BIG DATA

An Introduction To The Fields Of Data Engineering, Development And Architecture Of Data-Intensive Applications.

—Winter Semester 2018 —

Munich, September 19, 2018

Marcel Mittelstädt
Mail: mittelstaedtmarcel@googlemail.com
Web: www.marcel-mittelstaedt.com

Cooperative State University Baden-Wuerttemberg

Preface

This lecture will give you a brief introduction to so what is called 'Big Data'. We will quickly refresh the basics about databases, data models and data processing you have learned so far and compare those to the distributed world of Big Data.
After that we will take a deep dive into the foundations of distributed data storages and data processing as well as the belonging concepts of reliability, scalability, replication, partitioning, batch and stream processing.
Later on we will take a look at the most common used software and frameworks (mostly the hadoop ecosystem).
At the end, as you know the basic concepts and you are able to setup and work with distributed environments and huge data sets, there will be a short introduction to data science.

At the end of each lesson, there will be some hands-on exercises, which we will start together and which have to be finished till the next week. This lecture will only be about 36 hours in 12 weeks (1 slot each week), which is very little time to cover such an extensive topic. So pay close attention and if you can't keep up, feel free to ask questions at the end of each lesson.

You can find:

- this **script** (and LaTeX-sources),
- **slides** presented within the lecture,
- **excercises** and **solutions**,
- **docker images**, **scripts** as well as **sample data sets**

here:

```
https://github.com/marcelmittelstaedt/BigData
```

You can just download everything directly or install `git` and get everything by using:

```
git clone https://github.com/marcelmittelstaedt/BigData.git
```

and

```
git pull
```

to get the most recent version.

If you find any mistakes or misspellings feel free to send me a mail (**mittelstaedtmarcel@googlemail.com**) or if you are able to, commit a push request.

One last point, as you may have noticed, Microsoft as well as other commercial vendors successfully fail at developing and providing adequate solutions for highly data-driven applications, so almost any software or framework you will encouter during this lecture or later, will be open-source and only be runnable on a UNIX-based operating system. Either you are already familiar with UNIX (lucky you), otherwise you will learn something valuable that will improve your life.

> *"Microsoft is not the answer. Microsoft is the question. NO is the answer."*
>
> — Erik Naggum ✝ , *Philantropist and Developer (Emacs, Lisp and SGML)*

Contents

1 Introduction To Big Data

"Big Data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it."
— Dan Ariely, *Professor of Psychology and Behavioral Economics, Duke University*

1.1 Definition of Big Data

to-be-added

1.2 Challenges

to-be-added

1.3 Use Cases

to-be-added

1.4 Dissociation Datawarehousing

to-be-added

2 Fundamentals Of Distributed Data-Systems

"I'm not telling you it's going to be easy - I'm telling you it's going to be worth it."

— Arthur L. Williams Jr., *Founder of Primerica Financial Services*

In this chapter we will go trough the foundation of data-systems, requirements and concepts which apply to any (data-driven) system. This covers in particular following topics:

- Section 2.1 Requirements of **Non-Functional Requirements Of Data-Systems**, e.g.
  - 2.1.1 Scalability,
  - 2.1.2 Reliability,
  - 2.1.3 Maintainability.
- Section 2.2 **Storage Concepts** for databases
- Section 2.3 **Data Models And Access** concepts
- Section 2.4 **Challenges Of Distributed Data Systems**, e.g.
  - 2.4.2 Partitioning,
  - 2.4.1 Replication,
  - 2.4.3 Transactions,
  - 2.4.4 Consistency amongst others.

At the end you will have a basic understanding about the difference between common and distributed systems and databases, the basic concepts of each of them and which one theoretically fits best to solve a certain problem. A more hands-on deep-dive into related software, frameworks as well as specific problems and use cases will be

demonstrated later in chapter 4 Software and Frameworks.

2.1 Non-Functional Requirements Of Data-Systems

When we think about data-driven systems, we mostly think about the same requirements we expect of any other data system we already know:

- **Data Storage**: We need to store data and also need to be able to find it again later (*database*).
- **Data Querying**: We need to be able to query and filter data efficiently in certain kinds of ways (*transaction and indices*).
- **Retention and Performance**: We want results fast, especially of expensive read operations (*caching*).
- **Data Processsing**: We want to be able to process a huge amount of data (*batch processing*) as well as process data asynchronously (*stream processing*).

This sounds quite obvious, but remember those requirements are still the same as for the first database CODASYL[1] back in the 1960's. Even though there are and have been a lot of databases back in time, each of them with a diverse purpose and different approaches to solve e.g. indexing or caching - all of them still match those same requirements. Certainly those data systems evolved much further, especially within the last years, you may noticed:

- Relational Databases being able to handle NoSQL data (e.g. even "retirees" like IBM DB2[2] or Oracle[3]) as well as NoSQL databases being able to handle traditional SQL (e.g. ToroDB[4]) or
- databases becoming message queues (e.g. RethinkDB[5] or Redis[6][7]) and the other way around message queueing systems become databases (e.g. Apache

[1] https://en.wikipedia.org/wiki/CODASYL
[2] (IBM18), https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/json/src/tpc/db2z_jsonfunctions.html
[3] (ORC18b), https://docs.oracle.com/database/121/ADXDB/json.htm
[4] (TOR18), https://www.torodb.com
[5] (RDB18f), https://rethinkdb.com/docs/changefeeds/
[6] (RUD18), https://redis.io/commands/rpoplpush
[7] (Joh14), see 5. Adopting Redis for Application Data

Kafka[8]).

As you can see, boundaries between traditional databases and data-driven applications get blurred and in the same way more diversified. There is no one-size-fits-all solution, e.g. like you can find back in the past in the 1990's or early 2000s. At that time monolithic single-, 2 and 3-tier, architectures were state-of-the-art (see Figure 2.1 left-hand side).
Usually the **database layer** was represented by a data store like MySQL, Oracle, DB2 or even just files containing data stored on the local disk.
The **application layer** was usually a monolithic application developed in languages like PHP, Perl, C++ or Java and running on a web- or application server (e.g. Apache HTTP Server or IBM WebSphere).
And last but not least the **client layer**: a web browser like nowadays.

If you take a look at Figure 2.2 on page 6 you can see an example of this time

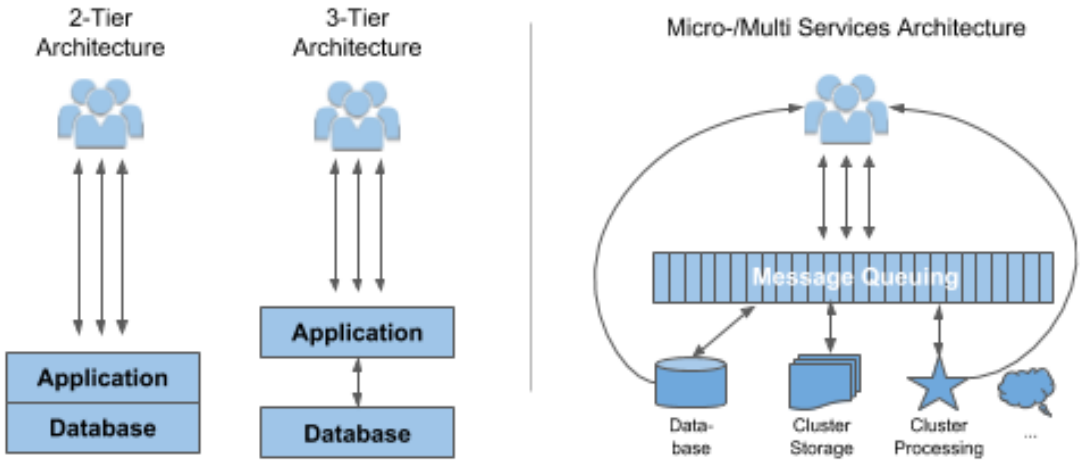

Figure 2.1: Schema - Application Architectures

you may know: ebay.com. They have used the classical 3-tier architecture as well: Oracle as the database running on Solaris as OS[9], C++ as application code running on the Microsoft IIS[10] web server. As you can already guess, this architecture won't scale very well today, in fact the only way to scale this application was to upgrade the single server (*scale up vertically*), in case of ebay.com they once switched from

[8](KFK18), https://kafka.apache.org/10/documentation/streams/developer-guide/interactive-

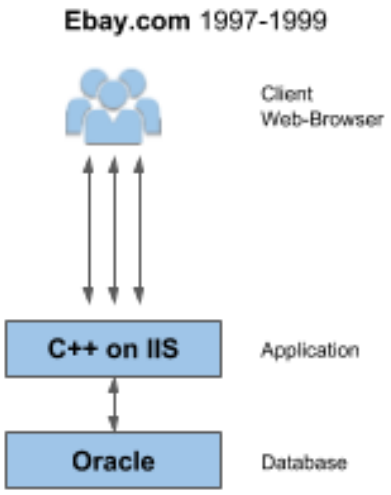

Figure 2.2: Schema - Architecture Ebay.com (1997-1999)

commodity hardware to a very pricey mainframe server (Sun Enterprise E10000[11]) to buy some time. But as you may have noticed there are much more ovious issues, e.g. if you think about:

- **Redundancy** does not exist at all (if the database itself or it's server suffers an outage the whole system will be unavailable.
- **Extensability** is not existent, the system is only able to scale up vertically and if this is needed, a downtime is inevitable, as no part of the data system is neither replicated nor virtualized.
- **Maintainablity** is also very limited as any maintenance of the database will require a certain amount of time in which the application will be unavailable.

But we will dicuss this later in the following chapters.

The previously mentioned issues are already sufficient reason but also the increasing amount of data as well as required features of data systems these days (becoming more diversified in the same way) make it unfeasible to rely on a single tool. Instead

queries.html

[9] OS, Operating System

[10] (IIS18), *Microsoft Internet Information Services, an extensible web server created by Microsoft for use with the Windows NT family.*

[11] (SP06), slide 11

each functionality is usually broken down into parts which can be done efficiently by suitable tools which are sticked together within the applications itself. This could probably look like as you can see in Figure 2.1 (right-hand side) on page 5, but that's just one plain example of many other.

Instead of having one single-purpose data store, there are several tied together, each one of them to fulfill it's specific part within the whole data system but all of them tied together as one application.
As you can see one part of the data system could be: a **database** (like you saw in Figure 2.2 on page 6), e.g. to store and serve:

- user data (e.g. in case of an application with login)
- product data (e.g. in case the applications is a web shop)
- user generated content (e.g. in case the application is a newspage, blog or forum)

Another part could be **cluster storage** like Hadoop, which could:

- keep a complete history of all raw data (e.g. page requests of a website or measured values of a sensor)
- serve for batch processing (e.g. crunching the whole history of data, which is impossible for a single database, as it couldn't even save the whole data and certainly wouldn't be able to process it later on)
- serve for analytic and reporting purposes (e.g. reports of how many people have visited the website within the last year based on the raw data)

Also frequently seen, an analytical **cluster processing engine**, e.g. Spark or Flink to:

- process data gathered in real-time (e.g. every page request of a website) for analytical purposes
- use processed data, to run data science models on it (e.g. to serve targeted

advertisements or customized content to a user on a website, based on his last page requests, browser user-agent or device)

- ...

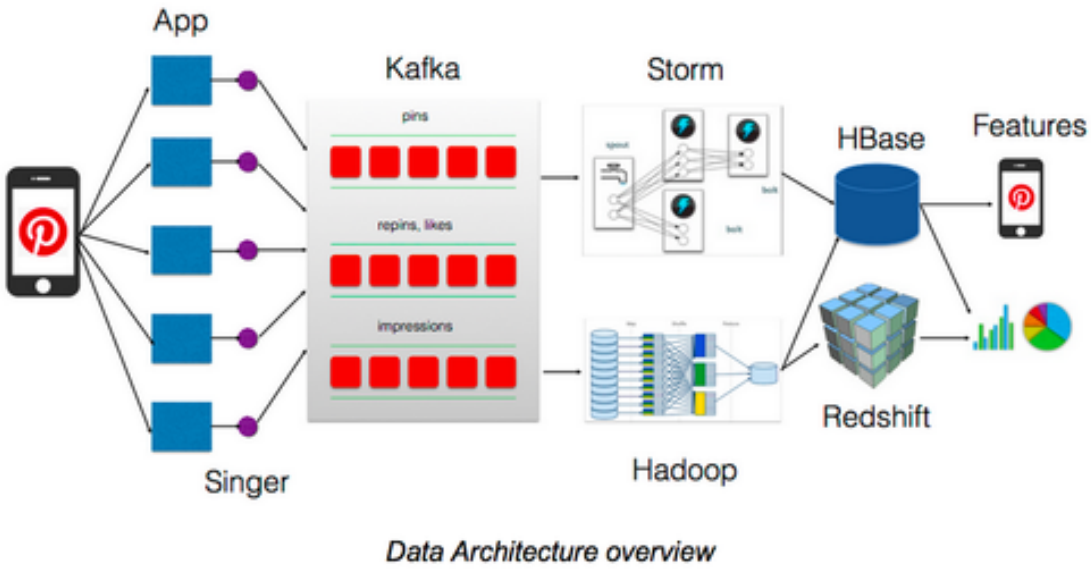

Figure 2.3: Schema - Architecture pinterest.com

If you take a look at Figure 2.3, you can see a comparable data system architecture, implemented by pinterest.com[12]. Redis as a database on top of the hadoop cluster storage to serve for analytical purposes (e.g. ad serving of pinterest's adbuyers) or HBase on top of Hadoop cluster storage and Storm to serve features for the actual end-user of pinterest.com.

But we need to take care here: by creating new and more complex data systems from special purpose data systems, complexity is growing with it. How to ensure the system is avalaible with a reliable performance if something crashes? How to make sure data remains consistent and complete if things go wrong? How to scale the data system to be able to handle increased load?...
There are many apects which are crucial and influence the architecture of a data

[12](PIN18), Architecture of Giants: Data Stacks at Facebook, Netflix, Airbnb, and Pinterest

system like regulary constraints like data security, location of servers, SLA's[13] or existing devlopment and operation skills - which very much depend on the specific situation.
Within the next chapters we will focus on the aspects which must be taken into account by any data system:

- Scalability (Chapter 2.1.1),
- Reliability (Chapter 2.1.2),
- Maintainability (Chapter 2.1.3).

As many people and companies usually mess around with those terms, firstly we will develop a clear understanding on what they mean and later on take a closer look on how to apply algorithms, development and architectures to fulfill them appropriately.

2.1.1 Scalability

As is evident from the introduction of this chapter: the fact that a system is working reliable today doesn't mean it will necessarily work reliable in the future. The data system of ebay.com in 1999 was maxed out at handling **50.000** active listings[14], imagine how the system would behave today at handling **1 billion** active listings[15]. Obviously handling increased load (e.g. a larger amount of data needed to store or request to handle) is a major factor of a scalable data system.

Scalability is the capability of data system to handle a growing amount of load (e.g. a larger amount of data needed to store or requests to handle). A data system is considered scalable if its capable of increasing it's total through-/output under an increased load when resources (typically hardware) are added.

[13](WKS18), *Service Level Agreement, a commitment between a service provider and a client. Particular aspects of the service – quality, availability, responsibilities – are agreed between the service provider and the service user.*
[14](SP06), slide 9
[15](EBA18)

Note that, "*scalability*" isn't a binary tag that could be attached to a data system. It's pointless to say "*a data system is scalable*" as well as "*a data system is not scalable*", in either way you must think about "*If the load of a data system grows in a certain way, what are the options on the table for coping with the growth?*" and "*How can we add ressources (hardware) to be able to handle the additional load?*". Therefore we will discuss the parameters and definition of **Load** (Section 2.1.1.1) and **Performance** (Section 2.1.1.2) within the next section as well as **Approaches** for coping with load to achieve a certain performance (Section 2.1.1.3).

2.1.1.1 Load

To get an idea what *load* of a data system actually means, how it could be described an measured, we will take a nother look at the example of ebay.com discussed so far. If we take a look back at the architecure of ebay at 1997-1999 (Figure 2.2 on page 6) we can already guess with increasing load (page requests to certain items listed on ebay.com and in this way calls to the database through the application) will reach its limit at the maximum amount of read requests the oracle database can serve. As the application tier (web server) has already been scaled horizontally to multiple nodes, the oracle database server reached its limit of physical growth in November 1999.

So ebay added an additional server to not just elmininate the SPOF[16] but to be able to failover but also they have splitted the database to be able to logically partition it into separate instances and in this way be able to scale horizontally. This was achieved in 2001 by splitting items by categories, as you can see in Figure 2.4 on page 11. In this simple way it was possible to distribute the load (mostly page requests for items) in an "equal" way to different physical nodes. Later on they segmented whole databases into functional areas like hosts for item, user, account or transaction data as well as partitioned the data by typical usage characteristics to scale horizontally.

They obviously did furher optimizations at the whole data system to be able to cope

[16](WPS18), Single Point Of Failure

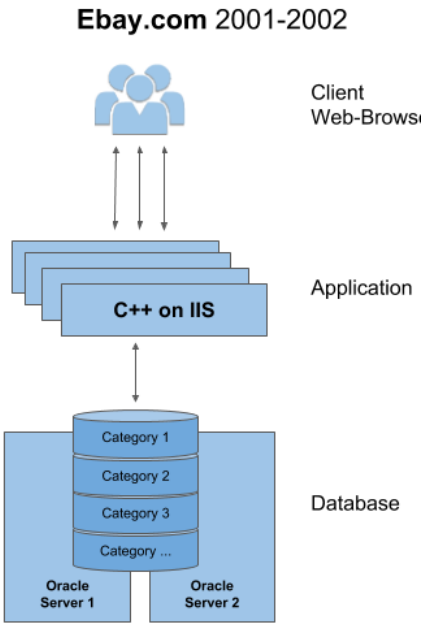

Figure 2.4: Schema - Architecture Ebay.com April (2001 - December 2002)

with the increasing load, like disabling transactions, moving CPU-intensive[17] work to the application tier (e.g. joins, referential integrity or sorting), extensive use of prepared statements...but as this techniques are not mainly specific for distributed systems and some of them not even recommened nowadays, we won't focuse on them within this lecture.

In the example of ebay.com, requests per item and category could be a valuable *load parameter* for discussing scalability, since it determines the database requests per data record and partition - as proven by the the data system structure of ebay at 2002 as we see. Your or other data systems you have seen so far most likely have very different characteristics, but you can apply similar principles to reason about their load.

[17]CPU, a processor or processing unit is an electronic circuit which performs operations on some external data source, usually memory or some other data stream is called central processing unit

The **load** of a data system is a measurement of the amount of computational work it performs (depending on the architecture in-place), e.g. the number of (concurrent) reads from a data storage, writes to a data storage or the ratio between reads and writes. The maximum load is defined by the weakest part of the architecture (=*bottleneck*).

2.1.1.2 Performance

Now that we have described what *load* of a data system means as well as what *load parameters* could be, we will examine more closely what happens when the load increases. Usually there are two important cases you need to think about while developing data systems:

- The load parameter increases, but all ressources (e.g. number of server, CPU or memory) stay the same - how is the performance of the data system affected?
- The load parameter increases - how much do you need to increase the ressources (e.g. number of server, CPU or memory) to keep the performance stay the same?

But how to answer them? Therefore we need performance numbers. In case of data systems measurement, methods usually are **throughput** (number of records that can be handled), e.g.:

- read/writes per second (in case of MongoDB up to 100.000 read/writes per second)
- messages processed (in case of Apache Kafka and Linkedin more than 2 million records per second on just 3 nodes)
- data processed (in case of Apache Hadoop and MapReduce terabytes of data within several seconds)

or if your buidling a data-system which works as the backend of a end-user facing application like a website, it's more about the **response time**, which means the time between sending a request and receiving the response. For instance if we think

about the example of ebay.com within the previous chapters, as of 2012 they had 1 billion items accessible at any time, needed to serve 2 billion page requests each day and had to fulfill each of them within fractions of a second[18].

Regardless of throughput or response time - if we think about performance we don't think about a single number, but a distribution of values that we can measure. If you will repeat the same page request, read or writes on the same data system: response time and throughput will inevitable vary somehow. There are simply too many factors you usually cannnot contain:

- network issues (e.g. latency or TCP packet loss and retransmission)
- os issues (e.g. a page faults, context switches or running background processes)
- physical issues (e.g. a damaged disk/ssd or overheating of a CPU and, associated therwith a decreased processing power)

Therfore it is common to use an average for measuring throughput or response times. *Average* doesn't refer to a particular formula, we will briefly discuss 3 that are commonly used:

- **arithmetic mean**[19] (easy to calculate but ignores ratios and is highly affected by statistical outliers, so it cannot tell you how many requests, reador writes actually have had a worse performance)
- **median**[20] (easy to calculate and less distorted by outliers)
- **percentiles**[21] (easy to calculate, not distorted by outliers)

As you can guess, evaluating symmetric distributions with no outliers, *arithmetic mean* will be the best choice, but as we are looking at performance parameters like throughput and response times, symmetric distribution is a whishful thinking. More usually throughput and response times will result in skewed distributions, so *median* seems to be the better choice, as it doesn't ignore ratio and outliers completely. For instance if you calculate the median for latency (y-axis) of reads in a

[18](EBA12), https://hughewilliams.com/2012/06/26/the-size-scale-and-numbers-of-ebay-com/
[19]Sum of values divided by the number of values.
[20]A median separates the higher half of values from the lower half.
[21]A measure used for indicating a certain percentage of scores falls below that measure.

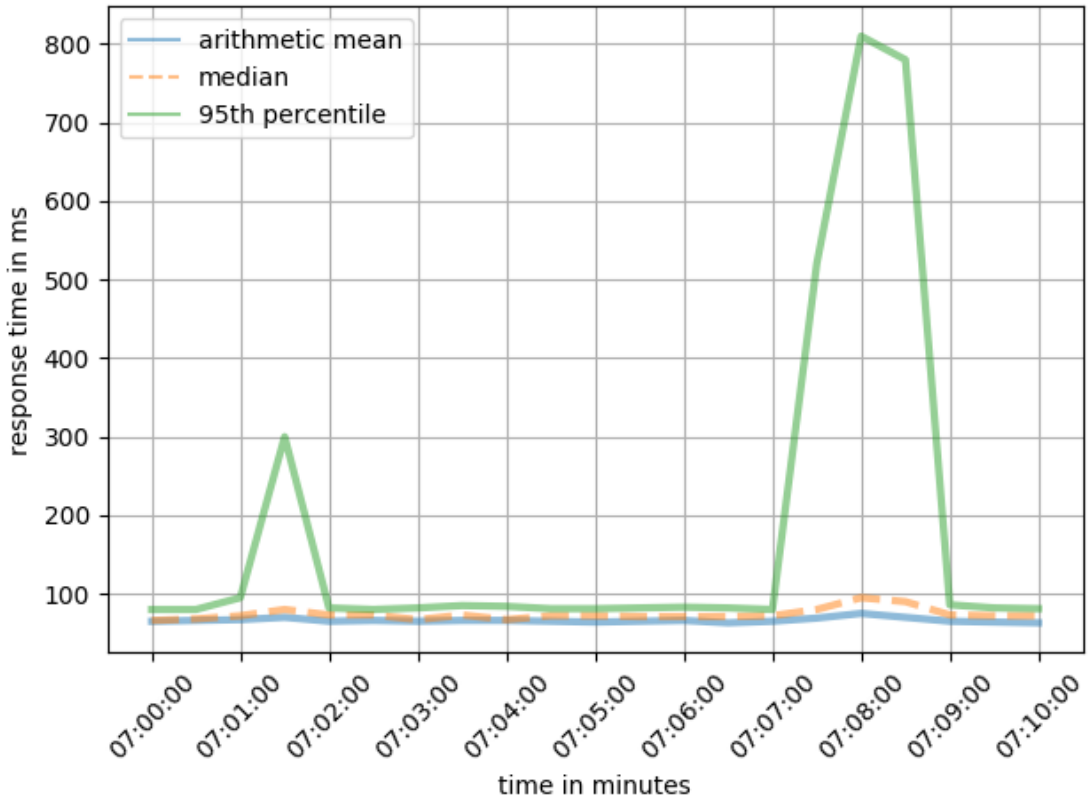

Figure 2.5: Schema - Arithmetic Mean, Median and Percentiles - Example

given timeframe (x-axis) from a data system as illustrated in figure 2.5 on page 14. You can see a small peak at *07:08:00* but it still looks fine as the median response time is $<$ 100ms. But what about the green graph (*95th percentile*)? That's the main reason why using percentiles is pretty common, especially the 95th, 99th or 99.9th percentile (abbreviated *p95*, *p99*, *p999*) is frequently used in SLA's[22]. Percentiles define thresholds at which 95%, 99% or 99,9% of requests, reads or writes are beneath that threshold. Looking back at figure 2.5 this would mean that 95% of all requests, reads or writes done by user are faster or equal 810ms and 5% will result in a response time $>$ 810 ms, which in case of a customer facing data system could mean: 5% unsatisfied users and in this way probably a loss of possible leads (e.g. purchases on a webshop or subscriptions for a video streaming platform) and ultimately loss of revenue.

[22](WKS18), *Service Level Agreement, a commitment between a service provider and a client.*

So why don't use 99,9th percentile every time as it is the best? This is a major cost factor, optimizing the last percentiles gets really expensive, especially as this usually involves a lot of hardware redundancy as well as eliminating factors outside of your control. At a certain point costs will be bigger than the benefits, so you need to make a trade-off.

Performance of a data system is defined by system throughput and response time, e.g. number of transactions (like read/write operations), processed records (like aggregations for analytical purposes) or even system commands (like an *update statistics* or rebalancing of several data nodes) under a given workload and for a specific timeframe. It usually depends on a variety of influencable as well as uninfluencable factors of the system itself, like network latency, a page fault or damaged disk.

2.1.1.3 Approaches For Scaling

Now that we are familiar with describing *load* and measuring *perfromance* we can answer the question: how to ensure performance, even if the load increases? As we have seen by the example of ebay.com within the previous chapters, a system which is able to cope with a load, won't be able to handle 10 times of that load. This needs us to think about the architecture right at the beginning as well as each time the load significantly increases. As we have learned there are two ways to scale an architecture of a data-system:

- **scale up/vertical** - replace a server by a more powerful one
- **scale out/horizontal** - distribute the load towards multiple server instead of one

A data-system running on a single server is easier to develop, as you can neglect a lot of factors that are specific to distributed systems (e.g. replication, partitioning or transactions and consistency across nodes), but more powerful machines are also more expensive and at a certain point you will reach the physical limit as ebay.com did.
A distributed data-system will require more development, test and operational effort

as well as result in complexity but servers will be much cheaper as you make use of less powerful machines (*"commodity" hardware*) and you are able to bypass the inevitable physical limit of a single server.
In practise you won't choose one pattern only (scale up or out) as well-working architectures need a carefully chosen mixture of both approaches, e.g. it doesn't make sense to make use of a lot of poorly powered servers (like a Raspberry Pi) instead of some more powerful machines in terms of unnecessary costs, network and software complexity. As you may guess, there is no one-size-fits-all solution and architecture for scalabale data-systems as the requirements are highly specific to each data-system itself. The *load parameter* may be strongly influenced by:

- the **volume of data** to store
- the **number of read or write operations**
- the required **throughput** or **response time**
- the **structure of the data** and **how it's accessed** (e.g. relational, document-oriented, graph)
- and many more.

Right now it shall be sufficient that you know the basics concepts of scalability, later within this lecture, we will make use of it when looking at distributed data storage and processing as well as related software and frameworks. At the end you should be able to apply those concepts to any data-system and be able to make reasonable decisions in terms of scalability.

2.1.2 Reliability

In general *reliability* represents the probability that something/someone will perform a required function without failure under stated conditions for a period of time, e.g. a test will be reliable when it gives the same repeated result under the same conditions.
Or more pragmatic: *something works correctly even if things go wrong.*

So what are the *faults*, mentioned in the previous defintions that we need to anticipate, about?

2.1.2.1 Hardware Faults

Obviously any hardware produced has a certain lifespan, buth that's not the only reason for hardware faults. If you're talking with operators of data centers, they will provide you with a broad list of common as well ase spine-chilling causes for hardware faults as:

- broken HDDs[23] or SSDs[24]
- faulty RAM[25] or CPUs[26]
- broken power adapters, switches or whole network outages
- unplugged network cables or even connected to the wrong port
- and many many more.

As this seems pretty unlikely at first sight - it's definitely not. For instance let's

[23]HDD, a hard disk drive is a non-volatile computer storage device containing magnetic disks or platters rotating at high speeds.
[24]SSD, a solid-state drive is a nonvolatile storage device that stores persistent data on solid-state flash memory.
[25]RAM, a Random Access Memory is the hardware in a computing device where the operating system, application programs and data in current use are kept so they can be quickly reached by the device's processor.
[26]CPU, a processor or processing unit is an electronic circuit which performs operations on some external data source, usually memory or some other data stream is called central processing unit

have a look at hard drives, especially in distributed data-systems you will have a lot of them. If you think about Apache Hadoop, you usually use low-class server (*"commodity" hardware*), e.g. *ProLiant DL380 Gen10 Server* as they provide a good ratio of:

- computing power (CPU) / Storage (HDD),
- rack space (2 RU[27]) / storage and
- benefit/cost.

Each of this servers can store 19 HDDs, if you build a hadoop cluster with those servers, e.g. with about 100 nodes, this means 1,900 HDDs. Based on a regularly study by BackBlaze[28] (a big data storage center provider like Amazon AWS) with a set of 82,516 HDDs, the average annual failure rate is about 2.11%. Regarding our previous Hadoop example containing 1.900 HDDs, we can suppose that nearly any week a HDD will fail. If we would make use of the particular HDD model *Seagate ST4000DX00* with a failure rate of 35,88% (also mentioned within the study) this would mean nearly 2 HDDs would fail each day.

In single server data systems it is possible to mitigate those problems by adding redundancy to individual hardware parts to minimize the failure rate of the whole system to a point where a failure is very unlikely, as at any time a redundant part can take over. This could mean, dual power adapters (like used by the previous mentioned *ProLiant DL380 Gen10 Server*), RAID[29] configurations or hot-swappable CPUs. As data volumes and computing demand increases, data-systems need to be distributed among several servers, which proportionally increases the rate of hardware faults and system failures, like discussed above regarding HDD faults. Therefore distributed data systems need to be able to tolerate the loss of entire machines, requiring software to be fault-tolerance additionally to hardware redundancy. But those distributed data systems have further advantages, a system that tolerates

[27](WPR18), Rack Unit is a unit of measure defined as 44.50 millimetres (1.75 in). It is most frequently used as a measurement of the overall height of rack frames.

[28](HDD17), https://www.backblaze.com/blog/hard-drive-failure-rates-q1-2017/

[29]RAID, is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both.

failure of single machines can be restarted, patched, updated (*rolling-upgrades*) or maintained - one node at a time - without a downtime of the whole data-system.

2.1.2.2 Software Faults

When we talk about software faults in terms of distributed data-systems, we don't talk about usual bugs but rather faults which affect the whole data-system integrity and reliability. Such faults are harder to anticipate than usual bugs of single-server applications, as they usually tend to be caused by the environment (e.g. multiple servers, network, dependencies, special and unusual circumstances), are difficult to test, and are even worse in their result as they can cause a failure of the whole data-system. Examples could be:

- a runaway and/or zombie process that extensively used up some shared ressource (e.g. network, CPU, RAM, disk space)
- a software bug causing the whole cluster to fail (e.g. the Hadoop Ressource Manager YARN[30] once had a bug[31], that if you removed a cgroup (*control group*) under some circumstances (*race conditions*) a kernel panic and in this way a failure of multiple server was caused
- a service the whole data-system depends on slows done, becomes unresponsive or fails
- cascading failures (e.g. one server of the data-system fails due to heavy network traffic, causing the other servers to take over, in this way increasing network traffic for them too and finally all server will fail)

As you can see, most of the reasons for software faults are caused by assumptions about the environment that may not be true at some time and at some special circumstances. To avoid suffering those issues you need to carefully think about assumptions and interactions within the distributed data-system, you will need a lot of measuring, monitoring and you will do a lot of analyzing of the system behaviour in any special circumstance as well as testing even with forcing some servers of the system to crash.

[30] YARN, Yet Another Resource Negotiator
[31] (YAR18), https://issues.apache.org/jira/browse/YARN-2809

2.1.2.3 Human Faults

We have been talking a lot about reliability so far, but what about the most unreliable factor: humans. We will briefly discuss some approaches to make a data-system reliable in terms of unreliable humans:

- decouple places where people make the most failures from places they can cause failures, e.g. using production and development environments or providing interfaces or frameworks for an API instead of direct API access
- use extensive testing (e.g. unit test, system tests, integration tests) and automize them
- measuring and monitoring (e.g. performance metrics, error rates) allows to check wether assumptions or constraints are violated at an early stage

To sum up the last chapters: why do wee need reliability? Reliability is not just a major topic for stock exchanges, air line reservations or military. Failures of a data-system can cause data loss, lost productivity or sales loss and therefore huge costs and loss of revenue. There are special circumstances when you may choose to reduce reliability for the sake of time, development effort or operational costs (e.g. *prototyping*), but you need to be very conscious and it's inevitable that at some point in the future you will need to invest the the saved effort, time and costs and probably a multiple of what it would have been before.

Reliability in terms of hardware, software or especially data-systems can be defined as the ability of a system to function as specified and expected. A reliable data-system also detects and tolerates faults due to mistakes of users, hardware or lower parts of the data-system itself as well as ensures the required performance under any expected load.

2.1.3 Maintainability

Maintenance is known as one of the biggest costs at software devlopment and in the same way a very unfamous topic to software engineers. Keeping a system running, investigating failures, fixing bugs, adding new features, adapting it to updates of underlying hard- and software - to name some usual tasks.

A data-system should be designe to minimize effort during maintenance and in this way making it more reliable. We will briefly discuss 3 major topics: *operability*, *simplicity* and *evolvability*.

2.1.3.1 Operability

The main goal of operability should be to make operations easy to keep the system running smoothly, this means making routine tasks easy and enable operations ressources to use their time for important tasks. This can be achieved by:

- good documentation and operational model (a data-system which can be understand easily can be operated more easily)
- transparency (visibility into the data system and runtime behaviour, e.g. by log files or monitoring tools)
- no dependencies between single services or server (allow single server to go down for maintenance tasks, e.g. patches, update or restarts)
- self-healing if possible, but also possibilities to override for operators

2.1.3.2 Simplicity

When you start a development project everything is pretty simple and probably well-documented but later on with multiple developers, features, services and servers, it gets more complex, hard to understand by a developer and especially more difficult to handle by administrators. As a lot of issues caused by this are not specific to data-systems we will focuse on complexity and abstraction, as reducing complexity should be the main goal when devloping distributed data-systems. Making a sys-

tem less complex doesn't require reducing functionaility, it's more about removing unnecessary complexity.
For instance if your data-system is crunching a lot of data for a very plain purpose, like parsing web server log files for analytical purpose to get to know how many people have visited your website - you could do this counting in Java (MapReduce) but this will probably be about 50 lines of code, a lot of libraries, testing and dependencies - making operations more difficult in the same way. If you would do this using Hive on HDFS your done with a one-line SQL statement.

However finding useful abstractions is not that easy and needs a lot of experience, but when you are developing something you should always ask yourself: do I make use of abstraction and will the complexity be at a manageable level?

2.2 Storage Concepts

to-be-added

2.3 Data Models And Access

Data Models are one of the most crucial part of any data-system as they significantly influence actually everything: development and required skills, operations, backends and frontends as well.
For instance, if you choose a document oriented data store (e.g. MongoDB) to serve a web-based application, as it is pretty easy to build a REST API on top of it and JSON data and JavaScript Frontends (e.g. based on AngularJS) are a perfect match but don't think about how the data will be queried later - you may run into a lot of trouble. For example if your data consist of many business objects which are highly complementary and the frontend usually needs multiple business objects within one request, you're probably out of luck as document-oriented storages perform pretty well when only reading from one collection but not at joining them. You could probably put all related business objects in one document, but as this is unnecesessarily redundant it will burst your data storage very soon.
To conclude: there are many different kinds of data models and every data model has it strengths but weaknesses as well. Some are easy to use - some not, some operations using them will be fast - others definitely not, some are very storage efficient - some note... As you can see it's not that easy to decide which data model to use as it has a profound effects on the whole data-system you are building, even that far that a wrong decision could be a show stopper at some time in development.
It needs a lot of experience and very forward-looking to choose a data model that fits best to your data-system. Within the next chapters we will talk about the most common data models (*relational*, *document-oriented*, *graph*), how to access them *SQL*, *MapReduce*, *SPARQL*), how they differ, advantages and disadvantages.
At the end you should have a basic understanding on how they work and which of them fits best to a certain problem.

2.3.1 Data Models

2.3.1.1 Relational Data Model

Firstoff we will take a look at the relational data model, originally introduced by Edgar Frank Codd in 1970[32].

The **relational model** is an approach to managing data using a structure and language consistent with first-order predicate logic where all data is represented in terms of tuples and grouped into relations.

Having its roots in the 1970's the relational data model had the idea to hide implementation details (e.g. internal representation of data in a data store) from developers by providing a cleaner, *declarative* and *read-on-schema* interface for specifying data and querying it. Developers can state what information the database contains and what information they want from it - the database itself will take care of describing, storing and retrieving data for developers. As you have already learned the basic about relational databases, indices, concepts like normalization and much more we will take a quick look at an example and afterwards conclude about advantages and disadvantages compared to other data models.

Let's take a look at figure 2.6 on page 26 as an example. Here you can see how a Facebook profile could be represented in a very simple (and not fully normalized) relational data model. The table `user` works as the main entity, as it's the profile page of a user. As a user is unique we also have a unique identifier (`id`) as well as some information regarding the user within the same table (e.g. `first` and `last_name` or who they are `married_to`). People may have worked for different companies (table `companies`), lived in several cities table `cities` or visited more than one university (table `universities`) so we put that into separate tables including a foreign key (`user_id`) of the referring table `user`.

[32](Cod70), 'A Relational Model of Data for Large Shared Data Banks" - IBM Research Laboratory, San Jose, California

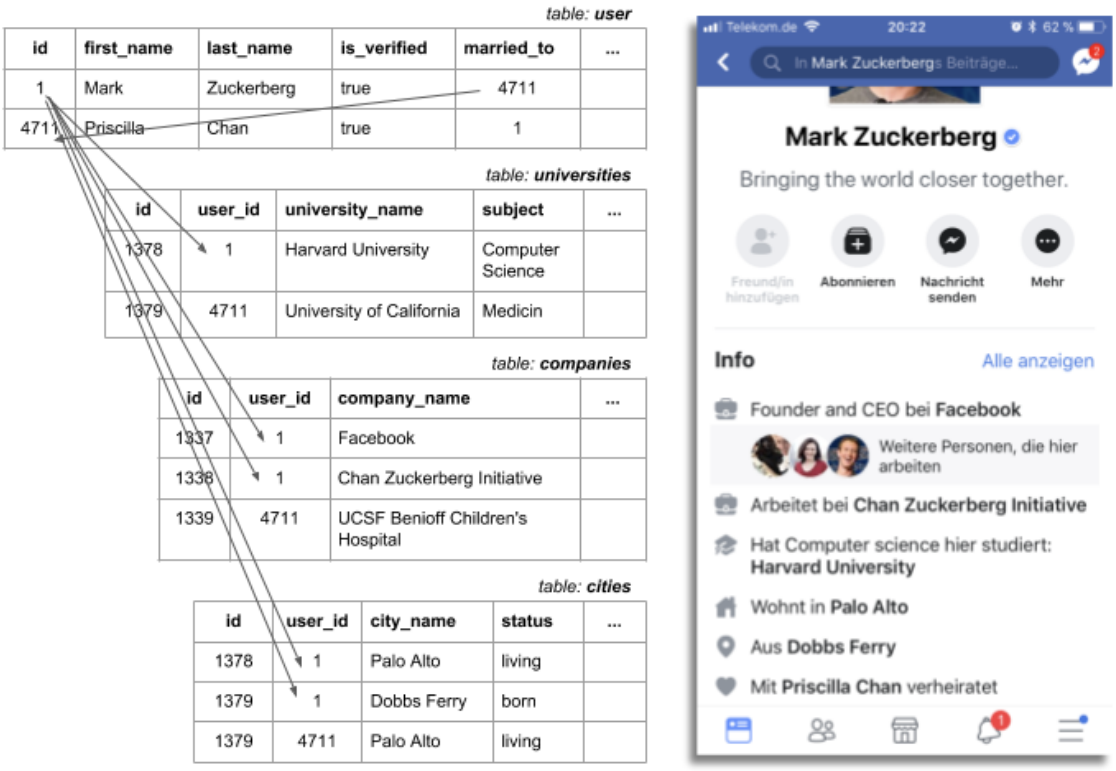

Figure 2.6: Schema - Facebook Profile - As Relational Model

Relations are represented by *tables* and tuples are represented by *rows*. Rows can easily be inserted or fetched all-at-once, by using an id or a *primary/foreign key* unlike *document-oriented* data models, where you:

- sometimes need to make use of access paths,
- need to think about nested structures,
- need to worry about unknown fields as those systems are usually *schema-on-read* or
- or intensively need to think about possible performance issues and how the system will probably execute your query as the execution engine is usually not that mature as the one of a relational system,
- it's more difficult to understand how the systems works, as sometimes you don't even have a query-explain feature like in relational databases, so you won't be able to investigate or guess in advance how it will behave in detail during execution.

But as relational data models are also mostly used by applications, we need to talk about a frequently discussed issue: *object-orientation*. As applications are *object-oriented* you need to make use of a translation layer between an applications and relational model to enable the applications to make use of the data. There are many frameworks available to serve this translation called ORM[33], e.g. Hibernate[34] in case of Java or SQLAlchemy[35] in case of Python to name only 2 of them. Those frameworks will require additional code, skill and development time, as systems using the document-oriented model are regularly used without additional frameworks for data translation, e.g. an AngularJS application using MongoDB.
But systems using a relational model are mostly more resilient as they usually gained several years or even decades of research, experience and development time, while document-oriented that are still widely used today just started in the early 2000's. For instance many of them have highly sophisticated query optimizer which automatically decide which part of the query needs to be executed when, in which order and which index e.g. `user_id` of the previous example is probably the best one to use - most of the document-oriented systems don't even have something that is worth to be called a query optimizer as their capabilities are far behind the relational ones.
But that's also due to the fact that relational and document-oriented data models are completely different in terms of relationship implementation: both of them are able to represent *many-to-one* or *many-to-many* relations but in a different manner. Relational data models make use of *primary* and *foreign keys* which can easily be used for indices, document-oriented data models need to make use of nested structures (most commonly used) or *document references* and joins to other collections (e.g. `\$lookup` in MongoDB[36], `populate()` in Mongoose[37] or Joins in RethinkDB[38]) - both of them are resulting in expensive IO and CPU operations as either you need to make use of features that are extremely immature compared to their relational counterpart or the whole document needs to be read, which can also be very wasteful

[33] ORM, Object-relational mapping is a programming technique for converting data between incompatible type systems using object-oriented programming languages.
[34] (ORM18a), http://hibernate.org/
[35] (ORM18b), https://www.sqlalchemy.org/
[36] (MDB18e), https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/
[37] (MGP18), http://mongoosejs.com/docs/populate.html
[38] (RDB18e), https://www.rethinkdb.com/docs/table-joins/

on large documents.

This behaviour of document-oriented systems also applie for writes, as they usually require to rewrite the whole document in document-oriented data-systems. But to be fair, most document-oriented systems weren't initially designed with serving relationale dependencies or querying in mind, but rather for the sake and benefits of:

- being object oriented,
- easy to be altered,
- being semi-/unstructured and schema-"free".

An in this way being the whole opposite of the relational data model.

2.3.1.2 NoSQL Data Model

As we have discussed the relational data model in the previous chapter we will now take a closer look on non-relational data models, probably known as *NoSQL*.
Even due to the fact that there have been some approaches and databases in the past (back to 1960's), the term *NoSQL* and data-systems using it, gained their popularity in the early twenty-first century. The NoSQL wave was initially triggered by the emerging wave of *Web 2.0* companies like Facebook, Google or Amazon, whose data-systems needed to cope with data sizes, throughput, response times and the need of high scalabilty, which could not be handled by using traditional relational data-systems anymore[39]. Beside that the limitations of the relational data model for *Web 2.0* kind of data, probably was on of the biggest issue, log files and textual data of social media websites or e-commerce shops are not very structured and even if they were, they probably changed their structure very frequently over time.
NoSQL data-systems like HBase, BigTable or Cassandra emerged by those needs, e.g.:

- **Cassandra**, initially developed by Avinash Lakshman and Prashant Malik (Facebook employees) because of the need to solve the inbox search problem of the Facebook messenger[40] and later on released to be *OpenSource* in 2008 or
- **BigTable**, initially developed by Google in 2004 to support several data intensive applications of google, such as web-indexing, GoogleMaps or GoogleMail and later on released as a public service in 2015[41].

As you can see NoSQL data-systems are increasingly used by BigData and real-time web and analytical applications as they can cope better with frequently changing data structures and are easier to scale horizontally, compared to their relational counterpart.

[39](Moh13)
[40](FBI08), https://www.facebook.com/note.php?note_id=24413138919
[41](BTR15), https://cloud.google.com/bigtable/docs/release-notes

NoSQL data-systems are somethimes also called *"Not only SQL"* to emphasize that they may support SQL-like query languages, even if their storage engine has "nothing" in common with traditional relational data-systems, e.g.:

- **Hive**, initially developed by Facebook, is an abstraction layer to access data on a distributed file system (e.g. Hadoop HDFS or Amazon S3) without the need of programming MapReduce jobs by using SQL-like queries (*HiveQL*[42]) and *schema-on-read* or
- **Cassandra**, initially developed by Facebook as a NoSQL distributed key-value data store, which also provides a SQL-like query language, called *CQL*[43].

NoSQL in terms of BigData, is a theorem for managing data using document-oriented, key-value or graph structures, where all data is represented in terms of documents, key-value pairs, graphs (nodes, edges, properties) or mixed approaches.

Let's take a another look at the example in Figure 2.7 on page 31, which we already discussed by regards of the relational data model in chapter *2.3.1.1 Relational Data Model*. Here you can see how the same Facebook profile could be represented in a very simple document-oriented data model (e.g. *JSON*[44]). Representing a data structure like a (Facebook) profile, which is mostly a self-contained document, using JSON could be vary appropriate. As you can see in this case the document structure has a better *locality* than the relational structure, where you would need to fetch rows from multiple tables and join them. Using the document-oriented structure, you just need to fetch one document as all the information you need to create the profile page, are at one place. The document-oriented structure allows you to easily access previously *normalized* data (*one-to-many relationship*), like `companies` a person worked for or `cities` a person lived in, just by it's explicit tree structure.

[42](AHW18), https://hive.apache.org/

[43](CQL18), http://cassandra.apache.org/doc/latest/cql/

[44](JSO18), JSON (JavaScript Object Notation), a lightweight data-interchange format, easy for humans to read/write and machines to parse/generate.

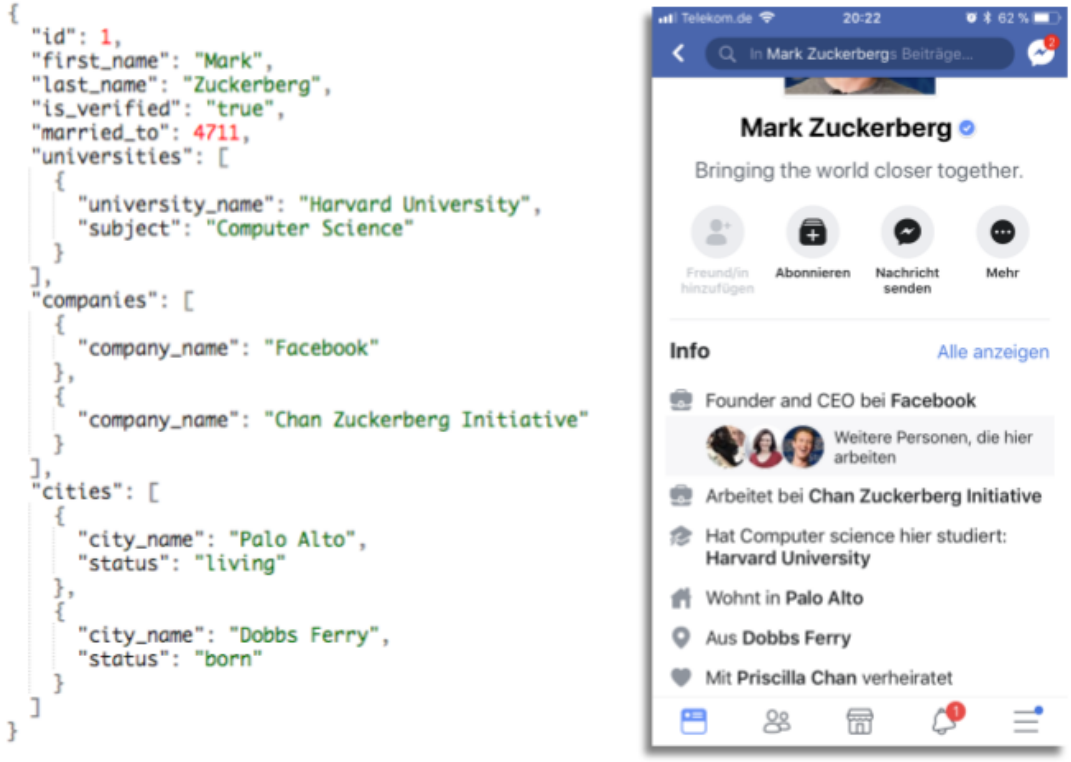

Figure 2.7: Schema - Facebook Profile - As Document Oriented Model

Let's have a closer look on 3 topics often discusses as advantages of the document-oriented data model:

2.3.1.2.1 Schema Flexibility

Most document-oriented data-systems do not enforce a defined schema unlike the relational data model. For instance if we take look at Figure 2.7 arbitrary keys and values, e.g. additional information like interests or favourite movies could be added to the document without causing any issues. But if your reading documents later on, you have no guarantee wether a certain document contains a required field or not, instead you need to be sure or make use of additional operators, for instance the `$exists` operator in case of MongoDB to check wether a required field exists or not[45].

NoSQL data-systems are often called *schemaless*, which is wrong, as the engine

[45](MDB18d), https://docs.mongodb.com/manual/reference/operator/query/exists/

which is processing the data usually assumes some kind of structure. This is called *schema-on-read*, the structure of the data is implicit and only interpreted when the data is read, whereas traditional relational data-systems are *schema-on-write* and explicit forcing the engine to ensure all data which will be written is conform to the schema. This approach is very valuable in case an application wants to change the format of it's data on a regular base. In case of a relational data-system you would need to run an `ALTER TABLE` or even `CREATE TABLE` statement, which are highly IO and CPU intensive operations. As a very drastic example, if you run an `ALTER TABLE` statement in MySQL, this will not only cause a lock for `INSERT` and `UPDATE` statements on the same table (if not using InnoDB), but also copying the entire table, which could take even hours depending on the size of the table[46].
To sum up, in case the data your data-system needs to store is heterogenous, whereby not all records share the exact same structure and/or the schema will frequently change for a subset or all of the data, *schema-on-read* is an appropriate approach.

2.3.1.2.2 Data Locality

As *schema-on-read* data is usually represented (*encoded*) as a single document (e.g. JSON or CSV[47]) or as their binary/serialized counterpart (e.g. Avro[48] or ORC[49]) and your application often needs to access the entire or a large part of the document, there is a huge performance advantage in terms of *storage locality* compared to relational data models, which need to read and join multiple tables (e.g. compare Figure 2.6 on page 26) requiring probably more IO (e.g. disk seeks and reads) and CPU time (e.g. join operation). But this could also be a pitfall, the advantage only applies if you need a large part of the required document at the same time, as the data-system usually needs to load the whole document, even if you only need one value, which could be vary expensive on large documents. Another disadvantage are updates, as the data-system usually needs to rewrite the whole document, even if you only change one value. As an example, if you update a field in MongoDB, e.g. incrementing a counter:

```
db.some_collection.update( { _id : ... }, { $inc : { y : 2 } } );
```

[46](MYS18a), https://dev.mysql.com/doc/refman/5.5/en/alter-table.html
[47]CSV, a text file that uses a comma to separate values
[48](AVR18), https://avro.apache.org/
[49](ORC18a), https://orc.apache.org/

MongoDB is even that optimized, that no rewrite of the whole document is caused. But if you add another field, which causes the document to increase in size, it will no longer fit into the previously allocated space, the whole document needs to be rewritten anyway[50].

2.3.1.2.3 Application Code Simplicity

Building a data-driven application obviously requires development and therefore application code that handles all interaction with the used data-system. Using a document-oriented data-system, where typically entire documents are loaded and written, usually requires less code within the application itself compared to relational data-systems where several tables need to queried, joined or updated at the same time. But in the same way it's a significant disadvantage of document-oriented data-systems, for instance if you would like to access a nested item within a document, or even the n-th tuple within the nested item, it often requires more code than an appropriate SQL would need. On the other way, just querying one single relational table requires way more code than querying one document, as the relational data model is not a natural fit to the world of object oriented programming languages. You need to write a lot of mappings to map the data to your application objects or use an ORM framework (as previously discusses) which also adds a lot of additional code in terms of libraries to your application. The document-oriented data model is a much more natural fit to objects uses in object-oriented programming languages and require less code to be linked to an objects, especially in the environment of data-intensive web applications, which for instance frequently make use JSON. But there are also cases like *many-to-many* relationships where a document-oriented data model will force you to write a lot of code. As we already discussed, document-oriented data-systems usually suffer and performing joins between different collections/entities, it's possible to reduce the need for joins by strictly denormalizing data - suffering all the disadvantages we have already discussed - or by moving the join operation from the data-system to the application code. This will in some way speed things up, but in the other way slow things down, as the application usually will not be as fast as a join within a database. This will also cause additional load on the application side as well as more complexity within the

[50](MDB18c), https://www.mongodb.com/blog/post/fast-updates-with-mongodb-update-in-place

application code itself, as it needs to take care about everything the data-system would usually do.

There is no universal answer to the question which data model leads to more or less application code, as it highly dependends on the data itself, kinds of relationships, how the data is usually queried, which operations need to be run on the data-system and many more factors.

2.3.1.3 Graph Data Model

to-be-added

2.3.2 Data Access

2.3.2.1 SQL

Let's take a look at SQL[51], originally introduced as "SEQUEL" by Donald D. Chamberlin and Raymond F. Boyce in 1974[52] and inspired by Edgar Frank Codd's relational data model in 1970[53]. Despite not entirely adhering to the relational model as described by Codd, it became the most widely used database language and even supported in a limited way by NoSQL data-systems like Cassandra (*CQL*[54]) or Hadoop (*HiveQL*[55]). Designed for handling structured data and making it easily accessible to users and application code, it quickly became standard (ANSI 1986, ISO 1987). It obviously has been revised several times since then, to include a larger set of features but it's still used by an kind of relational data-system. Despite the fact that most SQL code is not completely portable among different, especially commercial, relational data-systems.

SQL is a set-based and declarative programming language unlike imperative or mixed-up programming languages, paradigms or approaches for accessing data (e.g. MapReduce, Spark). An imperative programming language forces the data-system to perform certain operation in a certain order. Imagine any application code you have written so far, for instance in Java, C++ or Python, the code will be executed line by line, doing some calculations, evaluating conditions and probably looping around several times. A declarative language like SQL (or programming languages like *Prolog* or *Lisp*, or even build languages like *make/cmake*) is usually more easy to work with than an imperative language. In case of SQL it is due to the fact, that it hides implementation details of the data-system, you usually do not need to think a lot about how the data is stored or retrieved as the data-system will take care about this as well as doing it with the best performance possible (*query optimizer*). Declarative languages are also easier to parallize in terms of horizontal scalability, as they only specify the pattern of the result, not the algorithm used to get the

[51] SQL, Structured Query Language

[52] (DDC74), "SEQUEL: A STRUCTURED QUERY LANGUAGE" - IBM Research Laboratory, San Jose, California

[53] (Cod70), 'A Relational Model of Data for Large Shared Data Banks" - IBM Research Laboratory, San Jose, California

[54] (CQL18), http://cassandra.apache.org/doc/latest/cql/

[55] (AHW18), https://hive.apache.org/

result. Whereas imperative programming languages and approaches are way more complicated to parallelize across multiple CPUs or nodes. As they require a lot of effort and attention, as imperative code specifies operations that need to happen in a specific order, need to be kept in-sync, are dependent of the results of previous steps and much more.

2.3.2.2 MapReduce

MapReduce is a programming paradigm and an associated implementation for processing and generating large data sets in parallel and distributed on a cluster, originally introduced by Jeffrey Dean and Sanjay Ghemawat (Google Inc.) in 2004[56]. MapReduce is neither a declarative language nor a an imperative programming language, it's something in between - the logic of data querying is expressed with snippets of code, which are executed repeatedly by the processing framework/runtime system. The paradigm is based on specifying a *map* function, which performs filtering and sorting, resulting in an intermediate set of key/value pairs and a *reduce* function that merges all intermediate values associated with the same key (e.g. sum all values). Applications written in this paradigm are automatically parallized and executed on several nodes of a cluster. The runtime system (e.g. YARN[57] in case of Hadoop) takes care of:

- the details of providing and partitioning the input data,
- scheduling the application code execution across all cluster nodes,
- saving intermediate states,
- handling node failures,
- inter-node communication and much more.

This enables programmers, without any experience with parallel or distributed systems, at a minimum effort, to easily utilize the computing ressources of distributed data-system and build data-driven applications that are highly scalable.

[56](JD04), "MapReduce: Simplified Data Processing on Large Clusters" - Google Inc.
[57](VKV13), "Apache Hadoop YARN: Yet Another Resource Negotiator" - The Hong Kong University Of Science And Technology

```
1  map(String key, String value):
2    // key: document name
3    // value: document content
4    for each word w in value:
5      EmitIntermediate(w, 1);
6
7  reduce(String key, Iterator values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
12     result += v;
13   Emit(key, result);
```

Code Snippet 2.1: MapReduce Example - *Word Count*

Above you can see a very simple pseudo code example (Code Snippet 2.1) to count the occurences of single words within a document (which is almost impossible to do in a SQL query on a database). The `map` function is called once for each document and emits each word within it. The `reduce` function sums together all counts emitted for a particular word.

The `map` and `reduce` functions are both defined with respect to data structured in *key/value* pairs. The `map` function takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$map(k1,v1) \rightarrow list(k2, v2)$$

This produces a list of pairs (keyed by `k2`) for each call. After that, the MapReduce framework collects all pairs with the same key (`k2`) from all lists and groups them together, creating one group for each key. The `reduce` function runs in parallel for each group, which in turn produces a collection of values (`v3`) to an associated key (`k2`) within the same domain:

$$reduce(k2, list(v2)) \rightarrow list(k2, v3)$$

The returns of all `reduce` processes are collected as the desired result list.

Let's take a deeper look at the previous example and how the paradigm would work in detail, given the documents below:

```
1 document1 = "Da steh ich nun, ich armer Tor!";
2 document2 = "Und bin so klug als wie zuvor;";
3 document3 = "Heiße Magister, heiße Doktor gar";
4 document4 = "Und ziehe schon an die zehen Jahr";
5 document5 = "Herauf, herab und quer und krumm":
6 document6 = "Meine Schüler an der Nase herum.";
```

Code Snippet 2.2: MR Example - *Word Count (Input Documents)*

Which would result into 6 processes executing the `map` function (special characters removed ahead as well as converting everything to lower-case to avoid duplicates):

```
1 map(document1, "da steh ich nun ich armer tor");
2 map(document2, "und bin so klug als wie zuvor");
3 map(document3, "heiße magister heiße doktor gar");
4 map(document4, "und ziehe schon an die zehen jahr");
5 map(document5, "herauf, herab und quer und krumm");
6 map(document6, "meine schüler an der nase herum");
```

Code Snippet 2.3: MR Example - *Word Count (`map()` Calls)*

Creating the following partial results:

```
1 p1 = [ ("da",1), ("steh",1), ("ich",1), ("nun",1), ("ich",1),
2        ("armer",1), ("tor",1) ];
3 p2 = [ ("und",1), ("bin",1), ("so",1), ("klug",1), ("als",1),
4        ("wie",1), ("zuvor",1) ];
5 ...
6 p6 = [ ("meine",1), ("schüler",1), ("an",1), ("der",1), ("nase",1),
7        ("herum",1)];
```

Code Snippet 2.4: MR Example - *Word Count (Partial `map()` Results)*

Those partial results calculated by the `map` processes will be collected (if they have the same key/word) and handled over to the `reduce` processes, which summarize the partial results:

```
reduce("da", [1]); // = ("da", 1)
reduce("ich", [1,1]); // = ("ich", 2)
reduce("und", [1,1,1,1]); // = ("und", 4)
...
```

Code Snippet 2.5: MR Example - *Word Count (`reduce()` Calls)*

At the end, the desired output could look like:

```
result = [ ("und", 4), ("ich",2), ("heiße",2), ("an", 2), ("da",1),
    ("steh",1), ("nun",1), ("armer",1), ("tor",1), ("bin",1), ("so",1),
    ("klug",1). ("als",1), ("wie",1), ("zuvor",1), ("magister",1),
    ("doktor",1), ("gar",1), ("ziehe",1), ("schon",1), ("die",1),
    ("zehen",1), ("jahr",1), ("herauf",1), ("herab",1), ("quer",1),
    ("krumm",1), ("meine",1), ("schüler",1), ("der",1), ("nase",1),
    ("herum",1) ]
```

Code Snippet 2.6: MR Example - *Word Count (Final Result)*

Now that we know how *Map* and *Reduce* work in Detail, let's have a look at the whole process using the previous example.

Figure 2.8 on page 40 illustrates the whole process required to run MapReduce, usually managed and executed by framework and ressource manager like YARN in case of Apache Hadoop:

- **Input Phase:** Read Input, usually from a distributed file system (e.g. HDFS, GoogleFS or Amazon S3), divide input into parts of appropriate size and generate key/value pairs. As data on a distributed file system is already arranged in blocks (e.g. 128 MB as default per block on Apache Hadoop), the split size of the data passed to each `map` process is commonly equal, but can usually be customized within a MapRecude program. For the purpose of

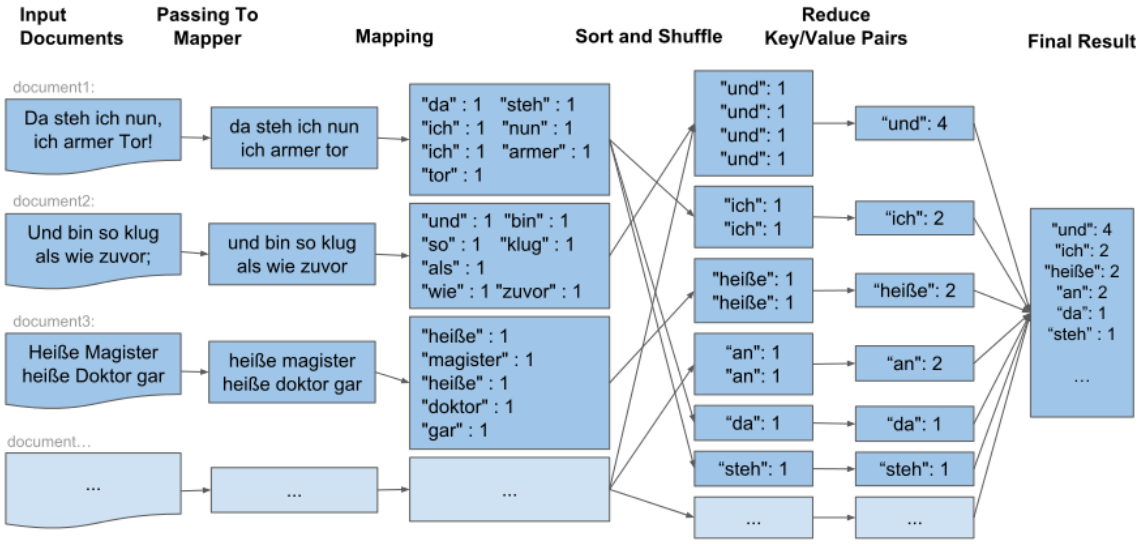

Figure 2.8: Schema - MapReduce Phases - Word Count Example

simplicity, the example data is to small, whereby documents doesn't need to be split up.

- **Map Phase:** As already discussed, each `map` function gets multiple key/value pairs and processes each of them separately. All Map processes run indepently of each other, faciliating the whole processing to run concurrently on multiple nodes. In our example resulting in to several lists of words with an associated count of one and represented as key/values pairs.
- **Sort&Shuffle Phase:** Phase in between Map and Reduce with the purpose of transferring data from `map` to `reduce` processes efficiently, which is usually done automatically by the MapReduce framework. All output produced by the `map` processes is grouped and sorted by their key, partioned and distributed to the `reduce` processes. A common way of partitioning is to hash the keys and use the hash value *modulo* amount of reduce processes to achieve an evenly distribution of load among all nodes of a cluster (the total number of partitions is equal to the number of reduce processes). Usually you do not need to make use of the frameworks default *Shuffle*, *Sort* and *Partitioning*, for instance using Hadoop you could implement your own ones[58], but as those parts are

[58](HDP18), https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html

centerpieces in case of the whole MapReduce performance, think twice about not using the default ones.

- **Reduce Phase:** As already discussed, each `reduce` function gets called once for each unique key. All Reduce processes run indepently of each other, facilitating the whole processing to run concurrently on multiple nodes. In this way the `reduce` functions iterates through all values associated to a key and produces zero or more output. In case of the WordCount example an increment happens for each value (count) associated to the same key (word).
- **Output Phase:** The output of all `reduce` processes are collected and consolidated by the MapRedcue framework and usually written to a distributed file system.

Beside the listed phases, most MapReduce frameworks also make use of a phase called **combiner**, which happens between *Map* phase and *Sort&Shuffle* phase. The *combiner* is also known as a "mini-reducer", it takes the intermediate output of the `map` processes - shuffles, sorts and reduces them partly by combining key/value pairs with the same key. In case of our WordCount example (Figure 2.8 on page 40) the output of the first mapper wouldn't be:

```
[ ("da",1), ("steh",1), ("ich",1), ("nun",1), ("ich",1), ("armer"
   ,1), ("tor",1) ];
```

but:

```
[ ("da",1), ("steh",1), ("ich",2), ("nun",1), ("armer",1), ("tor"
   ,1) ];
```

As we can see, the combiner reduces the intermediate result of the first `map` process as it combines both entries for key *"ich"* to one entry and by that the amount of data which needs to be written to disk and/or transferred to and processed by the `reduce` processes. This reduces network bandwith usage, reduces network congestion, increases reducers performance and in this way speeds up the whole MapReduce execution time.

Speaking about performance, MapReduce programs aren't designed for the purpose of being fast, but to handle huge amounts of data in a scalable way, this means amounts of data, that won't fit entirely into the main memory of a single server (in this case, any database will probably be faster than MapReduce). Also most MapReduce frameworks are designed for being able to recover from failure of entire nodes within the cluster, during the execution of a long running MapReduce program. This is achieved by writing intermediate results to the distributed file system, which is I/O expensive and only pays off when the computation involves many computers and a long runtime of the MapReduce program.
The main goal of this model is to only have to write the `map` and `reduce` functions and to exploit the optimized sort&shuffle operation. The *Sort&Shuffle* phase is a crucial part of MapReduce as the partition function and the amount of data produced by the `map` function has a huge impact on the overall performance and scalability. When developing a MapReduce program, you need to make a tradeoff between computation and communication costs. For instance: the *reduce* phase needs sorting upfront (grouping and sorting of the *keys*), which has nonlinear complexity. This could imply to make use of small partition sizes to reduce computation time required for sorting, but would also result into a large number of `reduce` processes and therefore a lot of time-consuming I/O operations and network congestion.
Also mind that `map` and `reduce` functions are very pure functions, they are very limited in what they can do and what not. Which for instance means they cannot perform additional asynchronous operations like queries to other data systems like databases or APIs, as this wouldn't allow MapReduce to run those functions on any node of a cluster in any order and in this way to *scale horizontally*.

As MapReduce is kind of a low-level programming model, it is most likely you will also make use of higher-level APIs and query languages such as Hive, as they will translate your queries into low-level MapReduce operations, without you having to write them.

2.3.2.3 SPARQL

to-be-added

2.4 Challenges Of Distributed Data Systems

to-be-added

2.4.1 Replication

Replication is the process of continuously copying data (usually via network) from one part of a data-system to one or multiple others and keeping them in-sync. This serves the purpose of, e.g.:

- **Availability and Redundancy:** Even if some parts or nodes fail the whole data-system is able to continue working, as it can make of another replica.
- **Scalability and Performance:** Using multiple replicas, for instance increases read performance and throughput as read queries can be distributed to any node of a replica set or even be handled concurrently by multiple nodes of the same replica set.
- **Reliability:** Using multiple replicas stored in different data centers and locations, the data-system even continues to run during a catastrophe like an earthquake, typhoon or just a construction worker, having a bad day and cutting of the power link of one of your data centers.Reduce
- **Low Latency:** Keeping replicas of a data-system geographically close to users or consuming applications reduces latency (e.g. in case of a multinational webshop, having a replica in every country).

But all of those advantages come at a price: keeping data replicated, requires a lot of overhead, especially on non-read queries like inserting, updating or deleting data. If the data set won't change, it's just about copying the data to all replicas but if we need to handle changes for the replicated data it gets difficult. Keeping all replicas in-sync even by using unreliable ressources like network, is a tough thing to to, as there are a lot of potential issues specific to those setups like failed replicas, partial network outages, concurrency and conflicts that need to be taken into consideration. We will discuss common approaches (*master*, *multi-master*, *masterless*) for replication as well as accompanied challenges within this chapter. For the sake of simplicity we will not think about *partitioning* right now, assume a dataset which fits entirely

on a single node and get back to partitioning within the next chapter 2.4.2.

2.4.1.1 Master Replication

Master Replication is the most simple approach for achieving replication and already used by a lot of relational data-systems, like MySQL[59] or PostgreSQL[60] but also NoSQL data-systems like MongoDB[61] or RethinkDB[62].
The Master replication (also known as *Primary/Secondary*, *Master/Slave* or *Single-Leader* replication) consists of two types of replicas:

- **Master:** Dedicated processing node, usually also a replica and also known as *primary* or *leader*. The master node serves read as well as write queries, is responsible for persisting changes locally and propagates changes (e.g. by using replication logs or change streams) to slave nodes.
- **Slave:** Slave (also known as *secondary*, *follower* or *hot-standby*) are dedicated for serving read queries only. They receive changes from the master node and update their local replica accordingly to and in the same order as the replication log provided by the master.

Let's take a look at Figure 2.9 on page 45, which illustrates the communication between every part of the data-system, with regards to the *user*, the *master* and the *slaves* (y-axis) and the time (x-axis). As you can see a user is able to query and receive data from any replica within the replica set, it does not matter wether the node is a *master* or *slave*, any option is possible.

[59](MYS18e), https://dev.mysql.com/doc/refman/8.0/en/replication.html
[60](PSQ18c), https://www.postgresql.org/docs/9.6/static/runtime-config-replication.html
[61](MDB18b), https://docs.mongodb.com/manual/replication/
[62](RDB18a), https://www.rethinkdb.com/docs/architecture/

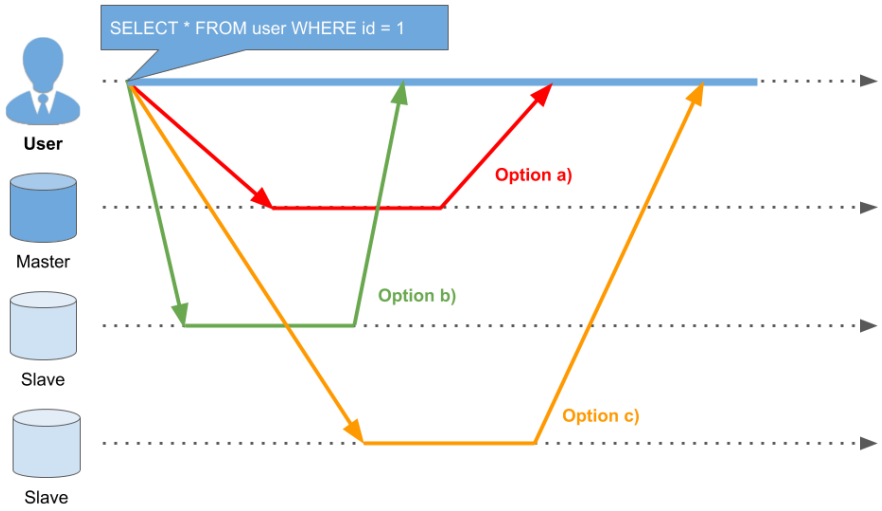

Figure 2.9: Schema - Master Replication - Read

Write operations as shown in Figure 2.10 are not that easy to handle, as only the *master* is allowed to process write queries. The master processes the requests and propagates it afterwords to all slaves. When all slaves have succeeded in processing the write query, the master reports success to the user as well as making the data change visible to all other users.

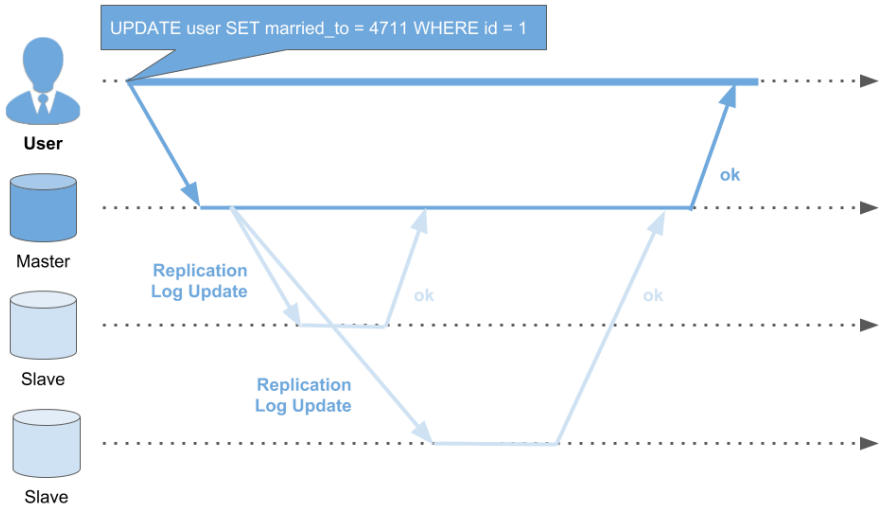

Figure 2.10: Schema - Master Replication - Write

This is also calles *synchronous* replication. This is usually configurable using relational databases but could also be hardcoded sometimes. There are 3 kinds of replication:

- **Synchronous:** As illustrated in Figure 2.10, using synchronous replication, the master waits for slaves to succeed before reporting success to the user.
- **Semi-Synchronous:** As illustrated in Figrue 2.11, using semi-synchronous replication, the master waits for one replica to report success, before reporting success to the user.
- **Asynchronous:** The master does not wait for any slaves to report success, before reporting success to a user.

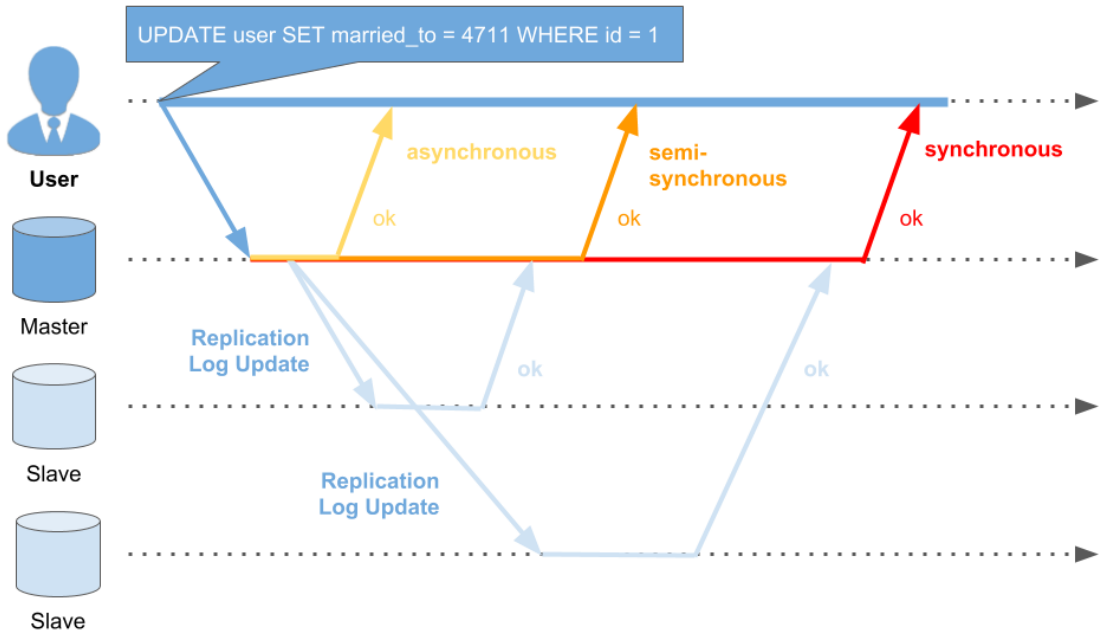

Figure 2.11: Schema - Master Replication - Synchrony

Synchronous replication ensures that all slaves have an up-to-date copy of the data that is shared with the master, which is a great advantage, as any slaves could take over if the master crashs. But this comes at a price: latency and fault tolerance. If there are networks latencies, they will directly affect the write performance, as the master needs to wait for the slaves to finish. If a slave crashs, the master won't

be able to process the write query at all and waits till the replica gets available again. Even if everything is working smoothely, it will still be slower than any other replications type, as slaves still add additional processing time to write queries.
As you can see it usually does not make any sense to keep the whole data-system synchronous, as the failure of one node would cause the whole data-system to basically stop working. In practise it's common to make use of **semi-synchronous** replication, which means one slave is running synchronous to the master and all other slaves are running asynchronous. If the synchronous slave gets slow or crashs, one of the other asynchronous slave takes over and becomes synchronous. In this way it is guaranteed that at least two nodes store a replica which is up-to-date. This approach is much faster than the synchronous one, as one slave response is received much faster than all and probability is high, that at least one slave will be slow.
But there are also data systems which are completely **asynchronous** and in this way not guaranteed to ensure durability. If the master crashs and cannot be recovered, any write requests processed but not replicated to slaves, are lost. Even this is not widely used, there are cases it makes sense, as the write performance is incredibly fast and the master can still process requests, even if all slaves have failed.

As already spoken about *performance* and *scalability* at the beginning of the chapter, *semi-synchronous* or *asynchronous* replication is for instance a great match for achieving scalability of read-intensive data-systems. As you can distribute all read requests across multiple nodes, just by adding more *slaves* nodes to the data-system, which are able to serve more read requests than a single node could handle. Beside that, getting all the benefits of data locality (like reduced *latency*) comes for "free", as you just need to put the slave nodes geographically close to your users and consumer applications.
Remember it's quite unlikely a setup like this would work reliable with a data-system using *synchronous* replication, as a network issue or single node failure would cause the whole data-system to be unavailable for write operations (as all writes need to be propagated to and acknowledged by every node within the data-system). Thats why most Master data-systems make use of asynchronous replication, for instance MongoDB[63] or MySQL[64].

[63](MDB18j), https://docs.mongodb.com/manual/core/replica-set-sync/
[64](MYS18g), https://dev.mysql.com/doc/refman/8.0/en/replication.html

As you can guess, if the replication is done asynchronously, there will be a certain timeframe (**replication lag**, see Figure 2.12) when the master and it's slaves will be *inconsistent*, as the slaves have not yet processed the most recent write request. If you would query the modified data within this timeframe from one of the slaves, you would probably get an unexpected result. But this inconsistency is just temporary (usually just fractions of seconds depending on the *load* of the data-system), if you would wait a while (without any new changes), the change will be eventually propagated to all slaves. This is also known as *eventual consistency*[65]. Let's have a look at 2 examples when *replication lag* is not just a theoretical issue but really causing your data-system to be inconsistent: **Reading Your Own Writes** and **Monotonic Reads**.

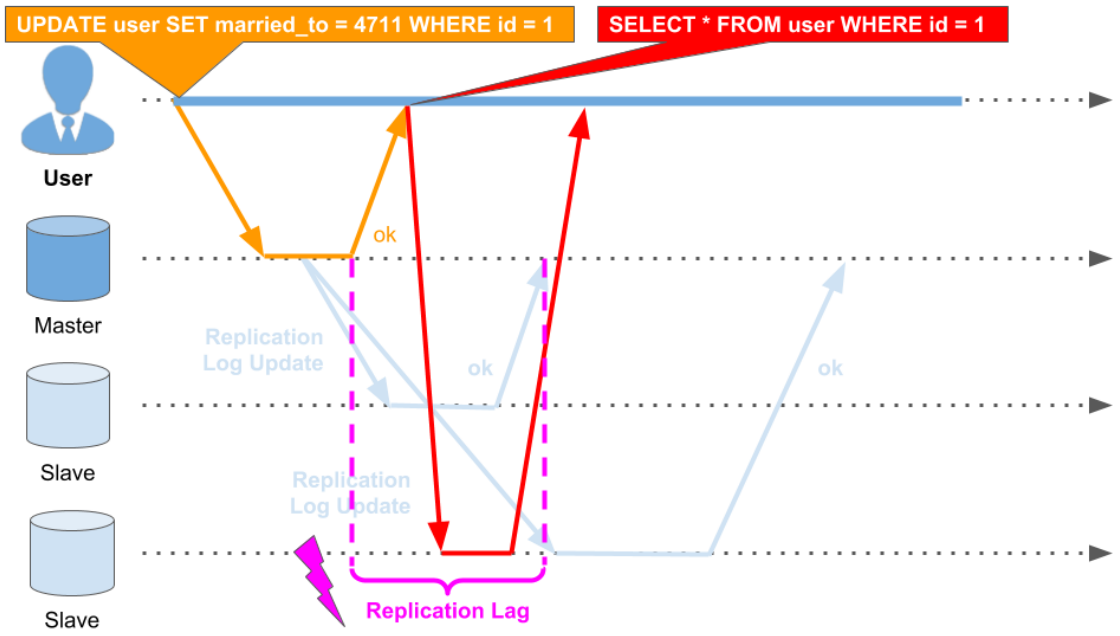

Figure 2.12: Schema - Master Replication - Replication Lag

[65](Ber15)

2.4.1.1.1 Reading Your Own Writes

You probably already know a lot of applications that let you submit or update some data and view it again later on. For instance a post to a Facebook timeline, post to a forum or something similar. In case of a master replication this submit/update request needs to be processed by the master node and propagated to all slave nodes. Using asynchronous replication it is possible that, if you query the data from a slave node (that has not yet received and processed the update), you will not get the desired result. In fact it would look like the data-system has not processed your previous write request at all (see Figure 2.12 on page 48).
To mitigate this issue we need *read-your-writes-consistency*, which ensures that any write requests submitted by a user will directly be seen on any further read requests, guaranteeing a user his write request has been processed successfully. Approaches for achieving *read-your-writes-consistency* could be, e.g.:

- Read data, a user may have modified, from the master, otherwise make use of a slave. Using this approach you need to know, whether the required data has been modififed by a user, without querying it. For instance in case of a Xing/LinkedIn/Facebook it is pretty easy, as those profiles can only be edited by the user itself.
- The previous approach does not work very well for data-systems where a user is able to edit almost anything of a data-system, as this would cause any read request to be executed on the master node (inhibiting scalability at all). Time or replication state could be a valuable criteria to decide wether use of he master or a slave is appropriate. The data-system could consider the time(-stamp) of the last update or processing of replication logs by slave nodes, e.g. if an update request is newer than X seconds or less than the *replication lag* (all slave nodes have not yet sucessfully processed the last write request), read from master node otherwise make use of a slave node.

Anyhow those approaches are just some and also very simple, if you think about the same user using multiple devices: How to synchronize the last update information between both devices? How to make sure both devices connect to the same master within the same data center? What if the application does not make use of users at

all?

2.4.1.1.2 Monotonic Reads

Another typical example, or rather an anomaly caused by *replication lag*, is moving backward in time. This is caused by reading from asynchronous slaves. Let's take a look at Figure 2.13 as an example. Here a user runs the same read query (SELECT) twice on two different slave nodes. The first query returns up-to-date values but the second one outdated values. This happens because the user is reading from 2 different slaves, one with less and one with more replication lag, whereas the last one is still missing the replication update.

Monotonic Read consistency ensures that this kind of divergence does not happen - a user will not read less recent data after reading new data. Approaches for achieving monotonic read guarantee are for instance: ensuring the same user always reads from the same replica (master or slave). Obviously different users can read from different replicas. A simple possibility to determine which replica node to read from. can be achieved by using the user id modulo the number of replica nodes - however, if a node fails, it's inevitable to reroute a users read request.

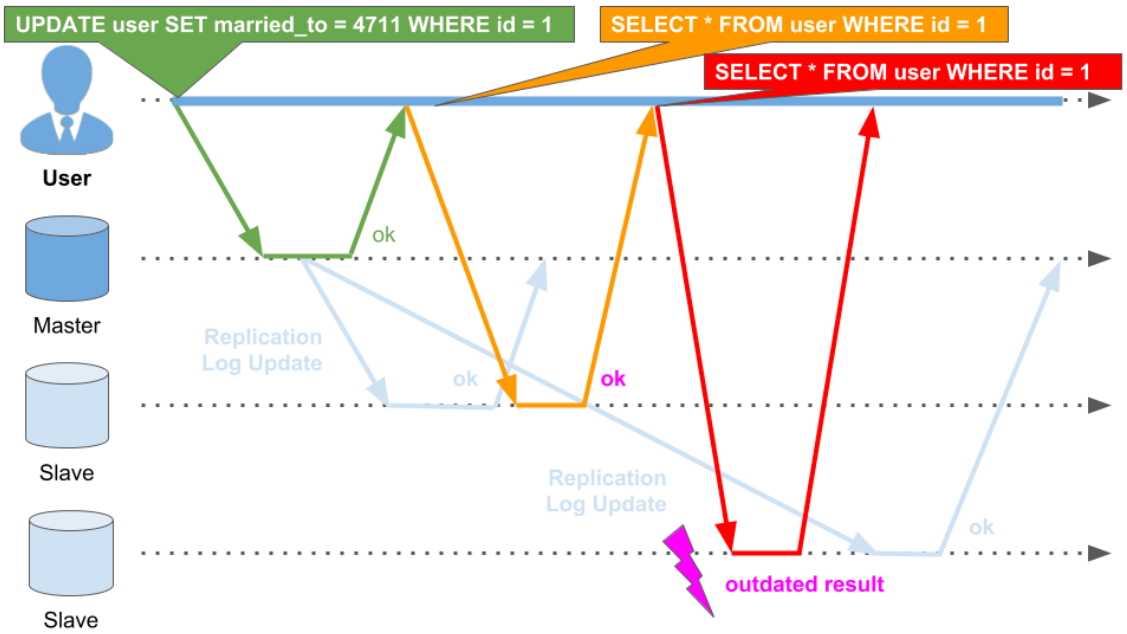

Figure 2.13: Schema - Master Replication - Monotonic Reads

2.4.1.1.3 Adding New Slave Nodes

As we have already spoken about adding new node to a replica set, due to outages of single nodes or because of the need to scale the *load* of read requests horizontally. Let's take a quick look how this is usually achieved.
In a data-system without frequently changing datasets, it is easily done by just copying data from one node (master/slave) to the new one. But as datasets are usually changing, escpecially within the timeframe of executing the dataset copy statement, a simple copy is not appropriate, as at the end of the copy process, the new node will not be up-to-date obviously. One could think about a write lock during the execution of the copy process, but this would strongly violate our previously discussed requirement of high availability. In practise most data-systems make use of following concept:

1. **Create Snapshot:** Take a snapshot of a master or slave node (if not already done, e.g. by backup processes). Usually done by using tools of the data-system itself (e.g. Ops Manager in case of MongoDB[66]) or by using typical ops storage system tools for snapshotting underlying data files (e.g. LVM[67] or Amazon EBS[68] for EC2 instances) or even just `cp`/`rsync`. It is important to notice that the last option would require to stop the data-system, running on the node the copy is taken from, as otherwise it is impossible to get a consistent snapshot. Another disadvantage of those plain copy snapshots: they are unnecessary big as they also include indexes and duplicate underlying storage padding and fragmentation.
2. **Copy Snaphsot:** Copy Snapshot to new replica slave node, for instance automatically by using tools of the data system or in a lot of cases even `rsync` or `cp` is used[69].
3. **Process Replication Log:** The new slave node connects to the master node and processes all dataset changes happened since the snapshot used, was created. This is usually done by using a replication log of the master node

[66](MDB18f), https://docs.opsmanager.mongodb.com/current/
[67](LVM18), http://www.sourceware.org/lvm2/
[68](AMZ18), https://docs.aws.amazon.com/de_de/AWSEC2/latest/UserGuide/EBSSnapshots.html
[69](MYS18c), https://dev.mysql.com/doc/refman/5.7/en/replication-howto-additionalslaves.html

(e.g. *Oplog* collection in case of MongoDB[70] or a binary *relay log* in case of MySQL[71]) or even slave nodes (e.g. in case of MongoDB[72] or MySQL[73]). It is important to mention that the oldest entry within the replication log needs to be less recent than or equal to the creation time of the snapshot used, as the replication logs are usually capped after some time. Otherwise you would have a gap between the snaphshot and the replication log, which would lead to an information loss. A Slave which is too far behind the replication log, and in this way requirering a complete resync, is also known as a *stale* slave. If this happens, you would usually need to start all over again.

4. **Go Live:** As soon as the new slave successfully processed the replication log and is up-to-date, it can start working again like any slave node, processing changes from the master as they happen.

2.4.1.1.4 Outages Of Nodes

As we have already spoken about adding new nodes to a cluster, it is also important to speak about the challenge of handling node outages.

Slave Outage:

As previously discussed, slave nodes make use of repliation logs to stay in-sync with the master. Those replication logs and processing state (usually timestamp, number or position of event within replication log) are usually stored on the slave, for instance in case of MongoDB *OpLog* (stored locally in a collection called `local.oplog.rs`[74]) or in case of MySQL relay logs[75]. If a slave node crashes and gets back up again or recovers from an network issue, it can just start right away where it stopped before the outage occured. It can connect to the master and request all dataset changes that have happende during the time of outage or which are newer than the last entry of the replication log before the outage. As soon as the new slave successfully processed all missing data changes and is up-to-date, it can start

[70](MDB18a), https://docs.mongodb.com/manual/core/replica-set-oplog/
[71](MYS18h), https://dev.mysql.com/doc/refman/5.7/en/slave-logs-relaylog.html
[72](MDB18a), https://docs.mongodb.com/manual/core/replica-set-oplog/
[73](MYS18c), https://dev.mysql.com/doc/refman/5.7/en/replication-howto-additionalslaves.html
[74](MDB18a), https://docs.mongodb.com/manual/core/replica-set-oplog/
[75](MYS18h), https://dev.mysql.com/doc/refman/5.7/en/slave-logs-relaylog.html

working again like before, processing changes from the master as they happen.

Master Outage:
Handling an outage of a master is not that easy, as a *Failover* needs to happen: one of the replica slave nodes needs to be promoted as the new master, all other slave nodes need to consume the replication log from the new master and and clients need to send their write requests to the new master as well. A Failover can be performed manually (e.g. in case of maintenance of the master node) or automatically, which would require following 3 steps:

1. **Determining Master Failure:** As there are a lot of thing that can potentially go wrong (e.g. network, power-supply or disk outages), data-systems as well as many high-available systems make use of timeouts to identify wether a node has failed. All nodes of a data-system usually send messages (also known as *Heartbeat*) between each other and if a node does not respond within a defined timeframe (e.g. 60 seconds) it is declared dead.
   But the timeout approach also has pitfalls, for instance if the outage happened because of heavy network *load*, a failover process will even increase the network load and make the whole situation much worse. This is also a reason some operation teams do not make use of automatic failover.
2. **Evaluating New Master:** There are different approaches to evaluating a new master, most common the new master is chosen by the majority of the remaining replica nodes or by a controller node (e.g. an *Arbiter* node in case of MongoDB[76]). The best candidate is obviously the one containing the most recent updates from the old master.
   This approach has pitfalls, for instance if the cause of the outage was a network split, leaving nodes in two separate networks, not able to communicate with each other. You could end up in a *split-brain* scenario, where there is a master in each network, thinking it is the only master, processing write requests and in this way corrupting all data.
3. **Reconfiguration:** As soon as the new master is elected, all clients need to send their write requests to the new master and all slaves need to receive their

[76](MDB18g), https://docs.mongodb.com/manual/core/replica-set-arbiter/

replication log from the new master as well. If the old master gets back up again, it needs to be ensured it recognizes the new master and will become a slave.

2.4.1.1.5 Types Of Replication Logs

As we have already spoken a lot about replication logs, let's discuss 3 of the most common approaches:

Statement-Based:
Statement-based replication is the most simple approach, the master processes and afterwards adds every write request (e.g. `INSERT`, `UPDATE` or `DELETE` statements in case of a relational data model) to the replication log. Those statements are later on executed by all replica slave nodes (like the master did). This approach has some pitfalls as you can guess:

- How to handle non-deterministic functions within a write request (like `RAND()`, `USER()` or `NOW()` used in an SQL query) or external functions like StoredProcedures, Triggers or user-defined functions?
  The master node would need to replace those non-deterministic functions with deterministic values (like the return value of the called function).
- What if statements depend on other data, like in an `UPDATE ... WHERE ...` statement.
  This requires all statements to be executed within the exact same order, otherwise they will end up in different results.

As there are a lot more potential issues that need to be taken care of, today most data-systems make use of other replication log types. For instance MySQL previously made use of statement-based replication but switched to logical (row-based) replication[77]. But MySQL also still supports a mixed approach[78], which (depending of the statement) dynamically decides whether to use statement- or row-based replication. This serves the purpose of still getting the advantages of statement-based

[77](MYS18f), https://dev.mysql.com/doc/refman/8.0/en/replication-formats.html
[78](MYS18d), https://dev.mysql.com/doc/refman/8.0/en/binary-log-mixed.html

replication, like less data that needs to be written to the replication logs, which increases overall performance, especially restoring data from backups/snapshots is done more quickly.

WAL (Write-Ahead-Log):
Using the write-ahead-log approach, the master logs all IO data changes to a WAL (e.g. (re-)writes of disk blocks, appends to files, etc.), writes the WAL to disk and sends it to all slaves. When a slave processes this log file, it builds an exact same copy of the data structures as found at the master.
Using WAL results in significantly reduced IO (disk writes), because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every statement or data file changed by the transaction. The main disadvantage of this approach is the dependence to the used storage engine, as the WAL is very low-level and defines which byte has changed within which disk block. This makes it impossible to run different versions of a data-system or storage engine on master and slave nodes and also increases complexity of maintenance tasks, especially rolling-upgrades of single nodes within a data-system are not possible any more, making downtimes inevitable.
Write-Ahead-Logs are for instance used by data-systems like ArangoDB[79] or PostgreSQL[80].

Logical Log Replication:
Another approach for replication logs is logical (row-based) replication and unlike WAL it is decoupled from the storage engine. In case of a relational data-system a logical replication log contains records describing changes of a dataset in a row-based way:

- **INSERT:** The log contains one record with all values for each inserted row.
- **UPDATE:** The log contains one record for each updated row as well as all new values and an information to uniquely identify the updated row (e.g. primary key).

[79](ARA18), https://docs.arangodb.com/3.3/Manual/Architecture/WriteAheadLog.html
[80](PSQ18b), https://www.postgresql.org/docs/9.6/static/wal-intro.html

- **DELETE:** The log contains one record for each row to be deleted as well as information to uniquely identify the row to be deleted (e.g. primary key).

As this approach is decoupled from the underlying storage engine (unlike WAL), it is impossible to run different versions of the data-system or storage engine on master and slave nodes and in this way to do rolling-upgrades without the data-system having any downtime.

Let's take a look at an example using MongoDB. At first we switch to our desired collection (`user`) on database `test`, then we insert a new document containing `user_id:1` and afterwords extending the same document by the value `married_to :4711`:

```
$ use test
 switched to db test
$ db.user.insert({user_id:1})
$ db.user.update({user_id:1}, {$set : {married_to:4711}})
```

This would lead to following replication log entries within MongoDB *Oplog* (`local .oplog.rs` collection):

```
$ use local
switched to db local
$ db.oplog.rs.find()
{
  "ts" : { "t" : 1534616696000, "i" : 1 },
  "h" : NumberLong("1342870845645633201"),
  "op" : "i",
  "ns" : "test.user",
  "o" : {
    "_id" : ObjectId("4cb35859543cc1f4f9f7f85d"),
    "user_id" : 1
  }
}
{
  "ts" : { "t" : 1534616699000, "i" : 1 },
```

```
  "h" : NumberLong("1233487572903545434"),
  "op" : "u",
  "ns" : "test.user",
  "o2" : {
    "_id" : ObjectId("4cb35859543cc1f4f9f7f85d")
  },
    "o" : {
      "$set" : { "married_to" : 4711 }
    }
}
```

Whereas:

- `ts` is the timestamp, when the operation occured.
- `h` is a unique ID of the operation. Each operation has a different value within this field.
- `op` indicates the operation to be done. (i = insert, u = update, d = delete)
- `ns` defines the database and collection affected by this operation (in our case database `test` and collection `user`)
- `o/o2` the operation to do or in case of an insert (`o2`) the document affected by the operation (using the docuent id).

If we would update multiple rows at once, MongoDB would create an entry for each row affected within the Oplog replication log.

2.4.1.2 Multi-Master Replication

As we have discussed Master-based replication within the last chapter we have already noticed the biggest weakness: we have only one master. All write requests must go through it (strongly limiting write *scalability*) and if the master is suffering a network outage, we are not able to process write requests anyhow. A multi-master replication mitigates those issues, as it allows multiple (master) nodes to accept write requests. The basic replication idea of *master* and *slave* nodes stays the same: each master processing write requests needs to propagate those changes to all slaves as well as each master acts as a slave to the other master nodes. The multi-master replication is also known as *Multi-Master*, *Active/Active* or *Master/Master* replication.

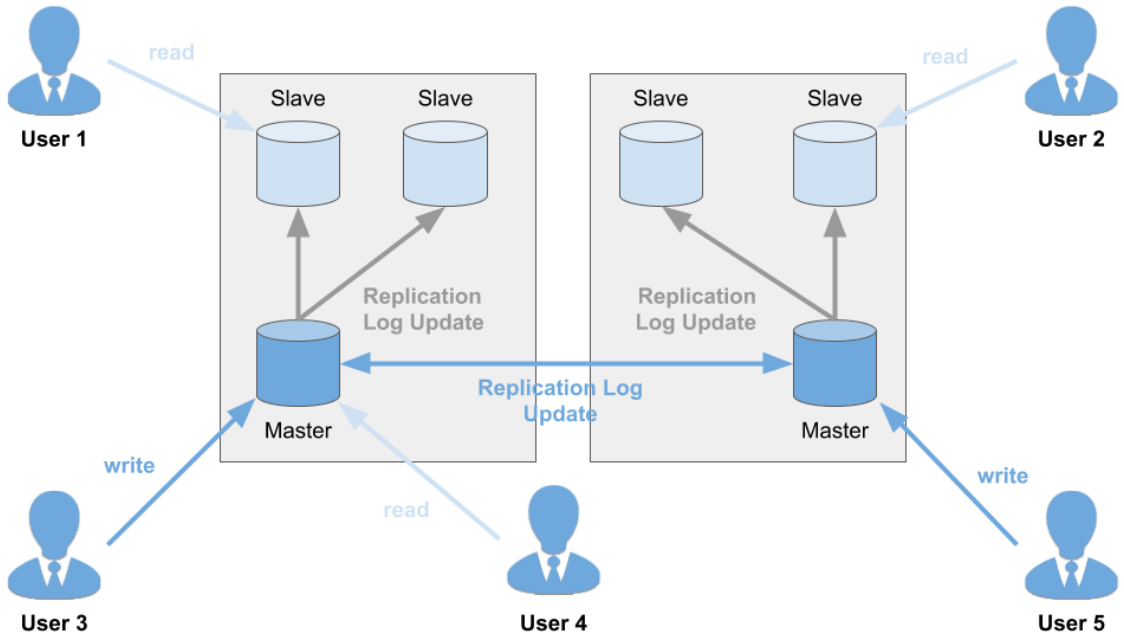

Figure 2.14: Schema - Multi-Master Replication - Example

Figure 2.14 illustrates an example of multi-master replication setup. As you can see there are multiple master, which are able to process read requests as well as multiple slave nodes which serve the purpose of processing read requests. Using this approach approach we are able to achieve a better write *performance* (*horizontal scalability*) as multiple nodes are able serve read requests. In the same way the whole data-system is way more *fault-tolerant* as if a single master node suffers an outage there are still

other master nodes available for serving write requests (while another slave node is already in the process of being elected, replicating and replacing the old master node).
Implementations of multi-master replication can be found at CouchDB[81], PostgreSQL (using BDR[82]) or MySQL (using Tungsten Replicator[83] or MYSQL native multi-source implementation, which is very limited and does not even provide any conflict detection or resolution[84]).
As also illustrated in Figure 2.14 on page 58 a multi-master is a perfect match for setting up data-systems on multiple data-centers and getting data for read as well as write requests geographically close to users and consumer applications. But there is also a big downside using multi-master setups, as multiple nodes are able to process write requests we need to think about conflicting write requests involving the same dataset to be changed within the same time. We will discuss this within the next chapter.

2.4.1.2.1 Conflicts

Write conflicts within multi-master setups happen if the same dataset is being edited in parallel using multiple master nodes. Imagine editing the same line of a document within Google Docs simultanously: user 1 is working on a master node based in Frankfurt and user 2 is working on a master node in Berlin. Each write gets processed successfully on each master but later at the asynchronous replication between the two master nodes a conflict will arise. This won't happen if you are using a master replication. One could think about making the write requests and conflict detection synchronous, but this would be foolish as you would abandon the main benefit of multi-master replication: *horizontal* write *scalability*, as multiple nodes are able to process write requests. But how to handle write conflicts?
Well the best approach would be to avoid those conflicts, for instance this could be done by the application using the data-system. Thinking about our Google Docs example: if the application would ensure that both users are pushing their write requests to the same master, write conflicts will no longer occur.

[81](CDB18), http://docs.couchdb.org/en/2.1.2/replication/intro.html
[82](PSQ18a), https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/
[83](TNG18), https://github.com/continuent/tungsten-replicator
[84](MYS18i), https://dev.mysql.com/doc/refman/5.7/en/replication-multi-source-overview.html

But what if the application is not able to prevent those write conflicts? Let's dicsuss some common approaches for conflict resolution:

- **LWW:** as each operation has a unique ID, timestamp or both one could think about only accepting the most recent operation (Last-Write-Wins). It is important to notice that this approach is highly vulnerable to **data loss**.
- **Merge:** merge values of multiple updates together (e.g. alphabetically or in order of appearance).
- **Application Managed:** persist the conflict in a way, that it preserves all information without loss and let the application, using the data-system, or rather the user of the application take care about it later on (e.g. implemented by SVN[85] or Git[86])

2.4.1.2.2 Topologies

If you think again about the example at the beginning of this chapter (Figure 2.14 on page 58) the replication topology of all master nodes is really straight forward as there are only two master nodes, which need to talk with each other. But what about setups of far more master nodes? Figure 2.15 on page 61 illustrates the most common approaches:

- **Star Topology:** one designated node acts as the single point of communication and forwards all write requests to all nodes.
- **Circle Topology:** every node receives write operations from one node and forwards it to another node, adding its own write operations at the same time.
- **All-To-All Topology:** every node sends it write operations to all other nodes.

One issue regarding *circular topology* and *star topology* is, if just one node fails (in case of the star topology the center node), the whole process of master node replication is interrupted. This would not happen using *all-to-all topology*. But *all-to-all topology* is also vulnerable to network latencies and issues, it is not very

[85](SVN18), https://subversion.apache.org/
[86](GIT18), https://git-scm.com/

unlikely that some write requests will overtake others in time, which requires a lot more effort in terms of conflict handling.

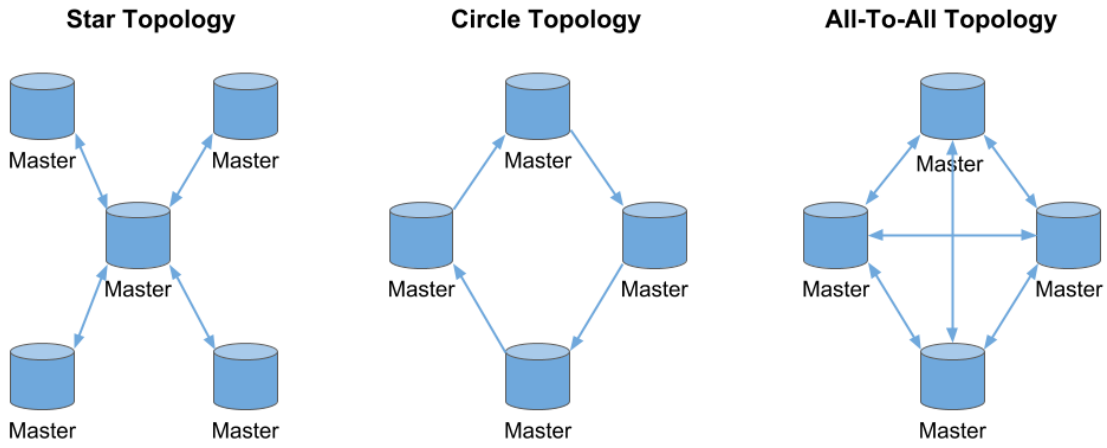

Figure 2.15: Schema - Multi-Master Replication - Topologies

2.4.1.3 Masterless Replication

Within the last wo chapters we have discussed (Single-/Multi-) Master-based replication, in which all write requests of users or consumer applications must go trough one or multiple dedicated master nodes. The master node will take care about propagating all changes regarding the dataset to all slave nodes. Let's take a look at another approach without any dedicated master node.
The basic idea of *masterless replication* (also known as *masterless* replication), like illustrated in Figure 2.16, is that there is no special kind of nodes - any node will not only accept read but also write requests from clients. This approach also works very well on multiple datacenter, with respect to reasonable quorum configuration, but we will talk about that later.

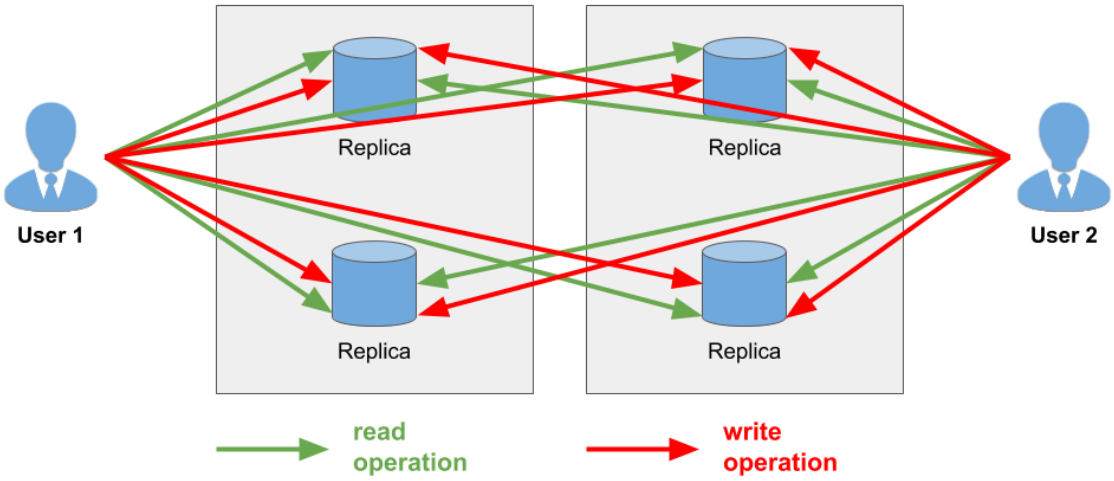

Figure 2.16: Schema - Masterless Replication - Example

Using masterless replication, clients can actual be users/consumer applications but also a *controller node* in between user/consumer applications and the data-system, which acts as a proxy and takes care of routing requests.
For instance Cassandra makes use of an approach called *coordinator* as a controller node[87]. Read or write requests of clients can be sent to any node within a cluster. The node receiving a request serves as the coordinator for the particular client operation. The coordinator determines which nodes within the cluster should process

[87](CAS18g), https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archIntro.html

the request based on how the cluster is configured and which *quorum* is used. This approach enables a better write performance as write requests can be parallelized as well as a better overall performance as failing and slow replica nodes can be tolerated without causing *latency* and/or *failover* processes.
Other examples of data-systems using masterless replication are for instance: VoldemortDB[88], Riak[89] or DynamoDB[90].
Data-systems using the masterless replication approach are usually able to achieve a better write *performance* and *horizontal scalability* as any node is able to serve write requests. In the same way the whole data-system is way more robust and *fault-tolerant* than master-based approaches, as there is no *single-point-of-failure* (master nodes), as multiple nodes can fail without causing the data-system being unable to serve write requests. A *failover* process is not required at all, as there is no master to failover. But this comes at a price:

- The client or rather the application using the data-system needs to handle a lot of tasks a master node would usually take care of (quorum, consistency and conflict resolution)
- Initial setup costs, as you need at least 3 nodes for a quorum to be able to ensure consistency and availability.
- Application code complexity, as masterless data-systems usually dont provide sophisticated querying engines or basic features like referential integrity, ACID[91], which will force you to take care about that within your application code.
- Almost any masterless data-system is a key/value store providing all advantages but also disadvantages and limitations of a key/value data system.

[88](VOL18), http://www.project-voldemort.com/voldemort/design.html
[89](RIA18d), http://basho.com/products/riak-kv/resiliency/
[90](DYN18c), https://aws.amazon.com/de/dynamodb/
[91](TH83), ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of failures (network issues, hardware failures etc.

2.4.1.3.1 Quorums

Let's take a look at a simple example of handling write requests on a *masterless* data-system (Figure 2.17). As you can see a user executes a read request, which gets distributed to all 3 replica nodes and executed in parallel. Two of three replica nodes successfully process the write requests, but one node fails (e.g. because of a failure or network outage). The overall write request returns success as at least two of the three replica nodes returne with a success message. Now what about the faulty node comes back online and starts serving read requests again, which may result in *stale* (outdated) results? Let's take a look at Figure 2.18 on page 66. As you can see the read request of a user is also distribted to all replica nodes and executed in parallel. The user gets a result set of two up-to-date and one updated result, being able to determine the most recent value (*last-write-wins*).

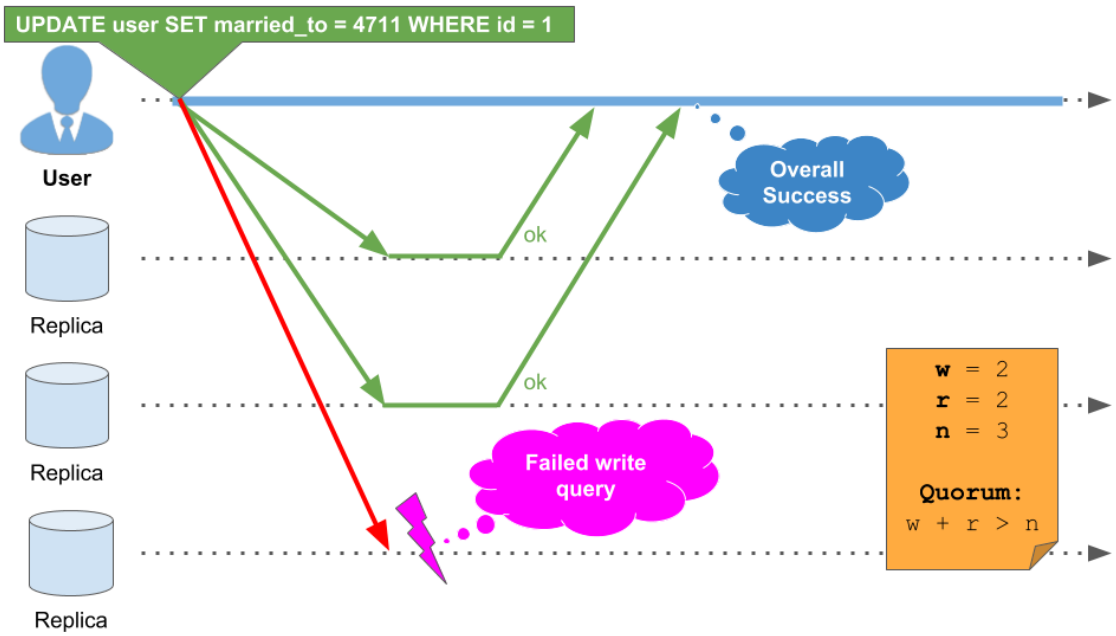

Figure 2.17: Schema - Masterless Replication - Quorum Write Example

But how to take care about outdated nodes to become eventually consistent? Let's briefly discuss 3 common approaches:

- **Read Repair:** If a user application or controller node executes a read requests

and recognizes a replica node with a stale value (see Figure 2.18 on page 66), the user application or controller sends it the most recent value afterwards. For instance used by Cassandra[92], RIAK[93] and VoldemortDB[94].

- **Anti-Entropy Repair:** Some data-systems are running background processes or provide tools (e.g. `nodetool repair` in case of Casssandra[95]) which run in background constantly looking for differences between different replica nodes and copying data from one node to another to keep everything up-to-date. For instance used by Cassandra[96] and RIAK[97] but not supported by VoldemortDB.
- **Hinted Handoff:** If a node is unable to process a particular write request, the user application or coordinator node (which executed the write request) preserves the data to be written as a set of hints. As soon as the faulty node comes back online, the user application or coordinator triggers a repair process by handing off hints to the faulty node to catch up with the missed writes. For instance used by Cassandra[98] and RIAK[99] but not supported by VoldemortDB.

It is important to notice that those approaches do not work like the previously discussed *replication log*. Any of those repair approaches don't run in a particular order (unlike *replication logs*) and there is no guarantee in terms of time when a replica node will be up-to-date again, just that it will be eventual consistent. Usually the delay is much bigger than data-systems using *master-based replication*. But in terms of consistency this is totally fine as long as a reasonable quorum is used. So what is a (reasonable) quorum?
A quorum is the minimum number of votes that a read/write request to a distributed data-system has to obtain in order to be sure the requests is successful (and the result consistent). For instance, if a data-system has `n` replica nodes (`n` **is not** the number of all nodes available within the whole cluster, just the number of nodes sharing

[92](CAS18c), https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesTOC.html
[93](RIA18a), http://basho.com/posts/technical/why-riak-just-works/
[94](VOL18), http://www.project-voldemort.com/voldemort/design.html
[95](CAS18b), https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsRepair.html
[96](CAS18c), https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesTOC.html
[97](RIA18a), http://basho.com/posts/technical/why-riak-just-works/
[98](CAS18c), https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesTOC.html
[99](RIA18a), http://basho.com/posts/technical/why-riak-just-works/

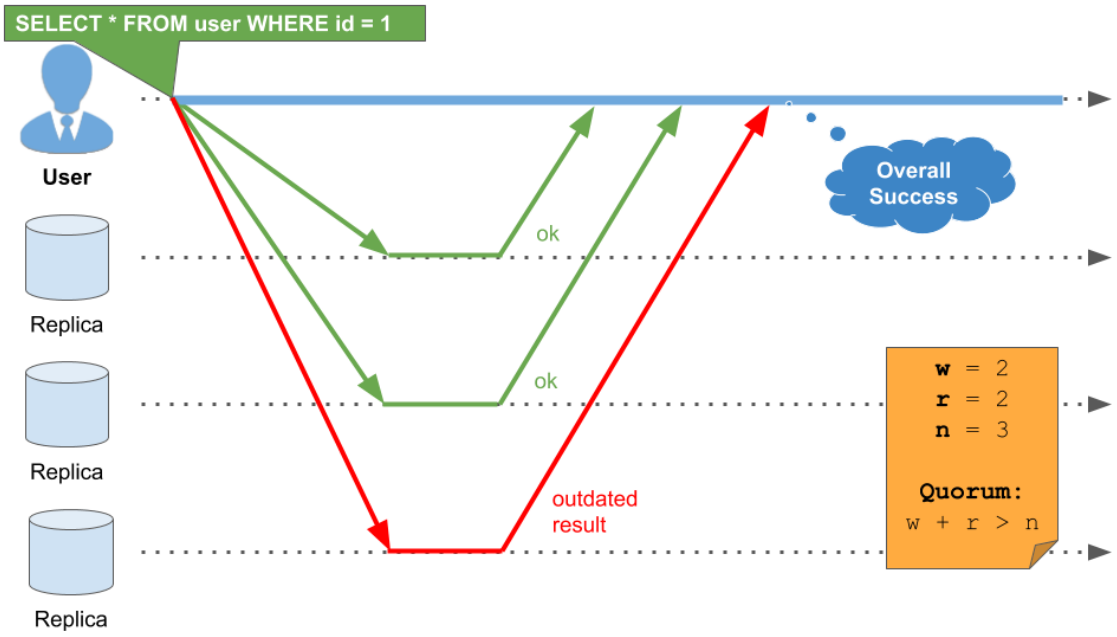

Figure 2.18: Schema - Masterless Replication - Quorum Read Example

the same replica), every read and write request must be processed and confirmed by at least `r` nodes (read) or acknowledged by at least `w` nodes (write). A reasonable quorum is `r + w > n`, as we can be sure to get an up-to-date result (e.g. in case of a read request) as at least one of the `r` replica nodes will have the most-recent value for our request. Read and write requests which adhere to those rules are called *quorum reads* and *quorum writes*.

The quorum parameters are configurable, based on your cluster size, a read/write performance and fault-tolerance you want to achieve. In terms of read performance it is reasonable to choose a smaller `r`, but resulting in a larger `w` which causes slower writes. Vice versa a smaller `w` will speed up writes but result in a larger `r`, which causes slower reads. The number of node failures a data-system can tolerate can easily be calculated by:

- `n - r = number` of nodes tolerated to be unavailable for read requests
- `n - w = number` of nodes tolerated to be unavailable for write requests

For instance a data-system of 5 nodes (`n = 5`, `r = 3` and `w = 3`) is able to tolerate 2 unavailable nodes.

2.4.1.3.2 Limitations of Quorums

As we discussed quorums and their advantages so for, let's have a look at some major limitations. Most quorum based data-systems allow to weaken up the quorum $\mathbf{r} + \mathbf{w} \leq \mathbf{n}$, for instance for the purpose of reducing *latency*, increasing *high availablity*, ensuring durability or just distributing the whole data-system among different datacenters, which are more closely to a user or consumer applications. This is totally fine, as long as the nodes used for read and write requests overlap at least in on node. In this way its more likely that a data-system is still able to process read and write requests, even in case of failure of multiple nodes, network or datacenter outages (unless the number of replica nodes does not get smaller than `r` and `w`). But even using the $\mathbf{r} + \mathbf{w} > \mathbf{n}$ quorum, there are some pitfalls resulting into reading stale values:

- **Concurrent Writes:** If two write requests are executed concurrently, it is not clear which one happened first, as both are executed from different controller nodes or applications. Both requests need to be merged, for instance based on a timestamp (*last-write-wins*), which is highly vulnerable to clock skew, causing older values to overwrite more recent ones.
  For instance Cassandra is based on *last-write-wins*[100] whereas Riak requires the admin to choose whether to make use of *last-write-wins* or handle write conflicts within the application using the data-system[101].
- **Concurrent Reads And Writes:** If read and write requests (regarding the same value) happen at the same time it is unclear whether the read request returns the stale or the new values. As the write request may succeeded only on some nodes during execution of the read query, the new value might be underrepresented, causing the old value to win the quorum.
- **Node Failure:** If a node previously processed a new value, crashed and comes back online and is restored from a node with the old value, the quorum might be violated as the number of nodes storing the new value might be < `w`.
- **Sloppy Quorum and Hinted Handoff:** There are cases like network or datacenter outages causing some user or consumer applications being cut off

[100](CAS18a), https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlWriteUpdate.html
[101](RIA18c), http://docs.basho.com/riak/kv/2.2.3/developing/usage/conflict-resolution/

from some nodes of a data-system (while the unreachable nodes are still online). In this case it is possible that some user or consumer applications won't be able to achieve a quorum. If the data-systems contains more than `n` nodes, it needs to make a crucial decision here, whether to ignore all requests that cannot achieve a quorum or still accept write requests and just write them to some nodes outside of `n` to ensure *write availability* and *durability*. A *sloppy quorum* still requires `r` and `w` nodes, but those do not need to be one of the original `n` nodes. As soon as the network or datacenter outage is fixed, all writes processed by nodes outside of `n` are sent to the appropriate nodes inside of `n` (*Hinted Handoff*). Using this approach it is no longer guaranteed a data-system will provide the most recent value as read and write requests may not overlap on `r` and `w` nodes. This could also be mitigated by using versioning of values, like *vector clocks* (e.g. used by DynamoDB[102]).
For instance sloppy quorums are enabled by default within Riak[103] and disabled by default within Cassandra[104].

2.4.1.3.3 Gossip

to-be-added

[102](DYN18e), https://de.wikipedia.org/wiki/Amazon_Dynamo
[103](RIA18e), http://docs.basho.com/riak/kv/2.2.3/learn/glossary/
[104](CAS18f), https://www.datastax.com/dev/blog/understanding-hinted-handoff

2.4.2 Partitioning

Partitioning is the process of continuously dividing data into subsets and distributing it to several nodes within a data-system. Usually each record or document within a partitioned data-system is distributed and directly assigned to certain partition. We will discuss approaches on how to *distribute* data and related topics like *routing* of certain data and *rebalancing* of nodes later within this chapter. Partitioning serves the purpose of, e.g.:

- **Scalability and Performance:** Distributing data to multiple nodes, for instance increases read/write performance and throughput as read/write queries can be distributed to multiple nodes and handled concurrently. In this way it is possible parallelize IO (disk), computing power (CPU) as well as scale the memory usage needed to run a certain operation on a part of the dataset.
- **Low Latency:** Using partitioning it is possible to place data close to where it is used (user or consumer applications).
- **Availability:** Even if some nodes fail, only parts of the data are offline.

To avoid confusion on the term *partition* or *partitioning*, let's list some other terms, you might have heard and which are frequently used synonymously:

- *shards/sharding* - (e.g. MongoDB[105], ElasticSearch[106] or RethinkDB[107])
- *Vnodes/Virtual Nodes* - (e.g. Riak[108] or Cassandra[109])
- *region* - (e.g. HBase[110])
- *tablet* - (e.g. BigTable[111])
- *vBucket* - (e.g. Couchbase[112])

[105](MDB18k), https://docs.mongodb.com/manual/sharding/
[106](ELA18), https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html
[107](RDB18d), https://www.rethinkdb.com/docs/architecture/
[108](RIA18g), http://docs.basho.com/riak/kv/2.2.3/learn/concepts/vnodes/
[109](CAS18d), https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archData DistributeVnodesUsing.html
[110](HBA18), http://hbase.apache.org/0.94/book/regions.arch.html
[111](FC06), "Bigtable: A Distributed Storage System for Structured Data", Google Inc.
[112](CBP18), https://developer.couchbase.com/documentation/server/3.x/admin/Concepts/concept-vBucket.html

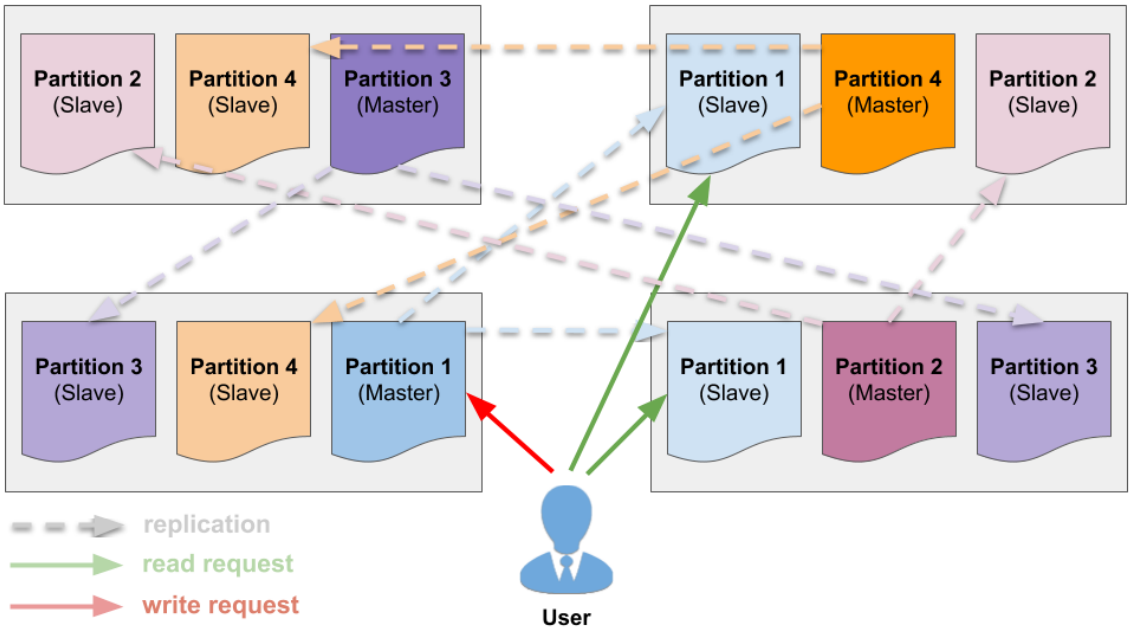

Figure 2.19: Schema - Partitioning (using Replication)

Partitioning and Replication are usually used together, especially when building data-intensive applications, as a dataset is to big to be stored on a single server or replica, and benefits of replication are required (e.g. *redundancy*, *fault-tolerance* or high read/write *throughput*). This can be achieved by storing partitions of a data set on multiple replica nodes. In case of *single-master* replication a usual approach looks like visualized in Figure 2.19. Each partition is replicated to multiple nodes, it has a master node as well as 2 slave nodes storing the same data. Each single node can be a master for a certain partition but also a slave for another partition. Using this approach, datasets and *load* can be easily distributed among multiple nodes as well as an outage of a node won't cause partitions to be unavailable for requests.

As **horizontal** and **vertical** partitioning are mixed up sometimes, it is important to notice: when we speak about partitioning within this lecture, we mean *horizontal partioning*. *Vertical partitioning* is an approach of traditional relational databases, usually done by splitting datasets into multiple entities (e.g. tables or databases) and using references (e.g. to achieve *normalization*).

2.4.2.1 Partitioning Of Key-Value Data

Partitioning is done with the purpose of distributing a dataset, but more important: distribute related *load* (read/write queries) evenly among several nodes of a data-system. This requires the way of determining the partition of a certain row or document to be choosen wisely, as it directly affecs the performance of a data-system. An improper choosen distribution key may cause some nodes to be idle and/or empty and a single node to be the processing bottleneck and hitting its space limitations as all read/write requests end up on that single node. Whereas an appropriate distribution key will distribute the data evenly and enable the data-system to (theoretically) scale linearily in terms of space utilization and request throughput. Let's discuss some approaches for partitioning:

- **Key Range Partitioning:** Derive a partition by determining whether a key is inside a certain value range. Dicussed within chapter 2.4.2.1.1.
- **Partitioning By Hash Value Of A Key:** Derive a partition by a certain hash of a given key to achieve a more even data distribution. Dicussed within chapter 2.4.2.1.2.
- **Partitioning By List:** Every partition to be used has an assigned list of values. A related partition is derived from the input dataset by checking whether it contains one of those values. For instance all rows containing *iPhone*, *Samsung Galaxy* and *HTC One* within a column `device_type` are assigned to partition *Smartphone*.
  As no data-intensive system makes use of partitioning by list (as it is very improper to provide even data distribution), we wont discuss this approach in detail.
- **Round-Robin Partitioning:** A very simple approach, which ensures even data distribution. For instance assignment to a partition can be achieved by `n` modulo `p` (`n` = number of inccoming data records, `p` = number of partitions).
  As no data-intensive system makes use of round-robin partitioning (as for instance the direct access to an individual data record or subset usually requires accessing the whole dataset), we wont discuss this approach in detail.

2.4.2.1.1 Key Range Partitioning

Key Range partitioning is done by defining continues ranges of keys and assigning each of them to a certain partition. If you are aware of the boundaries of each key range, you can easily derive a partition belonging to a certain data record (and in this way node of a data-system) just by using the key of the record. This approach can be compared with an encyclopedia (compare Figure 2.20), which is partitioned into books, of which everyone stores a certain range of articles partitioned by the first letters of the name of the article. For instance the article *"Arsenal F.C."*[113] will be found in partition *863 "ARS" - "ART"*.

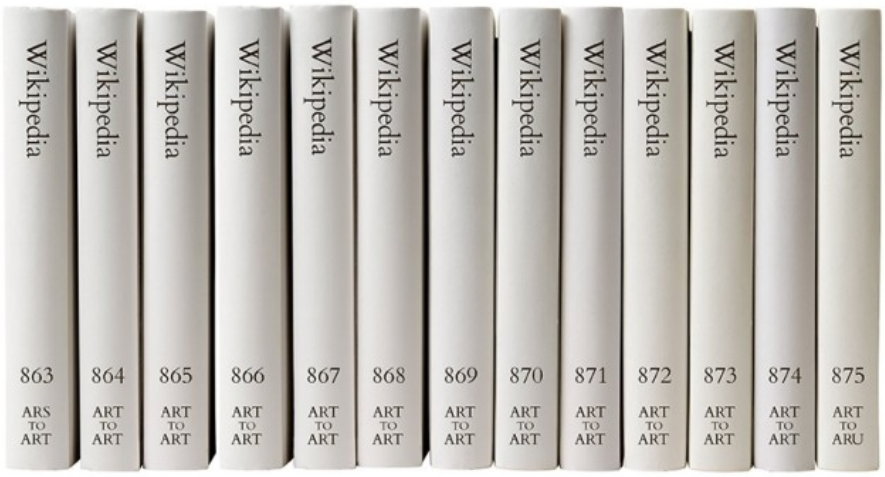

Figure 2.20: Image - Wikipedia as Books

As you can guess a partitioning only using the first letter (*A, B, C, ... Y, Z*) would lead to an unevenly data distribution. Therefore partition keys or rather boundaries need to be choosen wisely and suitable to a certain dataset, e.g. by a data-system admin or automatically. Another advantage of key range partitioning is, that some operations on a dataset are very easy, especially range scans (queries). If you partition a dataset by time (e.g. web server log files by day) it is very easy to fetch all log files related to a particular week. Disadvantages of key range partitioning are:

- **Datasets Are Changing:** A key range partitioning which was suitable in

[113] https://en.wikipedia.org/wiki/Arsenal_F.C.

the past might not be appropriate in the future. Expensive rebalancing or even repartitioning might be needed somewhen. For instance web server log files partitioned per ranges of the URL (`/products/[A-B]`, `/products/[C-D]`... `/products/[Y-Z]`) maybe improper in the future, as some products will have heavier traffic than others over time (*load skew*).

- **Hotspots:** Keys that seem very appropriate in terms of even distribution at first sight, for instance partitioning of web server logfiles over time (by using timestamp of data record), create hotspots within the data system. As all write requests end up on the same partition (e.g. today), a single partition (and node(-s)) will underly heavy load whereas other partitions or rather nodes are idle (*load skew*).
- **Query Performance:** As you do not know the size of a partition beforehand, query performance is impredictable as well as partition pruning and partition-wise joins are more complex and less efficient.

Nevertheless for instance RethinkDB makes use of *key range partitioning*[114]. RethinkDB calls this approach *range sharding* and its applied on the table's primary key. If a table is configured to make use of a certain number of partitions (*shards*), RethinkDB automatically examines the statistics for the table and finds the optimal set of *split points* to distribute the table's data evenly among all partitions. But this comes (as discussed) at a price: every time you need to add a shard to the data-system or the dataset changes significantly over time in a way that the primary key distribution is not even anymore, you need to *rebalance*[115] all shards of a table.

[114](RDB18b), https://www.rethinkdb.com/docs/architecture/
[115](RDB18c), https://www.rethinkdb.com/api/javascript/rebalance/

2.4.2.1.2 Hash Partitioning

Hash partitioning is used to spread data efficiently and evenly among several certain partitions. This is achieved by splitting data in a randomized way rather than by using information provided within the dataset (e.g. IDs) or derived by arbitrary factors (e.g. time of data receival). The hash value itself is derived by a hash function (on a certain key of a data record) and is used to determine the partition a data records should be saved on.

Hash Function is a function which takes input data of arbitrary size and usually provides an output of fixed size. The output of a hash function is called *hash*, *hash value* or *digest*. A hash function needs to be *deterministic* and *uniform*. Common use cases for hash functions are *cryptography*, *checksums* and *partitioning*.

Hash functions are commonly used for partitioning, as they are:

- **deterministic**[116] - as we need to be able to find records to be saved later on and
- **uniform**[117] - as we want to distribute the data as evenly as possible among the set of avaiable partitions and nodes, even if the inputs of the hash function (e.g. data record key) are very similar.

Unlike cryptography, partitioning makes use of hash functions that are not cryptographilly strong (as this is not needed) but fast and less CPU consuming. Examples of commonly used hash function for distributed data-systems are:

- **MD5** - For instance used and supported by MySQL[118] and Cassandra[119].
- **MurmurHash** - For instance supported by Cassandra[120].
- **SHA1** - For instance used and supported by Riak[121].

[116] A deterministic hash function will provide the same output, if it is executed several times for the same input.
[117] A uniform hash function will map the inputs as evenly as possible to the available output range
[118] (MYS18b), https://dev.mysql.com/doc/refman/5.7/en/partitioning-key.html
[119] (CAS18h), https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html
[120] (CAS18h), https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html
[121] (RIA18b), http://basho.com/posts/technical/why-riak-just-works/

- **CRC32** - For instance used and supported by Couchbase[122].

As an example for *determinism* and *uniformity* let's take a quick look at MD5 and how the hash value changes when a single character is added to the input value, when just a single character of the input value is changed or the same input value is hashed twice (see bash output in Code Snippet 2.7).

```
marcel$ md5 -s abc
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
// add a single character ("d")
marcel$ md5 -s abcd
MD5 ("abcd") = e2fc714c4727ee9395f324cd2e7f331f
// change a single character ("d" to "e")
marcel$ md5 -s abce
MD5 ("abce") = b9c4fe92c2a30ef69833ac8f53eebcec
// hash again with same input value
marcel$ md5 -s abce
MD5 ("abce") = b9c4fe92c2a30ef69833ac8f53eebcec
```

Code Snippet 2.7: Bash Output - *MD5 Hash For Several Input Values*

As you can see *uniformity* is ensured, as with just the change of a single character the resulting hash is completely different. *Determinism* is also fulfilled as executing the function with the same input value produces the same hash value.
Using this hash functions, distributed data-systems are able to distribute records among partitions, this is usually done by two common approaches:

Hash Modulo:
Take the calculated hash value V and calculate V `modulo` N (number of partitions). This allows the the data-system to easily distribute and receive records to and from a given number of partitions. Unfortunately this approach has a major disadvantage in terms of *scalability* especially *operability*. As data-systems usually grow significantly

[122](CBC18), https://docs.couchbase.com/server/5.5/understanding-couchbase/buckets-memory-and-storage/vbuckets.html

over time (which requires adding additional nodes (N) to a data-system to cope with the increasing data volume) *hash modulo* causes a lot of trouble, as an increased (as well as decreased) N results in different partition assignments for a lot of records (depending on size of N), which will require to shuffle and reassign already saved data again among all partitions. That's the main reason most data-systems we talk about within this lecture (unlike traditional databases) do not make use of *hash modulo* but *consistent hashing*. Nevertheless for instance elasticsearch makes use of it, a partition (called *shard*) is derived by[123]:

```
shard = hash(routing) % number_of_primary_shards
```

The number of shards for an *index*[124] can increased (by `_split`[125]) or decreased (by `_shrink`[126]) some time after creation but this is not a trivial task and will usually require recreating the same or even creating a new *index*.

Consistent Hashing:
Assign a range of hashes to every partition (and in this way node). Every record will be stored and read from the partition in charge for a given range of hash values. Imagine *consistent hashing* as a ring of keys (see Figure 2.21 on page 77). Each node (N_i) is in charge for serving all hash values v in between i and the position j of its clockwise predecessor N_j :

$$\mathbf{j < v \leq i}$$

Using *consistent hashing* even distribution and avoiding of hotspots is also ensured by using hash values of keys. Unlike *hash modulo* this approach is not vulnerable for a changing cluster size (number of nodes N), as for instance in case of removing a node (see Figure 2.21 on page 77, node 7), the higher neighbour (node 8) takes over and all other data does not need to be touched or reassigned. Node N_8 is no

[123](ESR18), https://www.elastic.co/guide/en/elasticsearch/guide/current/routing-value.html
[124]index - An index is a collection of documents. Compared to a relational database like MySQL: elasticsearch can contain multiple indices (databases), which in turn contain multiple types (tables). These types hold multiple documents (rows), and each document has properties (columns).
[125](ESS18b), https://www.elastic.co/guide/en/elasticsearch/reference/6.4/indices-split-index.html
[126](ESS18a), https://www.elastic.co/guide/en/elasticsearch/reference/6.4/indices-shrink-index.html

longer in charge of hash values v between $\mathbf{7 < v \leq 8}$ but $\mathbf{6 < v \leq 8}$.

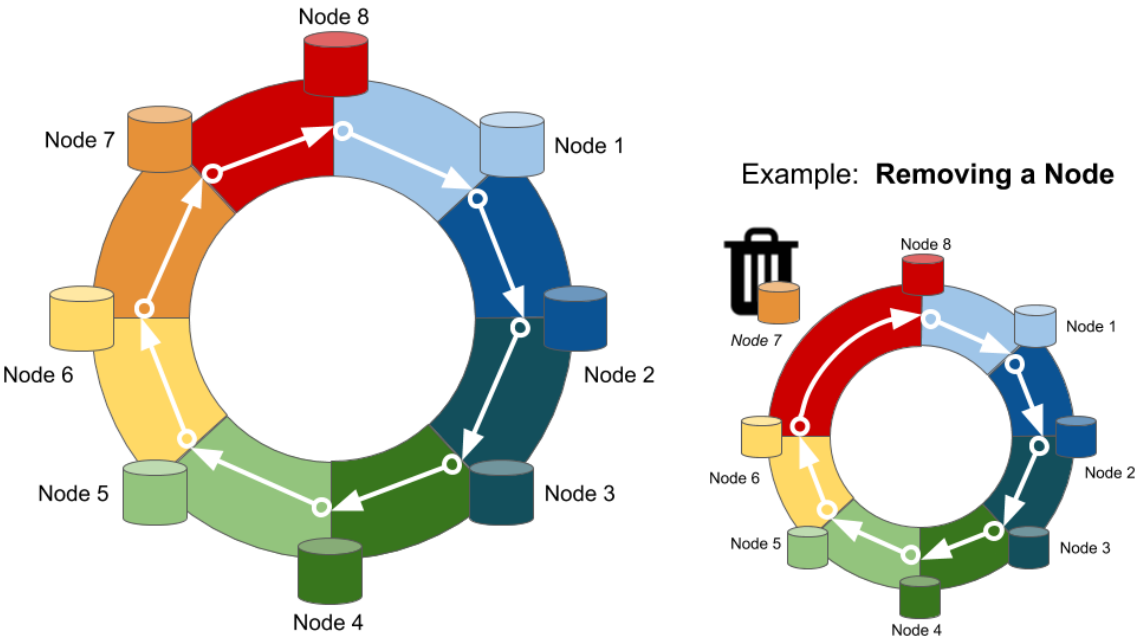

Figure 2.21: Schema - Consistent Hashing

In case of a node is added to the data-system, it just takes values from another node and all other partitions and nodes do not need to be touched. As you can see after every addition or removal of a node, only c/N keys need to be redistributed (where c is the count of hash values and N is the number of nodes within the data-system). Because of that *cosistent hashing* has become a de facto standard for all modern highly distributed data-systems (e.g. Cassandra[127], Riak[128], VoldemortDB[129] or DynamoDB[130]).

It is important to notice a disadvantage of *consistent hashing*: range queries. As keys are now randomly distributed among all partitions (to achieve even distribution) range queries will perform less efficient compared to *key range partitioning* as the query needs to be sent and processed by all partitions (and therefore nodes).

[127](CAS18i), https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeHashing.html
[128](RIA18h), http://docs.basho.com/riak/kv/2.2.3/learn/glossary/
[129](VDB18), http://www.project-voldemort.com/voldemort/design.html
[130](DYN18d), https://cloudacademy.com/blog/dynamodb-replication-and-partitioning-part-4/

For instance MongoDB provides *key range partitioning* (called *ranged sharding*[131]) as well as *hash partitioning* (called *hashed sharding*[132]) to efficently serve both use cases, but you still need to choose wisely and make a trade-off which one to use depending on your case.

[131](MDB18i), https://docs.mongodb.com/manual/core/ranged-sharding/
[132](MDB18h), https://docs.mongodb.com/manual/core/hashed-sharding/

2.4.2.2 Partitioning Of Secondary Indices

We previously discussed partitioning mostly relying on key-value data, in which a single key is not only used for partitioning but also as the primary key of a dataset. If you only need to access the data by the primary key, determination of the belonging partition can easily be achieved by using the hash function as described within the last chapter. But what if you need to access the data by attributes within a record or even do a range scan on those attributes? Let's take our previous discussed Example of Facebook profiles as an example (Figure 2.6 on page 26). If you want to receive a single profile and know the `user_id`, a belonging partition can easily be determined, but if you want to receive all profiles of people living in "*Palo Alto*" *secondary indices* are a good choice to efficiently access the required data by attributes that are other than the primary key.
You probably already know the idea of *secondary indices* from traditional relational databases like DB2, Oracle or Informix discussed within previous lectures. Applying this concept on distributed data-systems is a little bit more complex, as *secondary indices* need to take care of partitions of the data they link to as well as they even need to be partioned themself too. We will take a closer look on two common approaches for *secondary indeces* and partitioning within the next two chapters *2.4.2.2.1 Local Secondary Indices* and *2.4.2.2.2 Global Secondary Indices*.

2.4.2.2.1 Local Secondary Indices

A *local secondary index* (also known as *LSI*) is achieved by creating, storing and maintaining the index locally in every partition. Insert, update or delete operations are performed locally on the node and partition the index belongs to. Every partition basically manages its own index and all pointers reference to local data items. For instance if a data record is added to a partition, the partition also automatically adds the new entry to its secondary index. Maintaining a local index requires less overhead and is in this way usually much faster than maintaining a global index. But do not ignore that maintaining the local index will compete with the local workload and affect the throughput and if your cluster grows significantly in size (amount of nodes), *scattering* and *gathering* overhead will probably have a major impact on read request performance.

Let's get back to the previously mentioned example of Facebook profiles (Figure 2.6 on page 26) and how this would look like using *local secondary indices*: Figure 2.22 on page 80. As you can see every partition has its own local secondary indices. If you for instance want to query alle profiles of people living in "*Palo Alto*", you would need to query the secondary index of every partition (*scatter* and *gather*) as the data is not partitioned by `city` and related entries can be found in any partition. This is also a major weakness of *local secondary indices*, as they are expensive as you need to query all partitions. This can be done in parallel, as we are talking about distibuted, partitioned and probably replicated data-systems but you should keep that always in mind. It is recommended to partition a dataset in such a way, secondary indices can be served by one partition, but thats not always feasible. especially if you make use of multiple seconday indices within one query.
Local secondary indices are implemented and provided by e.g. Riak[133], Cassandra[134] and DynamoDB[135].

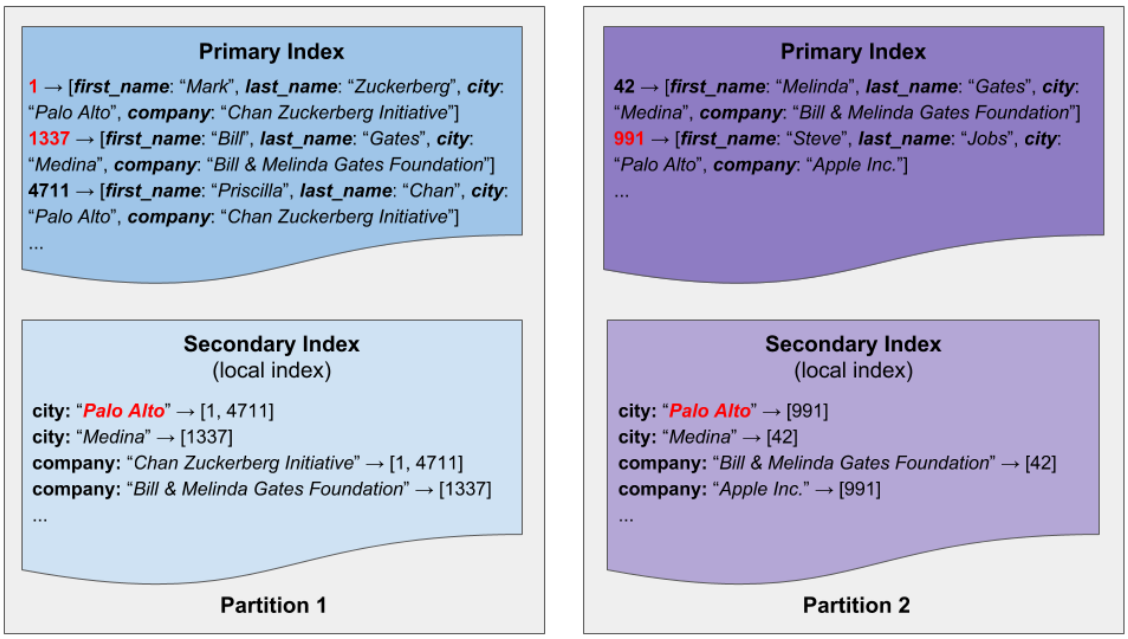

Figure 2.22: Schema - Secondary Indices (local)

[133](RIA18f), http://docs.basho.com/riak/kv/2.2.3/using/reference/secondary-indexes/
[134](CAS18e), https://www.datastax.com/dev/blog/cassandra-native-secondary-index-deep-dive
[135](DYN18b), https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html

2.4.2.2.2 Global Secondary Indices

A *global secondary index* (also known as *GSI*) is achieved by creating, storing and maintaining the index globally and independent of local data items of partitions. Insert, update or delete operations require remote updates of the index for the belonging data. A *global secondary index* could be saved on a single node but that would not only violate the basic idea of distributed data-systems, *scalability* and *reliability* but also will also be impossible at some time as the size of the index gets to big for one node. So every *global secondary index* is partitioned by its own key indepent of local data records. For instance if a data record is added to a partition, the partition also automatically adds the new entry to its secondary index.
Maintaining a global index requires more overhead and is in this way usually slower than maintaining a local index but also more efficent in terms of read requests as you only need to query one partition index instead of *scattering* and *gathering* all partitions. This approach usually weakens the read consistency, as indices updates take more time, for instance DynamoDB supports *strong consistency* for *local secondary indices*[136] but only *eventual consistency* for *eventual consistency* for *global secondary indices*[137].
Local secondary indices are mostly implemented by traditional relational databases (e.g. Oracle Database[138] or Microsoft SQL Server[139]) but rarely used by BigData data-systems. Nevertheless DynamoDB[140] and Couchbase[141] make use of them.

Let's get back to the previously mentioned example of Facebook profiles (Figure 2.6 on page 26) and how this would look like using a *global secondary index*: Figure 2.23 on page 82. As you can see every *global secondary index* is partioned and stored complety independant of the belonging data it points to. If you for instance want to query alle profiles of people living in "*Palo Alto*", you would query the secondary

[136](DYN18b), https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html
[137](DYN18a), https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
[138](ORC18c), https://docs.oracle.com/database/121/VLDBG/GUID-EE7C7B09-81BD-4996-8AC1-42A50D26FC25.htm
[139](MSS18), https://docs.microsoft.com/de-de/sql/relational-databases/partitions/partitioned-tables-and-indexes?view=sql-server-2017
[140](DYN18a), https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
[141](CBG18), https://docs.couchbase.com/server/5.5/indexes/indexing-overview.html

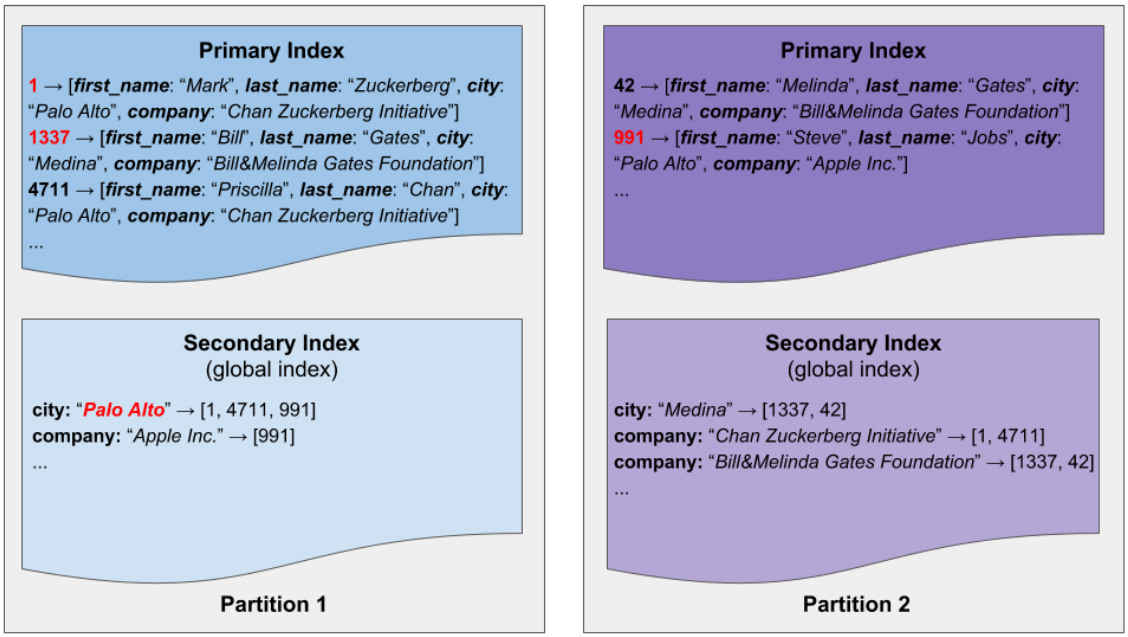

Figure 2.23: Schema - Secondary Index (global)

index of partition 1 once and be able to get all related records, irrespective of which partition they are stored in. The *global secondary index* itself is also partitioned, as you can see, but completely independent of the partions of the data records it points to.

This approach is usually faster in terms of read requests, as you only need to query one partition to get the full secondary index instead of *scattering* over all partitions. But it could also be slower at write operations, as if only one record changes, probably multiple indexes and partitions (probably on multiple nodes) need to be updated too.

2.4.2.3 Rebalancing Partitions

2.4.2.4 Request Routing

2.4.3 Transactions

to-be-added

2.4.4 Consistency

to-be-added

3 Data Processing On Distributed Systems

Begin at the beginning, the King said gravely, "and go on till you come to the end: then stop."

—Lewis Carroll, *Alice in Wonderland*

3.1 Batch Processing

to-be-added

3.2 Micro-Batch Processing

to-be-added

3.3 Stream Processing

to-be-added

3.4 Message Queuing

to-be-added

3.5 ETL and Workflow Automation

to-be-added

4 Software and Frameworks

> *Begin at the beginning, the King said gravely, "and go on till you come to the end: then stop."*
>
> —Lewis Carroll, *Alice in Wonderland*

to-be-added

5 Data Science

"Big data is not bout the data."

— Gary King, *Harvard University*

5.1 Data Cleaning, Integration and Preparation

to-be-added

5.2 Data Visualization

to-be-added

5.3 Regression

to-be-added

5.4 Classification

to-be-added

5.5 Clustering

to-be-added

5.6 Association

to-be-added

5.7 Neural Networks

to-be-added

5.8 DataScience on Distributed Systems

Spark ML PySpark

6 Outlook

"Big data is not bout the data."

— Gary King, *Harvard University*

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

7 Appendix

List Of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
BDR	Bi-Directional Replication
CPU	Central Processing Unit
CSV	Comma-Separated Values file
DHBW	Duale Hochschule Baden-WŸrttemberg
GSI	Global Secondary Index
HDD	Hard Disk Drive
JSON	JavaScript Object Notation
LSI	Local Secondary Index
LVM	Logical Volume Manager
ORM	Object-Relational-Mapper
OS	Operating System
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RU	Rack Unit
SLA	Service Level Agreement
SPOF	Single Point Of Failure
SQL	Structured Query Language
SSD	Solid-State Drive
YARN	Yet Another Resource Negotiator

List of Figures

List Of Code Snippets

Bibliography

[AHW18] *Apache Hive Website*. https://hive.apache.org/. Version: August 2018, Abruf: 04. 08. 2018 42, 55

[AMZ18] *AWS - EBS-Snapshots*. https://docs.aws.amazon.com/de_de/AWSEC2/latest/UserGuide/EBSSnapshots.html. Version: August 2018, Abruf: 18. 08. 2018 68

[ARA18] *ArangoDB Manual 3.3 - Write-ahead log*. https://docs.arangodb.com/3.3/Manual/Architecture/WriteAheadLog.html. Version: August 2018, Abruf: 18. 08. 2018 79

[AVR18] *Apache Avro Website*. https://avro.apache.org/. Version: August 2018, Abruf: 04. 08. 2018 48

[Ber15] Bermbach, David: *Messbarkeit und Beeinflussung von Eventual-Consistency in verteilten Datenspeichersystemen*. Technische Universität Berlin, 2015 65

[BTR15] *Google Cloud BigTable - Release Notes*. https://cloud.google.com/bigtable/docs/release-notes. Version: August 2015, Abruf: 04. 08. 2018 41

[CAS18a] *Cassandra 3.0 Documentation - How is data updated?* https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlWriteUpdate.html. Version: August 2018, Abruf: 25. 08. 2018 100

[CAS18b] *Cassandra 3.0 Documentation - nodetool repair*. https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsRepair.html. Version: August 2018, Abruf: 25. 08. 2018 95

[CAS18c] *Cassandra 3.0 Documentation - Repairing nodes.* https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesTOC.html. Version: August 2018, Abruf: 25. 08. 2018 92, 96, 98

[CAS18d] *Cassandra 3.0 Documentation - Virtual nodes.* https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeVnodesUsing.html. Version: August 2018, Abruf: 26. 08. 2018 109

[CAS18e] *Cassandra Blog - Cassandra Native Secondary Index Deep Dive.* https://www.datastax.com/dev/blog/cassandra-native-secondary-index-deep-dive. Version: September 2018, Abruf: 19. 09. 2018 134

[CAS18f] *Cassandra Documentation - Understanding Hinted Handoff.* https://www.datastax.com/dev/blog/understanding-hinted-handoff. Version: August 2018, Abruf: 25. 08. 2018 104

[CAS18g] *DataStax - Cassandra Architecture.* https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archIntro.html. Version: August 2018, Abruf: 19. 08. 2018 87

[CAS18h] *DataStax - Cassandra Partitioners.* https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html. Version: September 2018, Abruf: 11. 09. 2018 119, 120

[CAS18i] *DataStax - Cosistent Hashing.* https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeHashing.html. Version: September 2018, Abruf: 17. 09. 2018 127

[CBC18] *Couchbase Documentation 5.5.* https://docs.couchbase.com/server/5.5/understanding-couchbase/buckets-memory-and-storage/vbuckets.html. Version: September 2018, Abruf: 17. 09. 2018 122

[CBG18] *Couchbase Documentation 5.5 - Indexing*. https://docs.couchbase.com/server/5.5/indexes/indexing-overview.html. Version: September 2018, Abruf: 19. 09. 2018 141

[CBP18] *Couchbase Documentation - vBuckets*. https://developer.couchbase.com/documentation/server/3.x/admin/Concepts/concept-vBucket.html. Version: August 2018, Abruf: 26. 08. 2018 112

[CDB18] *CouchDB Documentation - Introduction to Replication*. http://docs.couchdb.org/en/2.1.2/replication/intro.html. Version: August 2018, Abruf: 18. 08. 2018 81

[Cod70] Codd, Edgar F.: *A Relational Model of Data for Large Shared Data Banks*. IBM Research Laboratory, San Jose, California, 1970 32, 53

[CQL18] *The Cassandra Query Language (CQL)*. http://cassandra.apache.org/doc/latest/cql/. Version: August 2018, Abruf: 04. 08. 2018 43, 54

[DDC74] Donald D. Chamberlin, Raymond F. B.: *SEQUEL: A STRUCTURED QUERY LANGUAGE*. IBM Research Laboratory, San Jose, California, 1974 52

[DYN18a] *Amazon DynamoDB Developer Guide - Global Secondary Indexes*. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html. Version: September 2018, Abruf: 19. 09. 2018 137, 140

[DYN18b] *Amazon DynamoDB Developer Guide - Local Secondary Indexes*. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html. Version: September 2018, Abruf: 19. 09. 2018 135, 136

[DYN18c] *AWS - DynamoDB*. https://aws.amazon.com/de/dynamodb/. Version: August 2018, Abruf: 25. 08. 2018 90

[DYN18d] *DynamoDB - Replication And Partitioning*. https://cloudacademy.com/blog/dynamodb-replication-and-partitioning-part-4/. Version: September 2018, Abruf: 17. 09. 2018 130

[DYN18e] *Wikipedia - DynamoDB*. https://de.wikipedia.org/wiki/Amazon_Dynamo. Version: August 2018, Abruf: 25. 08. 2018 102

[EBA12] *Hugh E. Williams (VP Engineering and Product at ebay.com 2009-2013)*. https://hughewilliams.com/2012/06/26/the-size-scale-and-numbers-of-ebay-com/. Version: July 2012, Abruf: 20. 07. 2018 18

[EBA18] *What we do - eBay is where the world goes to shop, sell, and give.)*. https://www.ebayinc.com/our-company/who-we-are/. Version: July 2018, Abruf: 15. 07. 2018 15

[ELA18] *Elasticsearch Reference - Basic Concepts*. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html. Version: August 2018, Abruf: 26. 08. 2018 106

[ESR18] *Elasticsearch The Definite Guide - Routing A Document To A Shard*. https://www.elastic.co/guide/en/elasticsearch/guide/current/routing-value.html. Version: September 2018, Abruf: 17. 09. 2018 123

[ESS18a] *Elasticsearch Reference 6.4 - Shrink Index*. https://www.elastic.co/guide/en/elasticsearch/reference/6.4/indices-shrink-index.html. Version: September 2018, Abruf: 17. 09. 2018 126

[ESS18b] *Elasticsearch Reference 6.4 - Split Index*. https://www.elastic.co/guide/en/elasticsearch/reference/6.4/indices-split-index.html. Version: September 2018, Abruf: 17. 09. 2018 125

[FBI08] *Cassandra – A structured storage system on a P2P Network*. https://www.facebook.com/note.php?note_id=24413138919. Version: August 2008, Abruf: 04. 08. 2018 40

[FC06] Fay Chang, Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. G. Jeffrey Dean D. Jeffrey Dean: *Bigtable: A Distributed Storage System for Structured Data*. Google Inc., 2006 111

[GIT18] *Git - Website*. https://git-scm.com/. Version: August 2018, Abruf: 18. 08. 2018 86

[HBA18] *HBase Documentation - Regions*. http://hbase.apache.org/0.94/book/regions.arch.html. Version: August 2018, Abruf: 26. 08. 2018 110

[HDD17] *Backblaze HDD Failure Study*. https://www.backblaze.com/blog/hard-drive-failure-rates-q1-2017/. Version: July 2017, Abruf: 20. 07. 2018 28

[HDP18] *Apache Hadoop Documentaion - Shuffle and Sort Documentation*. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html. Version: August 2018, Abruf: 07. 08. 2018 58

[IBM18] *IBM DB2 Knowledgebase*. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/json/src/tpc/db2z_jsonfunctions.html. Version: July 2018, Abruf: 13. 07. 2018 2

[IIS18] *Microsoft IIS Homepage*. https://www.iis.net/. Version: July 2018, Abruf: 14. 07. 2018 10

[JD04] Jeffrey Dean, Sanjay G.: *MapReduce: Simplified Data Processing on Large Clusters*. Google Inc., 2004 56

[Joh14] Johanan, Joshua: *Buidling Scalable Apps With Redis And NodeJS*. Packt Publishing, 2014 7

[JSO18] *JSON Website*. https://www.json.org/. Version: August 2018, Abruf: 04. 08. 2018 44

[KFK18] *Apache Kafka User Documentaion*. https://kafka.apache.org/10/documentation/streams/developer-guide/interactive-queries.html. Version: July 2018, Abruf: 13. 07. 2018 8

[LVM18] *Linux Logical Volume Manager*. http://www.sourceware.org/lvm2/. Version: August 2018, Abruf: 18. 08. 2018 67

[MDB18a] *MonggoDB Manual - Replica Set Oplog*. https://docs.mongodb.com/manual/core/replica-set-oplog/. Version: August 2018, Abruf: 18. 08. 2018 70, 72, 74

[MDB18b] *MonggoDB Manual - Replication*. https://docs.mongodb.com/manual/replication/. Version: August 2018, Abruf: 12. 08. 2018 61

[MDB18c] *MongoDB Blog - Fast Updates with MongoDB*. https://www.mongodb.com/blog/post/fast-updates-with-mongodb-update-in-place. Version: August 2018, Abruf: 04. 08. 2018 50

[MDB18d] *MongoDB Documentation - exists*. https://docs.mongodb.com/manual/reference/operator/query/exists/. Version: August 2018, Abruf: 04. 08. 2018 45

[MDB18e] *MongoDB Documentation - lookup*. https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/. Version: July 2018, Abruf: 23. 07. 2018 36

[MDB18f] *MongoDB Documentation - MongoDB Ops*. https://dev.mysql.com/doc/refman/5.7/en/slave-logs-relaylog.html. Version: August 2018, Abruf: 18. 08. 2018 66

[MDB18g] *MongoDB Manual - Arbiter*. https://docs.mongodb.com/manual/core/replica-set-arbiter/. Version: August 2018, Abruf: 18. 08. 2018 76

[MDB18h] *MongoDB Manual - Hashed Sharding*. https://docs.mongodb.com/manual/core/hashed-sharding/. Version: September 2018, Abruf: 17. 09. 2018 132

[MDB18i] *MongoDB Manual - Ranged Sharding*. https://docs.mongodb.com/manual/core/ranged-sharding/. Version: September 2018, Abruf: 17. 09. 2018 131

[MDB18j] *MongoDB Manual - Replication Synchronization*. https://docs.mongodb.com/manual/core/replica-set-sync/. Version: August 2018, Abruf: 17. 08. 2018 63

[MDB18k] *MongoDB Manual - Sharding*. https://docs.mongodb.com/manual/sharding/. Version: August 2018, Abruf: 26. 08. 2018 105

[MGP18] *Mongoose Documentation - populate*. http://mongoosejs.com/docs/populate.html. Version: July 2018, Abruf: 23. 07. 2018 37

[Moh13] Mohan, C.: *History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla*. IBM Almaden Research Center, San Jose, California, 2013 39

[MSS18] *Microsoft SQL Server 2017 - Partitioned Tables and Indexes*. https://docs.microsoft.com/de-de/sql/relational-databases/partitions/partitioned-tables-and-indexes?view=sql-server-2017. Version: September 2018, Abruf: 19. 09. 2018 139

[MYS18a] *MySQL 5.5 Reference Manual - ALTER TABLE*. https://dev.mysql.com/doc/refman/5.5/en/alter-table.html. Version: August 2018, Abruf: 04. 08. 2018 46

[MYS18b] *MySQL 5.7 Documentation - Key Partitioning*. https://dev.mysql.com/doc/refman/5.7/en/partitioning-key.html. Version: September 2018, Abruf: 11. 09. 2018 118

[MYS18c] *MySQL Documentation - Adding Slaves to a Replication Environment*. https://dev.mysql.com/doc/refman/5.7/en/replication-howto-additionalslaves.html. Version: August 2018, Abruf: 18. 08. 2018 69, 73

[MYS18d] *MySQL Documentation - Mixed Binary Logging Format*. https://dev.mysql.com/doc/refman/8.0/en/binary-log-mixed.html. Version: August 2018, Abruf: 18. 08. 2018 78

[MYS18e] *MySQL Documentation - Replication*. https://dev.mysql.com/doc/refman/8.0/en/replication.html. Version: August 2018, Abruf: 12. 08. 2018 59

[MYS18f] *MySQL Documentation - Replication Formats*. https://dev.mysql.com/doc/refman/8.0/en/replication-formats.html. Version: August 2018, Abruf: 18. 08. 2018 77

[MYS18g] *MySQL Documentation - Replication Synchronization*. https://dev.mysql.com/doc/refman/8.0/en/replication.html. Version: August 2018, Abruf: 17. 08. 2018 64

[MYS18h] *MySQL Documentation - The Slave Relay Log*. https://dev.mysql.com/doc/refman/5.7/en/slave-logs-relaylog.html. Version: August 2018, Abruf: 18. 08. 2018 71, 75

[MYS18i] *MySQL Documentation 5.7 - MySQL Multi-Source Replication*. https://dev.mysql.com/doc/refman/5.7/en/replication-multi-source-overview.html. Version: August 2018, Abruf: 18. 08. 2018 84

[ORC18a] *Apache ORC Website*. https://orc.apache.org/. Version: August 2018, Abruf: 04. 08. 2018 49

[ORC18b] *IBM DB2 Knowledgebase*. https://docs.oracle.com/database/121/ADXDB/json.htm. Version: July 2018, Abruf: 13. 07. 2018 3

[ORC18c] *Oracle Database Documentation 12c - Global Partitioned Indexes*. https://docs.oracle.com/database/121/VLDBG/GUID-EE7C7B09-81BD-4996-8AC1-42A50D26FC25.htm. Version: September 2018, Abruf: 19. 09. 2018 138

[ORM18a] *Hibernate Website*. http://hibernate.org/. Version: July 2018, Abruf: 20. 07. 2018 34

[ORM18b] *SQLAlchemy Website*. https://www.sqlalchemy.org/. Version: July 2018, Abruf: 20. 07. 2018 35

[PIN18] *The eBay Architecture (Randy Shoup and Dan Pritchett)*. https://blog.keen.io/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pint
Version: April 2018, Abruf: 14. 07. 2018 12

[PSQ18a] *Postgres-BDR - Website*. https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/. Version: August 2018, Abruf: 18. 08. 2018 82

[PSQ18b] *PostgreSQL Documentation 9.6 - Write-Ahead Logging (WAL)*. https://www.postgresql.org/docs/9.6/static/wal-intro.html.
Version: August 2018, Abruf: 18. 08. 2018 80

[PSQ18c] *PostgreSQL Manuals - Replication*. https://www.postgresql.org/docs/9.6/static/runtime-config-replication.html.
Version: August 2018, Abruf: 12. 08. 2018 60

[RDB18a] *RethinkDB - Architecture*. https://www.rethinkdb.com/docs/architecture/. Version: August 2018, Abruf: 12. 08. 2018 62

[RDB18b] *RethinkDB Architecture*. https://www.rethinkdb.com/docs/architecture/. Version: September 2018, Abruf: 17. 09. 2018 114

[RDB18c] *RethinkDB Documentation*. https://www.rethinkdb.com/api/javascript/rebalance/. Version: September 2018, Abruf: 17. 09. 2018 115

[RDB18d] *RethinkDB Documentation - Architecture*. https://www.rethinkdb.com/docs/architecture/. Version: August 2018, Abruf: 26. 08. 2018 107

[RDB18e] *RethinkDB Documentation - Joins*. https://www.rethinkdb.com/docs/table-joins/. Version: July 2018, Abruf: 23. 07. 2018 38

[RDB18f] *RethinkDB User Documentaion*. https://rethinkdb.com/docs/changefeeds/. Version: July 2018, Abruf: 13. 07. 2018 5

[RIA18a] *RIAK Blog - Why Riak Just Works*. http://basho.com/posts/technical/why-riak-just-works/. Version: August 2018, Abruf: 25. 08. 2018 93, 97, 99

[RIA18b] *Riak Blog - Why Riak Just Works*. http://basho.com/posts/technical/why-riak-just-works/. Version: September 2018, Abruf: 11. 09. 2018 121

[RIA18c] *Riak Documentation - Conflict Resolution*. http://docs.basho.com/riak/kv/2.2.3/developing/usage/conflict-resolution/. Version: August 2018, Abruf: 25. 08. 2018 101

[RIA18d] *Riak Documentation - Resiliency*. http://basho.com/products/riak-kv/resiliency/. Version: August 2018, Abruf: 25. 08. 2018 89

[RIA18e] *Riak Documentation - Riak KV Glossary*. http://docs.basho.com/riak/kv/2.2.3/learn/glossary/. Version: August 2018, Abruf: 25. 08. 2018 103

[RIA18f] *Riak Documentation - Secondary Indexes Reference*. http://docs.basho.com/riak/kv/2.2.3/using/reference/secondary-indexes/. Version: September 2018, Abruf: 19. 09. 2018 133

[RIA18g] *Riak Documentation - Vnodes*. http://docs.basho.com/riak/kv/2.2.3/learn/concepts/vnodes/. Version: August 2018, Abruf: 26. 08. 2018 108

[RIA18h] *Riak KV Glossary*. http://docs.basho.com/riak/kv/2.2.3/learn/glossary/. Version: September 2018, Abruf: 17. 09. 2018 128

[RUD18] *Redis User Documentaion*. https://redis.io/commands/rpoplpush. Version: July 2018, Abruf: 13. 07. 2018 6

[SP06] Shoup, Randy ; Pritchett, Dan: *The eBay Architecture (Randy Shoup and Dan Pritchett)*. http://www.addsimplicity.com/

downloads/eBaySDForum2006-11-29.pdf. Version: November 2006, Abruf: 14. 07. 2018 11, 14

[SVN18] *Apache SVN - Website*. https://subversion.apache.org/. Version: August 2018, Abruf: 18. 08. 2018 85

[TH83] Theo Haerder, Andreas R.: *Principles of Transaction-Oriented Database Recovery*. IBM Research Laboratory, San Jose, California, 1983 91

[TNG18] *Github Repository - Tungsten Replicator*. https://github.com/continuent/tungsten-replicator. Version: August 2018, Abruf: 18. 08. 2018 83

[TOR18] *ToroDB Website*. https://www.torodb.com/. Version: July 2018, Abruf: 13. 07. 2018 4

[VDB18] *VoldemortDB - Design*. http://www.project-voldemort.com/voldemort/design.html. Version: September 2018, Abruf: 17. 09. 2018 129

[VKV13] Vinod Kumar Vavilapallih, Chris Douglasm Sharad A. Arun C Murthyh M. Arun C Murthyh: *Apache Hadoop YARN: Yet Another Resource Negotiator*. The Hong Kong University Of Science And Technology, 2013 57

[VOL18] *VoldemortDB Documentation - Design*. http://www.project-voldemort.com/voldemort/design.html. Version: August 2018, Abruf: 25. 08. 2018 88, 94

[WKS18] *Wikipedia - SLA*. https://en.wikipedia.org/wiki/Service-level_agreement. Version: July 2018, Abruf: 13. 07. 2018 13, 22

[WPR18] *Rack Unit*. https://en.wikipedia.org/wiki/Rack_unit. Version: July 2018, Abruf: 20. 07. 2018 27

[WPS18] *Single Point of Failure*. https://de.wikipedia.org/wiki/Single_Point_of_Failure. Version: July 2018, Abruf: 20. 07. 2018 16

[YAR18] *YARN Jira - Ticket YARN-2809*. https://issues.apache.org/jira/browse/YARN-2809. Version: July 2018, Abruf: 20. 07. 2018 31