing and looking up tablet locations, such that master servers can redistribute tablets and client libraries can discover the tablet servers which are in charge of certain rows of table. Figure 6.2 depicts how tablet location information is stored in Bigtable (cf. [CDG+06, p. 4f]).

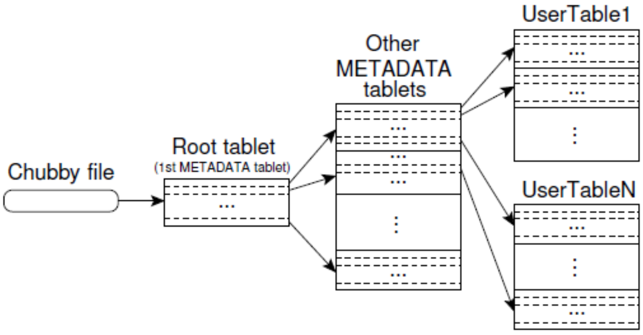


Figure 6.2.: Google's Bigtable – Tablet Location Hierarchy (taken from [CDG+06, p. 4])

The locations of tablets are stored in a table named `METADATA` which is completely held in memory. This table is partitioned into a special first tablet (*root tablet*) and an arbitrary number of further tablets (*other METADATA tablets*). The *other METADATA tablets* contain the location information for all tablets of user tables (i. e. tables created by client applications) whereas the *root tablet* contains information about

[6] Typical sizes of tablets after splitting are 100-200 MB at Google according to Chang et al..

the location of the *other METADATA tablets* and is never split itself. The location information for the *root tablet* is stored in a file placed in a Chubby namespace. The location information for a tabled is stored in row that identified by the "tablet's table identifier and its end row"[7]. With a size of about 1 KB per row and a tablet size of 128 MB Bigtable can address 2^34 tablets via the three-level hierarchy depcited in figure 6.2.

A client library does not read through all levels of tablet location information for each interaction with the datastore but caches tablet locations and "recursively moves up the tablet location hiearchy" if it discovers a location to be incorrect. A further optimization employed at Google is the fact that client libraries prefetch tablet locations, i.e. read tablet locations for more than one tablet whenever they read from the METADATA table.

Master Server, Tablet Server and Tablet Lifecycles

Tablet Lifecycle A tablet is created, deleted and assigned to a tablet server by the master server. Each tablet of a Bigtable is assigned to at most one tablet server at a time; *at most one* is due to the fact that tablets may also be unassigned until the master server finds a tablet server that provides enough capacity to serve that tablet. Tablets may also get merged by the master server and split by a tablet server which has to notify the master server about the split[8]). Details about how tablets are represented at runtime and how read and write operations are applied to them is discussed in the section on tablet representation below.

Tablet Server Lifecycle When a tablet server starts, it creates a uniquely-named file in a predefined directory of a Chubby namespace and acquires an exclusive lock for this. The master server of a Bigtable instance constantly monitors the tablet servers by asking them whether they have still locked their file in Chubby; if a tablet server does not respond, the master server checks the Chubby directory to see whether the particular tablet server still holds its lock. If this is not the case, the master server deletes the file in Chubby and puts the tablets served by this tablet server into the set of unassigned tablets. The tablet server itself stops serving any tablets when it loses its Chubby lock. If the tablet server is still up and running but was not able to hold its lock due to e.g. network partitioning it will try to acquire that lock again if its file in Chubby has not been deleted. If this file is no longer present, the tablet server stops itself. If a tablet server is shut down in a controlled fashion by administrators, it tries to its Chubby lock, so that the master server can reassign tablets sooner (cf. [CDG+06, p. 5]).

Master Server Lifecycle When a master server starts up, it also places a special file into a Chubby namespace and acquires an exclusive lock for it (to prevent "concurrent master instantiations"). If the master server is not able to hold that lock so that its Chubby session expires, it takes itself down as it cannot monitor the tablet servers correctly without a reliable connection to Chubby. Hence, the availability of a Bigtable instance relies on the reliability of the connection between master server and the Chubby service used by a Bigtable instance.

In addition to registering itself via a file and its lock in Chubby, the master server processes the following steps when it starts up:

[7] The tuple (table, end row) is sufficient to identify a tablet due to the fact that rows are kept in ascending order by their key.

[8] If such a split notification gets lost (because the tablet server or the master server crashed before it has been exchanged) the master server gets aware of the split as soon as he assigns the formerly unsplitted tablet to a tablet server. The tablet server discovers the tablet split by comparing the end-row of the tablet specified by the master server compared to the end-row specified in the `METADATA` table: if the latter is less then the first, a split has happened of which the master server is unaware. The `METADATA` table contains the latest information about tablet boundaries as tablet servers have to commit tablet splittings to this table before notifying the master server.

1. Discover tablet servers that are alive via scanning the Chubby directory with tablet server files and checking their locks.
2. Connect to each alive tablet server and ask for the tablets it serves at the moment.
3. Scan the METADATA table[9] to construct a list of all tables and tablets. Via subtraction of the tablets already served by the running tablet servers, it derives a set of unassigned tablets.

Tablet Representation

Figure 6.3 depicts how tablets are represented at runtime.

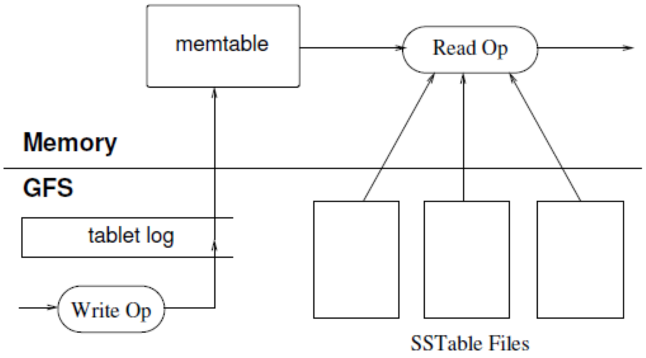


Figure 6.3.: Google's Bigtable – Tablet Representation at Runtime (taken from [CDG+06, p. 6])

All write operations on tablets are "committed to a commit log that stores redo records" and is persisted in the Google File System (GFS). The recently commited updates are put into a sorted RAM-buffer called *memtable*. When a memtable reaches a certain size, it is frozen, a new memtable is created, the frozen memtable gets transformed into the SSTable format and written to GFS; this process is called a *minor compactation*. Hence, the older updates get persisted in a sequence of SSTables on disk while the newer ones are present in memory. The information about where the SSTables comprising a tablet are located is stored in the METADATA table along with a set of pointers directing into one or more commit logs by which the memtable can be reconstructed when the tablet gets assigned to a tablet server.

Write operations are checked for well-formedness as well as authorization before they are written to the commit log and the memtable (when they are finally committed). The authorization-information is provided on column-family base and stored in a Chubby namespace.

Read operations are also checked whether they are well-formed and whether the requesting client is authorized to issue them. If a read operation is permitted, it "is executed on a merged view of the sequence of SSTables and the memtable". The *merged view* required for read operations can be established efficiently as the SSTables and memtable are lexicographically sorted.

Besides minor compactions—the freeze, transformation and persistence of memtables as SSTables—the SSTables also get compacted from time to time. Such a *merging compactation* is executed asynchronously by a background service in a copy-on-modify fashion. The goal of merging compacations is to limit the number of SSTables which have to be considered for read operations.

A special case of merging transformations are so called *major compactations* which produce exactly one SSTable out of a number of SSTables. They are executed by Bigtable regularly to "to reclaim resources

[9] The scanning of the METADATA table is only possible when it is actually assigned to tablet servers. If the master server discovers that this is not the case, it puts the *root tablet* to the set of unassigned tablet, tries to assign it to a tablet server and continues its bootstrapping with scanning the METADATA table as soon as the *root tablet* is assigned.

used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data".

During all compactations read as well as write operations can still be served. This is due to the fact that SSTables as well as a frozen memtable are immutable and only discarded if the compactation was finished successfully. In addition, a memtable for committed write operations is always present.

6.1.5. Refinements

In order to enhance the performance, availability and reliability of Bigtable, several refinements have been implemented at Google (cf. [CDG^{+}06, p. 6ff]):

Locality Groups are groups of column-families that a client application defines and that are expected to be accessed together typically. Locality groups cause Bigtable to create a separate SSTable for each locality group within a tablet. The concept of locality groups is an approach to increase read performance by reducing the number of disk seeks as well as the amount of data to be processed for typical classes of read operations in an application. A further enhancement of read performance can be achieved by requiring a locality group to be served from memory which causes Bigtable to lazily load the associated SSTables into RAM when they are first read. Bigtable internally uses the concept of locality groups served from memory for the `METADATA` table.

Compression of SSTables can be requested by client applications for each locality group. Clients can also specify the algorithm and format to be applied for compression. The SSTable-contents are compressed block-wise by Bigtable which results in lower compression ratios compared to compressing an SSTable as a whole but "we benefit in that small portions of an SSTable can be read without decompressing the entire file". Experiences at Google show that many applications "use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [. . .], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines." Although optimized for speed rather than space, this two-pass scheme results in good compression ratios like 10:1 for Bigtables with similar contents than the Webtable example of figure 6.1 on page 105. This is due to the fact that similar data (e. g. documents from the same domain) are typically clustered together as row keys are lexicographically ordered and are typically chosen such that the lexicographic distance of row keys representing similar data is small.

Caching at Tablet Servers In order to optimize read performance, two levels of caching are implemented at the tablet servers: a cache for key-/value-pairs returned by the SSTable API (called *scan cache*) and a cache for SSTable-blocks (called *block cache*). While the scan cache optimizes read performance if the same data is read repeatedly, the block cache "is useful for applications that tend to read data that is close to the data they recently read".

Bloom Filters To further enhance read performance, applications can require Bigtable to create and leverage bloom filters for a locality group. These bloom filters are used to detect whether an SSTables might contain data for certain key-/value pairs, thereby reducing the number of SSTables to be read from disk when creating the merged view of a tablet required to process read operations.

Group Commits "[The] throughput of lots of small mutations" to the commit log and memtable is improved by a group commit strategy (cf. [CDG^{+}06, p. 6]).

Reduction of Commit Logs Each tablet server keeps only two[10] append-only commit log files for all tablets it serves. This is due to the fact that number of commit logs would grow very if such a file would be created and maintained on a per-tablet base, resulting in many concurrent writes to GFS, potentially large numbers of disk seeks, and less efficiency of the group commit optimization. A downside of this approach appears if tablets get reassigned (e. g. caused by tablet server crashes or tablet redistribution due to load balancing): as a tablet server typically only servers part of the tablets that were served by another server before, it has to read through the commit logs the former server produced to establish the tablet representation shown in figure 6.3 on page 110; hence, if the tablets once served by one tablet server get reassigned to n tablet servers, the commit logs have to be read n times. To tackle this issue, the commits are stored and sorted by the key-triple `(table, row, log sequence number)`. This results in just one disk seek with a subsequent contiguous read for each tablet server that has to evaluate commit logs in order to load tablets formerly served by another tablet server. The sorting of a commit log is optimized in two ways. Firstly, a commit log is sorted lazily: when a tablet server has to serve additional tablets, it beckons the master server that it has to evaluate a commit log so that the master server can initiate the sorting of this commit log. A second optimization is how the commit logs are sorted: the master server splits them up into 64 MB chunks and initiates a parallel sort of these chunks on multiple tablet servers.

Reducing GFS Latency Impacts The distributed Google File Systems is not robust against latency spikes, e. g. due to server crashes or network congestion. To reduce the impact of such latencies, each tablet server uses two writer threads for commit logs, each writing to its own file. Only one of these threads is actively writing to GFS at a given time. If this active thread suffers from GFS "performance hiccups", the commit logging is switched to the second thread. As any operation in a commit log has a unique sequence number, duplicate entries in the two commit logs can be eliminated when a tablet server loads a tablet.

Improving Tablet Recovery *Tablet recovery* is the process of tablet-loading done by a tablet server that a particular tablet has been assigned to. As discussed in this section and in section 6.1.4, a tablet server has to evaluate the commit logs attached that contain operations for the tablet to load. Besides the aforementioned Bloom Filter optimization, Bigtable tries to avoid that a tablet server has to read a commit log at all when recovering a tablet. This is achieved by employing two minor compactations when a tablet server stops serving a tablet. The first compactation is employed to reduce "the amount of uncompacted state in the tablet server's commit log". The second compactation processes the update operations that have been processed since the first compactation was started; before this second compactation is executed, the tablet server stops serving any requests. These optimizations to reduce tablet recovery time can only happen and take effect if a tablet server stops serving tablets in a controlled fashion (i. e. it has not due to a crash).

Exploiting Immutability Bigtable leverages in manifold ways from the fact that SSTables are immutable. Firstly, read operations to SSTables on the filesystem do not have to synchronized. Secondly, the removal of data is delegated to background processes compacting SSTables and garbage-collecting obsolete ones; hence, the removal of data can occur asynchronously and the time required for it does not have to be consumed while a request is served. Finally, when a tablet gets split the resulting child tablets inherit the SSTables from their parent tablet.

To provide efficient read and write access to the mutable memtable, a partial and temporary immutability is introduced by making memtable rows "copy-on-write and allow reads and writes to proceed in parallel".

[10] Chang et al. first speak of only one commit log per tablet server in their section "Commit-log implementation" (cf. [CDG+06, p. 7f]). Later in this section, they introduce the optimization of two commit-log writer threads per tablet server, each writing to its own file (see below).

6.1.6. Lessons Learned

In the design, implementation and usage of Bigtable at Google, a lot of experience has been gained. Chang et al. especially mention the following lessons they learned:

Failure Types in Distributed Systems Chang et al. criticize the assumption made in many distributed protocols that large distributed systems are only vulnerable to few failures like "the standard network partitions and fail-stop failures". In contrast, they faced a lot more issues: "memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance". Hence, they argue that such sources of failure also have to be addressed when designing and implementing distributed systems protocols. Examples that were implemented at Google are checksumming for RPC calls as well as removing assumptions in a part of the system about the other parts of the system (e. g. that only a fixed set of errors can be returned by a service like Chubby).

Feature Implementation A lesson learned at Google while developing Bigtable at Google is to implement new features into such a system only if the actual usage patterns for them are known. A counterexample Chang et al. mention are general purpose distributed transactions that were planned for Bigtable but never implemented as there never was an immediate need for them. It turned out that most applications using Bigtable only needed single-row transactions. The only use case for distributed transactions that came up was the maintenance of secondary indices which can be dealt with by a "specialized mechanism [. . . that] will be less general than distributed transactions, but will be more efficient". Hence, general purpose implementations arising when no actual requirements and usage patterns are specified should be avoided according to Chang et al..

System-Level Monitoring A practical suggestion is to monitor the system as well at its clients in order to detect and analyze problems. In Bigtable e. g. the RPC being used by it produces "a detailed trace of the important actions" which helped to "detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the `METADATA` table when `METADATA` tablets are unavailable".

Value Simple Designs In the eyes of Chang et al. the most important lesson to be learned from Bigtable's development is that simplicity and clarity in design as well as code are of great value—especially for big and unexpectedly evolving systems like Bigtable. As an example they mention the tablet-server membership protocol which was designed too simple at first, refactored iteratively so that it became too complex and too much depending on seldomly used Chubby-features, and in the end was redesigned to "to a newer simpler protocol that depends solely on widely-used Chubby features" (see section 6.1.4).

6.2. Bigtable Derivatives

As the Bigtable code as well as the components required to operate it are not available under an open source or free software licence, open source projects have emerged that are adopting the concepts described in the Bigtable paper by Chang et al.. Notably in this field are Hypertable and HBase.

Hypertable

Hypertable is modelled after Google's Bigtable and inspired by "our own experience in solving large-scale data-intensive tasks" according to its developers. The project's goal is "to set the open source standard for highly available, petabyte scale, database systems". Hypertable is almost completely written in C++ and relies on a distributed filesystem such as Apache Hadoop's HDFS (Hadoop Distributed File System) as well as a distributed lock-manager. Regarding its data model it supports all abstractions available in Bigtable; in contrast to Hbase column-families with an arbitrary numbers of distinct columns are available in Hypertable. Tables are partitioned by ranges of row keys (like in Bigtable) and the resulting partitions get replicated between servers. The data representation and processing at runtime is also borrowed from Bigtable: "[updates] are done in memory and later flushed to disk". Hypertable has its own query language called HQL (Hypertable Query Language) and exposes a native C++ as well as a Thrift API. Originally developed by Zvents Inc., it has been open-sourced under the GPL in 2007 and is sponsored by Baidu, the leading chinese search engine since 2009 (cf. [Hyp09a], [Hyp09b], [Hyp09c], [Cat10], [Jon09], [Nor09], [Int10], [Wik11b]).

HBase

The HBase datastore is a Bigtable-clone developed in Java as a part of Apache's MapReduce-framework Hadoop, providing a "a fault-tolerant way of storing large quantities of sparse data". Like Hypertable, HBase depends on a distributed file system (HDFS) which takes the same role as GFS in the context of Bigtable. Concepts also borrowed from Bigtable are the memory and disk usage pattern with the need for compactations of immutable or append-only files, compression of data as well as bloom filters for the reduction of disk access. HBase databases can be a source of as well as a destination for MapReduce jobs executed via Hadoop. HBase exposes a native API in Java and can also be accessed via Thrift or REST. A notable usage of HBase is the real-time messaging system of Facebook built upon HBase since 2010 (cf. [Apa11], [Nor09], [Jon09], [Int10], [Hof10a], [Wik11a]).

6.3. Cassandra

As a last datastore in this paper Apache Cassandra which adopts ideas and concepts of both, Amazon's Dynamo as well as Google's Bigtable (among others, cf. [LM10, p. 1f]), shall be discussed. It was originally developed by Facebook and open-sourced in 2008. Lakshman describes Cassandra as a "distributed storage system for managing structured data that is designed to scale to a very large size". It "shares many design and implementation strategies with databases" but "does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format" (cf. [Lak08], [LM10, p. 1], [Cat10]). Besides Facebook, other companies have also adopted Cassandra such as Twitter, Digg and Rackspace (cf. [Pop10a], [Eur09], [Oba09a], [Aja09]).

6.3.1. Origin and Major Requirements

The use-case leading to the initial design and development of Cassandra was the so entitled *Inbox Search problem* at Facebook. The worlds largest social network Facebook allows users to exchange personal messages with their contacts which appear in the inbox of a recipient. The *Inbox Search problem* can be described as finding an efficient way of storing, indexing and searching these messages.

The major requirements for the inbox search problem as well as problems of the same nature were (cf. [Lak08], [LM10, p. 1]):

- Processing of a large amount and high growth rate of data (given 100 million users as of June 2008, 250 million as of August 2009 and over 600 billion users as of January 2011; cf. [LM10, p. 1], [Car11])

- High and incremental scalability
- Cost-effectiveness
- "Reliability at massive scale" since "[outages] in the service can have significant negative impact"
- The ability to "run on top of an infrastructure of hundreds of nodes [i. e. commodity servers] (possibly spread across different datacenters)"
- A "high write throughput while not sacrificing read efficiency"
- No single point of failure
- Treatment of failures "as a norm rather than an exception"

After a year of productive usage of Cassandra at Facebook[11] Lakshman summed up that "Cassandra has achieved several goals – scalability, high performance, high availability and applicability" (cf. [Lak08]). In August 2009 Lakshman and Malik affirm that "Cassandra has kept up the promise so far" and state that it was also "deployed as the backend storage system for multiple services within Facebook" (cf. [LM10, p. 1]).

6.3.2. Data Model

An instance of Cassandra typically consists of only one table which represents a "distributed multidimensional map indexed by a key". A table is structured by the following dimensions (cf. [LM10, p. 2], [Lak08]):

Rows which are identified by a string-key of arbitrary length. Operations on rows are "atomic per replica no matter how many columns are being read or written".

Column Families which can occur in arbitrary number per row. As in Bigtable, column-families have to be defined in advance, i. e. before a cluster of servers comprising a Cassandra instance is launched. The number of column-families per table is not limited; however, it is expected that only a few of them are specified. A column family consists of *columns* and *supercolumns*[12] which can be added dynamically (i. e. at runtime) to column-families and are not restricted in number (cf. [Lak08]).

Columns have a name and store a number of values per row which are identified by a timestamp (like in Bigtable). Each row in a table can have a different number of columns, so a table cannot be thought of as a rectangle. Client applications may specify the ordering of columns within a column family and supercolumn which can either be by name or by timestamp.

Supercolumns have a name and an arbitrary number of columns associated with them. Again, the number of columns per super-column may differ per row.

Hence, values in Cassandra are addressed by the triple (`row-key, column-key, timestamp`) with column-key as `column-family:column` (for simple columns contained in the column family) or `column-family:supercolumn:column` (for columns subsumed under a supercolumn).

[11] As of 2008 it ran on more than 600 cores and stored more than 120 TB of index data for Facebook's inbox messaging system.

[12] The paper by Laksman and Malik distinguishes columns and supercolumns at the level of column-families and therefore speaks of *simple column families* and *super column families*. In contrast, the earlier blog post of Lakshman distinguishes *columns* and *supercolumns*. As the latter approach describes the concept precisely and at column family level there is no further distinctive than just the type of assigned columns this paper follows Lakshman's earlier description.

6.3.3. API

The API exposed to client-applications by Cassandra consists of just three operations (cf. [LM10, p. 2]):

- `get(table, key, columnName)`
- `insert(table, key, rowMutation)`
- `delete(table, key, columnName)`

The `columnName` argument of the `get` and `delete` operation identifies either a column or a supercolumn within a column family or column family as a whole .

All requests issued by client-applications get routed to an arbitrary server of a Cassandra cluster which determines the replicas serving the data for the requested key. For the write operations `insert` and `update` a quorum of replica nodes has "to acknowledge the completion of the writes". For read operations, clients can specify which consistency guarantee they desire and based on this definition either the node closest to the client serves the request or a quorum of responses from different nodes is waited for before the request returns; hence, by defining a quorum client-applications can decide which "degree of eventual consistency" they require (cf. [LM10, p. 2]).

The API of Cassandra is exposed via Thrift. In addition, programming language libraries for Java, Ruby, Pyhton, C# and others are available for interaction with Cassandra (cf. [Cat10], [Int10]).

6.3.4. System Architecture

Partitioning

As Cassandra is required to be incrementally scalable, machines can join and leave a cluster (or crash), so that data has to be partitioned and distributed among the nodes of a cluster in a fashion that allows repartitioning and redistribution. The data of a Cassandra table therefore gets partitioned and distributed among the nodes by a consistent hashing function that also preserves the order of row-keys. The order preservation property of the hash function is important to support range scans over the data of a table. Common problems with basic consistent hashing[13] are treated differently by Cassandra compared to e. g. Amazon's Dynamo: while Dynamo hashes physical nodes to the ring multiple times (as virtual nodes[14]), Cassandra measures and analyzes the load information of servers and moves nodes on the consistent hash ring to get the data and processing load balanced. According to Lakshman and Malik this method has been chosen as "it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing" (cf. [LM10, p. 2], [Lak08]).

Replication

To achieve high scalability and durability of a Cassandra cluster, data gets replicated to a number of nodes which can be defined as a replication factor per Cassandra instance. Replication is managed by a *coordinator node* for the particular key being modified; the coordinator node for any key is the first node on the consistent hash ring that is visited when walking from the key's position on the ring in clockwise direction (cf. [LM10, p. 3]).

[13] Non-uniform load and data distribution due to randomly distributed hash values as well as non-consideration of hardware heterogenity (see section 3.2.1 on page 39ff)

[14] The number of virtual nodes per physical node depends on the hardware capabilities of the physical node.

Multiple replication strategies are provided by Cassandra:

- **Rack Unaware** is a replication strategy within a datacenter where $N-1$[15] nodes succeeding the coordinator node on the consistent hash ring are chosen to replicate data to them.
- **Rack Aware** (within a datacenter) and **Datacenter Aware** are replication strategies where by a system called ZooKeeper[16] a leader is elected for the cluster who is in charge of maintaining "the invariant that no node is responsible for more than N-1 ranges in the ring". The metadata about the nodes' responsibilities for key ranges[17] is cached locally at each node as well as in the Zookeeper system. Nodes that have crashed and start up again therefore can determine which key-ranges they are responsible for.

The choice of replica nodes as well as the assignment of nodes and key-ranges also affects durability: to face node failures, network partitions and even entire datacenter failures, the "the preference list of a key is constructed such that the storage nodes are spread across multiple datacenters"[18] (cf. [LM10, p. 3]).

Cluster Membership and Failure Detection

The membership of servers in a Cassandra cluster is managed via a Gossip-style protocol named *Scuttlebutt* (cf. [vDGT08]) which is favored because of its "very efficient CPU utilization and very efficient utilization of the gossip channel" according to Lakshman and Malik. Besides membership management, Scuttlebutt is also used "to disseminate other system related control state" in a Cassandra cluster (cf. [LM10, p. 3], [Lak08]).

Nodes within a Cassandra cluster try to locally detect whether another node is up or down to avoid connection attempts to unreachable nodes. The mechanism employed for this purpose of failure detection is based on "a modified version of the Φ Accrual Failure Detector". The idea behind accrual failure detectors is that no boolean value is emitted by the failure detector but a "suspicion level for [...] monitored nodes" which indicates a probability about whether they are up or down[19]. Lakshman and Malik state that due to their experiences "Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions" (cf. [LM10, p. 3], [Lak08]).

Cluster Management According to the experiences at Facebook Lakshman and Malik assume that "[a] node outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas". Hence, nodes have to be added to and removed from a cluster explicitly by an administrator (cf. [LM10, p. 3]).

Bootstrapping Nodes When a node is added to a cluster, it calculates a random token for its position on the hash ring. This position as well as the key-range is responsible for is stored locally at the node as well as in ZooKeeper. The node then retrieves the addresses of a few nodes of the cluster from a configuration file or by a service like ZooKeeper and announces its arrival to them which in turn spread this information to the whole cluster. By such an announcement, membership information is spread throughout the cluster so that each node can receive requests for any key and route them to the appropriate server. As the arriving node splits a range of keys another server was formerly responsible for, this part of the key-range has to be transferred from the latter to the joining node (cf. [LM10, p. 3f], [Lak08]).

[15] N is the replication factor specified for the Cassandra instance.

[16] ZooKeeper is part of Apache's Hadoop project and provides a "centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services" (cf. [Apa10f]).

[17] This metadata is called *preference list* in Amazon's Dynamo which has a similar concept.

[18] The term *preference list* is borrowed from Amazon's Dynamo which has a similar concept (see section 4.1.3 on page 53ff).

[19] For a Φ accrual failure detector this probability is indicated by the value of Φ, a logarithmically scaled factor.

Persistence

In contrast to Bigtable and its derivatives, Cassandra persists its data to local files instead of a distributed file system. However, the data representation in memory and on disk as well as the processing of read and write operations is borrowed from Bigtable (cf. [LM10, p. 4f]):

- Write operations first go to a persistent commit log and then to an in-memory data structure.
- This in-memory data structure gets persisted to disk as an immutable file if it reaches a certain threshold of size.
- All writes to disk are sequential and an index is created "for efficient lookup based on row key" (like the block-indices of SSTables used in Bigtable).
- Data files on disk get compacted from time to time by a background process.
- Read operations consider the in-memory data structure as well as the data files persisted on disk. "In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory". It "is first consulted to check if the key being looked up does indeed exist in the given file".

In addition, Cassandra maintains indices for column families and columns to "jump to the right chunk on disk for column retrieval" and avoid the scanning of all columns on disk. As write operations go to an append-only commit log and as data files are immutable, "[the] server instance of Cassandra is practically lockless for read/write operations".

6.3.5. Implementation

Cassandra Nodes

A server participating in a Cassandra cluster runs modules providing the following functionality:

- Partitioning
- Cluster membership and failure detection
- Storage engine

These modules are implemented in Java and operate on an event driven software layer whose "message processing pipeline and the task pipeline are split into multiple stages along the line of the SEDA [. . .] architecture" (cf. [WCB01]). The cluster the membership and failure module uses nonblocking I/O for communication. "All system control messages rely on UDP based messaging while the application related messages for replication and request routing relies on TCP" (cf. [LM10, p. 4]).

Request routing is implemented via a state machine on the storage nodes which consists of the following states when a request arrives (cf. [LM10, p. 4]):

1. Identification of the nodes that are in charge of the data for the requested key
2. Route the request to the nodes identified in state 1 and wait for their responses
3. Fail the request and return to the client if the nodes contacted in state 2 do not respond within a configured amount of time
4. "[Figure] out the latest response based on timestamp"
5. "[Schedule] a repair of the data at any replica if they do not have the latest piece of data"

The storage engine module can perform writes either synchronously or asynchronously which is configurable (cf. [LM10, p. 4]).

Commit-Logs

The commit log maintained by Cassandra node locally has to be purged from entries that have already been committed and persisted to disk in the data files. Cassandra uses a rolling log file which is limited by size; at Facebook its threshold is set to 128MB. The commit log also contains a header in which bits are set whenever the in-memory data structure gets dumped to this. If a commit log is purged as it has reached its size limit, these bits are checked to make sure that all commits are persisted via immutable data files.

The commit log can be either written to in normal mode (i. e. synchronously) or in a *fast-sync mode* in which the writes are buffered; if the latter mode is used, the in-memory data gets also buffered before writing to disk. “This implies that there is a potential of data loss on machine crash”, Laksman and Malik state.

Data Files

Data files persisted to disk are partitioned into blocks containing data for 128 row-keys. These blocks are “demarcated by a block index” which “captures the relative offset of a key within the block and the size of its data”. To accelerate access to blocks, their indices are cached in memory. If a key needs to be read and the corresponding block index is not already in memory, the data files are read in reverse time order and their block indexes get loaded into memory.

6.3.6. Lessons Learned

After three years of designing, implementing and operating Cassandra at Facebook, Lakshman and Malik mention the following lessons they have learned in this period:

Cautious Feature Addition Regarding the addition of new features Lakshman and Malik confirm experiences also made by Google with Bigtable, namely “not to add any new feature without understanding the effects of its usage by applications”.

Transactions Lakshman and Malik furthermore confirm the statement of Chang et al. that for most applications atomic operations per row are sufficient and general transactions have mainly been required for maintaining secondary indices—which can be addressed by specialized mechanisms for just this purpose.

Failure Detection Lakshman and Malik have tested various failure detection implementations and discovered that the time to detect a failure can increase “beyond an acceptable limit as the size of the cluster” grows; in an experiment with a cluster consisting of 100 nodes some failure detectors needed up to two minutes to discover a failed node. However, the accrual failure detector featured acceptable detection times (in the aforementioned experiment: 15 seconds).

Monitoring The experiences at Facebook confirm those at Google that monitoring is key to “understand the behavior of the system when subject to production workload” as well as detect errors like disks failing for non-apparent reasons. Cassandra is integrated with the distributed monitoring service Ganglia.

Partial Centralization Lakshman and Malik state that centralized components like ZooKeeper are useful as “having some amount of coordination is essential to making the implementation of some distributed features tractable”.

7. Conclusion

The aim of this paper was to give a thorough overview and introduction to the NoSQL database movement which appeared in the recent years to provide alternatives to the predominant relational database management systems. Chapter 2 discussed reasons, rationales and motives for the development and usage of nonrelational database systems. These can be summarized by the need for high scalability, the processing of large amounts of data, the ability to distribute data among many (often commodity) servers, consequently a distribution-aware design of DBMSs (instead of adding such facilities on top) as well as a smooth integration with programming languages and their data structures (instead of e. g. costly object-relational mapping). As shown in chapter 2, relational DBMSs have certain flaws and limitations regarding these requirements as they were designed in a time where hardware (especially main-memory) was expensive and full dynamic querying was expected to be the most important use case; as shown by Stonebraker et al. the situation today is very different, so a complete redesign of database management systems is suggested. Because of the limitations of relational DBMSs and today's needs, a wide range of non-relational datastores has emerged. Chapter 2 outlines several attempts to classify and characterize them.

Chapter 3 introduced concepts, techniques and patterns that are commonly used by NoSQL databases to address consistency, partitioning, storage layout, querying, and distributed data processing. Important concepts in this field—like eventual consistency and ACID vs. BASE transaction characteristics—have been discussed along with a number of notable techniques such as multi-version storage, vector clocks, state vs. operational transfer models, consistent hashing, MapReduce, and row-based vs. columnar vs. log-structured merge tree persistence.

As a first class of NoSQL databases, key-/value-stores have been examined in chapter 4. Most of these datastores heavily borrow from Amazon's Dynamo, a proprietary, fully distributed, eventual consistent key-/value-store which has been discussed in detail in this paper. The chapter also looked at popular open-source key-/value-stores like Project Voldemort, Tokyo Cabinet/Tyrant, Redis as well as MemcacheDB.

Chapter 5 has discussed document stores by observing CouchDB and MongoDB as the two major representatives of this class of NoSQL databases. These document stores provide the abstraction of documents which are flat or nested namespaces for key-/value-pairs. CouchDB is a document store written in Erlang and accessible via a RESTful HTTP-interface providing multi-version concurrency control and replication between servers. MongoDB is a datastore with additional features such as nested documents, rich dynamic querying capabilities and automatic sharding.

In chapter 6 column-stores have been discussed as a third class of NoSQL databases. Besides pure column-stores for analytics datastores integrating column- and row-orientation can be subsumed in this field. An important representative of the latter is Google's Bigtable which allows to store multidimensional maps indexed by row, column-family, column and timestamp. Via a central master server, Bigtable automatically partitions and distributes data among multiple tablet servers of a cluster. The design and implementation of the proprietary Bigtable have been adopted by open-source projects like Hypertable and HBase. The chapter concludes with an examination of Apache Cassandra which integrates the full-distribution and eventual consistency of Amazon's Dynamo with the data model of Google's Bigtable.

A. Further Reading, Listening and Watching

In addition to the references in the bibliography of this paper, the following resources are suggested to the interested reader.

SQL vs. NoSQL databases

- *One Size Fits All: An Idea whose Time has Come* and Gone by Michael Stonebraker and Uğur Çetintemel:
  http://www.cs.brown.edu/~ugur/fits_all.pdf
- *What Should I do? – Choosing SQL, NoSQL or Both for Scalable Web Apps* by Todd Hoff:
  http://voltdb.com/webcast-choosing-sql-nosql-or-both-scalable-web-apps
- *SQL Databases Don't Scale* by Adam Wiggins:
  http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/
- *6 Reasons Why Relational Database Will Be Superseded* by Robin Bloor:
  http://www.havemacwillblog.com/2008/11/6-reasons-why-relational-database-will-be-superseded/

Concepts, Techniques and Patterns

Introduction and Overview

- *Scalability, Availability & Stability Patterns* by Jonas Bonér:
  http://www.slideshare.net/jboner/scalability-availability-stability-patterns
- *Architecting for the Cloud – Horizontal Scalability via Transient, Shardable, Share-Nothing Resources* by Adam Wiggins:
  http://www.infoq.com/presentations/Horizontal-Scalability
- *Cluster-based scalable network services* by Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer and Paul Gauthier:
  http://www.cs.berkeley.edu/~brewer/cs262b/TACC.pdf

CAP-Theorem, BASE and Eventual Consistency

- *Brewer's CAP Theorem* by Julian Browne:
  http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
- *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services* by Seth Gilbert and Nancy Lynch:
  http://www.cs.utsa.edu/~shxu/CS6393-Fall2007/presentation/paper-18.pdf

- *Errors in Database Systems, Eventual Consistency, and the CAP Theorem* by Michael Stonebraker: http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext
- *BASE: An Acid Alternative* by Dan Prichett: http://queue.acm.org/detail.cfm?id=1394128
- *Eventually consistent* by Werner Vogels: http://www.allthingsdistributed.com/2007/12/eventually_consistent.html http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
- *I love eventual consistency but...* by James Hamilton: http://perspectives.mvdirona.com/2010/02/24/ILoveEventualConsistencyBut.aspx

Paxos Consensus Protocol

- *Paxos made simple* by Leslie Lamport: http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
- *Time, clocks, and the ordering of events in a distributed system* by Leslie Lamport: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.3682&rep=rep1&type=pdf
- *The part-time parliament* by Leslie Lamport: http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

Chord

- *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications* by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan: http://www.sigcomm.org/sigcomm2001/p12-stoica.pdf
- *The Chord/DHash Project*: http://pdos.csail.mit.edu/chord/

MapReduce (Critical Remarks)

- *A Comparison of Approaches to Large-Scale Data Analysis* by Michael Stonebraker, Andrew Pavlo, Erik Paulson et al.: http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf
- *MapReduce: A major step backwards* by David DeWitt: http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/

NoSQL

Overview and Introduction

- Directory of NoSQL databases with basic information on the individual datastores: http://nosql-database.org/
- Software Engineering Radio Podcast, Episode 165: *NoSQL and MongoDB with Dwight Merriman* by Robert Blumen and Dwight Merriman: http://www.se-radio.net/2010/07/episode-165-nosql-and-mongodb-with-dwight-merriman/
- heise SoftwareArchitekTOUR Podcast (German), Episode 22: *NoSQL – Alternative zu relationalen Datenbanken* by Markus Völter, Stefan Tilkov and Mathias Meyer: http://www.heise.de/developer/artikel/Episode-22-NoSQL-Alternative-zu-relationalen-Datenbanken-1027769.html
- RadioTux Binärgewitter Podcast (German), Episode 1: *NoSQL* by Dirk Deimeke, Marc Seeger, Sven Pfleiderer and Ingo Ebel: http://blog.radiotux.de/2011/01/09/binaergewitter-1-nosql/
- *NoSQL: Distributed and Scalable Non-Relational Database Systems* by Jeremy Zawodny: http://www.linux-mag.com/id/7579
- freiesMagazin (German), issue 08/2010: *NoSQL – Jenseits der relationalen Datenbanken* by Jochen Schnelle: http://www.freiesmagazin.de/mobil/freiesMagazin-2010-08-bilder.html#10_08_nosql

Key-/Value Stores

- *Amazon Dynamo: The Next Generation Of Virtual Distributed Storage* by Alex Iskold: http://www.readwriteweb.com/archives/amazon_dynamo.php
- Software Engineering Radio Podcast, Episode 162: *Project Voldemort with Jay Kreps* by Robert Blumen and Jay Kreps: http://www.se-radio.net/2010/05/episode-162-project-voldemort-with-jay-kreps/
- *Erlang eXchange 2008: Building a Transactional Data Store* (Scalaris) by Alexander Reinefeld: http://video.google.com/videoplay?docid=6981137233069932108
- *Why you won't be building your killer app on a distributed hash table* by Jonathan Ellis: http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html

Document Databases

- freiesMagazin (German), issue 06/2010: *CouchDB – Datenbank mal anders* by Jochen Schnelle: http://www.freiesmagazin.de/mobil/freiesMagazin-2010-06-bilder.html#10_06_couchdb
- *Introducing MongoDB* by Eliot Horrowitz: http://www.linux-mag.com/id/7530

Column Stores

- *Distinguishing Two Major Types of Column-Stores* by Daniel Abadi:
  http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html
- *Cassandra – Structured Storage System over a P2P Network* by Avinash Lakshman, Prashant Malik and Karthik Ranganathan:
  http://www.slideshare.net/jhammerb/data-presentations-cassandra-sigmod
- *4 Months with Cassandra, a love story* by Cloudkick:
  https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/
- *Saying Yes To NoSQL; Going Steady With Cassandra At Digg* by John Quinn:
  http://about.digg.com/node/564
- *up and running with cassandra* by Evan Weaver:
  http://blog.evanweaver.com/2009/07/06/up-and-running-with-cassandra/
- techZing! Podcast, Episode 8: *Dude, Where's My Database?!* by Justin Vincent, Jason Roberts and Jonathan Ellis:
  http://techzinglive.com/page/75/techzing-8-dude-wheres-my-database
- *Cassandra: Fact vs fiction* by Jonathan Ellis:
  http://spyced.blogspot.com/2010/04/cassandra-fact-vs-fiction.html
- *Why we're using HBase* by Cosmin Lehene:
  http://hstack.org/why-were-using-hbase-part-1/
  http://hstack.org/why-were-using-hbase-part-2/

Graph Databases

- *Neo4j - A Graph Database That Kicks Buttox* by Todd Hoff:
  http://highscalability.com/blog/2009/6/13/neo4j-a-graph-database-that-kicks-buttox.html
- JAXenter (German): *Graphendatenbanken, NoSQL und Neo4j* by Peter Neubauer:
  http://it-republik.de/jaxenter/artikel/Graphendatenbanken-NoSQL-und-Neo4j-2906.html
- JAXenter (German): *Neo4j - die High-Performance-Graphendatenbank* by Peter Neubauer:
  http://it-republik.de/jaxenter/artikel/Neo4j-%96-die-High-Performance-Graphendatenbank-2919.html
- *Presentation: Graphs && Neo4j => teh awesome!* by Alex Popescu:
  http://nosql.mypopescu.com/post/342947902/presentation-graphs-neo4j-teh-awesome
- *Product: HyperGraphDB – A Graph Database* by Todd Hoff:
  http://highscalability.com/blog/2010/1/26/product-hypergraphdb-a-graph-database.html
- RadioTux (German): *Sendung über die GraphDB* by Alexander Oelling and Ingo Ebel:
  http://blog.radiotux.de/2010/12/13/sendung-graphdb/

Conference Slides and Recordings

- NOSQL debrief San Francisco on 2009-06-11:
  http://blog.oskarsson.nu/2009/06/nosql-debrief.html
  http://www.johnandcailin.com/blog/john/san-francisco-nosql-meetup
- NoSQL Berlin on 2009-10-22:
  http://www.nosqlberlin.de/

Evaluation and Comparison of NoSQL Databases

- *NoSQL Ecosystem* by Jonathan Ellis:
  http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/
- *NoSQL: If only it was that easy* by BJ Clark:
  http://bjclark.me/2009/08/04/nosql-if-only-it-was-that-easy/
- *The end of SQL and relational databases?* by David Intersimone:
  http://blogs.computerworld.com/15510/the_end_of_sql_and_relational_databases_part_1_of_3
  http://blogs.computerworld.com/15556/the_end_of_sql_and_relational_databases_part_2_of_3
  http://blogs.computerworld.com/15641/the_end_of_sql_and_relational_databases_part_3_of_3
- *Performance comparison: key/value stores for language model counts* by Brendan O'Connor:
  http://anyall.org/blog/2009/04/performance-comparison-keyvalue-stores-for-language-model-counts/
- *Redis Performance on EC2 (aka weekend project coming)* by Michal Frackowiak:
  http://michalfrackowiak.com/blog:redis-performance
- *MySQL-Memcached or NOSQL Tokyo Tyrant* by Matt Yonkovit:
  http://www.mysqlperformanceblog.com/2009/10/15/mysql-memcached-or-nosql-tokyo-tyrant-part-1/
  http://www.mysqlperformanceblog.com/2009/10/16/mysql_memcached_tyrant_part2/
  http://www.mysqlperformanceblog.com/2009/10/19/mysql_memcached_tyrant_part3/
- *Redis vs MySQL vs Tokyo Tyrant (on EC2)* by Colin Howe:
  http://colinhowe.wordpress.com/2009/04/27/redis-vs-mysql/

B. List of abbreviations

2PC	**Two-p**hase **c**ommit
ACID	**A**tomicity **C**onsistency **I**solation **D**urability
ACM	**A**ssociation for **C**omputing **M**achinery
ADO	**A**ctiveX **D**ata **O**bjects
aka	**a**lso **k**nown **a**s
API	**A**pplication **P**rogramming **I**nterface
BASE	**B**asically **A**vailable, **S**oft-State, **E**ventual Consistency
BBC	**B**ritish **B**roadcasting **C**orporation
BDB	**B**erkley **D**ata**b**ase
BLOB	**B**inary **L**arge **Ob**ject
CAP	**C**onsistency, **A**vailability, **P**artition Tolerance
CEO	**C**hief **E**xecutive **O**fficer
CPU	**C**entral **P**rocessing **U**nit
CS	**C**omputer **S**cience
CTA	**C**onstrained **t**ree **a**pplication
CTO	**C**hief **T**echnology **O**fficer
DNS	**D**omain **N**ame **S**ystem
DOI	**D**igital **O**bject **I**dentifier
DBA	**D**ata**b**ase **a**dministrator
EAV store	**E**ntity-**A**ttribute-**V**alue **store**
EC2	(Amazon's) **E**lastic **C**loud **C**omputing
EJB	**E**nterprise **J**ava**B**eans
Erlang OTP	**Erlang O**pen **T**elecommunication **P**latform
E-R Model	**E**ntity-**R**elationship **Model**
FIFO	**F**irst **i**n, **f**irst **o**ut
GFS	**G**oogle **F**ile **S**ystem
GPL	Gnu **G**eneral **P**ublic **L**icence
HA	**H**igh **A**vailability

HDFS	**H**adoop **D**istributed **F**ile **S**ystem
HQL	**H**ypertable **Q**uery **L**anguage
IBM	**I**nternational **B**usiness **M**achines
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IETF	**I**nternet **E**ngineering **T**ask **F**orce
IO	**I**nput **O**utput
IP	**I**nternet **P**rotocol
JDBC	**J**ava **D**ata**b**ase **C**onnectivity
JPA	**J**ava **P**ersistence **A**PI
JSON	**J**ava**S**cript **O**bject **N**otation
JSR	**J**ava **S**pecification **R**equest
LINQ	**L**anguage **In**tegrated **Q**uery
LRU	**L**east **r**ecently **u**sed
LSM(-Tree)	**L**og **S**tructured **M**erge (**Tree**)
LZW	**L**empel-**Z**iv-**W**elch
MIT	**M**assachusetts **I**nstitute of **T**echnology
MVCC	**M**ulti-**v**ersion **c**oncurrency **c**ontrol
(Java) NIO	(Java) **N**ew **I**/**O**
ODBC	**O**pen **D**ata**b**ase **C**onnectivity
OLAP	**O**n**l**ine **A**nalytical **P**rocessing
OLTP	**O**n**l**ine **T**ransaction **P**rocessing
OSCON	O’Reilly **O**pen **S**ource **Con**vention
OSS	**O**pen **S**ource **S**oftware
PCRE	**P**erl-**c**ompatible **R**egular **E**xpressions
PNUTS	(Yahoo!’s) **P**latform for **N**imble **U**niversal **T**able **S**torage
PODC	**P**rinciples **o**f **d**istributed **c**omputing (ACM symposium)
RAC	(Oracle) **R**eal **A**pplication **C**luster
RAM	**R**andom **A**ccess **M**emory
RDF	**R**esource **D**escription **F**ramework
RDS	(Amazon) **R**elational **D**atabase **S**ervice
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
REST	**Re**presentational **S**tate **T**ransfer
RPC	**R**emote **P**rocedure **C**all

RoR	**R**uby **o**n **R**ails
RYOW	**R**ead **y**our **o**wn **w**rites (consistency model)
(Amazon) S3	(Amazon) **S**imple **S**torage **S**ervice
SAN	**S**torage **A**rea **N**etwork
SLA	**S**ervice **L**evel **A**greement
SMP	**S**ymmetric **m**ulti**p**rocessing
SPOF	**S**ingle **p**oint **o**f **f**ailure
SSD	**S**olid **S**tate **D**isk
SQL	**S**tructured **Q**uery **L**anguage
TCP	**T**ransmission **C**ontrol **P**rotocol
TPC	**T**ransaction **P**erformance Processing **C**ouncil
US	**U**nited **S**tates
URI	**U**niform **R**esource **I**dentifier
URL	**U**niform **R**esource **L**ocator
XML	E**x**tensible **M**arkup **L**anguage

C. Bibliography

[10g10] 10gen, Inc: *mongoDB*. 2010. –
http://www.mongodb.org

[AGS08] Aguilera, Marcos K. ; Golab, Wojciech ; Shah, Mehul A.: A Practical Scalable Distributed B-Tree. In: *PVLDB '08: Proceedings of the VLDB Endowment* Vol. 1, VLDB Endowment, August 2008, p. 598–609. – Available online.
http://www.vldb.org/pvldb/1/1453922.pdf

[Aja09] Ajatus Software: *The Future of Scalable Databases*. October 2009. – Blog post of 2009-10-08.
http://www.ajatus.in/2009/10/the-future-of-scalable-databases/

[Ake09] Aker, Brian: *Your guide to NoSQL*. November 2009. – Talk at OpenSQLCamp in November 2009.
http://www.youtube.com/watch?v=LhnGarRsKnA

[Ama10a] Amazon.com, Inc.: *Amazon Simple Storage Service (Amazon S3)*. 2010. –
http://aws.amazon.com/s3/

[Ama10b] Amazon.com, Inc.: *Amazon SimpleDB*. 2010. –
http://aws.amazon.com/simpledb/

[Apa09] Apache Software Foundation: *Lucene*. 2009. –
http://lucene.apache.org/

[Apa10a] Apache Software Foundation: *Apache CouchDB – Introduction*. 2008–2010. –
http://couchdb.apache.org/docs/intro.html

[Apa10b] Apache Software Foundation: *Apache CouchDB – Technical Overview*. 2008–2010. –
http://couchdb.apache.org/docs/overview.html

[Apa10c] Apache Software Foundation: *The CouchDB Project*. 2008–2010. –
http://couchdb.apache.org/

[Apa10d] Apache Software Foundation: *The Apache Cassandra Project*. 2010. –
http://cassandra.apache.org/

[Apa10e] Apache Software Foundation: *Introduction to CouchDB Views*. September 2010. – Wiki article, version 35 of 2010-09-08.
http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views

[Apa10f] Apache Software Foundation: *Welcome to Apache ZooKeeper!* 2010. –
http://hadoop.apache.org/zookeeper/

[Apa11] Apache Software Foundation: *HBase*. 2011. –
http://hbase.apache.org/

[BCM+10] Banker, Kyle ; Chodorow, Kristina ; Merriman, Dwight et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Tutorial*. August 2010. – Wiki article, version 22 of 2010-08-11.
http://www.mongodb.org/display/DOCS/Replica+Set+Tutorial

[Bem10] Bemmann, Dennis: *Skalierbarkeit, Datenschutz und die Geschichte von StudiVZ*. January 2010. – Talk at Stuttgart Media University's Social Networks Day on 2010-01-22.
http://days.mi.hdm-stuttgart.de/socalnetsday09/skalierbarkeit-datenschutz-und-geschichte-von-studivz.wmv

[Bez93] Bezdek, James C.: Fuzzy models—what are they, and why. In: *IEEE Transactions on Fuzzy Systems*, 1993, p. 1–6

[BH05] Box, Don ; Hejlsberg, Anders: *The LINQ Project*. September 2005. –
http://msdn.microsoft.com/de-de/library/aa479865.aspx

[Blo70] Bloom, Burton H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. In: *Communications of the ACM* 13 (1970), p. 422–426

[Blo01] Bloch, Joshua: *Effective Java – Programming Language Guide*. Amsterdam : Addison-Wesley Longman, 2001

[BMH10] Banker, Kyle ; Merriman, Dwight ; Horowitz, Eliot: *mongoDB Manual – Admin Zone – Replication – Halted Replication*. August 2010. – Wiki article, version 18 of 2010-08-04.
http://www.mongodb.org/display/DOCS/Halted+Replication

[BO06] Biswas, Rahul ; Ort, Ed: *The Java Persistence API – A Simpler Programming Model for Entity Persistence*. May 2006. –
http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html

[Bre00] Brewer, Eric A.: *Towards Robust Distributed Systems*. Portland, Oregon, July 2000. – Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on 2000-07-19.
http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[Bur06] Burrows, Mike: The Chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2006 (OSDI '06), p. 335–350. – Also available online.
http://labs.google.com/papers/chubby-osdi06.pdf

[C+10] Czura, Martin et al.: *CouchDB In The Wild*. 2008–2010. – Wiki article, version 97 of 2010-09-28.
http://wiki.apache.org/couchdb/CouchDB_in_the_wild

[Can10a] Canonical Ltd.: *ubuntu one*. 2008–2010. –
https://one.ubuntu.com/

[Can10b] Canonical Ltd.: *ubuntu*. 2010. –
http://www.ubuntu.com/

[Can10c] Canonical Ltd.: *UbuntuOne*. July 2010. – Wiki article, version 89 of 2010-07-21.
https://wiki.ubuntu.com/UbuntuOne

[Car11] Carlson, Nicholas: Facebook Has More Than 600 Million Users, Goldman Tells Clients. In: *Business Insider* (2011)

[Cat10] Cattell, Rick: *High Performance Scalable Data Stores*. February 2010. – Article of 2010-02-22.
http://cattell.net/datastores/Datastores.pdf

[CB74] Chamberlin, Donald D. ; Boyce, Raymond F.: SEQUEL: A structured English query language. In: *SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA : ACM, 1974, p. 249–264

[CB10] Chodorow, Kristina ; Banker, Kyle: *mongoDB manual – Databases*. Feburary 2010. – Wiki article, version 4 of 2010-02-04.
http://www.mongodb.org/display/DOCS/Databases

[CBH10a] Chodorow, Kristina ; Banker, Kyle ; Hernandez, Scott: *mongoDB Manual – Collections*. June 2010. – Wiki article, version 5 of 2010-06-07.
http://www.mongodb.org/display/DOCS/Collections

[CBH+10b] Chodorow, Kristina ; Banker, Kyle ; Hernandez, Scott et al.: *mongoDB Manual – Inserting*. August 2010. – Wiki article, version 12 of 2010-08-29.
http://www.mongodb.org/display/DOCS/Inserting

[CDG+06] Chang, Fay ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: *Bigtable: A Distributed Storage System for Structured Data*. November 2006. –
http://labs.google.com/papers/bigtable-osdi06.pdf

[CDM+10] Chodorow, Kristina ; Dirolf, Mike ; Merriman, Dwight et al.: *mongoDB Manual – GridFS*. April 2010. – Wiki article, version 13 of 2010-04-16.
http://www.mongodb.org/display/DOCS/GridFS

[CGR07] Chandra, Tushar D. ; Griesemer, Robert ; Redstone, Joshua: Paxos Made Live – An Engineering Perspective. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing*. New York, NY, USA : ACM, 2007 (PODC '07), p. 398–407. – Also avaialable online.
http://labs.google.com/papers/paxos_made_live.pdf

[CHD+10] Chodorow, Kristina ; Horowitz, Eliot ; Dirolf, Mike et al.: *mongoDB Manual – Updating*. November 2010. – Wiki article, version 72 of 2010-11-12.
http://www.mongodb.org/display/DOCS/Updating

[CHM10a] Chodorow, Kristina ; Horowitz, Eliot ; Merriman, Dwight: *mongoDB Manual – Data Types and Conventions*. February 2010. – Wiki article, version 8 of 2010-02-25.
http://www.mongodb.org/display/DOCS/Data+Types+and+Conventions

[CHM+10b] Chodorow, Kristina ; Horowitz, Eliot ; Merriman, Dwight et al.: *mongoDB Manual – Database – Commands – List of Database Commands*. September 2010. – Wiki article, version 52 of 2010-09-01.
http://www.mongodb.org/display/DOCS/List+of+Database+Commands

[Chu09] Chu, Steve: *MemcacheDB*. January 2009. –
http://memcachedb.org/

[Cla09] Clark, BJ: *NoSQL: If only it was that easy*. August 2009. – Blog post of 2009-08-04.
http://bjclark.me/2009/08/04/nosql-if-only-it-was-that-easy/

[CMB+10] Chodorow, Kristina ; Merriman, Dwight ; Banker, Kyle et al.: *mongoDB Manual – Optimization*. November 2010. – Wiki article, version 17 of 2010-11-24.
http://www.mongodb.org/display/DOCS/Optimization

[CMH+10] Chodorow, Kristina ; Merriman, Dwight ; Horowitz, Eliot et al.: *mongoDB Manual – Querying*. August 2010. – Wiki article, version 23 of 2010-08-04.
http://www.mongodb.org/display/DOCS/Querying

[CNM+10] Chodorow, Kristina ; Nitz, Ryan ; Merriman, Dwight et al.: *mongoDB Manual – Removing*. March 2010. – Wiki article, version 10 of 2010-03-10.
http://www.mongodb.org/display/DOCS/Removing

[Cod70] Codd, Edgar F.: A Relational Model of Data for Large Shared Data Banks. In: *Communications of the ACM* 13 (1970), June, No. 6, p. 377–387

[Com09a] Computerworld: *No to SQL? Anti-database movement gains steam*. June 2009. –
http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam

[Com09b] Computerworld: *Researchers: Databases still beat Google's MapReduce*. April 2009. –
http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9131526

[Cop10] Copeland, Rick: *How Python, TurboGears, and MongoDB are Transforming SourceForge.net*. Feburary 2010. – Presentation at PyCon in Atlanta on 2010-02-20.
http://us.pycon.org/2010/conference/schedule/event/110/

[Cro06] Crockford, D. ; IETF (Internet Engineering Task Force) (Ed.): *The application/json Media Type for JavaScript Object Notation (JSON)*. July 2006. – RFC 4627 (Informational).
http://tools.ietf.org/html/rfc4627

[CRS+08] Cooper, Brian F. ; Ramakrishnan, Raghu ; Srivastava, Utkarsh ; Silberstein, Adam ; Bohannon, Philip ; Jacobsen, Hans A. ; Puz, Nick ; Weaver, Daniel ; Yerneni, Ramana: PNUTS: Yahoo!'s hosted data serving platform. In: *Proc. VLDB Endow.* 1 (2008), August, No. 2, p. 1277–1288. – Also available online.
http://research.yahoo.com/files/pnuts.pdf

[DBC10] Dirolf, Mike ; Banker, Kyle ; Chodorow, Kristina: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Configuration – Adding a New Set Member*. November 2010. – Wiki article, version 5 of 2010-11-19.
http://www.mongodb.org/display/DOCS/Adding+a+New+Set+Member

[DG04] Dean, Jeffrey ; Ghemawat, Sanjay: *MapReduce: simplified data processing on large clusters*. Berkeley, CA, USA, 2004. –
http://labs.google.com/papers/mapreduce-osdi04.pdf

[DHJ+07] DeCandia, Giuseppe ; Hastorun, Deniz ; Jampani, Madan ; Kakulapati, Gunavardhan ; Lakshman, Avinash ; Pilchin, Alex ; Sivasubramanian, Swaminathan ; Vosshall, Peter ; Vogels, Werner: *Dynamo: Amazon's Highly Available Key-value Store*. September 2007. –
http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf

[DKE06] DeMichiel, Linda ; Keith, Michael ; EJB 3.0 Expert Group: *JSR 220: Enterprise JavaBeans^{TM}, Version 3.0 – Java Persistence API*. May 2006. –
http://jcp.org/aboutJava/communityprocess/final/jsr220/

[Dzi10] Dziuba, Ted: *I Can't Wait for NoSQL to Die*. March 2010. – Blogpost of 2010-03-04. http://teddziuba.com/2010/03/i-cant-wait-for-nosql-to-die.html

[Eif09] Eifrem, Emil: *Neo4j – The Benefits of Graph Databases*. July 2009. – OSCON presentation. http://www.slideshare.net/emileifrem/neo4j-the-benefits-of-graph-databases-oscon-2009

[Ell09a] Ellis, Jonathan: *NoSQL Ecosystem*. November 2009. – Blog post of 2009-11-09. http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/

[Ell09b] Ellis, Jonathan: *Why you won't be building your killer app on a distributed hash table*. May 2009. – Blog post of 2009-05-27. http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html

[Eri10] Ericsson Computer Science Laboratory: *Erlang Programming Language, Official Website*. 2010. – http://www.erlang.org/

[Eur09] Eure, Ian: *Looking to the future with Cassandra*. September 2009. – Blog post of 2009-09-09. http://about.digg.com/blog/looking-future-cassandra

[Eva09a] Evans, Eric: *NOSQL 2009*. May 2009. – Blog post of 2009-05-12. http://blog.sym-link.com/2009/05/12/nosql_2009.html

[Eva09b] Evans, Eric: *NoSQL: What's in a name?* October 2009. – Blog post of 2009-10-30. http://www.deadcafe.org/2009/10/30/nosql_whats_in_a_name.html

[F+09] Fitzpatrick, Brad et al.: *memcached – Clients*. December 2009. – Wiki article of 2009-12-10. http://code.google.com/p/memcached/wiki/Clients

[F+10a] Fitzpatrick, Brad et al.: *Memcached*. April 2010. – http://memcached.org

[F+10b] Fitzpatrick, Brad et al.: *memcached – FAQ*. February 2010. – Wiki article of 2010-02-10. http://code.google.com/p/memcached/wiki/FAQ

[F+10c] Fitzpatrick, Brad et al.: *Protocol*. March 2010. – http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt

[FAL10a] FAL Labs: *Tokyo Cabinet: a modern implementation of DBM*. 2006–2010. – http://1978th.net/tokyocabinet/

[FAL10b] FAL Labs: *Tokyo Tyrant: network interface of Tokyo Cabinet*. 2007–2010. – http://fallabs.com/tokyotyrant/

[Far09] Farrell, Enda: *Erlang at the BBC*. 2009. – Presentation at the Erlang Factory Conference in London. http://www.erlang-factory.com/upload/presentations/147/EndaFarrell-ErlangFactory-London2009-ErlangattheBBC.pdf

[FGM+99] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T. ; IETF (Internet Engineering Task Force) (Ed.): *Hypertext Transfer Protocol – HTTP/1.1*. Juni 1999. – RFC 2616 (Draft Standard). http://www.ietf.org/rfc/rfc2616.txt

[Fie00] Fielding, Roy T.: *Architectural styles and the design of network-based software architectures*, University of California, Irvine, Diss., 2000

[For10] Forbes, Dennis: *Getting Real about NoSQL and the SQL-Isn't-Scalable Lie*. March 2010. – Blog post of 2010-03-02.
http://www.yafla.com/dforbes/Getting_Real_about_NoSQL_and_the_SQL_Isnt_-Scalable_Lie/

[FRLL10] Ferguson, Kevin ; Raghunathan, Vijay ; Leeds, Randall ; Lindsay, Shaun: *Lounge*. 2010. –
http://tilgovi.github.com/couchdb-lounge/

[GL03] Gobioff, Sanjay Ghemawat H. ; Leung, Shun-Tak: The Google File System. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), No. 5, p. 29–43. –
http://labs.google.com/papers/gfs-sosp2003.pdf

[Goo10a] Google Inc.: *Google Code – protobuf*. 2010. –
http://code.google.com/p/protobuf/

[Goo10b] Google Inc.: *Google Code – Protocol Buffers*. 2010. –
http://code.google.com/intl/de/apis/protocolbuffers/

[Gos07] Gosling, James: *The Eight Fallacies of Distributed Computing*. 2007. –
http://blogs.sun.com/jag/resource/Fallacies.html

[Gra09] Gray, Jonathan: *CAP Theorem*. August 2009. – Blog post of 2009-08-24.
http://devblog.streamy.com/2009/08/24/cap-theorem/

[Gri08] Griesemer, Robert: Parallelism by design: data analysis with sawzall. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York : ACM, 2008 (CGO '08), p. 3–3

[Ham09] Hamilton, David: *One size does not fit all*. November 2009. – Blog post of 2009-11-03.
http://perspectives.mvdirona.com/2009/11/03/OneSizeDoesNotFitAll.aspx

[HAMS08] Harizopoulos, Stavros ; Abadi, Daniel J. ; Madden, Samuel ; Stonebraker, Michael: OLTP through the looking glass, and what we found there. In: *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2008, p. 981–992

[Haz10] Hazel, Philip: *PCRE - Perl-compatible regular expressions*. January 2010. – PCRE Man Pages, last updated 2010-01-03.
http://www.pcre.org/pcre.txt

[HCS+10] Horowitz, Eliot ; Chodorow, Kristina ; Staple, Aaron et al.: *mongoDB Manual – Drivers*. November 2010. – Wiki article, version 92 of 2010-11-08.
http://www.mongodb.org/display/DOCS/Drivers

[Hey10] Heymann, Harry: *MongoDB at foursquare*. May 2010. – Presentation at MongoNYC in New York on 2010-05-21.
http://blip.tv/file/3704098

[HL10] Horowitz, Eliot ; Lerner, Alberto: *mongoDB Manual – Admin Zone – Sharding – Upgrading from a Non-Sharded System*. August 2010. – Wiki article, version 6 of 2010-08-05.
http://www.mongodb.org/display/DOCS/Upgrading+from+a+Non-Sharded+System

[HMC+10] Horowitz, Eliot ; Merriman, Dwight ; Chodorow, Kristina et al.: *mongoDB Manual – MapReduce*. November 2010. – Wiki article, version 80 of 2010-11-24.
http://www.mongodb.org/display/DOCS/MapReduce

[HMS+10] Horowitz, Eliot ; Merriman, Dwight ; Stearn, Mathias et al.: *mongoDB Manual – Indexes – Geospatial Indexing*. November 2010. – Wiki article, version 49 of 2010-11-03. http://www.mongodb.org/display/DOCS/Geospatial+Indexing

[Ho09a] Ho, Ricky: *NOSQL Patterns*. November 2009. – Blog post of 2009-11-15. http://horicky.blogspot.com/2009/11/nosql-patterns.html

[Ho09b] Ho, Ricky: *Query processing for NOSQL DB*. November 2009. – Blog post of 2009-11-28. http://horicky.blogspot.com/2009/11/query-processing-for-nosql-db.html

[Hof09a] Hoff, Todd: *And the winner is: MySQL or Memcached or Tokyo Tyrant?* October 2009. – Blog post of 2009-10-28. http://highscalability.com/blog/2009/10/28/and-the-winner-is-mysql-or-memcached-or-tokyo-tyrant.html

[Hof09b] Hoff, Todd: *Damn, which database do I use now?* November 2009. – Blog post of 2009-11-04. http://highscalability.com/blog/2009/11/4/damn-which-database-do-i-use-now.html

[Hof09c] Hoff, Todd: *A Yes for a NoSQL Taxonomy*. November 2009. – Blog post of 2009-11-05. http://highscalability.com/blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html

[Hof10a] Hoff, Todd: *Facebook's New Real-time Messaging System: HBase to Store 135+ Billion Messages a Month*. November 2010. – Blog post of 2010-11-16. http://highscalability.com/blog/2010/11/16/facebooks-new-real-time-messaging-system-hbase-to-store-135.html

[Hof10b] Hoff, Todd: *High Scalability – Entries In Memcached*. 2010. – http://highscalability.com/blog/category/memcached

[Hof10c] Hoff, Todd: *MySQL and Memcached: End of an Era?* February 2010. – Blog post of 2010-02-26. http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html

[Hor10] Horowitz, Eliot: *mongoDB Manual – GridFS – When to use GridFS*. September 2010. – Wiki article, version 2 of 2010-09-19. http://www.mongodb.org/display/DOCS/When+to+use+GridFS

[HR10] Heinemeier Hansson, David ; Rails Core Team: *Ruby on Rails*. 2010. – http://rubyonrails.org/

[Hyp09a] Hypertable: *Hypertable*. 2009. – http://www.hypertable.org/index.html

[Hyp09b] Hypertable: *Hypertable – About Hypertable*. 2009. – http://www.hypertable.org/about.html

[Hyp09c] Hypertable: *Hypertable – Sponsors*. 2009. – http://www.hypertable.org/sponsors.html

[Int10] Intersimone, David: *The end of SQL and relational databases?* Febuary 2010. – Blog posts of 2010-02-02, 2010-02-10 and 2010-02-24. http://blogs.computerworld.com/15510/the_end_of_sql_and_relational_databases_part_1_of_3, http://blogs.computerworld.com/15556/the_end_of_sql_and_relational_databases_part_2_of_3, http://blogs.computerworld.com/15641/the_end_of_sql_and_relational_databases_part_3_of_3

[Ipp09] IPPOLITO, Bob: *Drop ACID and think about Data*. March 2009. – Talk at Pycon on 2009-03-28.
http://blip.tv/file/1949416/

[JBo10a] JBOSS COMMUNITY: *HIBERNATE – Relational Persistence for Java & .NET*. 2010. –
http://www.hibernate.org/

[JBo10b] JBOSS COMMUNITY TEAM: *JBoss Cache*. 2010. –
http://jboss.org/jbosscache

[Jon09] JONES, Richard: *Anti-RDBMS: A list of distributed key-value stores*. January 2009. – Blog post of 2009-01-19.
http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/

[Jud09] JUDD, Doug: *Hypertable*. June 2009. – Presentation at NoSQL meet-up in San Francisco on 2009-06-11.
http://static.last.fm/johan/nosql-20090611/hypertable_nosql.pdf

[K+10a] KREPS, Jay et al.: *Project Voldemort – A distributed database*. 2010. –
http://project-voldemort.com/

[K+10b] KREPS, Jay et al.: *Project Voldemort – Design*. 2010. –
http://project-voldemort.com/design.php

[Key09] KEYS, Adam: *It's not NoSQL, it's post-relational*. August 2009. – Blog post of 2009-08-31.
http://therealadam.com/archive/2009/08/31/its-not-nosql-its-post-relational/

[KLL+97] KARGER, David ; LEHMAN, Eric ; LEIGHTON, Tom ; LEVINE, Matthew ; LEWIN, Daniel ; PANIGRAHY, Rina: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1997, p. 654–663

[Lak08] LAKSHMAN, Avinash: *Cassandra - A structured storage system on a P2P Network*. August 2008. – Blog post of 2008-08-25.
http://www.facebook.com/note.php?note_id=24413138919

[Lam98] LAMPORT, Leslie: The part-time parliament. In: *ACM Transactions on Computer Systems* 16 (1998), No. 2, p. 133–169. – Also available online.
http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

[Lin09] LIN, Leonard: *Some Notes on distributed key value Stores*. April 2009. – Blog post of 2009-04-20.
http://randomfoo.net/2009/04/20/some-notes-on-distributed-key-stores

[Lip09] LIPCON, Todd: *Design Patterns for Distributed Non-Relational Databases*. June 2009. – Presentation of 2009-06-11.
http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

[LM09] LAKSHMAN, Avinash ; MALIK, Prashant: *Cassandra – Structured Storage System over a P2P Network*. June 2009. – Presentation at NoSQL meet-up in San Francisco on 2009-06-11.
http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf

[LM10] Lakshman, Avinash ; Malik, Prashant: Cassandra – A Decentralized Structured Storage System. In: *SIGOPS Operating Systems Review* 44 (2010), April, p. 35–40. – Also available online.
http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

[Mah10] Maher, Jacqueline: *Building a Better Submission Form*. May 2010. – Blog post of 2010-05-25.http://open.blogs.nytimes.com/2010/05/25/building-a-better-submission-form/

[Mat89] Mattern, Friedemann: Virtual Time and Global States of Distributed Systems. In: *Parallel and Distributed Algorithms*, North-Holland, 1989, p. 215–226. –
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.7435&rep=rep1&type=pdf

[MC09] Merriman, Dwight ; Chodorow, Kristina: *mongoDB Manual – Data Types and Conventions – Internationalized Strings*. July 2009. – Wiki article, version 2 of 2009-07-30.
http://www.mongodb.org/display/DOCS/Internationalized+Strings

[MC10a] Merriman, Dwight ; Chodorow, Kristina: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Sets Limits*. November 2010. – Wiki article, version 7 of 2010-11-11.
http://www.mongodb.org/display/DOCS/Replica+Sets+Limits

[MC10b] Merriman, Dwight ; Chodorow, Kristina: *mongoDB Manual – Optimization – Query Optimizer*. February 2010. – Wiki article, version 7 of 2010-02-24.
http://www.mongodb.org/display/DOCS/Query+Optimizer

[MCB10a] Merriman, Dwight ; Chodorow, Kristina ; Banker, Kyle: *mongoDB Manual – Querying – min and max Query Specifiers*. February 2010. – Wiki article, version 6 of 2010-02-24.
http://www.mongodb.org/display/DOCS/min+and+max+Query+Specifiers

[MCB+10b] Merriman, Dwight ; Chodorow, Kristina ; Banker, Kyle et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Configuration*. November 2010. – Wiki article, version 49 of 2010-11-18.
http://www.mongodb.org/display/DOCS/Replica+Set+Configuration

[MCD+10] Merriman, Dwight ; Chodorow, Kristina ; Dirolf, Mike et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets*. November 2010. – Wiki article, version 51 of 2010-11-12.
http://www.mongodb.org/display/DOCS/Replica+Sets

[MCH+10] Merriman, Dwight ; Chodorow, Kristina ; Horowitz, Eliot et al.: *mongoDB Manual – Admin Zone – Sharding – Configuring Sharding – A Sample Configuration Session*. September 2010. – Wiki article, version 16 of 2010-09-01.
http://www.mongodb.org/display/DOCS/A+Sample+Configuration+Session

[MCS+10] Merriman, Dwight ; Chodorow, Kristina ; Stearn, Mathias et al.: *mongoDB Manual – Updating – Atomic Operations*. October 2010. – Wiki article, version 42 of 2010-10-25.
http://www.mongodb.org/display/DOCS/Atomic+Operations

[MD10] Merriman, Dwight ; Dirolf, Mike: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Design Concepts*. October 2010. – Wiki article, version 17 of 2010-10-09.
http://www.mongodb.org/display/DOCS/Replica+Set+Design+Concepts

[MDC+10] Merriman, Dwight ; Dirolf, Mike ; Chodorow, Kristina et al.: *mongoDB Manual – Data Types and Conventions – Database References*. September 2010. – Wiki article, version 45 of 2010-09-23.
http://www.mongodb.org/display/DOCS/Database+References

[MDH$^+$10] Merriman, Dwight ; Dirolf, Mike ; Horowitz, Eliot et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding Introduction*. November 2010. – Wiki article, version 56 of 2010-11-06.
http://www.mongodb.org/display/DOCS/Sharding+Introduction

[MDM$^+$10] Murphy, Rian ; Dirolf, Mike ; Magnusson Jr, Geir et al.: *mongoDB Manual – Collections – Capped Collections*. October 2010. – Wiki article, version 22 of 2010-10-27.
http://www.mongodb.org/display/DOCS/Capped+Collections

[MDN$^+$10] Merriman, Dwight ; Dirolf, Mike ; Negyesi, Karoly et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding Limits*. November 2010. – Wiki article, version 37 of 2010-11-16.
http://www.mongodb.org/display/DOCS/Sharding+Limits

[Mer10a] Merriman, Dwight: *mongoDB Manual – Admin Zone – Replication – About the local database*. August 2010. – Wiki article, version 6 of 2010-08-26.
http://www.mongodb.org/display/DOCS/About+the+local+database

[Mer10b] Merriman, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Data Center Awareness*. August 2010. – Wiki article, version 3 of 2010-08-30.
http://www.mongodb.org/display/DOCS/Data+Center+Awareness

[Mer10c] Merriman, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Admin UI*. August 2010. – Wiki article, version 8 of 2010-08-05.
http://www.mongodb.org/display/DOCS/Replica+Set+Admin+UI

[Mer10d] Merriman, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Resyncing a Very Stale Replica Set Member*. August 2010. – Wiki article, version 9 of 2010-08-17.
http://www.mongodb.org/display/DOCS/Resyncing+a+Very+Stale+Replica+Set+Member

[Mer10e] Merriman, Dwight: *mongoDB Manual – Admin Zone – Replication – Replication Oplog Length*. August 2010. – Wiki article, version 9 of 2010-08-07.
http://www.mongodb.org/display/DOCS/Replication+Oplog+Length

[Mer10f] Merriman, Dwight: *mongoDB Manual – Querying – Mongo Query Language*. July 2010. – Wiki article, version 1 of 2010-07-23.
http://www.mongodb.org/display/DOCS/Mongo+Query+Language

[Mer10g] Merriman, Dwight: *On Distributed Consistency — Part 1*. March 2010. –
http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1

[MH10a] Merriman, Dwight ; Hernandez, Scott: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set FAQ*. August 2010. – Wiki article, version 4 of 2010-08-08.
http://www.mongodb.org/display/DOCS/Replica+Set+FAQ

[MH10b] Merriman, Dwight ; Hernandez, Scott: *mongoDB Manual – Indexes – Indexing as a Background Operation*. November 2010. – Wiki article, version 15 of 2010-11-16.
http://www.mongodb.org/display/DOCS/Indexing+as+a+Background+Operation

[MHB$^+$10a] Merriman, Dwight ; Horowitz, Eliot ; Banker, Kyle et al.: *mongoDB Manual – Admin Zone – Sharding – Configuring Sharding*. November 2010. – Wiki article, version 64 of 2010-11-20.
http://www.mongodb.org/display/DOCS/Configuring+Sharding

[MHB$^+$10b] Merriman, Dwight ; Horowitz, Eliot ; Banker, Kyle et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding and Failover*. September 2010. – Wiki article, version 9 of 2010-09-05.
http://www.mongodb.org/display/DOCS/Sharding+and+Failover

[MHC$^+$10a] Merriman, Dwight ; Horowitz, Eliot ; Chodorow, Kristina et al.: *mongoDB Manual – Querying – Dot Notation (Reaching into Objects)*. October 2010. – Wiki article, version 23 of 2010-10-25.
http://www.mongodb.org/display/DOCS/Dot+Notation+(Reaching+into+Objects)

[MHC$^+$10b] Merriman, Dwight ; Horowitz, Eliot ; Chodorow, Kristina et al.: *mongoDB Manual – Sharding*. October 2010. – Wiki article, version 37 of 2010-10-21.
http://www.mongodb.org/display/DOCS/Sharding

[MHD$^+$10] Merriman, Dwight ; Horowitz, Eliot ; Dirolf, Mike et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Internals*. November 2010. – Wiki article, version 43 of 2010-11-10.
http://www.mongodb.org/display/DOCS/Replica+Set+Internals

[MHH10] Merriman, Dwight ; Horowitz, Eliot ; Hernandez, Scott: *mongoDB Manual – Developer FAQ – How does concurrency work*. June 2010. – Wiki article, version 17 of 2010-06-28.
http://www.mongodb.org/display/DOCS/How+does+concurrency+work

[Mic10] Microsoft Corporation: *ADO.NET Entity Framework*. 2010. –
http://msdn.microsoft.com/en-us/library/bb399572.aspx

[MJS$^+$10] Merriman, Dwight ; Jr, Geir M. ; Staple, Aaron et al.: *mongoDB Manual – Admin Zone – Replication – Master Slave*. October 2010. – Wiki article, version 55 of 2010-10-22.
http://www.mongodb.org/display/DOCS/Master+Slave

[MKB$^+$10] Merriman, Dwight ; Kreuter, Richard ; Banker, Kyle et al.: *mongoDB Manual – Developer FAQ – SQL to Mongo Mapping Chart*. November 2010. – Wiki article, version 47 of 2010-11-07.
http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart

[MMC$^+$10a] Merriman, Dwight ; Magnusson Jr, Geir ; Chodorow, Kristina et al.: *mongoDB Manual – Admin Zone – Security and Authentication*. November 2010. – Wiki article, version 31 of 2010-11-20.
http://www.mongodb.org/display/DOCS/Security+and+Authentication

[MMC$^+$10b] Murphy, Rian ; Magnusson Jr, Geir ; Chodorow, Kristina et al.: *mongoDB Manual – Querying – Server-side Code Execution*. October 2010. – Wiki article, version 35 of 2010-10-24.
http://www.mongodb.org/display/DOCS/Server-side+Code+Execution

[MMD$^+$10] Murphy, Rian ; Magnusson Jr, Geir ; Dirolf, Mike et al.: *mongoDB Manual – Querying – Sorting and Natural Order*. February 2010. – Wiki article, version 14 of 2010-02-03.
http://www.mongodb.org/display/DOCS/Sorting+and+Natural+Order

[MMG$^+$10] Murphy, Rian ; Merriman, Dwight ; Geir Magnusson Jr et al.: *mongoDB Manual – Admin Zone – Replication*. August 2010. – Wiki article, version 83 of 2010-08-05.
http://www.mongodb.org/display/DOCS/Replication

[MMH$^+$10] Murphy, Rian ; Magnusson Jr, Geir ; Horowitz, Eliot et al.: *mongoDB Manual – Indexes*. November 2010. – Wiki article, version 55 of 2010-11-23.
http://www.mongodb.org/display/DOCS/Indexes

[MMM+10a] Murphy, Rian ; Magnusson Jr, Geir ; Merriman, Dwight et al.: *mongoDB Manual – Data Types and Conventions – Object IDs*. November 2010. – Wiki article, version 43 of 2010-11-01.
http://www.mongodb.org/display/DOCS/Object+IDs

[MMM+10b] Murphy, Rian ; Magnusson Jr, Geir ; Merriman, Dwight et al.: *mongoDB Manual – Inserting – Schema Design*. July 2010. – Wiki article, version 22 of 2010-07-19.
http://www.mongodb.org/display/DOCS/Schema+Design

[MMM+10c] Murphy, Rian ; Magnusson Jr, Geir ; Merriman, Dwight et al.: *mongoDB Manual – Querying – Advanced Queries*. December 2010. – Wiki article, version 118 of 2010-12-23.
http://www.mongodb.org/display/DOCS/Advanced+Queries

[MMM+10d] Murphy, Rian ; Magnusson Jr, Geir ; Merriman, Dwight et al.: *mongoDB Manual – Updating – Updating Data in Mongo*. September 2010. – Wiki article, version 28 of 2010-09-24.
http://www.mongodb.org/display/DOCS/Updating+Data+in+Mongo

[MMM+10e] Murphy, Rian ; Merriman, Dwight ; Magnusson Jr, Geir et al.: *mongoDB Manual – Databases – Commands*. September 2010. – Wiki article, version 60 of 2010-09-01.
http://www.mongodb.org/display/DOCS/Commands

[MMM+10f] Murphy, Rian ; Merriman, Dwight ; Magnusson Jr, Geir et al.: *mongoDB Manual – Indexes – Multikeys*. September 2010. – Wiki article, version 17 of 2010-09-22.
http://www.mongodb.org/display/DOCS/Multikeys

[MMM+10g] Murphy, Rian ; Merriman, Dwight ; Magnusson Jr, Geir et al.: *mongoDB manual – Querying – Aggregation*. September 2010. – Wiki article, version 43 of 2010-09-24.
http://www.mongodb.org/display/DOCS/Aggregation

[MMM+10h] Murphy, Rian ; Merriman, Dwight ; Magnusson Jr, Geir et al.: *mongoDB Manual – Querying – Queries and Cursors*. March 2010. – Wiki article, version 21 of 2010-03-01.
http://www.mongodb.org/display/DOCS/Queries+and+Cursors

[Mon10] MongoDB team: *MongoDB 1.6 Released*. August 2010. – Blog post of 2010-08-05.
http://blog.mongodb.org/post/908172564/mongodb-1-6-released

[MS10] Merriman, Dwight ; Stearn, Mathias: *mongoDB Manual – Querying – Retrieving a Subset of Fields*. July 2010. – Wiki article, version 12 of 2010-07-24.
http://www.mongodb.org/display/DOCS/Retrieving+a+Subset+of+Fields

[MSB+10] Merriman, Dwight ; Stearn, Mathias ; Banker, Kyle et al.: *mongoDB Manual – Updating – findAndModify Command*. October 2010. – Wiki article, version 25 of 2010-10-25.
http://www.mongodb.org/display/DOCS/findAndModify+Command

[MSL10] Merriman, Dwight ; Stearn, Mathias ; Lerner, Alberto: *mongoDB Manual – Admin Zone – Sharding – Sharding Internals – Splitting Chunks*. August 2010. – Wiki article, version 5 of 2010-08-16.
http://www.mongodb.org/display/DOCS/Splitting+Chunks

[N+10] Newson, Robert et al.: *couchdb-lucene*. 2010. – github project.
http://github.com/rnewson/couchdb-lucene

[Nor09] North, Ken: *Databases in the cloud*. September 2009. – Article in Dr. Drobb's Magazine.
http://www.drdobbs.com/database/218900502

[OAE+10] Ousterhout, John ; Agrawal, Parag ; Erickson, David ; Kozyrakis, Christos ; Leverich, Jacob ; Mazières, David ; Mitra, Subhasish ; Narayanan, Aravind ; Parulkar, Guru ; Rosenblum, Mendel ; M.Rumble, Stephen ; Stratmann, Eric ; Stutsman, Ryan: The case for RAMClouds: scalable high-performance storage entirely in DRAM. In: *SIGOPS Oper. Syst. Rev.* 43 (2010), January, p. 92–105. – Available online.
http://www.stanford.edu/~ouster/cgi-bin/papers/ramcloud.pdf

[Oba09a] Obasanjo, Dare: *Building scalable databases: Denormalization, the NoSQL movement and Digg*. September 2009. – Blog post of 2009-09-10.
http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabases-DenormalizationTheNoSQLMovementAndDigg.aspx

[Oba09b] Obasanjo, Dare: *Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes*. January 2009. – Blog post of 2009-01-16.
http://www.25hoursaday.com/weblog/2009/01/16/BuildingScalableDatabasesProsAnd-ConsOfVariousDatabaseShardingSchemes.aspx

[OCGO96] O'Neil, Patrick ; Cheng, Edward ; Gawlick, Dieter ; O'Neil, Elizabeth: The log-structured merge-tree (LSM-tree). In: *Acta Inf.* 33 (1996), No. 4, p. 351–385

[Ora10a] Oracle Corporation: *Interface Serializable*. 1993, 2010. – API documentation of the Java^{TM} Platform Standard Edition 6.
http://download.oracle.com/javase/6/docs/api/java/io/Serializable.html

[Ora10b] Oracle Corporation: *Java New I/O APIs*. 2004, 2010. –
http://download.oracle.com/javase/1.5.0/docs/guide/nio/index.html

[Ora10c] Oracle Corporation: *MySQL*. 2010. –
http://www.mysql.com/

[Ora10d] Oracle Corporation: *Oracle Berkeley DB Products*. 2010. –
http://www.oracle.com/us/products/database/berkeley-db/index.html

[Par09] Parallel & Distributed Operating Systems Group: *The Chord/DHash Project – Overview*. November 2009. –
http://pdos.csail.mit.edu/chord/

[PDG+05] Pike, Rob ; Dorward, Sean ; Griesemer, Robert ; Quinlan, Sean ; Inc, Google: Interpreting the Data: Parallel Analysis with Sawzall. In: *Scientific Programming Journal* Vol. 13. Amsterdam : IOS Press, January 2005, p. 227–298. – Also available online.
http://research.google.com/archive/sawzall-sciprog.pdf

[PLL09] Pritlove, Tim ; Lehnardt, Jan ; Lang, Alexander: *CouchDB – Die moderne Key/Value-Datenbank lädt Entwickler zum Entspannen ein*. June 2009. – Chaosradio Express Episode 125, Podcast published on 2009-06-10.
http://chaosradio.ccc.de/cre125.html

[Pop10a] Popescu, Alex: *Cassandra at Twitter: An Interview with Ryan King*. Feburary 2010. – Blog post of 2010-02-23.
http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king

[Pop10b] Popescu, Alex: *Presentation: NoSQL at CodeMash – An Interesting NoSQL categorization*. February 2010. – Blog post of 2010-02-18.
http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interesting-nosql

[RGO06] ROTEM-GAL-OZ, Arnon: *Fallacies of Distributed Computing Explained*. 2006. – http://www.rgoarchitects.com/Files/fallacies.pdf

[Ric10] RICHPRI: *diasporatest.com – MongoDB*. November 2010. – Wiki article, version 2 of 2010-11-16.
http://www.diasporatest.com/index.php/MongoDB

[Rid10] RIDGEWAY, Jay: *bit.ly user history, auto-sharded*. May 2010. – Presentation at MongoNYC in New York on 2010-05-21.
http://blip.tv/file/3704043

[RRHS04] RAMABHADRAN, Sriram ; RATNASAMY, Sylvia ; HELLERSTEIN, Joseph M. ; SHENKER, Scott: Prefix Hash Tree – An Indexing Data Structure over Distributed Hash Tables / IRB. 2004. – Forschungsbericht. – Available online.
http://berkeley.intel-research.net/sylvia/pht.pdf

[RV03] REYNOLDS, Patrick ; VAHDAT, Amin: Efficient Peer-to-Peer Keyword Searching. In: *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA : Springer-Verlag New York, Inc., 2003. – ISBN 3–540–40317–5, p. 21–40. – Available online.
http://issg.cs.duke.edu/search/search.pdf

[S+10] SANFILIPPO, Salvatore et al.: *redis*. 2010. –
http://code.google.com/p/redis/

[SBc+07] STONEBRAKER, Michael ; BEAR, Chuck ; ÇETINTEMEL, Uğur ; CHERNIACK, Mitch ; GE, Tingjian ; HACHEM, Nabil ; HARIZOPOULOS, Stavros ; LIFTER, John ; ROGERS, Jennie ; ZDONIK, Stan: One Size Fits All? – Part 2: Benchmarking Results. In: *Proc. CIDR*, 2007, p. 173–184

[Sc05] STONEBRAKER, Michael ; ÇETINTEMEL, Uğur: One Size Fits All: An Idea whose Time has Come and Gone. In: *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2005, p. 2–11. – Also available online.
http://www.cs.brown.edu/~ugur/fits_all.pdf

[Sch09] SCHMIDT, Stephan: *The dark side of NoSQL*. September 2009. – Blog post of 2009-09-30.
http://codemonkeyism.com/dark-side-nosql/

[Sch10] SCHMIDT, Stephan: *Why NOSQL Will Not Die*. March 2010. – Blog post of 2010-03-29.
http://codemonkeyism.com/nosql-die/

[Sco09] SCOFIELD, Ben: *NoSQL Misconceptions*. October 2009. – Blog post of 2009-10-21.
http://www.viget.com/extend/nosql-misconceptions/

[Sco10] SCOFIELD, Ben: *NoSQL – Death to Relational Databases(?)*. January 2010. – Presentation at the CodeMash conference in Sandusky (Ohio), 2010-01-14.
http://www.slideshare.net/bscofield/nosql-codemash-2010

[See09] SEEGER, Marc: *Key-Value stores: a practical overview*. September 2009. – Paper of 2009-09-21.
http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_-Stores.pdf

[Sha09a] Shalom, Nati: *The Common Principles Behind The NOSQL Alternatives*. December 2009. – Blog post of 2009-12-15.
http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principles-behind-the-nosql-alternatives.html

[Sha09b] Shalom, Nati: *No to SQL? Anti-database movement gains steam – My Take*. July 2009. – Blog post of 2009-07-04.
http://natishalom.typepad.com/nati_shaloms_blog/2009/07/no-to-sql-anti-database-movement-gains-steam-my-take.html

[Sha09c] Shalom, Nati: *Why Existing Databases (RAC) are So Breakable!* November 2009. – Blog post of 2009-11-30.
http://natishalom.typepad.com/nati_shaloms_blog/2009/11/why-existing-databases-rac-are-so-breakable.html

[SM10] Stearn, Mathias ; Merriman, Dwight: *mongoDB Manual – Inserting - Trees in MongoDB*. March 2010. – Wiki article, version 21 of 2010-03-10.
http://www.mongodb.org/display/DOCS/Trees+in+MongoDB

[SMA$^+$07] Stonebraker, Michael ; Madden, Samuel ; Abadi, Daniel J. ; Harizopoulos, Stavros ; Hachem, Nabil ; Helland, Pat: The end of an architectural era: (it's time for a complete rewrite). In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007, p. 1150–1160

[SMK$^+$01] Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. F. ; Balakrishnan, Hari: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: *Proceedings of the ACM SIGCOMM '01 Conference*. San Diego, California, United States, August 2001, p. 149–160. – Also available online.
http://www.sigcomm.org/sigcomm2001/p12-stoica.pdf

[Ste09] Stephens, Bradford: *Social Media Kills the Database*. June 2009. – Blog post of 2009-06.
http://www.roadtofailure.com/2009/06/19/social-media-kills-the-rdbms/

[Sto09] Stonebraker, Michael: *The "NoSQL" Discussion has Nothing to Do With SQL*. November 2009. – Blog post of 2009-11-04.
http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext

[Str10] Strozzi, Carlo: *NoSQL – A relational database management system*. 2007–2010. –
http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page

[Tay09] Taylor, Bret: *How Friendfeed uses MySQL*. February 2009. – Blog post of 2009-02-27.
http://bret.appspot.com/entry/how-friendfeed-uses-mysql

[Tec09] Technology Review: *Designing for the cloud*. July/August 2009. – Interview with Dwight Merriman (CEO and cofounder of 10gen).
http://www.technologyreview.com/video/?vid=356

[vDGT08] van Renesse, Robbert ; Dumitriu, Dan ; Gough, Valient ; Thomas, Chris: Efficient reconciliation and flow control for anti-entropy protocols. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. New York, NY, USA : ACM, 2008 (LADIS '08), p. 6:1–6:7. – Also available online.
http://www.cs.cornell.edu/projects/ladis2008/materials/rvr.pdf

[Vog07] Vogels, Werner: *Eventually consistent*. December 2007. – Blog post of 2007-12-19.
http://www.allthingsdistributed.com/2007/12/eventually_consistent.html

[Vog08] Vogels, Werner: *Eventually consistent*. December 2008. – Blog post of 2008-12-23.
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[WCB01] Welsh, Matt ; Culler, David ; Brewer, Eric: SEDA: an architecture for well-conditioned, scalable internet services. In: *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2001 (SOSP '01), p. 230–243. – Also available online.
http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf

[Whi07] White, Tom: *Consistent Hashing*. November 2007. – Blog post of 2007-11-27.
http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

[Wig09] Wiggins, Adam: *SQL Databases Don't Scale*. June 2009. – Blog post of 2009-07-06.
http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/

[Wik10] Wikipedia: *Fallacies of Distributed Computing*. 2010. – Wiki article, version of 2010-03-04.
http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

[Wik11a] Wikipedia: *HBase*. January 2011. – Wiki article of 2011-01-03.
http://en.wikipedia.org/wiki/HBase

[Wik11b] Wikipedia: *Hypertable*. January 2011. – Wiki article of 2011-01-21.
http://en.wikipedia.org/wiki/Hypertable

[Yen09] Yen, Stephen: *NoSQL is a horseless carriage*. November 2009. –
http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf