Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science III

Menthal
Managing Massive Sensor Data

Master's Thesis

Supervisor: Jun.-Prof. Dr. Alexander Markowetz

Christian Beilschmidt

Michael Mattig

Bonn, October 9, 2013

universitätbonn

Decleration of Authorship

I hereby certify that my part of this thesis, indicated by my initials, has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted. Every source of information, including graphs and data sets, has been specifically acknowledged.

Bonn, October 9, 2013 Christian Beilschmidt Michael Mattig

"Any sufficiently advanced technology is indistinguishable from magic."
Arthur C. Clarke

Contents

1 Introduction

Both psychology and psychiatry heavily rely on observing humans. They fundamentally assume that a person's mental state affects his interactions with the environment. The other way around it is possible to derive a person's mental state by observing his actions. If a driver maneuvers his car more aggressively than usual, we can safely assume that he is stressed. It is however inherently hard to gather such data.

Traditional observation methods are too expensive and often inapplicable. One approach is to let users collect data about themselves, e.g. by using a diary. Another one is to keep them in a controlled environment that makes it possible for external parties to note down their actions. A third approach is shadowing, where a follower observes a user's every move and collects this data for him. However all those methods inherently restrict the user and are biased towards the observer. They are also expensive and very time consuming which makes them infeasible for a broader audience. Recent technological developments allow the collection of data in a far less intrusive way.

Smartphones are a quantity rich source of data. In contrast to mobile sensors like wristbands, which e.g. measure the heart rate, they are already an everyday companion for many users. They take them wherever they go and use them throughout the day, mostly performing numerous short interactions. For many people it is the first and the last device they use during their waking hours. They possess a large number of different sensors which provide analyzable data. Additionally, they capture a big part of people's social interactions. This makes them especially suited for this use case.

Menthal (mental health analytics) is a research project on the intersection between psychology and mobile computing. At its core it contains a mobile software that logs interactions, locations and app usages of users. This application is used by several subprojects which all focus on human behavior. OpenMenthal is a smartphone app for Self Monitoring where people analyze themselves on their own. They are supported by automatic data collection and aggregation. The users can review their data as graphs of time series. MenthalDepressed tries to capture the mood level of a person by observing changes in their interactions with other people. If someone for instance starts talking to fewer and fewer people, it might be an indicator of some mental issue. Both deal with the analysis of potentially large data sets. This poses new challenges for storing, managing and processing this data.

In this thesis we present a structured approach for finding a suitable data storage technology to cope with the challenges of our particular use case. These are the Big Data scenario, the high frequency of inserts and the ability to handle continuous queries. The data is heterogeneous because the type and characteristics of data sources may change over time. We can think of including new sensors or adapting to different mobile phones. We expect high rates of new incoming data which a system has to handle. The arrival of new data triggers aggregations which we thus have to perform continuously in a given time frame. In order to fulfill these requirements in the limited time frame of our thesis we have to exhaust all given assumptions. For example, we do not join the data of different users. This level of optimization requires revisiting every single aspect of traditional database system architectures.

This thesis consists of twelve chapters which can be divided into three parts. The first part builds the foundation by introducing the mobile data scenario and data management in general. Chapter 2 introduces the fundamentals of capturing data on mobile devices. It also discusses the common characteristics and particularities of time series data in this context. Chapter 3 presents the basic aspects of data storage. It addresses the question of how to store and retrieve data efficiently. Chapter 4 compares the most common database models, based on the previously introduced concepts. It discusses traditional approaches like the relational model as well as the more novel NoSQL systems. Chapter 5 turns to the real world and provides an overview of the database system market. In particular, it outlines four archetypical systems we employ throughout the rest of this thesis. These are the EventStore from Marburg University, the column-family store HBase, the relational database system PostgreSQL and the in-memory key-value store Redis.

The second part deals with the requirements of our particular scenario and proposes a suitable architecture. The sixth chapter takes a closer look at the two main use cases within the Menthal project, namely Quantified Self (Self Tracking) and illness detection. It declares the focus on Quantified Self and extracts functional and nonfunctional requirements from the specified use case description. On this foundation Chapter 7 proposes an architecture for a Menthal backend. It describes the different components and their basic responsibilities. It states problems which arise and sketches solutions. However it does not provide a full implementation description.

The third part deals with the most compelling question of what database technology fits the best within this architecture. To this end Chapter 8 describes a new benchmark for testing different database systems. In contrast to existing benchmarks it focuses on the data and query load we expect in the scope of the project.

The chapter begins by describing an algorithm for artificial data generation based on empirical studies. Furthermore, it defines queries to review certain aspects of the investigated systems. Chapter 9 deals with the implementation of the benchmark and the adaptation of the different database systems. It presents a detailed description of the data generation and specifies for each system a way of storing and querying this data. The tenth chapter presents the results of an experiment based on the benchmark. It interprets the observations and points out the differences in the performance of the systems. Chapter 11 treats the non-performance aspect of the database systems. It presents our experiences during the implementation phase of the benchmark and gives an impression of how well the systems support the development process. Finally, Chapter 12 concludes the thesis by summarizing our work and providing an outlook on future research questions.

2
Mobile Sensor Data

This chapter introduces the gathering and analysis of mobile sensor data from smartphones. It begins with a short introduction to smart phones and continues with the problem of capturing data on them. It then explains some general considerations of this scenario with respect to the resources of a smartphone. The captured data has some specific characteristics. Section 2.4 briefly introduces them. The last section finally describes time series analysis and practical problems in dealing with time based data.

2.1 Smartphones [MM]

The traditional capabilities of a mobile phone are calls and text messages. They reveal the interaction of a user with other people. The amount of calls and text messages shows how communicative a user was. The number of distinct people he talked and wrote to tells us how many relationships he maintains. Changes in these characteristics are far more interesting than the absolute values. If a person starts talking to fewer and fewer people, he might suffer from depression. A smartphone has several additional features, including sensors.

Sensors

Those sensors include GPS[1], cellular and WiFi[2] connection, accelerometer, camera and microphone. GPS and connectivity signals enable us to track the position of a user. An accelerometer gives movement information on a more microscopic level. Camera and microphone are helpful to gather information about the environment of a user. Brightness and noise level are two examples for this. In contrast to feature phones, smartphones are always associated with Internet access. This mainly enables the user to surf to websites and check and write e-mails. It also enables the installation of apps.

[1]Global Positioning Service
[2]A wireless LAN connection

App Apps are software applications, specifically designed to run on mobile devices. They can make better use of the available screen size, touch-gestures and other sensors, than traditional websites can. Apps are often specialized for a single task like organizing notes, looking up train arrivals or playing a game. Usually third parties and not the phone manufacturer develop these apps. Users can download and install them via an app store. *App Store* An app store is a centralised service, usually operated by the producer of the phone's operating system. It is a catalogue and a shop for available apps, but also a source of control. On the one hand it handles the distribution of apps, and the billing between app author and user. On the other hand it only allows apps which meet certain criteria. App stores manage their products in a catalogue. This catalogue classifies the apps for example as education, entertainment, communication, news, travel, sports and so on. This classification provides us already some valuable information, without looking at the particular apps. A usage of apps from the education category hints to a more productive use of the phone. The entertainment category displays a more recreational use.

2.2 Gathering Usage Data on Smartphones [MM]

To analyze smartphone usage, we first have to gather the data on the device and collect it on a server. The first step is to develop an app, that collects information from sensors and other apps. The development of such an app is platform specific. The different mobile phone operating systems have different architectures and APIs[1] and make different kind of restrictions with respect to the app development. Common is the execution of apps in sandboxes, which poses difficulties in our scenario. Android and iOS together already make up for two thirds of the market. Android alone runs on half of the devices [Gar]. We therefore restrict our investigation to those two.

iOS Apple's operating system iOS is known to be rather restrictive. Apps are completely isolated from each other. There is even no common accessible file system which they could use to exchange data between them. Apps are only allowed to execute code, if they are currently active (in foreground). Once an app is closed, it gets a limited time frame to finish calculations or store data. It is not allowed to perform any long running calculations in the background. The only thing it can do is notifying the user. The two ways to achieve this are local notifications and push notifications [Inc]. Local notifications are time scheduled. Before a user closes the app it specifies a time and a notification message. The operating system handles the execution of this notification. Push notifications require a server to *push* a notification to the smartphone. The operating system maintains a connection to the server in the background to this end. This is useful, if a particular app is only part of a service, for example a chat app. The users begins a conversation and closes the app. If the user gets a new message, the server pushes a notification to his smartphone. The operating systems then shows a notifications about the new message and the user can open the app to continue the conversation.

[1]Application Programming Interface

In conclusion, apps are completely isolated from each other and they are not allowed to run in background. Capturing smart phone usage is thus not feasible on iOS devices at the moment. Android on the other hand is not as restrictive as iOS. It also yields a different architecture and app model.

Android

Android differentiates between apps and services [And]. An app consists of one or more activities. An activity is a single screen of an app and has a user interface. If it is hidden, the operating system pauses it after giving it time to finish some small calculations as iOS allows its apps. Services on the other hand support long running operations with no interface. A classic example is a music player that plays songs while other apps are in foreground. Services provide the necessary tool to schedule a transmission of data at regular intervals. But they still do not allow an inspection of other apps. As in iOS, android apps are also isolated. Each app runs in its own virtual machine. There are however two opportunities, which iOS does not offer: custom launchers and the accessibility service.

Launcher

A launcher is the part of the smartphone's user interface that allows to start apps. It is common for phone providers to install a custom launcher on their android smartphones. Typically this launcher includes special services of the provider into the UI and maintains the corporate identity by including the logo. As the launcher is responsible for starting apps, it can also log those events.

Accessibility Service

The accessibility service is designed to assist users with disabilities. For example it could read out content which is currently displayed on the screen to a blind person. To achieve this, it has to allow one app access to the content of another one. We can also gather usage information this way. While this is not the purpose of the interface it provides the best approach to collect such data.

2.3 General Considerations [MM]

There are some considerations when collecting usage data on smartphones. They fall into three categories: resource-dependant, financial and ethical. The first category concerns with the scarce resources that originate from the mobile nature of the device. The second one comes from the costs of cellular data transfers. The last one deals with the sensitivity of the collected data.

Battery

Smartphones have to rely on a battery. Feature phones could last for more than a week in a normal usage scenario. Smartphones offer much more functionality and are used more frequently. Thus they have a higher power consumption. They are typically designed to last a few days or only a single day when used heavily. Having only limited energy supply means that also the processing power is limited. This means we should not perform complex calculations on the smartphone, because they would take too long and use too much energy. The phone can also die, when the battery is empty. Such a scenario yields incomplete data about the usage behaviour. For example the current app is never closed. We have to fix such issues either on the phone or later on the server.

Data Transfer

The quality of a smartphone's cellular Internet connection fluctuates heavily. It depends on the capabilities and load of the current cell (total bandwidth vs. number of active users), the location of the user, the weather and other factors. Even if the connection is stable, the data transfer is typically more expensive than on wired-lines. A real time submission of usage data is therefore not feasible. It is beneficial to collect data over a period of time, for example a day, and submit all of it at once. We can avoid slow and error prone cell transmissions and possibly high fees by using a WiFi-network. A good strategy is therefore to wait for a WiFi connection to transfer the data.

Clock

Usage data is always associated with the point in time, when the user did something. The internal clock of a smartphone provides this measurement. The accuracy of the time does not have to be perfect. Whether a call took place at 12:30 or 12:37 makes no significant difference in the analysis. The time should be somewhat correct though, such that we can assign a measurement correctly e.g. to morning or afternoon. All clocks show some kind of drift, meaning they run slightly faster or slower than they should do. Over time, the accuracy of the clock's measurements thus gets worse. We can use external services to regularly correct the time of of the clock. The network time protocol (NTP) offers one possibility to do so. It requires an Internet connection though. GPS also provides time information, that we could use to synchronize the clock. In the end time measurements will be more or less inaccurate and we have to consider this in the analysis of the data.

Malware

Mobile operating systems are as susceptible to malware like viruses or trojan horses as desktop operating systems. The risks are somewhat contained by the distribution of apps through an app store. All apps in the store are tested for malicious behaviour. Users are thus more save to install apps from a store than arbitrary software on their PC they downloaded from the Internet. There are still no guarantees though. Android users can also install apps from outside the app store. Even iOS devices have been contaminated previously by manipulated websites. In the end,we can not be sure the smartphone behaves like it is supposed to. An infected device might send thousands of messages without any user action. An analysis component should be aware of this and could even be capable of detecting such behaviour and notifying the user about it.

Privacy

Collecting usage data on a smartphone raises privacy issues. The data, that leaves the phone, contains highly sensitive data, that must not get into the hands of a third party. The systems, that stores the data, thus has to fulfill high security demands. The collected itself data should contain as few personal information as possible. Text messages should e.g. not be transferred to a server. Uncritical information like the number of characters, or the time to write a response already yield a lot of information. Phone calls should not be recorded. Extracting information like the talking speed or the proportion of time each person talked, require much less space and preserve privacy.

2.4 Sensor Data Characteristics [CB]

The sensor data we collect in our scenario yields certain characteristics. Everything we collect can be seen as an *event*. Events are inherently temporal data. They are either instantaneous or time spanning. Instantaneous events occur at a single point in time with no duration. An example is to unlock a phone or switch to a new app. Time spanning events have a start time and duration, e.g. app sessions which state when and how long an app was used. Events contain a set of attributes and corresponding values. The set of attributes form the schema of an event type. An app session event e.g. would have attributes `app name` and `category`. A GPS event would yield the current `location` as coordinates.

Append Only

Event data is timely ordered and *append only*. This means events are created and received in the order they occur and the data is never deleted or modified. This has positive implications on the efficient storage of such data. We can store them in the original order and do not need to rearrange them later on.

A single event only contains little information. This manifests in lots of small pieces of data, which is again important to consider for efficient storage. It also means that it rarely happens, that only a single event is needed in an analysis. The most common access patterns are range queries on the time dimension, like: *"Give me all phone calls of a user of the last 30 days"*.

2.5 Time Series Analysis [CB]

Analysis of mobile sensor data belongs to the field of time series analysis. A time series is a sequence of data points over time. We visualize it as graph with the time on the horizontal axis (cf. Figure 2.1). In the analysis one tries to find characteristics, patterns and statistics within the data. It typically results in a clustering or classification of the data, or shows detected anomalies. In time series forecasting, a model is formed to predict the future development of the series based on previous observations. [Ham94]

Types of Time Series

There are two types of time series: continuous and discrete time series. Continuous time series assign to each point in time a value. Discrete ones do so only for specific points in time. They can be further distinguished in equally and unequally spaced time series. As the name indicates, data points in equally spaced time series occur in regular time intervals, e.g. every minute. This constraint does not hold for unequally spaced time series. For analysis purposes we often transform unequally spaced time series into equally spaced counterparts using interpolation. Then we can use the tools of equally spaced time series. Common operations on time series include finding the average, maximum or minimum of a time series on a given interval. Also finding trends, smoothing the curve or calculating its derivative are typical.

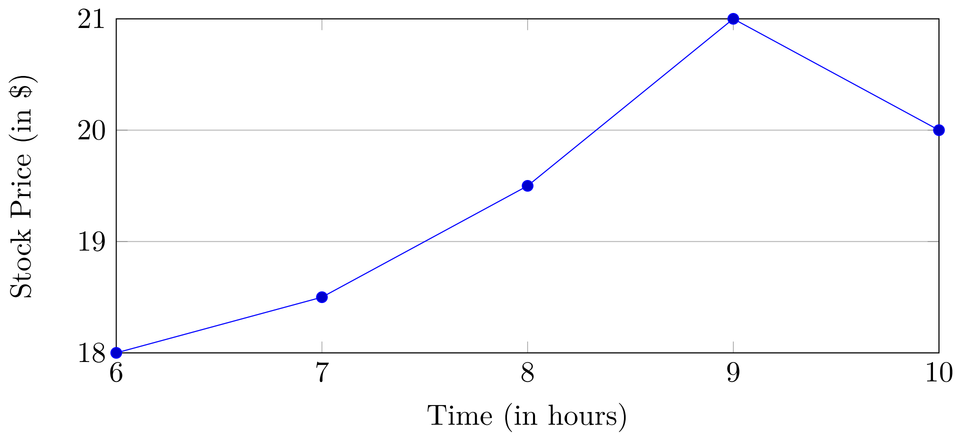

Figure 2.1: Example of a discrete time series

Time Zones

There are some further considerations regarding the management of time series. Time is not a global measure. The earth is separated into time zones, to adjust to the different times in rise and set of the sun at different locations. When we manage data from multiple time zones together, we have to take the different time zones into account. Our time measures and our calender are based on astronomical phenomena like the time it takes for the earth to rotate around itself and around the sun. To cope with deviations between the two parts, leap seconds and days are introduced. Then there is the more recent development of the daylight saving time which leads to some hours being skipped or occurring twice if the clock is adjusted.

The granularity of time is crucial when capturing and also when storing data. Typically smartphones deliver measurements with milli- or microsecond granularity. As previously discussed, the measurement will not be accurate with respect to a reference clock. A reference clock is a reliable clock, which's time we considered to be correct. The granularity is more interesting for ordering the events of a single phone. If the granularity is fine enough, the time is unique for all events of the phone. This can then be exploited to identify this event by the occurrence in time.

3 Aspects of Data Management

This chapter presents different ways to store and retrieve data efficiently. Database systems are a common way to achieve this. They play an important role in storing information in- and outside of today's companies.

Definition Database Management System

A database system consists of a set of data and the programs necessary for processing this data. The stored data is often described as the database. This database contains with each other related information units, necessary for controlling and managing an area of responsibility (e.g. a whole company). The database *management system* is the collection of programs we use for accessing the database, for checking the consistency and for modifying the data. Often literature differentiates these components explicitly, such that with database management system (or shorter database system) we mean the database as well as all management processes regarding the database.

This chapter describes the most important aspects of a typical database system. It begins with a discussion about the term data model, which characterizes the approach of defining and accessing data. It continues describing different ways to store and persist data on a physical medium. Section 3.2 enumerates the established storage technologies and their differences. The chapter then describes ways to access the data. These define the means to interact with a database system. The following section describes different architectures like client-server and embedded. It also mentions a clustered operation of a database system and presents different types of data distribution across multiple machines. The next section deals with maintaining the integrity and consistency of data. This includes preserving stored data despite unexpected failures. It also includes the definition of a schema and the denial of operations that would violate it. Finally the section discusses concurrent access and security related aspects. The last section of this chapter discusses what makes database systems fast and presents methods for tuning the performance.

As a final remark: every database system consists of several components. This section describes aspects, which are common among most of the database systems. Not every part in the following section is obligatory for a database system. Some parts might be missing and there may exist several additional aspects, which are not mentioned here.

3.1 Data Model [CB]

A fundamental characteristic of a database management system is data abstraction. This means that the underlying physical storage differs from the conceptual way of describing the data. Literature calls this the data model. There are several levels of abstraction. In more detail the data model provides all necessary tools for describing and accessing the data. Typically it also contains a set of basic commands for retrieving and modifying data. [SKS06]

Abstraction Levels

The data model yields three different abstraction levels namely the physical, the logical and the view level. Literature defines them also as internal, conceptual and external level. The physical level describes all details of the data storage like the internal structure and layout on an actual storage medium. It also describes all access paths to the data. Low-level model is another term for the data model concerning the physical layout.

Conceptual Level

The model at the conceptual level is called high-level model. It describes the primary way for users of interacting with the database system. It defines all semantics of the data, their relationships and integrity as well as consistency constraints. This description hides the physical layer below. There are models on top of this level like the entity relationship model for relational data. They form a view on the concrete implementation of the schema in the database system.

View Level

The view level models base on the high-level model. There can be multiple views for the data model of a database system. System wide models like entity relationship are an abstract basis for these models. A view forms a subset of the overall data and recombines parts of the schema. It focuses on important parts of the data model and hides details which are unimportant e.g. for the user. [EN99]

In any model it is important to differentiate between the database itself and the meta data describing it. This meta model is then referenced as the database schema. Not every database has an explicit schema, however every database has some structure within its data. Database systems where the schema is not stored in the meta data are often called schemaless. The name is unfortunately misleading, as it is meant that you cannot easily retrieve the schema and not that there is none. There is the notion of an implicit schema where the user has to look at the data for figuring it out. [SF12]

Data Independence

Another important notion in the context of the data model is data independence. It defines that the change of the schema in one of the previously explained levels does not affect the schema at the next higher level. If we change the way we store data, it must not have implications on the access on the logical level. The same holds for changes on the logical level with respect to the view level. Expanding a schema by e.g. adding more records or fields must not change the view of the application on top of it. However reducing the schema has an impact on this level, as the data is not available anymore. In this case views referring only to the remaining data should not be affected.

3.2 Physical Storage Formats [CB]

We can store a database on different physical storage mediums. This section briefly describes the most common formats with their individual properties. Furthermore it compares their differences and classifies them regarding their utility for database systems.

3.2.1 HDD

A hard disk drive (HDD) is a storage medium that uses rotating disks with magnetic coating. It is a nonvolatile storage type as the data remains on the disks even after power shutdown. HDDs can be used for storing and retrieving data, using a magnetic head on a moving arm for accessing the right data position. They store data in blocks, which are bit sequences of equal length. They are currently the default method to store data.

Characteristics HDDs are characterized by the storage capacity, the performance and their price. Today's disks have a capacity in a range from several 100 gigabyte to a few terabyte of memory. Additionally they have a disk buffer consisting of 8 to 128MiB of faster volatile memory. This buffer caches common reads to the disk and is located between the actual memory and the disk controller.

Pricing HDDs are rather inexpensive compared to other storage types. Besides the growing disk capacities the price per unit is constantly decreasing over the last decades (cf. Figure 3.1). This makes it possible to store large amounts of data within a database system on a single machine. HDDs are connected via different bus types like Serial ATA, IDE and Serial Attached SCSI (SAS). They differ in the transmission rates between the disk and the rest of the computer and in the price of the HDD.

Performance The performance is a major downside of the HDD with respect to other storage types. Common characteristics are the seek time, the latency and the data transfer rate. The seek time describes the time for finding the desired block on the disk. Latency is the delay of the rotation for bringing the required sector of a disk under the read-write mechanism. The data transfer rate describes the time for reading consecutive data from the disk to the buffer. These three attributes form the time to access data as common measure for comparing different HDDs. The mechanical restrictions of the seek time and latency lead to a poor performance compared to other memory types.

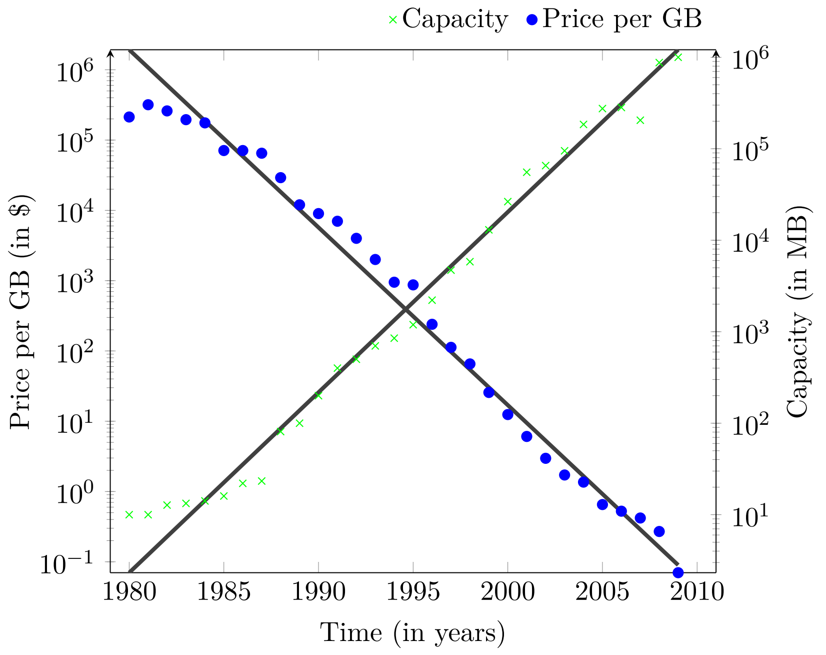

Figure 3.1: HDD prices and capacities (numbers from [Kom])

3.2.2 SSD

The solid-state drive (SSD) is a different form of memory, using integrated circuit assemblies to store data (flash memory). SDDs store data persistently like HDDs. Moreover they use a common block input/output as an interface. This technology requires no moving mechanical components for the storage of data. This leads to fewer complications with physical damage and a noiseless operation.

The performance of SSDs is in general better than that of HDDs. It omits seek time and the latency is lower. As a result they can achieve a faster access time, which is the main advantage of SSDs compared to HDDs. Today's HDDs read and write data at about 200MiB/s and access it within a few milliseconds. SSDs achieve up to 600MiB/s transfer rate and access times of fractions of a millisecond.

SSDs in the consumer class cost \$0.70 to \$1.00 per gigabyte of storage. This is far more expensive than current HDDs. The possible disk capacity is far lower than the capacity of hard disks. SSDs typically yield capacities starting from 60GB up to 512GB with prices heavily growing in the space they offer. However similar to the HDDs they show a trend of constantly shrinking overall prices.

Block Storage

The storage layout consists of blocks like in hard disks. Blocks usually have a size of about 512KiB and pages of 4KiB size subdivide them. This means each block contains 128 pages. On writing the systems takes a block and is able to write individually to each of the pages. It cannot override pages though and has to delete the whole block before it can reuse them.

TRIM Command

Usually delete commands do not affect the SSD until a new write command occurs. The operating system marks the blocks as available and does not propagate this directly to the SSD controller. This affects the performance of SSDs, as the memory fills up and the controller has to reorganize the internal structure upon further writes. This reorganization consists of writing blocks to disk, deleting blocks and distributing the data again. Current operating systems utilize the so called TRIM command which propagates the delete actions to the SSD controller. This avoids this undesired slowdown as the SSD's memory fills up.

Wear Leveling

Single cells of an SSD have, depending on their concrete type, a lifespan of about one thousand to one hundred thousand write cycles. This means it is not recommendable to write to only a few cells over and over again. To bypass this phenomenon the controller of an SSD utilizes a so called wear leveling mechanism. As the operating systems addresses concrete memory blocks, the controller proxies the operations to a custom internal structure. This structure distributes the write operations equally using hashing algorithms. Using this technique the lifetime of an SSD is in average as long as that of an HDD. The difference is that HDDs fail usually because of mechanical wear and SSDs because of the phenomenon described above.

3.2.3 In-Memory

When we mention in-memory storage, we mean the use of Random-Access Memory (RAM) for data storage. This memory type differs drastically from the two previously mentioned as it is a volatile storage. Random access describes the possibility of accessing each memory cell individually without the need to reading block-wise. RAM uses integrated circuits for storing data like SSDs. However the lack of persistence comes with an even increased access speed. This makes in-memory storing of data in a database system the fastest way for reading and writing.

Along with the highest performance RAM comes with the highest costs of all storage types. The average prices are about \$12.50 per gigabyte of memory. Contrary to prior limitations it is nowadays possible to install up to 2 terabyte of RAM into a single machine. Specific mainboard and RAM types allow these configurations although they are highly expensive. Consumer mainboards allow up to 64 gigabyte of memory to be installed. The price grows strongly with the size of the RAM, but like the other storage types the price per gigabyte decreased over the years.

3.2.4 Trade Offs

Cost vs. Speed and Capacity

The decision of a storage type for a database system depends on the simple trade-off between the costs and the speed and capacity. If it is necessary to store a lot of data, but the access time does not matter, it is sufficient to use HDDs. Having the need for fast access and computation times, in-memory database systems become a common technique. By using fast compression schemes we can store a lot of data in RAM even on a single machine. A good compromise is the use of SSDs which yield a good performance with moderate prices.

Durability Constraints

While the in-memory technique offers a very good performance, the storage type is still volatile. To achieve persistence the installation of a second storage layer consisting of nonvolatile memory is necessary. This leads to an additional layer of SSD or HDD devices in combination with the RAM storage. Current database systems offer systems for backing up the current state of the data automatically or manually.

3.3 Access and Interfaces [CB]

Every database system provides a way to access its data, however there are different approaches. Common ways are to provide either a database system language or an application interface. Each access type usually consists of two parts. The first one describes the storage layout, often called the schema. The second part enables accessing and manipulating the data. The term data definition describes the creation of meta data for specifying the contents of the database. Retrieving and changing of data could be seen as two different parts as well. But the main notion for this is data manipulation. [EN99]

Literature divides languages for manipulating data into high-level and low-level DML [SKS06]. A user can perform complex operation by only using the high-level DML on its own. He can enter it into terminal or use it embedded within a general-purpose programming language for preprocessing. Low-level DML must be embedded in a programming language in order to allow complex operations. This is because they offer only a one record-at-a-time access. In contrast high-level DML offer directly a set-oriented access. [EN99]

Declarative

Some database systems implement declarative query types as their domain specific language [Fowa]. The user specifies *what* data he needs and not *how* the data is processed. SQL is a famous example for this. The left hand side of Figure 3.2 shows a query to a database system which returns a number of customers of a company grouped by their country. The query filters out customers who do not meet a certain conditions, in this case that the gender is male. SQL is a query language for defining, manipulating and accessing the data in a relational database management system.

```sql
SELECT
  country, COUNT(country)
FROM
  customers
WHERE
  gender = 'male'
GROUP BY
  country
;
```

```python
customers = connection \
        .smembers("customers")

resultSet = []
for customer in customers:
  if fitsCondition(customer):
    resultSet.append(customer)

return resultSet
```

Figure 3.2: Examples for a declarative language and a procedural language (left SQL and right Redis embedded in Python)

Procedural

Low-level DML are also often called procedural languages. A lot of database systems do not offer a declarative language for accessing the data. The user thus needs to specify what data he needs *and* how to get the data [SKS06]. Procedural languages can either be embedded completely into a programming language or also be a domain specific language (cf. Figure 3.2). Both ways provide access for a general purpose language to loop over the contributed data and perform complex actions. A third way would be embedding programming languages into a procedural language (an example is Apache Pig of the Hadoop suite). The critical point for programming is the row-at-a-time access.

There are several other types of query possibilities for database systems. One could be a form-based approach where the user accesses the data via a GUI. A similar way would be browsing in a web-based access. A completely different approach is natural language interfaces. Systems offering these interfaces try to understand the meaning of the user input. They refer to standard words in the language or dictionary and return results on a successful interpretation. Part of this field deals with keyword-based querying. Systems use predefined indexes on words and ranking functions for retrieving the most-matching results. [EN99]

Speech input is another way of querying a database system. Often customer support systems like telephone hotlines use this approach where they give e.g. credit card information depending on the customer's input. Queries utilize mostly a limited vocabulary and use certain keywords as parameters. The database system does not only use speech as an input, but also as an output method. Such higher level query approaches often transform input into a query on a lower level. This means e.g. that speech input is first transformed to text and then transformed into an SQL query. [EN99]

3.4 Database Architectures [CB]

Database systems can be set up in different architectures. There are systems that restrict to a single structure. Others allow using different approaches depending on the usage scenario is. Several attributes characterize the different architectural concepts. They mainly differ in the number of DBMS instances or the number of users working with it in parallel.

3.4.1 Overview

Embedded

Embedded database systems are instantiated as a part of an existing program. They run whenever this program is executed and stop when it terminates. Hence they are alive in the context of this program. Such a database system has no installation process by itself and does not run in a standalone mode. As the program is the only one working with the embedded instance of the database system there is only a single user. Nevertheless multiple threads may access the database system concurrently.

The common embedded database systems use the same language as the surrounding program. They provide an API for access with all interfaces and commands using this language. Other systems are also embedded but use a different language for accessing the database. Here the API provides some wrapper for using this domain specific language. One important aspect is that embedded systems can avoid the access via the network card which is often a bottleneck in the overall performance of a database system.

Client-Server

The most common approach is the client-server concept. Here a single instance of the database system is set up on a single machine, called the server. Different programs can use the system directly because they allow a standalone installation. Programs using the instance are called clients. Following this paradigm, we have a single server with multiple clients. These client may connect at the same time, such that there is a concurrent access to the data (treated in Subsection 3.4.2).

The language (generic or domain specific) for accessing the database system may not correspond to the language used within the client programs. This requires a wrapper to enable communication. The systems often provide APIs for different general purpose programming languages such that the connection setup is rather easy. A common example is ODBC[1] which is a generic wrapper for relational database systems. It provides an interface for accessing different database systems from numerous programming languages. It allows to send SQL commands to the server and retrieve result sets.

Cluster

For several reasons (discussed in more detail in Subsection 3.4.2) there might be a need for more than one database system instance. Each of the instances has its own resources. They always use different storage space, in some cases also CPU and buffers (memory). Distinctions have the notion shared-memory, where the instances share the buffers, and shared-nothing, where they can use all components individually.

Clustering leads to a multi-server and multi-client architecture where we need to take care of the synchronization between the instances. This results in a more complex scenario as the client-server principle and depends on the specific use case. The access is similar to the client-server architecture, as we need wrappers to connect to the database system. Some systems hide the distribution from the user of the database system. We can as well consider them to be a cloud architecture.

Cloud

The cloud architecture is a similar approach to the cluster with the addition of a middleware in between the database system model and the system underneath. This middleware is characterized by aspects like transparency of the location and used resources. In opposite to the cluster it is unknown to the user what and how many resources are used to run the database system. The resources can also change at runtime, like the addition of more CPU power or more storage volume. This allows the database system to scale easily as we can adjust the resources and fit them to the current needs of the system. The ease-of-use however is a trade off to the control of the exact resources.

[1] Open Database Connectivity

3.4.2 Distribution

Now that we are aware of the possibility to distribute a database we want to take a closer look at why we would want to do this in the first place. There is a point where one instance of a database system on one server computer might not be sufficient. If this point has been reached, there are several general solutions for it. They depend on the original problem and the usage scenario. Two basic problems can be reasoning the decision to use multiple instances:

1. Physical limitations

   There are two primary limiting factors. The first one is the lack of sufficient storage capacity, such that new data cannot be stored within the database system. A solution would be to buy new and/or bigger disks for storing the data. The second one is missing CPU power and buffer space. Here we can either buy a faster CPU with more cores and bigger RAM modules or install a new machine.

2. The Total Cost of Ownership (TCO) is too high for upgrading the machine.

   Running out of resources leaves two opportunities. Either we buy a new machine to extend the overall resources or we upgrade the existing machine's hardware. This depends often on the costs assumed for upgrading compared to the costs of a completely new machine. At a certain point it is wasteful to extend an existing system as the costs become prohibitively high. From this point on distribution is necessary.

There are several approaches to achieve a distributed database system architecture. The following subsections introduce three different approaches. They discuss their main motivation and introduce some arising challenges. Additionally they briefly describe some of the most common techniques to solve them.

Replication

Replication means keeping data redundant over several database system instances. The other cluster nodes are full copies of the original one and are kept equally. There can be several reasons for the need of data duplication. One reason is that there is too much load such that one machine cannot handle it on its own. Load means the amount of retrieval or manipulating operations within the database system. An additional machine can reduce the load for each instance. *Overload*

Another reason is availability which we can divide into two arguments. The first argument is failure prevention. In order to always have a responding system, multiple instances with the same data can stand in on the crash of a node. Net splits, where the network (mainly the Internet) is separated into multiple subnets for a short period of time, can also be counteracted. However there should be sufficient nodes in the cluster to have at least one instance reacting to user calls. The second argument is the data locality within a network and thereby also an availability *Availability* *Data Locality* argument. In this case the response time is the measure which should be as low as possible to answer user requests satisfactorily. Locating equal nodes in different places makes sense in this case. We could for example set up servers in Europe and North America to lower the network delay.

Synchronization

Whenever we use replication the synchronization of the database system instances becomes a major concern. Ideally it should not matter from which instance someone reads data or writes modifications and inserts. It requires however time to forward changes someone writes to one node to the others nodes. If someone reads during the synchronization from another node, he will retrieve old data. This would yield inconsistency and could reach to a faulty state of the whole system.

Master-Slave

There are methods for dealing with network synchronization of data. One is arranging the nodes in a master-slave manner such that one node is the master server and every other node acts as a slave. In this case changes are only initiated on the master server which propagates these changes to the slaves. Writing cannot take any advantage of the scaling because of the single server limitation. Reading only from the master would solve the consistency problems, but vanishes again the scaling effect of the cluster. The cluster would exist only for backup reasons.

Reading from slaves speeds up the reading, but there is again a time span until a synchronization happens. However it is manageable as there can only be two states, one newer state from the master and the previous state on the slaves. Without the master-slave architecture there could be different states depending on the number of nodes in the cluster, because changes can occur on every node.

Quorum

QUORUM is a common technique for assuring consistency in a cluster. Considering N nodes there are two rules:

1. $W > N/2$
2. $R + W > N$

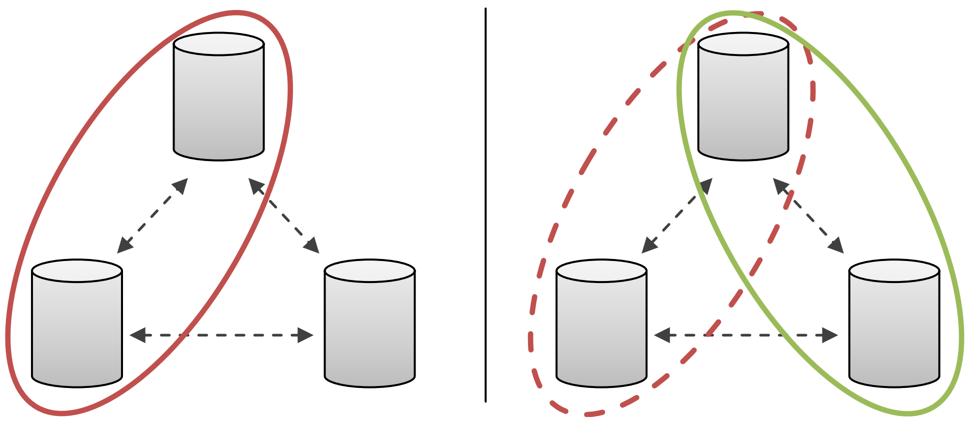

Figure 3.3: QUORUM example with $N = 3$, $W = 2$ and $R = 2$. Writing (left) to two nodes and reading (right) from any other two nodes ensures at least reaching one node that has the latest version of this piece of data.

The first rule states that we need to perform writing (modifications) simultaneously on more than half of the nodes. We can then read from R nodes such that we are reading and writing in sum to more than the N nodes. Figure 3.3 shows an example of a quorum with three nodes. Writing to two nodes and reading from two nodes ensures to read the current state of the system at least once. Having two different states of the data, the request has to dismiss the older one based on the timestamp.

Changing the amount of reads and writes puts the advantage of the scaling effect either on reading or writing. Writing to a lot of nodes leads to reading from only a few ones, which increases the performance of read requests. The settings of QUORUM depend heavily on the scenario of usage. A special case of QUORUM is the write-to-all/read-from-one principle. This architecture allows reading only from a single instance while assuring consistency. The trade off in this case is the write performance.

If consistency is not essential for an application, we can synchronize the database in a lazy fashion. *Eventual consistency* assures that at some (future) point in time all nodes are synchronized [Vog09]. Other systems allow a manual start of synchronization. This allows flexibility, but also bears more effort for the administrator of the system.

Sharding

Sharding is defined as splitting the data across multiple cluster nodes. This is necessary if the data does not fit on a single machine. Thereby a cluster of n machines is necessary to store the data. In contrast to replication the data is never held redundantly on two nodes. Combining all the data of the different nodes, we end up with the whole set of data in our database system.

The problem with sharding is the model of how to distribute the data. For finding a fitting approach we need to look at it differently. How should we layout the data throughout the sharded database system such that we can perform all operations for modifying and retrieving data efficiently? This heavily depends on the concrete scenario for which we use the database system. Generally it is important to consider network traffic as costly or slow.

For modifying the data we have to touch only a single machine. We can avoid any network traffic for synchronizing these machines after inserting or changing the data. However the database system should offer a single entrance point or interface for manipulating data. At this it has to decide to which cluster node it forwards the operations.

Hash Functions

Hash-functions are a good way to equally shard the data and operations. Based on the uniformity property a good hash function should map its inputs as evenly as possible over the output range. The output range is in this case the number of cluster nodes. Other functions depend on the specific usage scenario of the database system with respect to efficiently retrieving the data.

A system has to minimize network traffic in order to efficiently compute results on existing data in the cluster. To achieve a good parallel execution of computations it should shard the data in a way that as many nodes as possible can contribute to it. As a result the computation effort for a single machine gets lower. On the other hand the effort for managing the computation should be as low as possible. Additionally the single results, which have to be transmitted finally to one node for aggregating these results, should be minimized. A good model for sharding yields a reasonable trade-off between those two factors.

The efficiency of computations on sharded data depends on the data layout on the one hand and on the type of computation on the other hand. Some functions are easy to distribute while others have strict limitations. For aggregation functions there are three subdivisions in this context. *Distributive Functions* Distributive functions can work on partitioned data. We can apply such a function on its results again to achieve the final result. A good example is the `SUM`-function, where the sum of partial sums equals the overall sum. In this case we only have to transmit the partial results via network.

Algebraic Functions Algebraic functions need certain information about the partition for further computation. The `AVERAGE`-function e.g. needs the size of the partition for computing the average across all partitions. We thus have to transmit the result and additional partition information. *Holistic Functions* Holistic functions in contrast need the information about the data in its totality. Computing the median requires to know all data partitions in detail to compute the final median. The intermediate result (in this case the median of the partition) does not contribute to the final result. Computing holistic functions requires the most network traffic and is thereby critical in distributed computation.

Distribution

The term distribution does generally not distinguish between sharding and replication. We can see it as the combination of both approaches or an availability concerned sharding of the data (cf. Figure 3.4). The clear distinction that data only resides on one cluster node or all data resides on a single machine is no longer valid. Thereby the problems and solutions have to be slightly modified, but are again heavily dependent on the usage scenario.

CAP Theorem While talking about distribution, it has to be clear that there are limitations. One big trade-off is described by the CAP theorem. This theorem describes three different attributes, namely **c**onsistency, **a**vailability and **p**artitioning. It implies that it is always only possible to choose two out of three properties for a system. As this section is about distribution, the property of partitioning (of the data between nodes) is the presupposition. As a result we are left with the choice between availability and consistency (Figure 3.5).

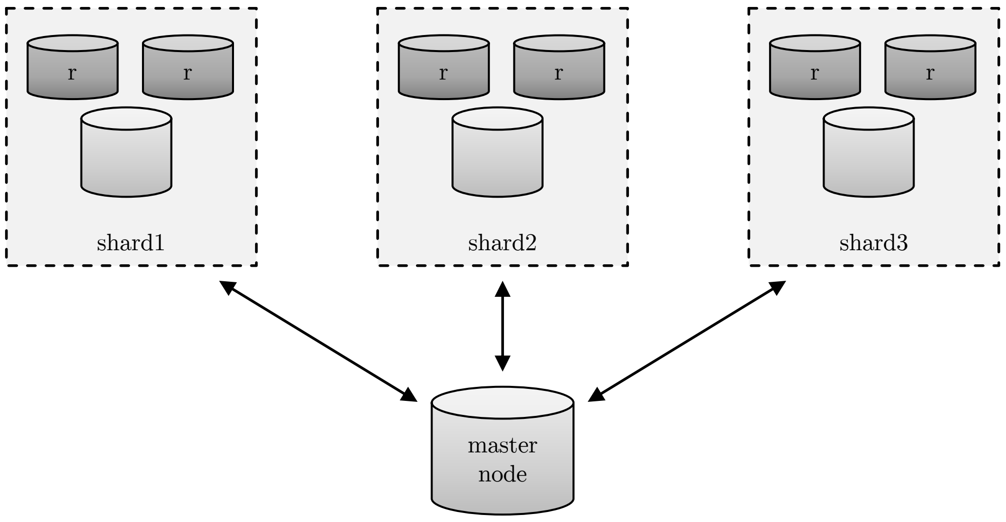

Figure 3.4: Distribution example as a combination of sharding and replication, r = replica sets

A classical example for visualizing this trade-off is a hotel booking system. This system is spread across the world with nodes for instance in the USA and in Germany. Now a net split happens such that there is no communication anymore between the US and the German node. The hotel has two possible options for coping with this problem. One option is to stop taking any new reservation for hotel rooms. This avoids double bookings for a single room and thereby yields consistency in the system. The trade-off is that the system is not usable anymore, not reachable and hence lacks of availability. The other option is that it continues taking new reservations. Double bookings for a room can occur and must be solved manually. Thereby it does not preserve consistency, but it is available all the time.

The decision between the *C* and the *A* of the CAP theorem is not a technical one. There exist both possibilities which have to fit to the need of the corresponding use case. Furthermore we cannot always clearly decide between consistency and availability for the whole system. Often it is more a gray scale where certain parts tend to require one of the possibilities.

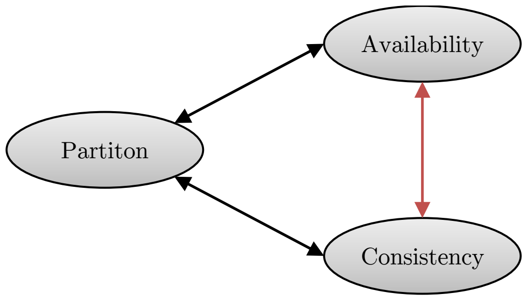

Figure 3.5: CAP theorem trade-off

3.5 Integrity and Consistency [MM]

The term consistency appeared more than once in the previous section. This section deals with the challenges of maintaining integrity and consistency in more detail. Beyond simply storing and manipulating data, we want to ensure that data is and remains meaningful. The first step is to achieve durable storage, which means preventing loss of any data. When we achieved this, we have to deny operations which would violate some given constraints. The section thus discusses the semantic integrity of data which is defined by a schema. It then turns to issues that arise with concurrent access of multiple users to the same database. Finally it treats the security aspects, which should ensure that read and write access is only performed by users who are allowed to do so.

3.5.1 Durable Storage

Durable storage is a significant aspect of preserving the consistency of data. It means a consistent state of the database is never disrupted by a sudden loss of data. More precisely if we store data in a database system, it is available forever, until a new intended manipulation command for this data occurs. A delete command is also affected by this statement as data must not reappear accidentally at a future point in time. Nonvolatile storage mediums are a necessary precondition to achieve durable storage. Using a non-persistent storage does not support durability in general. Even databases which use persistent storage types can possibly violate durability. The problem concerns preserving consistency even on system failures. We have to consider unintended shutdowns and restarts. The notion for this is recovery because we have to regain a consistent system status after the occurrence of an error.

Logging

When a system failure occurs, a normal shutdown of the system cannot be guaranteed. It is not justifiable to assume the data durability after this point has passed. We can counteract faulty data storage by using RAID[1] systems to provide loss of data by corrupt hardware. But we can not handle all errors this way. There are other systems errors and even abrupt software errors, which can be countervailed by using logging mechanisms within the database system.

WAL

A common technique for this is the write-ahead log (WAL) principle. Here the system writes every modification of the data to a log before applying them. The resulting logs are commands for applying the modification and for undoing it. If the system crashes right within an operation and the recovery procedure takes place, it can compare the logs to the current state of the data. In the case that the previous commands have succeeded, the system can restore the data by using the logs. Otherwise it can leave the data as it is (when the database equals the log information) or change it to the previous consistent state.

[1]Redundant Array of Independent Disks, allows to distribute data across multiple HDDs.

Shadow Pages

Another technique is the use of shadow pages, which is a temporary copy of parts of the data during modification. When a modification starts, the system copies the current page (unit of physical storage) of the data. It applies the modification first on the copy and after success it replaces the page with the newer version. With this technique only successful modifications become part of the database storage.

In-memory vs. HDD

A lot of modern database systems use in-memory storage for a faster access compared to magnetic storage formats (HDD). As the main memory is not a persistent storage, durable storage is not possible on first glance. Using logging principles on a nonvolatile medium helps in this case, but there are some physical limitations. As the main memory is faster than the hard disk, changes will first succeed in the former storage. Thus a time window occurs, where the two devices are not in sync. A solution is to wait for the completion on the hard disk. However this perishes the benefits of the in-memory approach for writing. In the end there is a trade-off between the performance gain and the data consistency. The goal is to minimize this gap by using techniques like compression for faster writing of the log data.

Different Paradigms

There are database systems which do not support a durable storage by choice. One example are databases systems which are used as a cache. These systems are modeled to only have a short-term memory of the data in it. Here it is no problem to renounce logging to achieve higher writing rates. Apart from this there are database systems which can accept a certain loss of data. This depends heavily on the scenario where the system is used. An example would be a view counter on a big social network. A certain amount of loss does not affect the overall consistency of the network as it is insignificantly in this case. This design could emphasize on write latency instead of consistency. Disabling logging makes sense in this case, as the system needs fewer write operations to persist the data.

3.5.2 Semantics and Schema

A schema defines constraints on the content of data. It could e.g. require a piece of data to consist of 32 bytes. A more complex schema would e.g. require each line of a text file to require a number between 1 and 10. Some database systems offer the definition of such a schema and enforce that the constraints are always valid. The schema has to be consistent, i.e. with no data present all constraints hold. The system can then check if they would still hold after a given operation took place and only perform it if they do.

We differentiate between the definition of a data type and the restriction of the value. The first case is syntactical and the second one semantical. If we store the age of a person, it is reasonable to only allow integer values. Additionally we could enforce the number to be positive. Inserting a negative number or some random text is then not possible. This avoids errors later on. This is an example of a static value constraint. On the other hand constraints can depend on other values in the database semantically. If we store a list of persons we want to prevent that the same person occurs twice. Thus the constraint would be that the piece of data to insert is not already contained in the database.

3.5.3 Concurrency

Concurrent access to the same database endangers the integrity of its data. As long as multiple users are only *reading* the same data there is no problem. When modifications come into play it gets more complicated. It might happen that one user reads some data, which is currently being modified. This can result in reading some value which is partly modified and partly in the previous state. The user thus reads something which was never supposed to be in the database. The key to solve this is to make operations atomic, meaning they are either completely done or not done at all before another operation begins. Other users can thus only read the state before or after the modification.

Atomicity

In practice we often want to perform multiple operations as a unit. For example a person gets promoted and we update first her job and then her salary. Another user, reading the persons information meanwhile, should either see the new job and the new salary or the old job and the old salary. The new job and old salary do not make sense and should not be read. Multiple operations that are performed as a unit are called *transactions*. A solution to enable the atomicity of transactions is to put multiple transaction in a sequential order (serializability). Then we can execute them one after another. This however leads to a non optimal performance, as many transactions will not interfere with each other.

Transactions

The idea is to interweave transactions in a way that represents any sequential execution. This way we can ensure that all users read and write only meaningful data into the database. A common technique are write locks, which can be acquired by a particular transaction. Until the release of this lock only this transaction is allowed to access the particular piece of data. There are different approaches like optimistic and pessimistic locking. These two make assumptions about the likelihood of read write conflicts. The optimistic approach assumes that they are rare and handles them only in the case of an error. Many database systems allow the specification of a concurrency level. *Serializable* is the the most extensive one, but it also costs the most performance.

Serialization

Locking

An alternative to locking is multiversion concurrency control. It avoids transactions waiting for locks to be released in order to perform their operations. To achieve this, the database internally stores multiple versions of a piece of data. It distinguishes them by a timestamp or an ascending transaction id. Each data object also maintains a read timestamp, that stores when a transaction performed a read. If later on it tries to modify a data object whose write timestamp is higher than the corresponding read timestamp, the transaction is reset. Read access to the database never gets blocked this way. This is advantageous, especially if a database is mostly read and rarely written. On the other hand this benefit is achieved at the expense of storing multiple versions of a data object which in turn takes up more space.

Multiversion Concurrency Control

The two most important approaches to handling concurrency are *ACID* and *BASE*. The letters ACID stand for **A**tomicity, **C**onsistency, **I**solation and **D**urability. Atomicity means operations are performed as a whole. Consistency guarantees the integrity of data (meaning defined constraints are obeyed at any time). Isolation

ACID vs. BASE

means maintaining the illusion of being the single user of a database when performing operations, essentially shielding concurrent transactions from each other. Durability in the end demands the persistence of modifications from now on until the data is changed again.

ACID provides a high level of security with respect to concurrent changes of a database. It ensures, that the results of modifications are meaningful in the real world. Providing such a high level of control however is hard to implement, even more so, if the database is distributed across multiple machines. A contrasting approach that addresses this issue is called BASE.

BASE (**B**asically **A**vailable, **S**oft State, **E**ventual Consistency) has a much lower demand on the result of operations. Changes to the database have to reach all nodes of a distributed database *eventually*. In the end all nodes should have the same data, if no further changes are performed. In comparison to ACID it trades performance against the level of consistency. ACID is usually found in relational databases, while NoSQL databases stick to BASE.

There are many scenarios where transactions and concurrency control in general are not important. If only a single user uses the database this is obviously no problem. If a database is read only, like in analytical databases, we also do not have to worry about this. If a database is append only, meaning we only insert new data and leave old data untouched, we can also give up write locks. Not having to deal with these issues results in a more performant system.

3.5.4 Security

Confidentiality

Security of data is a concern in most database systems. The keyword is confidentiality and describes preserving the integrity of the database in form of access restrictions to certain data. Generally this can be divided into three different scenarios. The first one is a database system open for everyone. We could think of a public search engine or a public phone book, where everyone can find everything. In most cases they are not really public, because they are only public in retrieving all data. Certainly there are limitations for inserting, modifying and deleting the data.

The second and more common scenario is a general protection of the whole database. There exists exactly one set of login credentials, which allow all manipulation on the data. Users can either access the database system or are restricted from it. Further distinctions are not made, thereby it is a rather puristic approach. Nevertheless it is sufficient for several database systems, especially if they do not perform the authentication within the system. Often an upper software layer is responsible for authorization and grants the distinct rights to the user by presenting him only information he is allowed to see.

Authentication

There is a difference between authentication and authorization. Authentication means proving the truth about a fact or an entity, in this case it is the identity of the person logging in. This is done against a protection system with a software

Authorization

layer, e.g. the database system. Authorization is the process of granting the access to certain resources. In a database this can be a certain entity, which allows the access from a specific set of users. Systems first require an authentication and then can use authorization for granting the correct rights.

Data Units

This leads to the third scenario which deals with the protection of certain data units. Here we have a set of users and a set of data assigned to units like a table in a relational database system. Rights are specified by defining associations between users and the data units and applying rules on them. E.g. a user is allowed to view one part of the data, but is restricted to see everything else.

CRUD Operations

It is important to distinguish between the different possible rights in form of user-data associations. Common distinctions are in rights for inserting, modifying, deleting and retrieving data. Often this is referred to as CRUD operations (**C**reate, **R**ead, **U**pdate and **D**elete). Being able to set different rights for one data unit, a lot of combinations are possible. Some users are only allowed to read the data, while other users can also modify it. There can also be combinations, which are generally not that meaningful. One can doubt that it makes sense for a user to have the right for creating and modifying the data, while not being allowed to read them.

Access Control Lists

A common example for these kind of assignments are access control lists (ACL). The concept consists of one list for every data unit, where all access rights are inserted. Table 3.1 shows a simple example for the three users Alice, Bob and Sam. While Bob has only read rights for the employee data, he is also able to modify customer data. This shows how rights can differ for every data unit and every user.

ACL is a positive list. It lists all rights, which were granted to a user for a data unit. Missing rights, which would indicate a negative list are not inserted there. A missing user entry indicates that the user is not allowed to access the data at all. A data unit however has to have at least one entry for inserting data, otherwise there would be no data all. The list is always accessed first for the data unit, looking up the user and granting the rights for the current session.

Data unit	*Access rights*
Employee data	
	(Bob, read)
	(Alice, read + modify)
Customer data	
	(Bob, read + modify)
	(Alice, read + modify + insert + delete)
	(Sam, read)

Table 3.1: ACL example

Another aspect is the possibility to grant rights to users. As new users are created within the system, they do not have any rights for data units. Other users can grant rights to the new ones, specifying which rights should be associated. The user himself must have the necessary rights before he can grant these rights to another one.

A further restriction is to regard granting itself as a right on its own. We thus extended the set of rights to the CRUD operation with one granting right each for operation. Users can have the read-right and are able to retrieve the data, but they can not grant this right to others. It is necessary to have the grant-read right to grant reading rights to another user. Furthermore he can pass the granting right to another user, such that he is also able to grant rights.

Access Control Types

Discretionary Access Control

A system that allows such a flexible assignment of rights is called discretionary access control (DAC). It is at the user's discretion who gains the rights for a data unit. There is no limitation for propagating rights because a user with rights for granting can always grant this right to any other user. An important fact is the identification of rights is always done via the user. A slightly different approach is the role-based access control (RBAC), where several users are combined into one role. In this system the role is assigned the different rights and the user inheriting this role inherits the access rights automatically. [RSRS98]

Mandatory Access Control

Mandatory access control (MAC) describes a different approach. The rules for access are set by the system and can not be changed by the user. The lack of a single user having all rights (trusted party) can be seen as an advantage here. As treating this topic would go beyond the aim of this section, we can refer to [SC96] for further reading.

Cryptography

Another way of ensuring confidentiality within a database system is cryptography. Cryptographic algorithms alter (encrypt) a piece of data, such that only the person who knows a certain secret (password) is able to read (encrypt) the original information. This means we theoretically do not have to restrict the access to this data anymore, as an unauthorized user is unable to read the contained information. In practice however we still do not want to have the data publicly available, as any cryptographic algorithm is potentially breakable.

The standard approach of using cryptography in database systems is to en- and decrypt data within the system itself. This means the user sends his data unencrpyted to the system and also retrieves it in plain format. It is only within the database system that an encryption takes place. The administrator of the system thus needs to know the password for these processes. Such a strategy helps against the theft of the database's files. However the administrator is the weakest point of the system as he has access to all of the data although he does not need it.

A more sophisticated approach is to encrypt the data using a key that is only known to the user *before* it is inserted into the database system. The user or the application he works with is responsible for this. The database can now be accessed by anyone as even the administrator is unable the decrypt the data. It raises however a number of new challenges. As the database system has to work on data it can not decrypt, it is for example hard to retrieve a specific piece of data. If a user wants to retrieve all documents which contain the word "mental" the system is not able to answer this query on its own. It has to send the data to the user first such that he can decrypt it. We can think of incorporating additional data structures such that we can limit the amount of information the database system has to send to the user. Such techniques are rather complex and still afflict the performance of a system heavily. For this reason none of the current database systems support this approach.

3.6 Performance [MM]

The performance of a database system is an important factor. Read and write throughput are the most significant figures. There are many techniques to optimize the amount of data which we can store or query in a given time frame. We briefly introduce the most common ones here. It always depends on the scenario and expected access patterns which techniques are beneficial.

3.6.1 Physical Layout

The physical layout describes the way how the data is actually written to disk. Often times there is a trade off between storing and retrieving the data efficiently. It is for example trivial to store newly inserted data by appending it to the old data. For querying it might however be better to have the data sorted by some attribute other than the insertion date. It is then be easier to find the desired piece of data if we are looking for specific values. Additionally we might want to compress the data before we store it.

Compression

Compressing data means reducing the required physical space for storing it. In the context of database systems we strive for a *lossless* compression, which means avoiding to lose any part of it in the process. Compression is mostly associated with saving disk space which by itself is not so much a performance, but more of an economic factor. However compression also increases (mainly) read performance of a database system. If we store data in a more compact fashion it is smaller and can thus be read faster from disk into main memory. However we have to decompress it again before we can use it. This might diminish the time profit of loading the data faster. When loading data from disk we get some computation time for free though. This is due to the fact that the loading process from disk to memory is done by the system bus[MRS08]. The processor has to wait for the next piece of data to be delivered and can meanwhile decompress the last piece it got.

The compression algorithm has to be fast enough to be computed within the mentioned time frame. Very simple strategies like variable byte length or gamma encoding are very fast to compute. They already provide a significant compression factor. Recent open source implementations of compression algorithms for this use case are Google's Snappy and LZO. Files compressed with Snappy are typically 20% to 100% larger compared to standard libaries like zlib, but the process is an order of magnitude faster[Sna].

3.6.2 Redundancy

In some cases it is reasonable to store data redundantly. Redundancy means keeping data in different forms simultaneously to support different access patterns. This gets feasible as disk space has become cheap as discussed in Section 3.2. Redundant storage is however always implemented at the expense of higher write costs. We have two write one piece of data multiple times and keep the copies in sync.

Keeping redundant data in sync means preserving the integrity of the data. This might be easy if data is simply stored in a different order. It might however be more difficult if data is for example aggregated. Having a collection of sales stored in a database system, an interesting query is how many items were sold per day. We can calculate this from the individual sales. If this query occurs all the time we might want to store its result redundantly in order to save computation time. If an old order is updated or deleted, we have to perform the aggregation again. Another aspect is the composition of data. Having stored all employees on the one hand and all of their positions on the other, a useful composition is to have all employees *and* their positions stored together. Doing all of this semantically correct and also efficiently is part of the research concerned with view updating.

3.6.3 Indexes

When storing data we typically have pointers to stored data. An (inverted) index maps values back to pieces of data, which contain these values. The index at the end of a book is a good example for this. A book consists of pages which contain words. The index of the book states for each (interesting) word all of the pages which contain this word. It allows thereby an efficient access to the desired information. Without the index we would have to scan the whole book page by page to find it.

In a database system we often look for specific information, e.g. the details of a given employee identified by his ID. Also when joining data we need support for this kind of access pattern. For example if we want to find all positions a given employee held at a company. Another typical scenario is querying a *range* of an attribute. An example would be to find all employees between the age of 20 and 30. A good index structure should support queries like this as well.

B-Tree

Tree structures are the most prominent type of index. Each node of a tree represents a value and has multiple children. If the tree is kept balanced, we can access and insert data in logarithmic time. Balanced means here that all leaf nodes of the tree have the same depth. The dominant data structure of this kind is the B-tree. A modification of this is the B^+-tree which stores data only in the leaf nodes. It enables efficient range querying by finding the first leaf node that is in the range and following the sibling leaf nodes until the end of the range. The B^+-tree is used in most file and database systems [SKS06].

Bitmap Index

Bitmap Indexes are an example for indexes which are organized as arrays. Having a table of data, a bitmap index indicates the presence of a value with a 1 entry and the absence with a 0. For every value which occurs in the indexed column of the table such an array is created. Queries on the column can then be implemented by binary operations, e.g. AND. This allows a faster lookup of data, is however only feasible for attributes with few values. If the value is e.g. a decimal number it might be unlikely that two rows have the exact same value.

Hash Table

Another type of index are hash structures. Data is stored in a hash table by computing the hash of the key and using this as the index of the table. A hash function is responsible for performing this computation. Hash functions map some input data to some output. The output length is fixed and usually significantly shorter than the input. If it maps two different inputs to the same output we call this a collision. To search for a given key, we again compute its hash. The corresponding row in the hash table contains the searched data. As long as there are no collisions we can find data in constant time. There are different techniques to handle collisions, one of them is linear probing. If a collision happens while inserting, we use the next free slot in the table. The lookup can still be implemented in constant time. [KE06]

Bloom Filter

Bloom Filters are an example for a probabilistic data structure. They determine the existence of data in a given data set. They consist of a sequence of bits of length m (initially all 0), called the *filter*, and a set of k hash functions. When data is inserted, the value is hashed by all k hash functions. The results of the hash functions address the bits in the filter. All of those addressed bits are set to 1. Checking the existence of a value works similarly. Instead of setting the bits to 1 though, we check if they are set to 1 already. If one of the corresponding filter bits is 0, the value is not contained in the data set. If however all corresponding filter bits are 1, the value *might* be existent. It could as well be a false positive, as the 1s could be introduced by other values, whose hash is the same as the current value. Using more hash functions lowers the chance of getting false positive results. On the other hand more hash functions have to be computed on insertion. In the end bloom filters can be beneficial, as they consume little space and allow a fast check for existence of data. They can thus avoid complete scans of a data set.

3.6.4 Caching

Caching means keeping data redundantly in temporary data structures in order to improve performance. There are different scenarios or levels in which caching is used in different ways. For example there might be a web cache in a company that stores websites when they are first retrieved. If a second access is done within a short time frame, the answer comes from the local web cache and not from the remote webserver. This saves network traffic and response time but might result in an outdated version of the website.

Caching in database systems and also operating systems is mainly concerned with the levels of data storage on a single machine. Access to main memory is very fast, but there will be more space available on an SSD. The SSD is slower than main memory but faster than an HDD. The HDD again offers more storage. The first choice is thus to keep everything in main memory. But if we have too much data this in not possible anymore. We thus have to store it elsewhere, e.g. on disk. The memory should still hold as much data as possible for faster access. But we have to make a decision about *which* data we keep there.

Caching Strategies

The strategy for caching read as well as write operations is crucial. For read operation we want to have as many cache hits (meaning the requested piece of data is actually in the cache) as possible. This requires an anticipation of what is going to be read next. A good assumption is that something that is read once will be read again. We thus want to cache everything we read. Of course at one point the cache will be full and we have to remove some data. A simple approach is FIFO (first in first out) which means removing the oldest entry. More sophisticated strategies are to remove the least recently used or the least frequently used element. Write caching is more concerned with collecting data and writing them all at once. This is for example beneficial if we write on HDD and can do sequential writes.

Multi-level Cache

Extending a single cache can get either too expensive or impossible due to hardware restrictions. Then it is often reasonable to use multiple caches in a hierarchy (multi-level cache). For example we could use the SSD as a cache for a HDD and the main memory as a cache for the SSD. HDDs themselves have a cache nowadays built into them in varying sizes. Some HDDs even contain an SSD for caching. On a lower level this is also done with the caches of a CPU and the access to main memory.

Caching data is harmless as long as no data is modified. If modifications happen, we have to update the cache as well. Otherwise we will read outdated data when accessing the cache. When we cache modifications, we endanger the durability of the database system. If we have not executed a modification yet and the system fails, we lose the modification.

In the end caching is necessary to offer some data intensive service in real time. Google's search engine for example is able to search through billions of websites and provide results within milliseconds. It also provides some snapshot of the website which increases the delivered data size. This is only possible due to massive caching as it would already take longer to just read the corresponding data from disk.

4 Approaches to Data Management

Based on the fundamentals presented in the previous chapter, this chapter now looks at the differences of database systems. It introduces the most common types and presents their models. This distinction is important as Chapter 8 compares systems of different types in a benchmark. The most simple model for storing data is the file system. The first section thus discusses the abilities and drawbacks of the file system and whether it is reasonable to call it a database system. The chapter continues with an introduction to the relational model and relational database as the most prominent kind of databases. In practice a so called impedance mismatch occurs when object oriented programming languages and relational databases come together. The next section discusses object oriented extensions to the model for bridging this gap. Subsequently the chapter introduces pure object oriented databases. It is furthermore assessed why they failed to capture the market. After a short presentation of graph and hierarchical models the chapter turns to aggregate oriented database systems. They are better known as NoSQL data stores and form the newest generation of database systems. They tackle concerns which classical database systems do not cope well with, namely rapid development and scalability. In this NoSQL section the chapter presents the different aggregate oriented data models key-value, document and column-family.

4.1 File System [CB]

The file system is a data store which we generally do not see as a database system. However the distinction is in our sense not that meaningful as all characteristics of database systems more or less apply to file systems as well. Actually every common database systems uses the file system of the operating system for storing their data. In this section we will investigate different file system models and see their differences and drawbacks in the context of database systems.

4.1.1 Hierarchical File System

The hierarchical file system stores data in a tree-structure. It stores the actual data in file, which are the leaf nodes of this tree. For structuring the tree it uses so called directories or folders. These directories can contain either several files or even other directories.

Additionally for every file there exists meta data describing it. The most important information is the file name which identifies the storage location within the file system. This file name is unique within one directory but not in the whole system. However the combination of the directory path (path from root node to leaf node of the tree) and the file name is again unique. There exist additional meta data to a file that contains attributes like the file length and the modification date. This can as well include supplementary models like ACL to introduce a security model to a file system.

The file system uses blocks for allocating space on a physical storage in a unit of even granularity. We have to determine this value before creating the structure on disk. This leads to e.g. blocks of 64kb size that the file system allocates. Storing files of less than this 64kb nevertheless requires this amount of space on disk. Literature refers the unused space often as slack space which is avoidable by choosing the block size based on the average expected file size. If the file exceeds the block size, multiple blocks are necessary to store the data. The file system connects these blocks by using pointers to the corresponding next block. File system fragmentation occurs when blocks for one file do not layout contiguously on the disk. This happens if there are no adjacent blocks available which can happen e.g. during the modification (extension) of a file. Depending on the physical storage format this behavior leads to slower access speed which makes reorganization (defragmentation) necessary.

Fragmentation

The file system uses the pointing mechanism mentioned before also when it moves files to another location. Transferring a file logically to a different location (e.g. a different directory) does not necessary cause a relocation on disk. Instead only a pointer is necessary that refers to the initial storage location. Dependent on the implementation of the file system this leads to a split between the logical location and the physical one.

XFS

The uni**X** **F**ile **S**ystem is one example of a hierarchical file system. It is currently the main file system used on a Linux distribution and is licensed under the GPL[1]. Block sizes are possible from 512 bytes to 64 kib. File names can have lengths of up to 255 byte and it supports the usage of ACL.

The directory structure of the XFS locates itself in a B^+ tree which allows a quick access to single files. It uses the 64 bits for addressing, which leads to a maximum file capacity of 8 exbibyte ($8 * 2^{60}$ bytes). A journal mechanism allows consistency of the data even on power loss or a system crash. This mechanism uses a journal file to write changes at first to this as a buffer (similar to the write-ahead log mentioned

[1]General Public License, one of the most widely used open source licenses.

in Subsection 3.5.1). The journal has a fixed size up to 128MiB using a circular buffer of blocks. On a crash the file system is recoverable using the data of the journal to reach a consistent state.

4.1.2 Semantic File System

The semantic file system is a different way for organizing files logically. We can consider an e-mail from a friend with the topic holiday. Putting it into the directory friends would be reasonable for finding this mail at a future point in time. Besides that there exists a directory holiday where we expect everything regarding holidays (e.g. travel providers, etc.). Storing this mail in this location would be reasonable as well and may even necessary. On a hierarchical file system there is no option to achieve this as we cannot find a hierarchy for this two places.

A semantic file system addresses this problem and solves it. It does not store files in fixed directories but assigns them to different tags. In this case the tags would be `friends` as well as `holiday`. Figure 4.1 illustrates the differences of both models. At the right hand side one file is reachable from different locations. The logical storage of a file is thus no longer location-based but has a semantic intention by the meaning or content of a file.

The file system stores in this case the file physically in one location and accesses it via index structures. These index structures allow the access via the tag or words and point to the underlying file. The challenges are to update a file and all its indexed locations on modifications as well as the random access of files. A good caching mechanism is necessary to achieve a good performance on HDDs which have rather slow random access rates.

Tagsistant

Tagsistant is one implementation of a semantic file system for the Linux Kernel. The user can apply a set of tags to all objects like files, directories, devices, etc. Afterwards he can use a combination of one or more tags to list all objects attached

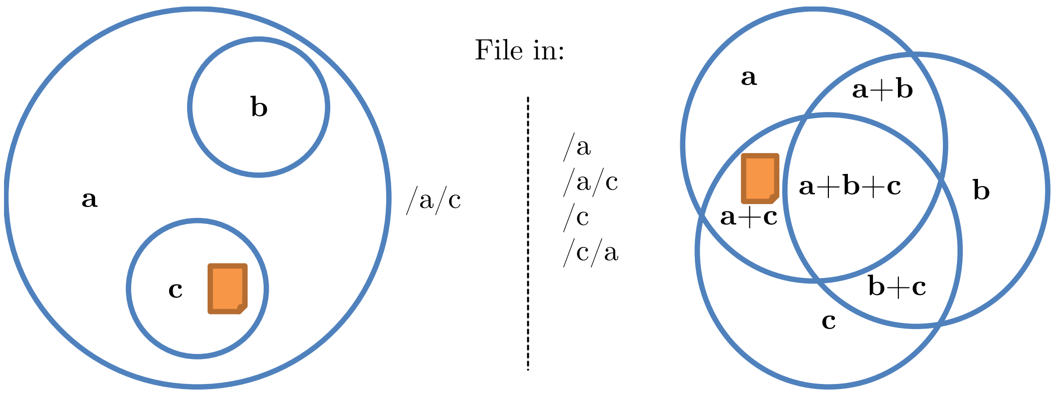

Figure 4.1: Comparison between hierarchical (l.) and semantic (r.) file system. The orange file indicates the file location and the paths describe ways to find it.

to it. We call this process querying. An auto-tag function allows the file system to tag certain document types like *html* or *txt* automatically. The tag repository of Tagsistant consists of a SQLite or MySQL database system for storing all its tags and location paths.

4.1.3 Distributed File System

A distributed file system is a shared and simultaneously used file system on multiple machines. Literature refers it often with terms like network file system or clustered file system which is describing the exactly same idea. This type of file system offers features like location transparency for accessing files which reduces the complexity in a network cluster. A parallel file system is a special kind of distributed file system which replicates the files across multiple servers to achieve redundancy and a performance gain.

Transparency

Generally transparency in multiple ways is the main aim of a distributed file system. Besides the before mentioned location transparency it provides access transparency such that the client remains unaware of using a distributed file system. Concurrency transparency leads to an equal view of the system for each participant and propagates all changes made in one place. The system should also be working after failures of one or more cluster nodes. A distributed file system achieves additionally several other issues for transparent replication or migration of files.

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a famous example of a recently developed distributed file system within the Hadoop family. The developers programmed it in Java and we cannot mount it directly into an operation system. Rather we have to access it via a Java program or the Apache Thrift API. A master node (called NameNode) is responsible for managing the heterogeneous storage cluster and storing the file system layout. The layout itself is a hierarchical file system with mostly immutable files that have the restriction that it does not intend concurrency. An advantage of the HDFS is the location awareness for the job scheduler of the Hadoop MapReduce (cf. Subsection 4.6.7) ecosystem. We call this component the JobTracker. It communicates with the NameNode to execute the computation on the same cluster node where the required files physically are.

4.1.4 Shortcomings and Notes

There are several shortcomings of a file system which we want to mention rather briefly. Generally the file system achieves a minimal data redundancy. This means it stores one file exactly at one place. Most systems do not provide any means for implementing redundancy. A distributed file system would be a possibility only in the case that we have multiple physical server machines.

We access data mainly by its the file name. More complex queries for retrieving data are mostly not available. This is a drawback in comparison to other (database) systems. Missing semantics can lead to inconsistencies in a file system. As files can be in principle any binary object there are only a few restrictions to store anything in one place. The common file system provides no possibilities for attaching semantic constraints as it is not the aim of it.

The focus on a file system is on a single file. Simultaneous operations of multiple users on a single file are not possible. We can estimate the disallowing of concurrent access as one major drawback. A file system generally does not provide transactions such that subsequent operations do not have the advantage of data isolation. Maintaining security is possible with ACL on a single file but not on a finer granularity within a file.

4.2 Relational [MM]

Relational database systems are the dominant database species ever since their development out of the the relational model in the 1970s. Only recently it is that other forms of data stores become increasingly popular under the NoSQL flag. The main idea is to model data in a way that supports queries in a declarative way. This means one states *what* shall be retrieved and not *how* it is done. In the following we describe the relational model and the foundations of relational database systems. There is additional information in the standard literature of Ullman et al. [GMUW09] and Stonebraker et al. [HSH07].

4.2.1 The Relational Model

Relation, Tuple

In the relational model data is modelled as *tuples* which are part of *relations*. A tuple is an ordered set of attribute values. A relation consists of a heading and a body. The heading is a list of attribute names. All tuples in the relations body follow the schema that is defined by a relations heading. A common view of a relation is a duplicate free table with rows (tuples) and columns (attributes).

Key-attribute Functional Dependency

A *key* attribute uniquely identifies a tuple within a relation. This is closely related to the term functional dependency. A functional dependency between attribute A_1 and A_2 states that every time a particular value occurs for attribute A_1 another particular value occurs in A_2. An example is the social security number (SSN). If a relation contains tuples with names and SSNs, the SSN uniquely identifies a tuple. That is while two people can have the same name, the SSN determines the name.

Normalization

Identifying functional dependencies is a key step to *normalize* a relation. Normalization of relations is done to remove redundancies. A simple example is a relation of people and their phone numbers. The relations head might look like this: (SSN, name, phone). We include the SSN because of the ambiguity of names. Today most people will have more than one phone number, e.g. one at home and one for their mobile phone. To store two phone numbers for a person we have to introduce two tuples, duplicating SSN and name. Each tuple contains one of the two phone numbers.

To normalize this relation we would split it into two. One relation containing all tuples of SSN and name and one which stores all phones for a given SSN. One tuple for each phone number. On the other hand we will have to *join* the two relations if we want to get all of the information back. This leads to the *relational algebra* which defines operators for relations, tuples and attributes.

Relational Algebra

The relational algebra is a language for querying data in the relational model. It is part of the theoretical foundation but is also used in relational database systems. The main operators for defining queries are set operations, projection, selection, cross-product and join. Set operators provide a union or intersection of relations. Projections extract particular attributes from a relation. Selection removes tuples from a relation which do not fulfill a given predicate. The cross or Cartesian product of two relations produces all combinations of tuples of the two relations. Finally the join combines a given tuple of a relation R_1 with all *matching* tuples from a relation R_2. Whether a tuple matches or not is determined by the type of join.

A join can be a *natural* join. A tuple would then match if it has an attribute with the same name *and* value. An *equi* join gets two attribute names as input and the tuples match if the corresponding values are equal. A *semi* join only tests a match and returns the left hand side relation's tuple on success. The *outer* join finally returns all tuples which do not have a matching partner in the other relation.

4.2.2 Relational Databases

Table

Relational database systems (RDBMS) use the relational model for storing and accessing data. They use however a different terminology. Relations have the notion of a head and a body that contains the tuples. Instead of this we speak of *tables* which have a *schema* of *columns*. A table contains *rows* that obey this schema.

SQL

The **S**tructured **Q**uery **L**anguage (SQL) is the interface of relational database systems. SQL is a standard and the ISO[1] as well as the IEC[2] is responsible for the development. However there is no relational database system that strictly implements SQL according to the standard. Each one deviates in several aspects which led to a variety of SQL dialects.

We can subdivide SQL into the data definition language (DDL) and the data modification language (DML). We use the DDL for defining tables and constraints. The DML covers the insertion and manipulation (modification, deletion) as well as the retrieval of data. The core idea of SQL is to formulate operations declarative.

DDL

The creation of a schema is the first step in using a relational database system. The DDL provides operations to create tables for storing data. A hierarchy of *catalogs* and *schemata* contain these tables. Creating a table starts with defining columns and their data types (numeric, character, temporal, array, etc.). A particular construct is the `NULL` value which can take on any data type. A common interpretation

[1]International Organization for Standardization
[2]International Electrotechnical Commission

is that *there is a value, but we do not know it yet*. The concrete interpretation is application specific though. We have to remark that the `NULL` value does bring problems in many cases. For example if we compare some specific value with a `NULL` value it is not clear what the result is.

Constraint

The next step in defining a schema is to declare constraints. Constraints can refer to a particular table or span across multiple tables. In the latter case they we call them *assertion*. Unfortunately they are not available in any major relational database system implementation. The former case enables constraints on the data within the scope of a table. The most important ones are primary and foreign key constraints.

Primary Key

A primary key of a table uniquely identifies a row. It can either be a single column or a composition of multiple columns. A value in this column (or a combination of values in the columns) must thus only occur in a single row of this table and must not be `NULL`. The primary key is most of the time also a good candidate for indexing as we access a table typically by the primary key.

Foreign Key

We use foreign key constraints in order to connect information across multiple tables. It enforces a subset relationship between two (sets of) columns from different tables. This means a value in the referencing column has to be present in the referenced ones. They play an important role in preserving consistency of data that spreads across multiple tables in order to reduce redundancy.

Check Constraint

Finally there are check constraints. We consider a table that might have a column *age* of type integer and a constraint that this age must be between 0 and 120. The relational database system has to fulfill check constraints at any moment in time. It will abort any operation that would lead to an inconsistent state with respect to the defined constraints.

DML

We use the DML to retrieve and manipulate data within the database. There are `INSERT`, `UPDATE` and `DELETE` commands for manipulation. We can retrieve data by formulating declarative queries. Formulating queries requires some rethinking in comparison to procedural approaches. Often times SQL allows a short and simple query formulation. On the other hand some types of queries are rather hard to formulate.

A query consists of three main parts that correspond to relational algebraic operators. The first part is a `SELECT` clause which corresponds to the projection operator. It states which columns the query shall retrieve to the user. Next comes the `FROM` clause where we state which tables we want to include in the query. We connect those tables by other relational algebraic operators like the cross product or the join. Finally the selection equivalent appears in form of the `WHERE` part. A simple query that is returning the names of all female employees could look like this:

```sql
SELECT name FROM employees WHERE gender = 'female';
```

Aggregates

The SQL query language provides a lot of additional features. It allows the calculation of aggregates like the sum of values in a column. We can also calculate them on grouped values, e.g. calculating the sum of employees grouped by gender.

Queries can also contain other queries as sub-queries for intermediate results. This makes makes the language very powerful. Window functions allow calculations based on subsequent rows which are otherwise not easy to achieve in a set-based access that SQL enforces. Usually the database system sets scope during query processing to one row at a time. Recursive statements allow the performance of an unknown amount of consecutive joins. This allows us for example to achieve a graph search.

Query Execution

The database system is responsible for executing the queries. This processing consists first of parsing the SQL query into a relational algebraic version. Then the system optimizes this by reordering the operators to achieve an efficient execution plan. Afterwards it chooses the best fitting implementations for the operators, e.g. a Hash-Join[1] for joining two tables. Finally the system executes the query. This approach provides advantages and disadvantages. On the one hand the user can formulate queries on a high level which saves time and effort. On the other hand he cannot control the execution. Meanwhile the database system most likely chooses a good execution plan as it knows a lot about the quantity and distribution of data. The system gathers these information by collecting statistics about the data.

ODBC

Cursor

ODBC (**O**pen **D**ata**B**ase **C**onnectivity) provides the means to execute queries from an external program and to retrieve the results. It is a standardized database interface (or middleware) and basically all products of major relational database system vendors support it. After establishing a connection we can execute a query and receive a *cursor*. This cursor allows us to iterate over the result set of a query within the programming language of choice.

Stored Procedure

Another part of the database interface is defining stored procedures. These are procedures that we can execute within the database system. They allow the retrieval using SQL but also a manipulation of the data in a procedural way. In contrast to the access from an external program we can avoid the network transfer of the data,. This is because we can perform the computations without leaving the database system itself. We can use stored procedures even within SQL queries. They can return single values as well as tables. Stored procedures may also make use of DDL statements or perform manipulations on a table's data.

View

A view is a named query. Creating a view means storing a statement and making it available in other queries. A *view* provides a new view on the data. For example a view called `management` on an `employees` table could return all employees who are managers. We can then perform queries that only include managers on this view. This saves the effort of first restricting the employees table on the managers within the actual query. The system has to recompute the result of a named query on every access though.

Materialized View

Some database systems support so called *materialized views*. They store the result of the named query separately. As we can as well compute the view's data from existing data, this is a redundant storage. If the underlying data changes we have to update the view accordingly. In case of the employees table someone might get

[1]Efficient join technique for checking on equality by using hashes.

promoted to a management position. We would have to adjust the management view then to reflect this fact. Doing this efficiently is one aspect of the field of view updating. The other one is to support the manipulation of views. This is difficult without knowing the semantics of an operation. If we would e.g. delete an employee from the management view it is not clear if this means that he got demoted or that he is fired? Thereby we cannot say for sure if the database system should change his position value or delete him as well from the employees table. There has been a lot of research in this area. Current relational database systems either do not support this aspect at all (like PostgreSQL) or only support cases where operations are unambiguous (like Oracle).

View Updating

Another way to realize the refreshment of materialized views are *triggers*. They are useful for much more though. A trigger is a rule that states: *if X happens, do Y*. X is typically a data manipulation operation on a specified table and Y an operation (in form of procedural code) that ensures the integrity of the database. A collection of triggers could e.g. maintain the `management` table. They would update it whenever a user performs an operation on the `employees` table.

Trigger

Beneath the database system interface the system has to physically store the data. Relational database systems support two kinds of storage: row-based and columnar storage. In the first case they store a table row after row while storing all columns of a given row sequentially. This is beneficial if we want to access all or most of the columns of a row within a query. For analytical purposes we often have to aggregate values of a single column of a table. For instance we want to compute the average age of all persons in a corresponding table. By having a row based storage the system has to skip a lot of data between the consecutive age columns which are of interest. In a columnar layout the system stores all values of a column next to each other. This allows a faster execution of such queries.

Physical Storage

Apart from the physical layout and the optimization of the sequence of operators in a query, relational database make use of other performance gaining techniques. They make use of caching and indexes. There are also different implementation of operators that vary in efficiency and utility with respect to the scenario. For instance a relational database system would prefer Hash-Joins over Nested-Loop-Joins[1] whenever possible. Part of the reason why relational databases are so successful is the management of transactions by maintaining the ACID paradigm. We already covered this briefly in Subsection 3.5.3 and thus mention it here again only for the sake of completeness.

ACID

4.2.3 Limitations of the Approach

The long time of research and successful business of developing relational databases has led to a multitude of performance as well as convenience features. This makes them a good choice for most applications. There are however some inherent limitations in this approach. Those led to the raise of the NoSQL era. The limitations

[1]Using two loops for comparing each row of table A with all rows of table B.

lie in the performance and scalability of relational databases. More precisely it is difficult to distribute a relational database system across multiple machines.

First of all SQL provides a lot of possibilities to define constraints on tables. In order to maintain these constraints, the DBMS needs access to *all* the data in this table. Otherwise it e.g. could not determine the uniqueness of a given value. Foreign keys even require the access to the data of other tables. This is difficult if we distribute data across many machines. The ACID paradigm defines hard constraints on the execution of concurrent transactions. Managing transactions across multiple nodes is hard to implement and expensive to operate.

The complex queries that SQL supports are also difficult to implement and manage across multiple nodes. Some functions are easy to distribute, like `COUNT`. All nodes can simply count the data they possess. The system can then add the individual results. Other operations e.g. `COUNT DISTINCT` are much harder to realize. `COUNT DISTINCT` counts the number of different values in a data set. We cannot implement this by simply adding individual results of different nodes.

There is typically a restriction regarding the distribution of a relational database in several ways. One restriction is to stick to replication. Each node contains all data and can therefore answer queries and accept modifications. The crucial point is to keep the different nodes in sync. Using replication allows to scale the performance of a relational database. However all the data still has to fit on one machine.

Another approach is to limit the use of features of the database system. Solutions like PL/Proxy[1] for PostgreSQL help in distributing operations across multiple instances. The instances themselves however are stand-alone and do not know about the other ones. Unfortunately we cannot define node-spanning constraints this way. We can distribute the queries across the instances. Afterwards we have to combine the results manually.

Legacy systems did not support a distribution functionality from the start but added them later on. Modern systems like SAP HANA consider it from the start on [Wor13]. The architecture is thus more clean and the query optimizer is able to cope with distributed queries. Nevertheless those systems are inherently complex and thus expensive to develop and maintain.

Aside from the problem of distribution we do not need the rich feature set of relational databases often times. Losing a piece of data might be okay sometimes. An application may even be able to cope with duplicate or corrupt entries. Even though the relational model is able to store virtually anything there are many scenarios where it is not very natural to store the data in a relational way. Having to define a schema first, hinders rapid development of prototypes. Often times the schema of data evolves or is very heterogeneous. All this encouraged the development of other database systems.

[1]http://wiki.postgresql.org/wiki/PL/Proxy

4.3 Object Relational [CB]

Many relational database systems offer functionality to work with the data in an object oriented matter. This feature extends the relational model with another paradigm and also extends SQL with new language attributes. Often we use object-relational parts of a general purpose RDBMS without explicitly knowing it. In this section we describe briefly the reasons why we often extend the relational model and how the differences look like.

4.3.1 Impedance Mismatch

OOP

Object-oriented programming languages (OOP) like Java or C++ are very popular nowadays. It is thus a common scenario to use a relational database system within an object-oriented programming language. In this combination we have to store objects into the database and also retrieve them from the database again. We summarize the conceptual problems and technical difficulties that arise from this under the term object-relational impedance mismatch. The term originated from impedance mismatching in electrical engineering where we have to transmit energy from a source to a sink with maximum power.

The first problem emerges because of the different type systems of object oriented languages and the relational languages like SQL. It is not always possible to find a one-to-one mapping between each data type. We then have to reconstruct missing mappings by using more complex data type combinations in the target system. There are also scenarios where this is not possible at all. Another paradigm difference arises from the navigational access via the use of pointers which are common in object-oriented programs. The relational system does not allow by-reference pointers, such that the mapping of objects to relational tables is quite complex. A good example is the difference of handling strings. Relational database systems typically limit strings in their length and store them in a fixed collation. Most object-oriented languages offer strings in variable (indefinite) length. They consider additional meta data e.g. of the collation only in certain operations like strings sorting.

Composition

We can compose objects in an object oriented language from other objects. This can lead to a high degree of dependencies. Unfortunately this is hard to model in a relational database system as it consists of flat tables with no direct hierarchical information. Relational database systems also cannot easily adopt constraints on those dependency hierarchies. Constraints using several relations and relationships are difficult to model declaratively on the relational side. It is even more complex since the system does not model constraints explicitly attached to objects. These constraints are for instance triggers that exist out of the scope of a table. We notice them often only when the system raises exceptions if a problem occurs. Inheritance as a special case of hierarchical dependencies and object composition is not a part of the relational model. Because of this we cannot realize this by using constraints.

Information Hiding

In object-oriented programs we encapsulate objects such that we can hide the (private) implementation for other accessing objects. Mapping such a private implementation to the relational database system makes it difficult to be capable of changing the implementation on the object-oriented side. This is even though the object stays the same in public. Furthermore we cannot transfer the private and public visibility of object methods and types to the meta data of the relational database system.

By looking at the manipulation of objects we can see differences as well in comparison to the relational model. SQL defines a set of a few well defined and limited operations to modify the data in different tables. In contrast we can attach complex manipulation functions to an object. We have to create the access to values of collections like hash sets by using custom functions. This is because there is no available access model for each object-oriented collection.

Transactions

Transactions are a key function in a relational database system while concurrency control is generally not part of an object in an object-oriented program. Atomic operations on objects are usually very fine grained such that it does not support concurrency control without using special mechanisms. E.g. we could borrow concurrency control from methods regarding multi-threading. Furthermore there is no sophisticated model for isolation. We can achieve durability in this case by using a relational database system. Unfortunately this suffers from the problems mentioned above.

4.3.2 O/R-Mapping

O/R-mappers are applications for persisting objects from an object-oriented program into a relational database system. They are capable of handling the conceptual problem of impedance matching. They are in a way a translator which takes information of the objects (commonly in forms of code annotations or specification files) and maps the instances to an existing relational schema. Some tools even directly create the schema out of the object specifications. The simplest case is storing each class as one relational table whereas each field corresponds to a column in the table. However this is not always possible. We noticed this in the impedance mismatch discussion previously.

The advantage of using an O/R-mapper instead of mapping objects for each program individually by hand is the use of common techniques for mapping objects to tables. One example is the mapping of the inheritance of objects into a relational database. There are two common techniques for achieving this. The first one is choosing a single table which provides columns for every field in the inheritance hierarchy. This table will be very sparse. The second one is creating class tables such that there exists one table per hierarchy. It saves the information of a specific field to the table of the parent class or the class itself. Another approach is saving everything in one table per concrete class. In this way we completely separate the data of the parent class and the child into two tables. The user of an O/R-mapper only has to specify the classes of the hierarchy and the type of inheritance model that he wants to choose.

4.3.3 Object-relational Database System

An object-relational database system is a relational database system with an object-oriented database model. Such relational database systems support classes, objects, types, functions and inheritance directly within the system. The main advantage is that we can extend their data model by custom functions and data types. We call them User Defined Types (UDT).

SQL Extension

The query language is also capable of working with objects, i.e. there exists an SQL extension in the standard (SQL:1999 standard). It mainly emphasizes on using structured data types (or structured user-defined types) which are basically the same as an object in the object-oriented world. Additionally it supports single inheritance (exactly one parent class). This corresponds to the inheritance model of most object-oriented programming languages like e.g. Java and C#.

UDT

User defined types allow navigational access to their fields even within the SQL syntax. Figure 4.2 shows the difference between the classic SQL approach (on the left) and the object-relational style (on the right). Navigational access allows us to reference fields of objects directly. This prevents us to use a normalized approach to store relationships. Additionally this strategy avoids using a join in a lot of cases. This leads to cleaner code and faster access depending on the implementation. Also Figure 4.2 shows custom functions on UDTs and their fields which enhance the possibilities of using more than only the standard built-in functions of the database system.

As mentioned before we can encapsulate relationships of objects into one single object. This allows a direct mapping from objects to UDTs and avoids normalizing entities and relationships. With the object-relational approach we can persist objects of object-oriented languages in a more natural schema. The mapping is closer to the object-oriented model but still has several differences because of the mixed languages.

```sql
SELECT C.surname || ', ' ||
       C.first_name AS name,
       A.country
  FROM Customers C
  JOIN Addresses A
    ON A.customer_id = C.id
 WHERE A.country = "Germany"
```

```sql
SELECT make_listentry(
         C.name,
         C.address.country
       ) AS entry
  FROM Customers C
 WHERE A.address.country
         = "Germany"
```

Figure 4.2: Object-relational SQL: navigational access and functions. On the left standard SQL, on the right SQL with object-relational extensions.

4.4 Object Oriented [CB]

Object oriented database systems allow to store objects from a programming language as a whole. They provide methods to store and retrieve objects from the database. The two main concerns are inheritance and associations of objects. In comparison to the object relational approach object oriented databases provide advantages, but they also suffer from some disadvantages. Object-relational database systems disassemble objects in an inheritance hierarchy and store them in multiple tables or introduce a multitude of `NULL` values. They translate the associations by using foreign key references to other tables. In order to retrieve a single object with all its associated objects, we have to access multiple tables at once.

Storage

Object oriented database systems use a more natural way of storing and accessing objects because they use a specialized model for this. They store an object basically as it is and introduce an ID for identification. It defines the object additionally to the subclass of a base class that determines its attributes. Other objects use this ID to reference this object. We can find an object by specifying its type and additionally filtering or sorting it according to its attributes. Furthermore we can avoid joins because we can access other associated object by following the pointers in form of their IDs. When we store a new object or modify an existing one, the system can lock the whole hierarchy which allows better concurrency control.

Distribution

In contrast to relational systems we can distribute object oriented database systems more easily. We can simply store objects that have no connection with each other on different machines. Relational database systems however struggle with table- and schema-wide integrity constraints. This makes distribution complicated or often unfeasible.

Object oriented database systems were never really established in practice. This might be due to the dominance of relational databases which also provide a number of advantages in form of their feature set. While the object model is closer to reality, the relational model is rather simple and well known. Furthermore the research of relational database systems is more advanced and they are highly optimized. Thus they outperform object oriented database systems especially if data and the relationships are simple [Nic07].

Relational database systems experience most of the time also a better support than object oriented database systems. For instance there are more user tools existent for relational database systems. But also the standardization is superior as it is more stable. The huge relational database system community also ensures better chances for a long term prevalence.

4.5 Graph and Hierarchical [MM]

The hierarchical database model is a very early (one of the first) database model that roots in the 1960s [Wika]. It closely relates to the file system that we have introduced in Section 4.1. The hierarchical database systems store data as records in a *one parent, many children* fashion. This model is simple but inflexible.

A more flexible approach provide so called network database systems. This model allows a more flexible way of storing objects and their relationships. It was standardized in 1969 but had little impact on the database system market [Wikb]. Nevertheless in recent years network database systems experienced an increasing popularity as so called graph database systemss.

Graph database systems allow storing and retrieving graph data efficiently. They support storing objects and arbitrary relations between them. Objects are in this sense the nodes and the relationships the edges that form the graph. Examples are social networks where people are the nodes and the edges represent relationships between them (like the Facebook friend status). Navigation systems use spatial data for storing places and planning routes between them. We can interpret this also as a graph. Another example is a knowledge graph where the nodes are concepts and the edges are properties which we share among the concepts. We can consider the two concepts *dog* and *animal*, which we interconnect by the *is a*-property. In this graph we store the knowledge that *a dog is an animal*.

Generally there are multiple ways to store graph data in an database system. E.g. it is possible to store this data in a relational database. One possibility is to store all edges in one relation which has two fields, namely *from* and *to*. This would implicitly model the nodes as well. Another way would be to store all nodes with additional information in one table and all edges with their information in another table and state their connection using foreign keys.

Query Language

With SQL and imperative languages it is not comfortable and not efficient to query graph data. Finding indirect connections between nodes (e.g. the friend of a friend) or computing all reachable nodes is complicated. In SQL for instance we would achieve this by using self-joins of the edge table. But if it is not known how many hops (connections from node to node) there are between the start and end node, it is not clear how often one has to join to find the target(s). There are recursive additions to the SQL standard (e.g. the `CONNECT BY` clause) which support formulating these kinds of queries. In current database systems they are not necessarily performant and are generally not easy to formulate. This means in general it is possible but not the best way to store and process graph data in relational database systems.

Graph database systems store the data in formats that are specific to this use case. As they do not use a general purpose storage type, current products can achieve an efficient layout and access of the data. There are also specific query languages that have a sophisticated design for retrieving graph data and formulating queries.

Currently there is no standard graph query language. Instead each system offers its own language for accessing the data. The languages are on the one hand mostly similar to SQL on the other hand very specific to answer queries on graph data.

A popular graph query language is Cypher of the database system Neo4j. Listing 4.1 shows an example of this query language. The system executes the query on the graph illustrated in Figure 4.3. This figure shows an acquaintance graph in which the *KNOWS* relationship interconnects two nodes. In this example we consider everyone who knows someone in a *friend of a friend* way to be in the acquaintance circle of this person.

The Cypher query returns all acquaintances of the person (node) *Marc*. We indicate this by the `START` clause which is pointing to a node with the corresponding name. The `MATCH` clause defines the connection type between the nodes. Neo4j will match every node with the *KNOWS* relationship. Furthermore the star right after the relationship name indicates that Neo4j should match all length of relationship chains. Without this star the system would only consider direct connections from *Marc*. The `RETURN` clause lists the results. Here it lists all acquaintances from the starting node. The distinct enhancements wipes out any duplicate entry which is similar to the distinct clause of SQL.

In comparison it is rather complicated to formulate this query in SQL. It is not possible to formulate a query with the SQL core syntax that uses an unknown amount of joins. This is the case in the acquaintance relationship between two persons in our example. The query has to utilize recursion which has to stop at the relationship most distant to the relevant person. This may be again rather inefficient. The undirected relationships cause that we need to investigate all concerning persons

Listing 4.1: Cypher language example

```
 START n=node:node_auto_index(name='Marc')
 MATCH n-[r:KNOWS*]-acquaintance
RETURN DISTINCT acquaintance
```

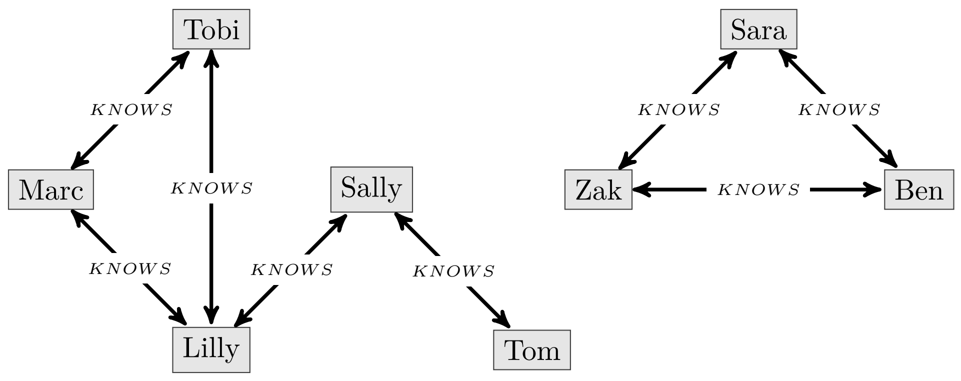

Figure 4.3: Acquaintance graph example

multiple times. This is because there is no limiting factor at the first glance. We can think of a stopping mechanisms in which we only advance if the degree of acquaintance exceeds a temporary threshold. This leads to a solution of this problem but not to a simpler query. Thereby it is possible to formulate the query in SQL with its extensions but it is more difficult because of the differences in the data model.

4.6 Aggregate Oriented (NoSQL) [CB]

NoSQL is a rather new approach for database systems. It is not a unified model but a set of models for use cases, where relational systems are not the best choice. Today's challenges make systems necessary which are different than the common SQL or more explicitly relational systems.

4.6.1 Challenges

Rapid Development

A fast program development requires a good workflow of implementing and modelling. It therefore makes a flexibility of the used systems necessary. The process of developing a fixed schema for data makes it hard to change that model frequently. There is always a cycle of changing the program and then changing the schema and transforming the current data if necessary. Most NoSQL systems do not require to determine a fixed schema before inserting or accessing the data. They tend to be flexible and are capable of handling any structure of data which arrives. It is important to note, that there is always a schema in a database system. However this schema is often implicit in NoSQL systems. This means the application needs to know for instance which fields of a stored objects are accessible and what data types describe those fields. Relational database systems store these information in their meta data, while NoSQL systems avoid to store this schema information. As the application always needs to know about the data as well as the database system this duplication tends to be an overhead when working with relational systems.

Implicit Schema

Aggregate Storage

Often it is the more natural way to store objects of the application context as they are. A good example is a shopping cart, where the system can store the cart itself and all products with prices, etc. as one unit. This makes it easy to work with the data at the first place. Relational systems with the often used normalization have a different approach where the data spreads across several tables. There the application has to collect it again for further usage. The impedance mismatch (cf. Subsection 4.3.1) is a result of this approach. NoSQL provides a different way for solving this problem compared to object-oriented or object-relational database systems. Often NoSQL database systems allow storing each object independently from the others with a unique schema.

Administration

The administration of a big, full-featured relational database management system often requires an extended knowledge of it. For this reason there exist jobs only concerned with this type of task. NoSQL systems are in general rather puristic and hence do not have that much administration effort. The easy setup and maintenance are also attributes of these type of systems.

Web-scale Data

The notion of BigData describes a phenomenon of gigantic data load for database systems. This implies that one single server computer cannot handle all the data and computations on its own. Web-scale data requests systems which can cope with distribution easily. Relational database systems were generally invented to work as a standalone instance. Most of the database systems offer a distribution functionality but they more or less attach it to the existing core. NoSQL systems are at front designed to work distributed by having simple but powerful mechanisms for handling multiple nodes in a cluster.

High Availability

As today's applications can grow really big the database systems have to cope with a lot of simultaneous work. Social networks like Facebook offer a 24/7 available service for millions of users with subsecond response times. Therefore NoSQL systems have to work highly parallel and therefore have a smart architecture to cope with that phenomenon. It is important to allow growing systems with varying data load. This requires easy to scale systems which again requires a different data model than classical database systems.

4.6.2 Storing Aggregates

NoSQL Term

The vague term NoSQL lacks of any meaning or semantics. Being precise NoSQL would include every database system which is not using SQL as its query language. But this does not describe this term in its actual meaning. The origin of NoSQL roots in a simple Twitter hashtag which denoted a conference about alternative database systems in the early 2000s. This means the term is not representative, but it is unchangeable and the definition does not arise directly from the title. Other notions say that NoSQL means *not only SQL* such that SQL stands for the old-fashioned properties of relational database systems. This could also include again relation databases with a shifted aim towards network architectures. The term of NewSQL comes up with this definition but we will not investigate it further in this section. However the term NoSQL represents all lightweight task-centered database systems that allow an easy distribution of their data. It does include all newer database management systems of the 20th century while excluding for instance key-value stores which where already present e.g. in the 1970s.

Aggregate Oriented

Martin Fowler shaped the notion of aggregate oriented databases [SF12] which is actually a better notion than NoSQL for the presented concepts in this section. It describes the idea that a database system should store data, which semantically belongs together, as a unit in the database as well. The before mentioned shopping cart is one example for this technique, other instances could be weblogs with comments or newpaper articles with images and layout information. For defining these aggregates, there is one simple question based on Fowler's definition: How should the data look like when I retrieve it? In relational systems we have to collect the data for one aggregate throughout multiple tables. In contrast an aggregate combines everything into one object.

This approach has certain benefits. First of all it tackles the relational impedance mismatch because the system can store objects basically one-to-one in its database. There in no need for conversions or normalizations and even the possibility to store the objects in exchange formats like JSON or XML exist in a lot of NoSQL stores. This also supports the rapid development of programs as there is no need to think about a complex schema for newly introduced objects. Objects do not introduce several new tables but we can simply write them and read them back as a whole. The aggregate oriented model makes distribution for NoSQL systems often a lot of easier. We can use one aggregate standalone because all necessary data is within this aggregate. This means that we can divide multiple aggregates across several severs as long as one aggregate stays on one cluster node as a whole. Sharding of aggregates does not affect other aggregates.

Distribution Benefits

Aggregate storage has also some drawbacks. This approach fits well when there is one view on the data or a subset of this aggregated data. The application needs to retrieve only one object which is straight forward and simple. Different views on the data are hard to realize in aggregate oriented systems. The shopping cart can serve as an example again. If the system should list all shopped products it has to retrieve this data from every aggregate in the database. In the worst case the system has to read all data to answer this query. This would require e.g. map-reduce steps to reorganize the existing data. We will introduce Map-reduce later on in detail (c.f. Subsection 4.6.7). To cope with this problem the system has to store the data accordingly to each view. This implies redundancy of the data as it is duplicated to create different aggregated views on the same set of information.

Aggregate Drawback

There exist other problems with NoSQL systems in comparison to relational database systems. NoSQL products are rather puristic database systems. They offer a less extensive feature set compared to the classical relational systems. This leads to the insight that we cannot tackle every problem with a single NoSQL database system. Every system is more or less designed for some specific use cases. For designing a complex system one might have to integrate several NoSQL stores to cope with different problems (cf. Polyglot Persistence in Subsection 4.6.6). The puristic approach also comes along with less constraining systems. Applications must ensure the integrity of data with only little or no support of the underlying database system. Also the less strict BASE consistency (cf. Subsection 3.5.3) and the lack of transactions confronts the developer with new difficulties. However there are NoSQL systems with atomic or transaction-like operations. The main difference is that the messing ACID mechanism does not solve most of the problems anymore. Instead the developer has to take care about this.

Puristic Feature Set

Relational database systems share in most of the cases SQL as their query language. In NoSQL database systems there is often only a puristic key-value lookup. There exist more complex system languages but they still contain only few operators in comparison to SQL. Some scientist have thought about defining a common database language for NoSQL systems [MB11]. But this does not actually cope with the NoSQL idea of task specific systems. Having a general language would

No Common Language

actually undermine this task-specific advantages of optimizing a database system for a specific use case. The approach also fails as there is no unified data model for NoSQL systems.

NoSQL database systems do not share one data model. There are several concepts contributing to the idea of flexibility and easy distribution of data. Nevertheless there are a few core models for certain use cases. The concrete implementation differs for each NoSQL system such that there is a unique model for each store. The following subsections introduce the main models of the NoSQL world.

4.6.3 Key-Value Stores

Key-value stores have the simplest model of storing a value under a certain lookup key. The key is either binary or a string and serves as a hash for saving and retrieving a connected value. Values can be primary data types or complex types. Primary data types are integers, strings, floats or binary raw data. Complex types are for instance JSON fields or sets and lists which allow specific operations for the retrieval. Even the systems only allow primary data types it is possible to store any object serialized as a string or raw binary data. Figure 4.1 shows an extract of a key-value store's data with different types as value. The difference is basically the advanced possibilities of manipulating and querying these values or only parts of these values. Otherwise the application itself has to implement the necessary operations. In general there are only a few possible operations like setting a value to a key, getting it back or deleting it.

Amazon Dynamo Paper

The Amazon Dynamo paper [DHJ+07] set the foundations for distributed key-value stores. The paper introduced a model for solving several problems with Amazon's shopping cart use case. It explained techniques for a super high availability including the eventual consistency idea, consistent hashing for data distribution and object versioning. It also introduced Dynamo, a new key-value store which uses this model and the paper explains its architecture.

Eventual Consistency

Dynamo's architecture consists of a server cluster with multiple nodes that organizes as a ring. The eventual consistency model contributes to the availability and thereby the reliability of this system. It means that after a modification the system will be consistent at some point in time. Every node will thus reflect the changes *eventually*.

Key	Value
Limit	4
Aaron	"{points: 4.8, interests: 'Sports'}"
Popular name	Karl
File	19 f2 b6 54 29 41 e2 7d 27 3f

Table 4.1: Key-value example

On a cluster node failure the system does not have to stop answering new requests because of the strict consistency model. This idea is implemented by a gossip based protocol.

Gossip

Gossip is similar to social conversation, where some information is distributed during one-to-one interactions of people. Each cluster node regularly (e.g. every 10 seconds) chooses another node to exchange their change histories. News are for instance nodes which are not reachable at the moment as well as new inserted or recurring instances. It is only necessary for a few cluster nodes to know about something and it spreads virus-like throughout the system.

Consistent Hashing

Consistent hashing is a special kind of hashing where resizing affords only a small number of reorganization steps. Dynamo uses this technique for partitioning and sharding the data among all cluster nodes. We can interpret the output of the hash function as a ring. Each cluster node takes randomly multiple positions on this ring. If the system retrieves a new item it hashes the key and scans the ring clockwise until the nodes position is smaller than the hashed value. The multiple positions within this ring contribute to a better division of the data. Furthermore each node on the ring is a virtual node and each real node is responsible for numerous virtual nodes. This ensures failsafe performance as the remaining nodes can take over the load if one real node crashes.

Vector Clocks

Dynamo uses vector clocks for synchronizing the keys on different cluster nodes. This is necessary because of replication. A vector clock is a vector of logical clock values for each item in the database. These logical clocks work like Lamport Timestamps[1] for ordering events throughout distributed systems. The system can distinguish if the object versions have a causal ordering or are parallel branches of it. In the first case the system can forget the older version and in the second case it has to order and arrange them manually. To ensure a good writing performance the synchronization follows a QUORUM-like approach (cf. Subsection 3.4.2) with only a few nodes necessary for writing. The system uses this vector clocks approach for retrieving the correct results on reading.

The advantage of key-value database systems is their simplicity that allows an easy implementation. Each operation is atomic only regarding one key which avoids complex transactions. As mentioned previously they have sophisticated distribution mechanisms for well scaling database systems. Use cases share the necessity of only accessing the data by its primary key. Problems occur in the lack of data relationship specification. Key-value stores do not have foreign key or similar semantics in general. The application has to realize references to other keys by itself.

[1] Simple algorithm for providing a partial ordering of events. We increment the counter on every change. When a piece of data arrives with a timestamp greater than the local one we apply this as the current one.

4.6.4 Document Stores

Document stores are another type of aggregated oriented database systems. We use them for structured data. The term document indicates this structure to be e.g. XML, JSON or BSON, the binary counterpart to JSON. This means that we do not consider real documents like PDFs, Scans or Microsoft Word files. Instead we think of any kind of serialization or exchange format. The main difference to key-value stores is the the support in storing more complex data instead of having simple value fields for primary data types. But there are also semantically differences.

One big difference is that document stores support operations for structured data. While most key-value stores can store basically everything within the data field, they do not specify any semantics about the content. Document stores provide operators for querying specific fields within the structured data. They also distinguish data types of the single fields which adds semantic to the data.

Most document stores offer to create indexes on fields in the data. This allows a quick access to one or multiple documents at a time. This access is necessary as there is no key field pointing at a value, which servers as an implicit index as in key-value stores. Often document stores offer to store different collections of items as a further distinction. This poses components of a schema like *tables* in the SQL world. However this is not necessary as there exists an implicit schema, but it supports a rough differentiation. Often times there exists an identifier item in the data such that an (indexed) access to this field is semantically not a big difference than accessing a key in a key-value store. Thus the distinction between key-value stores and document stores is often rather vague. One can argue about a specific product whether is is a document store or rather a key-value store without reaching a consensus.

The queries for the data use generally the same structured data format for their specification. There are special keywords for operators like *greater or equal* or aggregations. The structured data describes, like in SQL, the way the data should look like as an answer of the system. This answer is then a structured data item itself. Most document stores use the data format thereby for any kind of syntax of the database system.

Listing 4.2: MongoDB data model

```
{
        "id": "5711",                        "city": "Cologne",
   "country": "Germany",               "population": 1017000,
  "location": [50.957245, 6.967322]
}

{       "id": "4213",                        "city": "Bonn",
   "country": "Germany",               "population": 309869,
  "location": [50.737324,7.098541]
}
```

Listing 4.3: MongoDB query

```
db.locations.aggregate({
                         "avgPopulation": {$avg: "population"}
                       });
```

Listing 4.4: MongoDB result

```
{
   "avgPopulation": 663434.5
}
```

MongoDB Example

Listings 4.2, 4.3 and 4.4 show a query example using MongoDB syntax (JSON format). The data model describes two different structured items that represent two German cities, namely Cologne and Bonn, with some attributes. The system stores these object in this case in the *locations* collection. Formulating a query requires accessing the collection and specifying the kind of query. In this case it is an aggregation, but it could be as well a *find* operator for single item access. For the aggregation there exist several operators like the average function in this example. The system applies this operator to the *population* field. The result is again in JSON format and shows the previously specified field including the desired result.

4.6.5 Column-Family Stores

Column-family stores are the third type of aggregate oriented store. They arrange their data in tables. These tables are organized in column families, which are sets of columns, that are physically stored together. A column has a name and stores values of a certain primitive data type. This means the user can group several columns together as families based on the access patterns. If the application often retrieves certain columns together, the system can group them together on the storage level to improve read performance.

Tables contain a number of rows. Each row has a unique row key which the system uses to individually distinguish and access it. If a user inserts a new row with the same row key the system overwrites the prior row data. The row key is generally either a string or a binary value of variable length.

There exist three keys for one row: the row key, the column family and the column qualifier (the name of a column). The user needs to specify these keys to retrieve the value field of the row. This leads to the observation that the column-family store could be similar to a key-key-key-value database system. One can see this storage type as a more complex version of the key-value store. It is note-worthy that some column-family stores persist their data in single files per column family. This layouts in pairs of a combined key (row key + column qualifier) and a value.

Columns are part of the implicit schema of the column-family store, while the column-families are defined explicitly. This means an application has to know the column qualifier to access a single cell. Not every row has to contain the same set of columns compared to the other rows in the table. Furthermore columns with the same name can hold different types of data within separate rows. Column A can be an integer in row 1 whereas it can be a string value in row 2. Figure 4.2 shows such an example where Lewis' final points column is an string while the other rows contain double values.

Index Structures

Column-family stores offer two types of implicit indexes on the data, the row key and the column key. The row key is the primary used index which sorts the rows lexicographically by its string value. There are operations regarding the row key which allow range queries to this ordering. The second index is the qualifier of the column which is again sorted lexicographically by its name. In contrast to the row key there exist an separate index for each column family, because the system sorts these qualifier per family individually. As systems support range queries on this qualifier as well, they can utilize this index depending on the access pattern. Figure 4.2 shows this sorting behavior where the rows are arranged in order of the row key and the qualifiers of the first family's columns sorted by their name.

Distribution

Like all NoSQL database systems column-family stores can work in a cluster by design. A table may exceed the capacity of a single machine and gets distributed. Split points or functions determine how the data scatters across multiple instances. These functions can for instance be hash functions that use the row key as their input. Generally systems do not split a single row such that the single row access is fast as it avoids network traffic. This supports the aggregate oriented approach of keeping aggregates together despite distribution.

Google BigTable

As Amazon Dynamo was the starting point for the NoSQL key-value stores, Google's BigTable paper [CDG+06] set the foundations for column-family database systems. They presented the resulting BigTable database system as a "sparse, distributed, persistent multidimensional sorted map". A composition of the row key, the column family and the column qualifier forms the access to the value fields. Additionally the BigTable uses a timestamp for accessing a field. It stores multiple versions of one table row such that an additional reference is necessary and enhances at the same time the functionality.

Row key ↓	***class entries***		***interests***	← column family
	entry test	***final points***	***topic***	← column qualifier
Aaron	4	3.9	sports	
Karl	9	4.3	computers	
Lewis	10.1	no value	theory	

Table 4.2: Column-family example

For distribution BigTable organizes its row ranges in so called tablets. These tablets are split points for distribution and the system defines them to achieve a good data locality. This means range reads of data should include as few server nodes as possible in the cluster. An example are webpages of the same domain that form together one tablet. The Google File System (a distributed file system, see Section 4.1) is responsible for the distribution of the data files. BigTable stores its meta data and the table data files into this distributed file systems which is responsible for the organizational part.

BigTable organizes the data files in a B^+-tree-like structure that consists of three layers. The first layer contains the master meta data information and is called root tablet. It stores information about the location of all tablets and is never split such that the tree does not exceed three layers. On the second layer there are again meta data files which are basically the same as the root node. Yet the system can split those tablets where each row contains the location of one user tablet. This three-layer construct can address up to 2^{34} user tablets [CDG+06]. A user table is simply a tablet on one cluster node which stores part of a concrete table.

Compactions

The BigTable system persists the data on the disk, but new data is cached in main memory at first. While this data resides in volatile storage, log files ensure persistence on this level. As soon as the used memory exceeds a certain threshold the system transfers the data to disk. This is called minor compaction and leads to a reduction of main memory as well to a reduction of the log file. There exists a bound of compaction files on disk for BigTable because the system has to perform read operations on multiple files. This behavior leads at some point to a reduction of the performance. After the system has written multiple files to disk it performs a merging compaction. This compaction merges two minor compaction files into a single file. There is a method called major compaction which processes merging compactions until there is only one file left.

BigTable uses compression for improving disk I/O and processing time. We discussed this topic already in Subsection 3.6.1. The used Snappy compression allows compression and decompression rates of 400-1000MB/s on modern hardware. As additional index structures BigTable is using Bloom Filters. We explained this filter technique in Subsection 3.6.3.

4.6.6 Polyglot Persistence

Traditionally developers use relational database systems for everything. These systems are designed to cope with nearly every problem. There are numerous solutions to adjust a problem to the relational world. In contrast the functionality of NoSQL systems is rather puristic while relational systems offer a wide variety of extensions.

Special Purpose Systems

NoSQL database systems are special purpose systems. There exist several systems for solving a concrete use case but do not support others well. This means often it is not possible to find one system to fulfill all needs within a bigger-scale project or application. Moreover often it is not the best solution to use one database system for two use cases if it fits only to the first one but does not support the second one well. The aim is to find the best database for a single problem and a set of database systems to solve the whole use case.

Polyglot System

This leads to the term polyglot persistence where polyglot refers to the multitude of languages for different database systems. The term originated in an article by Martin Fowler in 2006 [Fowb]. It is the idea of using numerous database systems together in a bigger-scale system. The system integrates each database management system to solve a subset of the problems in the overall system. Altogether the single systems need to work hand in hand for solving the overall task. A good example is an online social network where a graph search for connected people requires a different database system than the user chat. For the former one a graph database system could be a good choice while the second one maybe requires a fast in-memory key-value store. The different aspects of the use cases require different things from a suitable database system.

Integration

When using multiple database systems it is not sufficient or even possible to use the management tools of the single systems. There is a need of orchestration such that these system work on one task together. There are several tools which avoid implementing a custom solution for each polyglot project. One open source example of such an integration framework is Apache Camel[1]. This framework offers describing life cycles for tasks and different connectors for database systems. It uses among other things a domain specific language in Java for implementing those parts. There exist also graphical solutions like the Talend Open Studio[2] which offer the process definition by drag-and-drop. These products come along with connectors for the most common database systems and often support the creation of custom ones. Another approach specific for querying the data of multiple data stores is Apache Drill[3]. This idea originated from Google's Dremel [MGL+10] that is a distributed query engine for interactive query evaluation. It uses a tree like structure for parallelizing queries where at the leafs there are connectors for the different database systems.

The polyglot approach comes along with several problems. It can be difficult to administrate one database system but it can cause even more maintenance effort for multiple systems. As NoSQL systems are rather puristic the complexity per database system is not that high compared to relational systems. Nevertheless there are multiple setups and multiple systems to update in a polyglot environment. The update process requires orchestration on its own. Furthermore the system has to control backups for multiple systems where it has to adjust the strategy to work with each data store. Knowing the products requires more versatile database administrators who have to be experts in several products.

[1] Apache Camel https://camel.apache.org/
[2] Talend Open Studio http://de.talend.com/products/talend-open-studio
[3] Apache Drill https://incubator.apache.org/drill/

The consistency between the data of different systems is also difficult. It is necessary that all modifications of the data propagate to every database system. Time deferred changes of the multiple system can be dangerous as queries could yield different results with respect to the accessed data store. As every database system has its own data model, the developer has to fit the overall data to every type. The integration of numerous data models is the most difficult problem regarding polyglot persistence.

4.6.7 MapReduce

MapReduce is a programming model for computing distributed queries in a multinode network cluster. NoSQL systems often provide a connection for applying MapReduce jobs on their database. Google fellows Jeffrey Dean and Sanjay Ghemawat described in their paper in 2004 [DG04] the algorithm for the first time. The common functional programming functions `map()` and `reduce()` inspired the idea for this approach. The most popular implementation is Hadoop of the Apache foundation. This is an open source product for distributed bulk processing that is written in Java.

Hadoop

Hadoop offers a whole ecosystem for working with the MapReduce paradigm. The core consists of job trackers and data nodes which are available on each cluster node. A centralized master node coordinates the execution of the distributed queries. The **H**adoop **D**istributed **F**ile **S**ystem (HDFS) is also important as a file system which organizes the data in the Hadoop cluster. We mentioned it earlier as an example in Section 4.1. There are system extensions with HBase as a NoSQL database system or Hive and Pig as relational data wrappers on top. [Lam10]

Distributed Processing

Figure 4.4 shows an example workflow of a distributed query. The system splits the accessed data via a custom defined split point (e.g. *per file* or *per text line*). A Mapper then processes one or several splits. It is basically a function that gets an input and maps this input to output results consisting of key-value tuples. The system aggregates for each mapper all tuples via the key, sorts them and stores them to the HDFS. The system transmits these files then to one or a few Reducers over network that process these intermediate result to final results. A Reducer is again a function which gets tuples consisting of a key and a set of values and outputs key-value result tuples. The system stores to final result, which is one file per reducer, to the HDFS again.

Data Locality

The main idea of MapReduce is the efficient exploitation of data locality and reduction of network traffic. To achieve this the system pushes the processing algorithms to the data. The job tracker of a node gets the instructions from the global job tracker. It contacts the data node and basically tells where the data is. It then places roughly one mapper to each involved data node. Each data node can process the data locally and thereby overall in parallel. For achieving low network

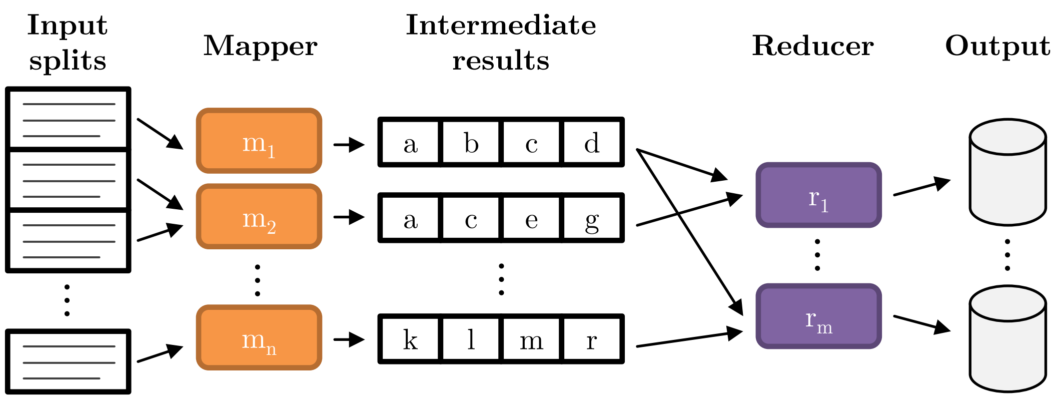

Figure 4.4: MapReduce processing example

traffic the intermediate results should be as small as possible. For this it is possible to pre-reduce the data locally on each data node before sending them to the Reducers.

Wordcount Example

One MapReduce example is word counting in texts. There are several data nodes with text files which map locally to each node. The mapper functions get the text line by line (splits) and splits the line into single words. As an output the Mapper sends out `(word, 1)` tuples. The system combines (or aggregates) these tuples after the map phase to `(word, (1, 1, 1, ...))` tuples. It then sorts the tuples lexicographically by word and sends the results to a single Reducer. This reducer receives these tuples and sums the second value (ones) up. It outputs as a final result `(word, #occurences)` tuples which solves the task of counting words in texts. The system does not sort tuples after the reduction phase. This means to sort the results e.g. descending on the number of occurrences a second MapReduce algorithm has to exploit the Mapper's sorting phase. The new setup consists only of a single Mapper with a custom sorting algorithm (sort by the number of occurrences) and one Reducer that only passes through the input tuples. This method enhances the word count with a sorted result set.

Aggregate oriented database systems can utilize MapReduce algorithms to create different views on their data. We can consider again the shopping cart example where we store all user information and the purchased products as one aggregate. A new view shall show the amount of purchases for a specific product. To achieve this we have to browse through every shopping cart aggregate for this product. The mapping function would receive each time one shopping cart and outputs the tuple `(product, amount)`. The reducer can then sum up all amounts for one product, output `(product, totalAmount)` tuples and create thereby a new view with redundant information.

5 Data Storage Market Overview

We already looked at the general approaches to data storage in Chapter 3 and the main database system models in Chapter 4. Now we want to close the circle and take a look at actual available software products. This chapter starts with an overview of the database system market in Section 5.1. It introduces some of the most popular databases systems while trying to cover as many different models as possible. Additionally it divides the market according to other features like the physical storage format and the licencing model.

The second section takes a closer look at the four database systems we found the most interesting. The first one is the EventStore, a puristic relational database system focused on write throughput. The second is HBase, a column-family database system based on Google's BigTable paper. The third is PostgreSQL, one of the most popular open source relational database systems with a vast feature set. And last but not least Redis, an in-memory key-value store. The section introduces the systems and relates them to previously introduced aspects of data storage. It also describes particularities of the systems and gives a rough impression of how to work with them.

5.1 Market Overview [MM]

The database system market is vast and hard to get a handle on. It contains numerous systems for every imaginable scenario and each one promises to be the best one of them all. We decided to focus on the most popular systems while still reflecting the heterogeneity of the market. The ranking of database systems (DB-engines ranking [sol]) provides numbers on the popularity of systems. Based on this we selected some systems as representatives for each database system type we discussed in Chapter 4.

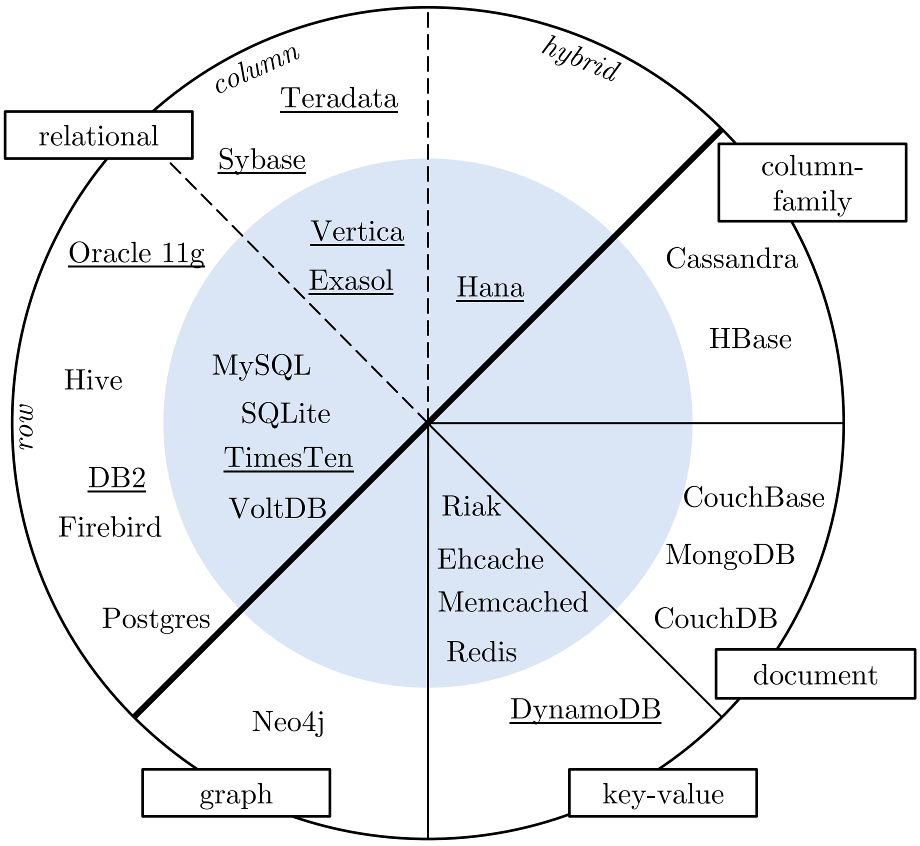

Figure 5.1: Database system market overview. Underlined products are proprietary, blue inner circle indicates in-memory technology

Figure 5.1 shows a graphical representation of our database market excerpt. It is firstly divided into relational and non-relational database systems. Within the relational database systems it differentiates between the physical storage format (row, column, hybrid). We divided the non-relational database systems into column-family, document, key-value and graph stores. The circle in the middle contains all systems that are able to hold all data in main memory. Underlined names mean that the system is proprietary and not open source.

Classic relational database systems are Oracle's 11g and IBMs DB2, which are both proprietary. There are however many open source implementations, above all MySQL and PostgreSQL which both have a huge user base. Examples for columnar database systems are Vertica or Sybase. SAP's HANA offers both storage formats and keeps its data in main memory [PZ12].

Even though the NoSQL movement is rather young there are already lots and lots of products which are mostly open source. The most popular column-family database systems are HBase and Cassandra. CouchBase and MongoD lead the document stores. DynamoDB is the only popular proprietary key-value store we present here. It originates of the paper in which Amazon introduced that built the foundations for many other products. A widely used graph database is Neo4j.

5.2 Database System Selection [CB]

The database system market is too big to allow a closer look at all systems. Even if we restrict ourselves to the most popular ones it is still not feasible to investigate all of them. We chose four database systems by their type, popularity and also our personal preference for further inspection. The following subsections introduce them in more depth.

5.2.1 EventStore

The EventStore is a database system for event processing. It is developed by the database research group of the University of Marburg. The system is an embedded database, which is implemented in and connected to the Java programming language. It is part of the JEPC stack as one of the default storage types.

JEPC

Java **E**vent **P**rocessing **C**onnectivity is a middleware supporting event processing in Java [Dat]. It is an open source project that provides a unified API and query language for processing events. These events arrive at the interface as continuous data streams of a fixed number of sources. There furthermore exists a fixed amount of queries that work on arriving data in form of continuous static computations. JEPC makes it possible to react on incoming events by having a set of operators like pattern matchers, aggregators and filters. The higher-level name of these operators is Event Processing Agents. A pattern matcher filters a series of events by defined search specifications. Aggregators take the input and compute an aggregation based on operators like the sum. Filters reduce the input by applying user-defined boolean expressions on the given input. Operators allow chaining in order to compute a final output out of the input stream. A group of Event Processing Agents form an Event Processing Network that serves as a reusable components in the event processing model.

The goal of JEPC is to allow the specification of these processes independent of the underlying event processor or store. To achieve this, it provides a decoupling of the event processors from JEPC. It is then possible to plug in different processors into the system. JEPC translates its schema and specifications to these processors by using connectors. These connectors are available for a small number of systems, but are also extendable by custom connectors. The connectors have to implement certain interfaces and follow the specifications. They also have the notion of Bridges in the JEPC context. As the default setting JEPC uses an in-memory storage engine for caching and processing events. The EventStore is a free to use persistent storage engine. Furthermore JEPC utilizes JDBC for plugging in any ODBC-compliant database system.

Realtime Scenario

A realtime scenario is the primary use case of the EventStore. This means the data arrives as soon as the source systems produces it. A good example is the live evaluation of log files of numerous system components. Each component directly forwards its logs to the EventStore which targets a high write throughput for being capable to process much data. The EventStore executes the queries frequently but on a rather small set of data to notice peculiar behavior.

Streams

The EventStore organizes its data as streams. Each stream is a B$^+$-tree which the EventStore stores in two files in the file system. One consists of meta data of the schema and the other one contains the data itself. The schema is relational such that each stream represents a relation and each event corresponds to one entry. Each stream restricts its events to the same schema which consists of a fixed number of attributes and corresponding data types. It only supports primitive data types, namely byte, int, long, double and string. Strings are not variable character arrays, but have a fixed size length. The user can specify this size up to 2000 characters.

The B$^+$-tree uses the insertion timestamp as its key and sorting attribute. There are two modes for treating the events: as instantaneous events or time range events. The former one uses a single timestamp while the latter one uses a tuple with start and end time. The system can create secondary indexes on other schema attributes and utilize them in the query process. Additionally, the EventStore compresses the events. For this it uses LZ4 compression which leads to a small data size and better CPU utilization. This compression compensates the overhead of unused space in string due to their fixed size.

Append Only

The EventStore allows only append operations on the data of one stream. This means that it can only store newer data based on the insertion timestamp. Being more precise this disallows any update or delete modification of the data. This restriction enables the fast creation of the B$^+$-tree, because no re-organization is necessary. This approach leads to a write-optimization of the EventStore, because this constraint does not improve the read performance. The user has to specify the size of a stream in terms of entries or disk space. After data exceeds this boundary, the EventStore deletes 10% of the old data from the stream.

API

There exists a simple API with only a few operations for data definition and manipulation. Data definition allows to create new streams where we likewise have to specify a schema. Additionally, it is possible to suspend (close) streams if we do no longer actively use them. Closed streams reopen automatically on the next access. There is an additionally method for creating a secondary index on one of the attributes. For data manipulation the EventStore has one procedure for inserting an event in form of a tuple. It has three operations for querying the data. The first returns all events of a stream while the second one returns events in a time range regarding the insertion timestamp. The third operation allows querying on other attributes of the schema. These queries utilize secondary indexes if we have created them beforehand.

XXL Library

Because the query capabilities of the EventStore are very limited, one can make use of an existing library to enhance the functionality. The e**X**tensible and fle**X**ible **L**ibrary (XXL) of the University of Marburg [vdBDS00] is a Java library containing database system operators. It consists of a cursor algebra and a package with aggregating functions. This means we can realize filters, aggregations and more complex operations like joins by using this library in combination with the EventStore.

Parts of the core functionality of the EventStore depend on the XXL library as well. Thereby it is necessary to use this library and at the same time it is available for further usage.

5.2.2 HBase

HBase is part of the Hadoop zoo consisting of Hadoop MapReduce, Pig, Hive and other tools. It is a column-family store and we can consider it as a Java implementation of Google's BigTable. Being a column-family store HBase stores data in tables, which users have to define explicitly. Tables consist of columns and rows. HBase groups multiple columns in column-families and stores and accesses them as a unit. The columns themselves are not part of the schema definition. We can create them ad-hoc and they thus do not to be present in all rows. All entries also contain the timestamp of their insertion. A lookup of a single value in HBase therefore is $(Table, Row, Family : Column, Timestamp) \rightarrow Value$. This subsection explains the inner structure of HBase as well as the usage from an application developer's perspective.

Architecture

The main parts of HBase are the API, the Master server and Region servers. This subsection describes those three aspects of the architecture. The API provides the means for defining and accessing tables. The Master Server coordinates an HBase cluster. RegionServers are responsible for managing a part of the data of a cluster.

API

HBase exposes its functionality through three different APIs: command-line, RESTful and Java. We can use the command-line by design mainly for short interactions of a human with the database. Applications either use the RESTful or the Java interfaces. RESTful interfaces make use of the HTTP protocol that is one of the foundations of the World Wide Web. Because of that they are flexible and obtain mostly a well support, but they also have high overhead. The main interface of HBase is therefore the Java interface. It consists of a library that provides all required functionality to define and access the data. HBase realizes these functions as remote procedure calls (RPC) and executes them on the cluster.

Table Definition

The definition of a table in HBase consists of a name and a set of column-families. A column family also has a name and additionally a set of options. One option allows the specification of the number of version, which HBase stores for values inside of columns in this family. We can also specify whether the system compresses this column-family and if it should keep it in memory with higher priority than others. HBase does not care about the content of the data that it stores. All values in cells are just byte arrays and the application that accesses them has to know the semantics. It also does not provide any transaction semantics like ACID. HBase guarantees the access to a single row to be atomic though.

Put, Delete, Get

The Java library provides Objects to store (`Put`), delete (`Delete`) and retrieve (`Get`) data from a cluster. Modifying a piece of data is the same as storing it. The system overwrites the old data. When we delete something the system remembers this by using tombstone markers. It does not actually delete the data, but it only marks it as removed. When retrieving a row via a `Get` object, we can specify the columns which the system shall return. If a query does not request any columns of a given column-family, the system can skip the whole corresponding file.

Table Scans

The previous methods address single rows only. A query that returns multiple rows must use table scans. A `Scan` always corresponds to one specific table. Additionally, it can contain the definition of a start and a stop regarding the row key. All rows whose keys are greater or equal than the start and smaller than the stop row key are part of the result. The row key design is crucial to achieve such range queries. It has to contain the right information in the right order to allow meaningful queries. Common practice is therefore to design the row key while having the expected access patterns in mind. Apart from this, as for `Get` objects, specifying the requested columns avoids the retrieval of unneeded data. The result of a scan is *not* a snapshot of the data as it existed at the time of the query. If changes in the data happen while the system gathers the result of a scan, they might occur in the result.

Filter

`Filter`s provide a possibility to remove undesired rows from a scan before they make their way to the client. There are standard Filters, like `RowFilter` and `PrefixFilter`, which work out of the box. Custom Filters have to implement an interface and be bundled as a jar file and then deployed on all Region servers. This hinders performing ad-hoc queries. A scan may include multiple Filters in a *must pass all* or *must pass one* fashion. More complex combinations of Filters are only possible by implementing custom Filters.

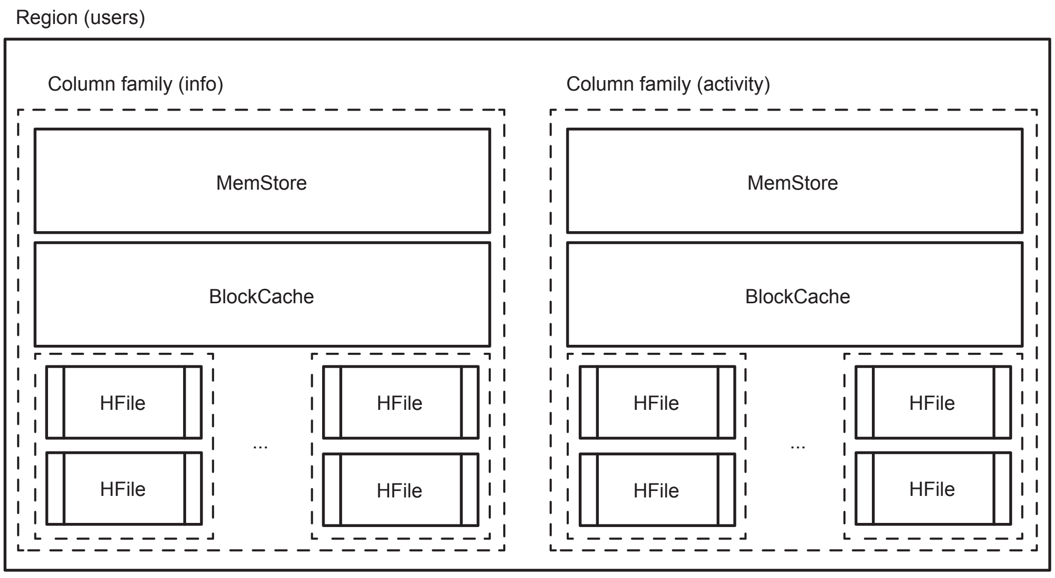

Figure 5.2: HBase region according to [DK12]

Regions

As tables in HBase grow in size, the system splits them into regions. We can either let the system decide about the split point or specify it manually in order to control which part of the data HBase stores as a unit on a single machine. A region server hosts the part of the data in an assigned region. It consists of several components which we can see in Figure 5.2. When we insert data, the system stores it in the memory-based MemStore. It guarantees the durability of the data by also writing it to a Write-ahead Log. Once the data in the MemStore reaches a certain size it gets flushed to disk as several `HFiles`, one file per column family. An `HFile` is a file that stores data as key-value entries. The key consists of the row key, a column qualifier and a timestamp. When the number of HFiles gets too large, so called compaction operations merge them.

Compaction

HBase distinguishes between minor and major compactions. Minor compactions fold multiple HFiles together such that there are fewer files to read on access. Major compactions operate on all HFiles of a column family in a given region and merge everything into a single file. They also clean up deleted records which the system marked previously by a tombstone. Major compactions are more expensive operations than minor ones. They block the access to the table for the time HBase performs them.

HDFS

HBase typically uses HDFS to persist data although it can operate on the regular file system as well. HDFS provides reliability and availability (cf. Section 4.1). To make use of data locality an HBase node usually also serves as an HDFS node. Having access to the HDFS is the only requirement though.

Considerations and Experiences

Tall-Narrow vs. Flat-Wide Tables

Two common table designs either use many columns and few rows (flat-wide) or many rows with few columns (tall-narrow). For example we could have a table with one row per user and many columns describing the user. Or we could have multiple rows for a user, each containing only few attributes. In HBase both variants yield the same storage footprint as the HFiles store key-value pairs and both cases store the same amount of cells. Rows are however never split across regions. Putting everything into a single row thus ensures that HBase stores the data the same location. Access to a row is atomic and thus guaranteeing a consistent read of a row's content. [Geo13]

Hotspotting

We should equally distribute data across the regions. If a regions A has to handle too much load, while others only handle few, we call region A a hotspot. We can use the row key for splitting a table into regions. Intelligent row key design combined with a manual split policy can avoid hotspots. Possible techniques are a randomization or hashing of a row key to achieve a good distribution of the data.

5.2.3 PostgreSQL

PostgreSQL is an open source relational database management system which also provides object-relational extensions. Its history roots back to the 1980s as a split of the Ingres project by Michael Stonebraker. The Post-Ingres project released its first stable version in 1989 as PostgreSQL, extended itself with an SQL interpreter and changed to open source in 1994. The change to the name PostgreSQL happened in 1996 with the version 6.0 and since then continuous development has taken place. PostgreSQL is written in C and is released under its own PostgreSQL license. The current version is 9.2. It has a vast community and a lot of participants all around the world.

Data Types

As PostgreSQL is a relational database management system it is interesting that it offers an extensive set of data types compared to other relational database systems. Because of the conformity to the SQL-2008 syntax, it preserves primitive types like numbers, varchars and dates and also new data types like intervals. Furthermore, array types and sets are part of the supported SQL-types. Since PostgreSQL has object-relational extensions, the user has the possibility to specify user-defined types (UTD) to create object like types in the SQL world. We explained the reasons for supporting user-defined types in Subsection 4.3.1.

HStore

In addition to these standard types PostgreSQL offers additional ones which are helpful in specific use cases. The HStore data type allows storing a map of key-value pairs in a single cell. Any SQL type can serve as a value. PostgreSQL stores and parses the value as a string on write and read access. There are particular operators which give this data type additional benefits. Set operators allow the combination of two sets in order to generate new HStore instances, e.g. by using the union operator. Other operators work on just one instance with operations like *is in* or *contains*. The operators are part of the HStore syntax which integrates into the SQL. This means we can create and modify HStore values within select or insert statements.

While HStore offers one possibility for storing unstructured information, JSON as a de facto standard for serialization is also part of PostgreSQL's data types. The user can create a JSON value by either passing the JSON string representation or by converting an SQL row or array type. Likewise the system allows to convert JSON values back to arrays. While previous versions did not allow query support for this data type, version 9.3 will allow similar operations as HStore does already.

A rather classical approach is storing XML data in table cells. The system allows the common query language XPATH for modifying and retrieving XML data. Thereby the user formulates predicates for retrieving certain fields in the structured data. Within XML we can store and retrieve every SQL type, so the abilities are similar to HStore. A unique feature is the mapping of complete relational tables to XML within the PostgreSQL database system.

Physical Storage

PostgreSQL offers row storage, i.e. it stores row data one after another on the disk. It arranges the data in pages which are spaces on disk of an equal 8 kb length.

While the page has still free space, the system stores the tuple (or row) in this area. Big field values, e.g. large variable data type values, can exceed the limit of a single page. The system cannot break down a single field at the first place, but utilizes the so-called TOAST technique for solving this issue. This notion curiously stands for “the best thing since sliced bread”. It compresses the values of large fields and can then split those values across different pages. For this it uses a very fast and simple LZ compression type as long as the data types are TOASTable. TOASTable are all variable length data types which implement strategies for compressing and splitting its values. Each table with TOASTable attributes creates automatically a new TOAST table for storing pointers to the consecutive splits. This technique allows saving space on large PostgreSQL tables.

Procedural Language

PL/PgSQL is a procedural language used in the PostgreSQL ecosystem. It is an extension which enhances the possibilities of writing custom functions and procedures. This language allows a mixture of procedural and set-based operations. Functions can either return a SQL or a user-defined type or they can return sets which are also SQL rows. SQL statements can include set returning functions as their source in the FROM-part by using an implementation of the TABLE operator. This allows flexible and parametrized table functions. It is also possible to use other languages as procedural extensions, e.g. Java or Python.

Data Definition Language

The data definition language of different relational database systems is often very similar. PostgreSQL offers some special features additionally to the standard ones. This means there are default index types like B-trees and bitmap indexes. Other index types are GiST and GIN, where the first one (**g**eneralized **s**earch **t**ree) is a generalized B$^+$-tree. This tree is also height bound and we can use it for creating indexes on spatial data or as a fulltext indexes. The SP-GiST is an index for space partitioning on geographical data. The **G**eneralized **IN**verted Index is similar to the previous one, but faster on access and slower in creation. This makes it more likely to be used for full text indexes. Furthermore PostgreSQL offers partial indexes. Queries can utilize such an index even if it does not contain all indexed fields. The user can also use these index types for specifying a function-based index.

Triggers, Rules

There are two possibilities for reacting on SQL statements, namely triggers and rules. Rules were introduced prior to triggers and allow rewriting queries and statements. While rewriting to nothing is possible, such that statements on false conditions do not do anything, it is not possible to throw exceptions. A user thus cannot react on failures because he cannot receive a notification. Triggers allow to react on statements by using the procedural language, which allows the user to either change it or throw custom exceptions. Version 9.1 introduced *instead-of triggers* for reacting on queries and statements. *Instead of* performing the requested action the system can change the behavior completely on its own specified conditions. With these extensions triggers are more powerful than rules.

Clustering

The system enables the clustering of tables for supporting efficient access patterns. Clustering advises the system to layout the data according to a specific index. While the table is in clustering mode afterwards, PostgreSQL does not cluster newly inserted or updated data automatically. The user has to initiate the clustering.

As mentioned in Section 4.2 the queries performance depends on the query optimizer as well as on the implementation of the relational operators. Statistics on the distribution of data helps the PostgreSQL optimizer to choose the right execution plan. The statistics include information about the size of the tables, the distribution of the data within the tables, relationship continuities for joins and numerous additional attributes about the data. PostgreSQL offers a function for keeping the statistics correct and current, namely *analyze*. This function collects statistics from all tables in a specified tablespace. Another important function in this case is *vacuum* which cleans up existing tables from dead entries. Dead entries are deleted or moved tuples which are no longer in use, but they still remain in pages close to alive tuples. Because sequential reads are more effective if these tuples are removed, *vacuum* provides means to achieve this. Vacuum either cleans up those entries to provide space for newly inserted tuples or rewrites the whole table to a new position in the tablespace. A common technique is to use first vacuum and then analyze. PostgreSQL provides a function to do this by using one command.

Foreign Tables

PostgreSQL offers a possibility to include tables from external PostgreSQL instances. They have the notion of *foreign tables*. This technique roots from the dblink extension and is part of the core since version 9.1. It enables the specification of a connection string and schema name of the external instance such that the system can create a table for the user which keeps this fact transparent. Queries can simply address this table and the query optimizer automatically performs the external lookup. This is especially interesting if the user performs joins with two foreign tables or a local table and one foreign table.

Data Manipulation Language

The possibilities for data manipulation are similar to the general operations discussed in Section 4.2. One special command is the COPY operator which allows bulk inserts of data from the file system to the database and vice versa. While copying tables to the file system may not be often used in production systems, the bulk insertion method offers performance benefits if the data does not arrive in an online way. It is possible to insert text files where one line corresponds to one tuple or respectively one row. These text files use special characters as delimiters. The most common format is Comma Separated Values (CSV), but any other character is possible as well. Also binary formats are possible and there is a documented way for specifying signatures and flags for compactly representing the data as tuples. If the data is available in bulks, this method is preferable because of the shorter insertion time.

Right Management

PostgreSQL offers a standard right management system with basic principles explained in 3.5.4, but with custom notions. We can achieve authentication via password, LDAP, PAM or one of several other options. For authorization PostgreSQL uses the notion of roles, which subsumes users and groups. Creating a user means creating a role with the ability to log into the database system. Rules can additionally serve as groups as they are inheritable. One role thus can inherit rights from one or more parent roles. For convenience reasons *user* is a synonym for *role*. The permissions are fine-grained such that we can grant operations like selecting or modifying data are individually for each data unit (table). Each schema has individual rights and each table distinguishes between read and modify operations.

Extensions

Numerous extensions enhance the functionality of PostgreSQL. The development team takes often used or famous extensions to the core functionality over time. There is *PostGIS*, an extension for spatial databases, which offers enhanced functionality for querying spatial data and building up spatial indexes. The *fuzzystrmatch* package supports fuzzy matching of strings and includes Soundex[1] functions as well as Levenshtein[2] operations. There is the *MADlib* which offers several analytic extensions to PostgreSQL. It allows to create easy data analysis by providing implementations for complex operations. PL/proxy provides a way to route connections and queries to a set of PostgreSQL instances in a cluster.

Backup

PostgreSQL offers with `PG_dump` and `PG_restore` some simple backup mechanisms. The dump allows the compressed storage of the instance's data to the file system by specifying all the schemata and resources which it should include in the backup. The restore function does the opposite by reading and optionally overwriting the current database with the data in the dump file. As PostgreSQL uses a write-ahead log for its persistence, there is a possibility to do an online backup by storing these files. For this reason we have to set the database system into archive mode. This allows a point-in-time recovery from then on. This also means we can restore the database to any previous state if a system crash happens. The technique is more complex than a simple dump, but offers more functionality.

5.2.4 Redis

Redis, acronym for **Re**mote **di**ctionary **s**erver, is an advanced key-value store. It is an open-source project under BSD-license and as such free to use. The system implements as an in-memory store and because of that is often referred as a cache database system. Redis is written in C as a standalone network application with numerous connectors for different programming languages like Java or Python. As a key value store the system aims for storing simple structured data rather than supporting complex schemata.

String Values

The notion advanced key-value store refers to the different data types that are available in the Redis data model. Redis defines every key as a string of variable size. The basic value type is also the string. The user can save a variable length string under a key. However there are multiple commands for advanced operations regarding these string values. The user can append a string to an already existing string, or read and write substrings with the `GETRANGE` or `SETRANGE` commands. Besides that the system provides means for altering the string value bit-wise, such that operations for setting or reading single bits exist. The `BITOP` operator allows performing bit-wise operations between two strings. Furthermore, Redis treats the values automatically as integer or float values if they match the format of containing either only numbers or numeric values separated by a dot. If the format

[1]A method that maps similar sounding words to a common hash-value

[2]Determines the number of modification to transform one word into another one

matches, the operations `INCRBY` or `INCRBYFLOAT` allow changing the value with numeric semantics. The user can exploit this behavior to use string values as counters for example.

Hashes

Another data type is the hash value which allows having another key-value store within one key in Redis. The notion for the keys of the hash is the term *field* and is a string value. The field's value is again a string, which has basically the same possibilities than the basic string value. Operations for integers and floats are possible, but the system does not support bit-wise operations here. It is possible to store values to multiple fields in one atomic operation. Additionally, there are other hash-related operations like determining the existence of a key, getting the length of the hash and retrieving all values at once.

Lists

Lists allow storing a variable amount of strings under one key. There are several operations for inserting values to the list. One can simply store a value to the front or the end of the list or override another value at a specific index position. Another way is storing a value before or after a specified pivot element of the list. There is the possibility to retrieve one element or a list of elements by specifying an index range, but it is also possible to treat the list as a queue. For this reason there are `POP` operations for the start or the end of the list. By this we can use it either as a *FIFO* or as a *LIFO* queue. The `POP` operations are either non-blocking or blocking, so that the operations waits until at least one element exists in the list. The blocking operations require the specification of a timeout. Remarkable are atomic operations for popping one value from one list and appending it to another list in one command.

Sets

Sets are a data type for storing unique values under a key and performing set operations between multiple keys. There are operations for storing a value, called member in this case, and retrieving the cardinality of the set or simply the whole set. The `SRANDMEMBER` operation randomly retrieves one or more elements of the set. The system supports basic set operations like union, intersect and difference. There exist atomic variants for performing a set operation and storing the result into another set respectively another key with set data type. *Sorted Sets* Sorted sets extend the functionality of sets by adding a score (or weight) to the value. The set is then sorted according to this score and operations exist to retrieve set items in an ascending or descending order. Optional parameters allow range queries by specifying a start and end point of the resulting sorted set. Additionally we can treat the members of a sorted set as integer values like in the basic string type. There exists an operation to increase or decrease the integer value.

Redis generally overwrites values when insert operations occur on an already existing key. This applies only to keys of the same data type, so it is for instance not possible to overwrite a string with a list. Because this behavior is not always intended, there exist operations for each data type to insert the new value only if the key does not yet exist. Furthermore, a key can have an expiration time which the user sets optionally during insertion. Redis deletes the key and its value automatically deleted after this amount of time. During this time to live we can remove the expiration time by using the PERSIST operation. Besides that the user can rename existing keys or DUMP and RESTORE them to and out of serialized representations.

Compression

In Redis we can activate compression for the values of each key data type. Since Redis is an in-memory store this is not only good for I/O optimization, but also for memory consumption. The system compresses hashes, lists and sets up to a certain amount of keys and size of values. The administrator can specify this limit in the configuration parameters depending on the type of data. If collections contain only integer values, the compression is most effective and can save up to 90% of the original data size. However the system compresses values always as a whole such that large value may not benefit from it. This is true if e.g. the user queries only ranges of sorted sets because Redis would have to decompress the whole set which can be a large overhead. Also the time that it takes for compressing large values can diminish the positive effect.

Transactions

Redis has the option of using transactions. First of all there is a possibility for each data type to insert and retrieve multiple values at once. As every Redis operation is atomic, this is the first hint to ACID-like behavior (cf. Section 3.5). Furthermore, the MULTI command starts a transaction. The system queues each command that the user inserts afterwards and executes everything atomically by calling EXECute. The execution is also serialized and sequentially such that it preserves the order and no other command interferes in the processing. There are no rollbacks in Redis and it does not guarantee that each operation of a transaction succeeds. This behavior on the one hand improves the performance of Redis transactions and on the other hand is not that critical because commands can only fail due to bad syntax. Otherwise commands succeed as there are no constraints such as failures because of inserts on existing keys.

The system can preserve consistency between read and write operations by WATCHing keys which is an optimistic locking mechanism. On the execution of the transaction the system checks the watched keys for modifications and decides if it has to abort the transaction. Listing 5.1 shows one example of such a transaction. An application reads the key *myKey* and a user modifies it. After this a transaction should increase a counter and write the modified value back to the key. In this case the transaction will only be executed if *myKey* was not modified in the time between

Listing 5.1: Redis transaction example with WATCH operation

```
WATCH mykey
myValue = GET mykey
myValue = change(myValue)

MULTI
INCR counterVar
SET mykey $myValue
EXEC
```

the `WATCH` command and the `EXEC` command. This optimistic locking mechanism works well if conflicts of multiple users are unlikely. On failure the procedure has to be repeated until it succeeds.

Scripting

Besides using external client programs, Redis provides a scripting language to perform server-side complex computations. Redis uses Lua as its scripting language and provides a small number of packages to perform operations. One package provides a connector to Redis for executing operations, which is similar to using ODBC on other languages for connecting to a database system. The user can either execute scripts directly or store them for later usage. The system automatically determines a hash value of this script and we can identify it by this value afterwards. The scripts can optionally return values compliant to Redis' data types like lists or single string values. Redis executes a Lua script the same way as a regular transaction, providing atomicity and sequential processing. Thereby a script can do everything what a transaction is able to do with even more options because of the more powerful language.

Security

Redis does not provide a complex security system. Generally the server is free to accept connections without any authentication. However there is the possibility to create a password that is optional for setting up the connection. Furthermore there is no authorization system for granting specific rights to certain users. The application layer is responsible for any further distinction and complex security concerns.

Architecture

Each instance of Redis uses exactly one process and can utilize at most one CPU core. If processing power is the bottleneck, it can be useful to partition data even on a single server node. This means adding a new instance to either the same server or a new one. To achieve this, the developer has to implement all necessary functionality by himself. One idea is hashing on the client-side or the use of a proxy for receiving and transmitting the commands. There will be a default Redis cluster implementation, which is not available in a stable version yet.

Replication

Replication on the other hand is an implemented easy-to-use feature in Redis. It uses a master-slave replication model for creating new instances as exact copies of the original one. The replication is done asynchronously using *SYNC* messages to exchange changes on the data throughout the cluster. The synchronization is non-blocking such that nodes can serve answers even by using data from old versions. Slaves can be set up in read-only mode to enhance scalability and improve the query performance of the system. Allowing to write on the slaves endangers the consistency of data within the system. Similar to QUORUM it is possible to specify the number of nodes which a client has to write to. This improves data consistency to a certain degree. We can use Redis' replication either for scalability issues or simply for redundancy reasons.

Publisher/ Subscriber Model

The Publisher/Subscriber model is a technique to enhance the client-server communication. It allows an n-to-n communication over channels while keeping the subscribers list transparent. Clients can subscribe to any amount of channels via their name. Additionally it is possible to use wildcards with subsets of the channel name. The client subscribes in this way channels are then subscribed to by pattern matching. Finally any Publisher is then able to publish messages which are automatically sent by the system and received by each subscriber of the channel.

Persistence

As Redis is an in-memory store, consistency is an issue. Redis allows the user to choose between two different persistence models which are called RDB (Redis Database Backup) and AOF (Append Only File). RDB enables to use point in time snapshots of the data set which is performed at specific interval times. AOF uses a logging mechanism similar to the write ahead log which stores every operation in a file. The advantage of RDB is that it is a very compact representation which needs less space than AOF. It makes backups simple because the resulting file contain all necessary information to restore the server. Furthermore it is beneficial for the performance of the instance as we can choose the time for persisting the data according to the requirements of the application. While AOF-files are bigger in size, the system can restore its data at any point in time. The system writes the files append only, so there are less seek times and a lower chance of corrupt data. Redis is even able to fix partially written log file lines. The user can choose either the RDB or AOS for persistence, but it is also possible to combine both techniques. Furthermore, it possible to disable persistence completely. In this case the system holds the data only in memory and everything disappears after a shutdown of the system. In scenarios where this is useful, Redis serves as a cache. Every other case requires storing the data from memory to disk which slows down the overall performance of the system in some way. Overall the user is responsible for setting up the proper consistency model for his application.

6 Requirements

This chapter discusses the requirements of data management with respect to the context of the Menthal project. First it describes the two main use cases. The first one deals with Self Monitoring and the Quantified Self. This mostly concerns the gathering of smartphone usage data and the presentation of this data back to the user. The other use case operates on the same data, but inspects the data further to achieve for instance stress or depression detection. The chapter finishes with the definition of the scope of our work which mainly focuses on the first use case. To this end it formulates the functional and nonfunctional requirements of a system that solves this use case.

6.1 Use Cases [MM]

The following section describes the two general use cases within the Menthal project. It starts with the Quantified Self use case, where a smartphone gathers usage data and sends it to a server. The server processes the data and provides an interface for the user to analyze his historic behaviour. The system has to cope with new kinds of devices and data and a growing user base. It also has to provide means to deploy new analysis tools. As users wish to view recent behaviour rather soon, the system has to process data in a reasonable amount of time.

A more in depth analysis on the whole data set is the goal of the second use case. This includes the learning of classifiers that we can e.g. incorporate to determine the stress level of a person. Ad-hoc queries are also part of the use case. The computation of answers does not comply to time constraints like real time. Instead it is acceptable that it can take multiple days.

6.1.1 Quantified Self

Quantified Self is a movement that deals with the usage of technology to gather data about oneself. This data can be anything from vital functions over location information to the food that a persons eats. We can use this information for instance to identify misbehaviour and improve ones way of life. In our case we are interested in anything that we can extract from the smartphone usage of a user.

The use case starts with the installation of an app on a smartphone. This app is responsible for gathering different kinds of usage data for further analysis. Usage data includes the use of apps and the performance of phone calls. It also considers sent and received messages as well as gathered location information. We collect all data in form of events. They have a type (e.g. app usage), a set of attributes (e.g. app-name and duration) and a timestamp.

As the resources of a smartphone are limited in processing power and battery life, we have to perform the analysis on a remote machine. The smartphone collects a *decent* amount of data and then transfers it to a server. A real time transfer of data is not possible because of issues with the connectivity and pricing of mobile Internet connections. The delivery of data to the server has a certain overhead. Because of this we should limit the amount of transfers. On the other hand users will demand an up-to-date view on their data. If a WiFi connection is available we should use it in order to avoid mobile bandwidth fees. If no WiFi connectivity exists over a longer period of time we should nevertheless use the mobile connection. By this we can ensure that we transfer the usage information at least once a day to the server. A regular delivery of data also eliminates problems in the storage of the data on the device. While most events are small, some sensors like the accelerometer are able to produce events in a high time resolution.

The server accepts the data via a standard protocol. Once the data is acknowledged we can delete it on the device. The server now processes the data to transform the device specific events to generic ones. We can also derive complex events, e.g. two window change events marking the beginning and end of an app session. The system has to be flexible enough to allow the subsequent enhancement of support for other smartphone devices. It also has to provide the means to define new derivation methods.

The server runs regularly aggregations. This includes counting events and the computation of sums, minimum and maximum values for attributes of all event types. We calculate the aggregations for different granularities: 15 minutes, 30 minutes, one hour, one day and one month. Additionally we should calculate greater intervals from the finer granularities to avoid redundant computations. We do not need to perform the aggregations in or close to real time. A delay of e.g. 1 hour after the user submitted the data is reasonable as we do not want to encourage the user to check his statistics all the time anyhow. We finally store the aggregations for later access.

The user can then view the history of his usage data in a separate mobile phone application or on a website. He may filter for time ranges and event types. The app, or the web server respectively, requests the data from the server. It might make use of caching to reduce the load. The user's client (in form of app or website) then renders the data as graphs.

The other side of the use case is the management on the server side. Administrators need the means to define different devices and how to interpret their data. The data processing has to be flexible enough to be extensible to include new data analysis algorithms. Administrators are also responsible to prevent data loss. The system should provide them the necessary tools e.g. in form of backup or replication functionality.

6.1.2 Detection of Stress, Depression and Illness

The second use case also bases on the data that we gather on the smartphone. In contrast to the first one the users do not analyze their behaviour themselves. Instead of this scientists try to find out general patterns across the user base. Additionally they attempt to discover anomalies in the behaviour of a single user.

A first difference to the Quantified Self use case is that we include the data of all users into our queries instead of looking only at a single user at a time. This demands for all data to be in one huge database (cluster). This allows us to perform simple queries like finding the average app session length across all users. It also enables learning more complex classifiers like stress detection or illness diagnosis on this data. In contrast to the Quantified Self use case there are no strict time constraints. A query may run for hours or even days until finishes.

6.2 Scope of our Work [CB]

The focus of this thesis is the management of sensor data. We therefore leave out everything which happens on the smartphone and on the user side in general. This work begins with data that a smartphone delivers to a server. The server then processes it, stores it and aggregates the data. It is apparent that the first and the second use case require different storage solutions. In the first case we can handle a single user in isolation of other users. The second case includes performing operations over the whole user base. In the first case we only work on recent data while we process the whole data without time bounds in the second case. We call the storage solution for the first use case *Data Store 1* and for the second use case *Data Store 2*. The emphasis of this thesis is Data Store 1 while considerations about a connection to the other store are only secondary. The following subsections describe the functional and nonfunctional requirements of Data Store 1 which we have to consider in the scope of this thesis.

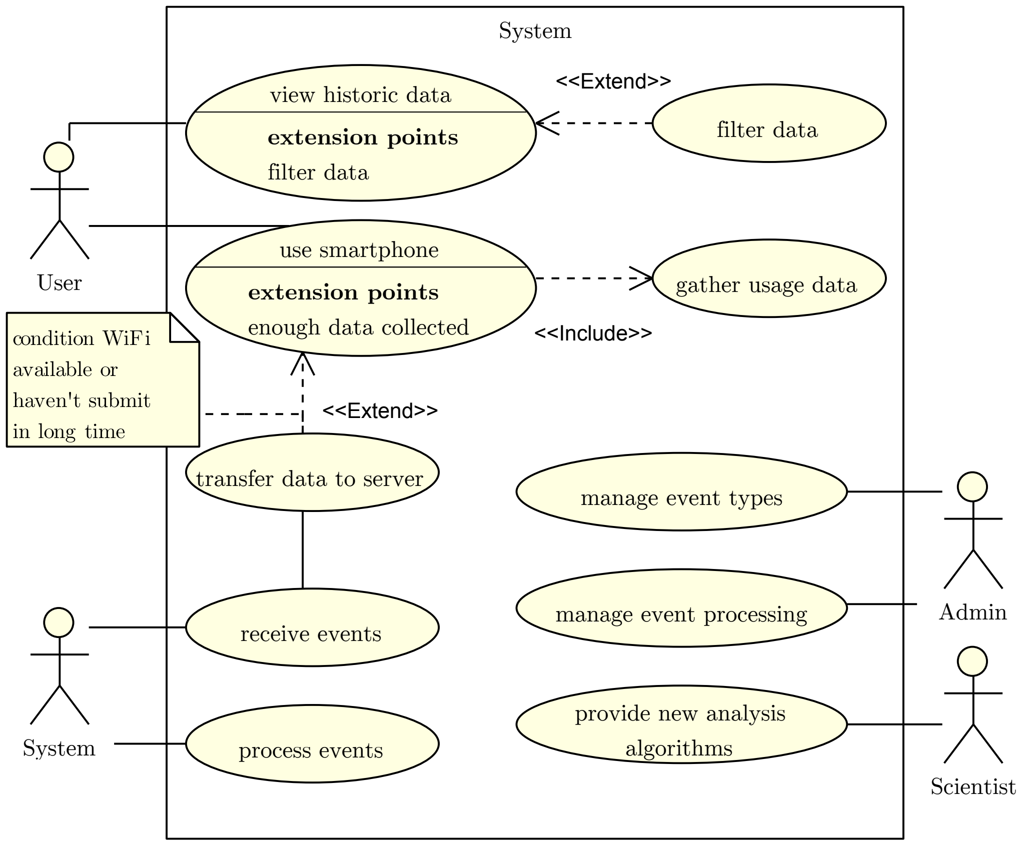

Figure 6.1: Use case diagram Quantified Self

6.2.1 Functional Requirements

Data Store 1 requires four main components. The first one is an API that receives data from a user's smartphone and provides aggregates to external applications. The second component is an event processing unit which resolves device specific details and provides a generic event format. The third one is the actual storage solution which persists event data and provides the means to query them. An aggregation unit is the fourth and final component and is responsible for providing a higher level view on the collected data.

API The API consists of two parts: receiving event data and providing aggregates. Both APIs should be platform-independent such we can use them regardless of the client technology. This is necessary as the data should be presentable on apps for different smartphone operating systems as well as web and possibly desktop applications.

Event Processing The events that we collect on the smartphones are highly device dependent. Different version of the Android OS might indicate a phone call for instance completely differently. The systems determines the semantic of those events in this case. Additionally there are events that yield information in connection with other events. The system has to be able to derive information, i.e. new events, from one or more existing events. The basis for those two operations are meta data, which system administrators have to maintain. The meta data consists of event schemata and derivation rules.

Aggregations

Single events do not offer too much information by themselves. Aggregations of multiple events from the same type however yield insights on a higher level. One of the systems main tasks is to compute aggregates of all events of a user for different granularities. We have to store the aggregates in addition to the originally events. We also have to provide them to the users later on.

Event Data Storage

The core point of the Data Store is the actual storage of the event data and the aggregations as well. A storage engine has to persist all incoming events durably. It also has to provide an interface to query events according to the user it belongs to and according to its event type and attributes.

Management

The administration of the Data Store mainly concerns the maintenance of the meta data. An interface is required that provides the means to manage event types and their schemata. Those are dependent on the smartphone characteristics like the device itself and the operating system. We have to manage the derivation of events from the basic events that the smartphones provide as well.

6.2.2 Nonfunctional Requirements

Scalability

Apart from the functional perspective, there are a number of nonfunctional requirements to the first Data Store. First of all it should be able to handle a huge amount of users. More so it should be able to *scale* to more users if necessary. As covered previously (cf. Section 3.4) this makes it necessary to distribute the system. Designing a distributed systems brings up a number of additional tasks. Users and requests need an assignment to one of the available machines. We should do this in a way that all machines have to handle approximately an equal load. Individual nodes or the connections in between them can fail. The system has to be able to cope with that availability issue.

Security/Privacy

The usage data that we collect contains data that tackles the privacy of users. We have to ensure that the access to the Data Store complies to several security policies. First of all only a user himself should be able to submit usage data under his account. Before the server can accept delivered events the user thus first has to authenticate himself. We should enforce encryption of the transmission of the data itself. This is especially critical in wireless networks. Smartphones use these typically and unfortunately they are often unencrypted. This means that every user of the network would be able to capture all the data of every user. The same policies apply of course for the retrieval of the aggregated data. A user has authenticate to the server in order to access the data. Access to the Data Store's management interface is even more critical. In this case a right management is necessary that differentiates between users that maintain different aspects of the system. This way we can ensure that they are not able to influence other parts of the system.

Data Preservation

In general data must not be lost. The value of the data does not lie in every single event but in the overall information. Losing a single event poses no threat. Instead we want to overall keep the data. Missing a single phone interaction does not significantly influence analysis. However we have to avoid losing all collected events from a longer period of time.

Performance Efficiency

We should store events in a space efficient manner if possible. As disk space has become cheap (cf. Section 3.2) this is not the most important factor though. The storage engine first of all has to be fast enough to persist all incoming data. Otherwise it would refuse connections from users as data must not be lost due to an overload of the Data Store. The write throughput is thus a first performance factor of the storage engine. The storage engine also has to provide efficient access to the data. It furthermore has to be able to compute aggregates within a given time frame.

Codability

A critical factor nowadays is the cost of development. In comparison to machines (i.e. servers) human work is much more expensive. In many cases it is cheaper to buy a additional hardware instead of paying a person to optimize a system. This leads to a focus on the ease of development rather than the performance of a system. While it is not clear how we should weigh both aspects in our context we have to consider the codability of a system. One requirement of a database system is thus how easy it is to work with it. This includes the setup and maintenance but first of all the development effort.

7 Menthal Data Storage Architecture

After the previous chapter introduced the requirements of the system, this chapter proposes an architecture that is able to fulfill those requirements. Figure 7.1 shows an overview of this architecture of the Menthal backend. It consists of two APIs, one which receives data from users' smartphones and another which provides them with aggregates. The security layer maintains confidentiality and integrity of the user

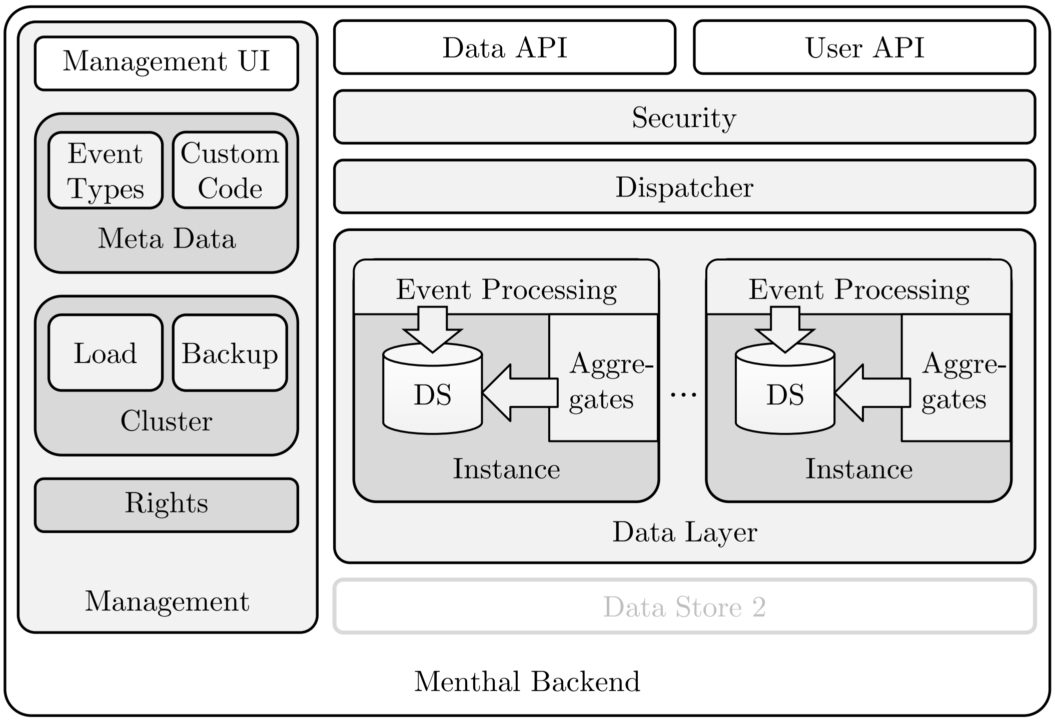

Figure 7.1: Overview of the Menthal backend

data. A Dispatcher is responsible for distributing the workload across multiple instances of the Data Store. These instances are part of the Data Layer. The instances of the Data Layer are responsible for processing, storing and aggregating the event data. Aside from that we also intend an uplink to the Data Store 2. The management layer is orthogonal to those architectural elements. It is responsible for managing meta data and the cluster which is the Menthal backend itself.

This chapter introduces the aforementioned components in more detail. It neither provides a complete description nor a solution that is ready to implement. It rather describes the components responsibilities and the problems we need to solve in order to realize them. Moreover it points out approaches to solve these issues or states that they need further investigation.

7.1 APIs [CB]

The APIs are the link between the backend and the smartphone. They provide means for setting up the communication between the user (client) and the server. This means sending the data to the server for evaluation and retrieving the computed results.

A RESTful[1] interface provides all necessary functionality to achieve this. This protocol is stateless, which is sufficient in our use case, because the communication is rather simple and requires only one-step processes. The server providing this interface listens on port 80 for incoming connections. This port is the default one for HTTP connections and thereby usually never blocked on devices. An advantage of REST is that it is easy to use and has a small complexity. Furthermore it is platform or language independent because the HTTP standard is universal and every language can adapt to it. There exist libraries for working with REST for almost any programming language.

For securing the communication channel we can use the HTTPS standard. In this sense we can incorporate the TLS encryption mechanism for setting up the communication between the smartphone and the backend server. Third-party certifications (trust-based) are available by using SSL certificates of trustworthy authorities. Securing the user data at this point is also a concern. For this we have to introduce an authentication mechanism. As the REST protocol is stateless, we either can achieve this by using cookies or by sending authentication credentials together with any procedure call. These credentials can be a username and password. After authentication the system authorizes the user to either send the event data to the server or retrieve computed analytic results from the server. The APIs allow only sending or retrieving data for one user at a time. Thereby the user is only able to modify or view its own data.

[1]Representational State Transfer

7.1.1 Data API

The data API is responsible for receiving usage data from the smartphones. It basically has to provide only one function for inserting events. Before submitting the data, the smartphone should merge all events that occurred since the last time. By doing this we are dealing with bulk inserts from a single user. This leads to a function with a signature that requests getting a list of events in one call.

The mobile phone application submits the event list in form of a JSON string. This string consists of a list of serialized event objects. It is necessary to send an acknowledgment back to the mobile phone after processing the data. For doing this an algorithm checks the integrity of the data. This bases on the schema of the data which derives out of the meta data of the system. Furthermore, the backend system has to ensure that it persists the data after it retrieves it. We can implement acknowledgments by using HTTP codes like the `200 OK` notification. After getting a positive acknowledgment, the mobile phone can delete the data on the device. It is evident that this is not part of the API such that the mobile phone application has to care about this.

It is possible that one user operates multiple devices. This makes it necessary to distinguish the submitted events. To achieve this the user has first of all to register his devices at the backend. This includes sending information about the device and the operating system. While sending the data to the server, the user also has to submit the ID of the corresponding registered device.

7.1.2 User API

The user API provides all computed analytical results about a user. This includes in the first place aggregations of the usage data. For this the API consists of several functions for choosing different categories. These include different types of events, e.g. application or phone usage events. Beneath this the user can choose between different types of aggregations like sums or averages and is able to choose a time range. This means looking at the past day, the past week or the whole year. There are also means that allow comparisons between different time ranges, for instance comparing this month to the past one.

There are generally different access types to the user API. This includes on the one hand the RESTful interface for communicating with the mobile phone. On the other hand this includes a web interface which is this accessible from other devices like the home computer or tablets. Both interface types present the data in form of diagrams. To allow interactivity on these diagrams, the RESTful API provides rather raw data then pre-compiled diagrams forms of graphics. This makes it possible to dynamically incorporate the data by means of zooming into or quick filtering it. It should be always possible to highlight anomalies in these diagrams. This incorporates data of classifiers, that provide data additionally to the default aggregations.

The user API can make use of external components beyond the data stores. This can include a web server that provides additional functionality. The relevance of this is that the caching of the user's aggregated data is possible apart from the data store. By use caching we can achieve faster response times of the API to provide the results of the aggregations. This is because we can avoid computing a result set twice or more often. It also allows us decoupling a caching strategy in the data node, because we leave the responsibility to the API component.

7.2 Dispatcher [MM]

The Dispatcher directs data and requests to the responsible data store instance. In principle this consists of a simple lookup and a forward. A user either wants to store new or query existing data. The Dispatcher looks up which data store instance is responsible for the given user. This can either be the calculation of a function or a table lookup. A simple function could, according to the user ID, direct the first 100 users to instance A the next 100 to instance B and so on. In a table we would look for the row that contains the current user and get its assigned instance. Once the particular instance is identified, the Dispatcher forwards the request to it. The actual work is done in the data store layer. Even though this process is not very complex, there are some issues to discuss. First of all it is not clear *how* the users are distributed across the instances. Secondly the Dispatcher is a single point of failure in the distributed systems and we have to deal with this fact.

Single Point of Failure

The Dispatcher is a single point of failure as it is the entry point to the system. Applications communicate through it with the actual data store instances. If we only have one Dispatcher it has to be able to handle all incoming requests. While we can introduce new data store instances to handle the data and query load, the Dispatcher would not scale accordingly. If the machine running the Dispatcher fails, no data gets out or into the system anymore. We can distribute the Dispatcher rather easily though. To run multiple instances we have to take care of two issues. First of all each instance has to know where to forward requests to. For this we need a shared distribution function or table. In order to keep the dispatching process fast, each Dispatcher instance would cache the *user-to-data-store* assignment locally. Now that the individual Dispatchers are able to forward requests, we have to ensure that the original requests are distributed across the Dispatchers. A typical solution is to perform DNS load balancing. The DNS server needs to know the IP of all Dispatchers. Upon a request for resolving the (unique) hostname for the Dispatcher (e.g. `http://dispatcher.self-monitor.de`) the DNS server answers with multiple IPs. By shuffling the order of these IP addresses, we are able to distribute the load across all the Dispatcher instances. We can thus avoid a performance bottleneck and a single point of failure.

Load Balancing

Hotspotting

It remains the question of how to distribute the users across the available data store instances. The goal is to distribute the user base *equally*. More precisely to distribute the *load* of the system as equally as possible. This way we can avoid hotspotting, meaning that one instance is kept very busy while other instances are idling. A simple heuristic to gain equal load across the system, is to assign each instance the same amount of users. But not all users behave the same. Some heavy user might create more load (size of produced data, number of requests over time) than dozens of regular users. A more sophisticated approach would take the (average) load of a machine into account when a new user is added to the system. The user would be assigned to the machine that is exhausted the least. Of course the load of an instance can change over time. Users can change their habits, some may generate more data, others will quit using the application all together. A future-proof system has to be able to cope with this by altering the user-instance-assignment over time. This includes the shipping of data from one instance to another and the update of the assignment table.

Cluster Topology

Not only the user base but also the systems topology can change. A machine can fail and new ones can be added. If a machine fails, we can no longer accept data of any user which is assigned to this machine. We are also unable to provide the data which already has been received. We could postpone all actions until the machine is restored. If the data on the faulty machine was lost, this requires the existence of a backup (cf. Section 7.4). If we can not allow such downtimes, we need to maintain a redundant storage of a users data. A user thus would not only be assigned to one instance but at least to one additional. The addition of a new data store instances also requires some work. At first there are no users assigned to the new instance. In order to exploit the new resources, the current load must be redistributed. This is essentially the same process as reassigning a user because of the load of a particular instance.

7.3 Data Layer [MM]

The data layer is called Data Store 1 and consists of multiple uniform data store instances. They get data and requests from the Dispatcher. When events arrive, the data store has to resolve the specific events from concrete devices into generic events which it can then aggregate (cf. Figure 7.2). We call the responsible component for these processes the EventEngine. It stores the original (raw) events, the generic events and the aggregates and provides them on request back to the users. This

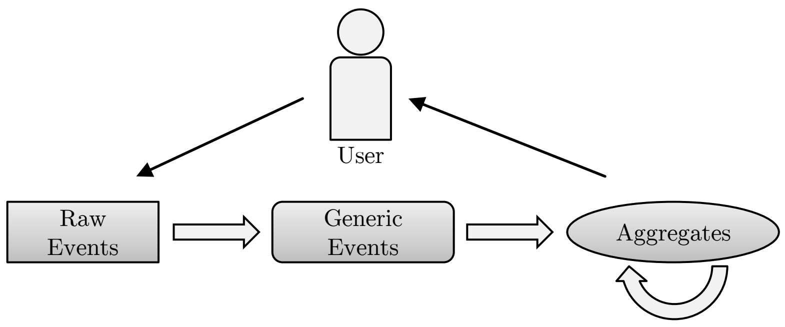

Figure 7.2: Process of event transformation in the data layer

section first discusses the actual storage of the data. Then it introduces the problem of event processing and shows approaches to solve it. Finally it deals with the aggregation of event data.

7.3.1 Data Store

A single instance of the data store is responsible for a subset of the users of the overall system. User data can stand alone. This means there are no relationships or dependencies between two user's data sets and we do not want to join them. We also assume that we can fit all data of a user onto a single machine. This is reasonable because a single user does only a manageable amount of data even over extended periods of time. Thus we need no communication between the different nodes of the data store to answer queries.

All data is present in the form of events with a temporal reference. Each event is of a given event type and each event type has a fixed schema. There is no overall schema for all event types though. The events arrive in the correct time order for a single user. They are bulk-inserted and do not arrive in real time. Before they are inserted they should be checked against the schema of the given event type, which requires access to the overall systems meta data (cf. Subsection 7.4.1). For processing the events and computing aggregates we typically want to access the events with a range query over the time. We also want to be able to filter for specific events in type and attribute values. This allows an efficient computation of event derivation and aggregates.

We want to realize the core functionality of storing and querying events using existing database technology. This way we can avoid (re-)implementing the different aspects we introduced in Chapter 3. It is however not obvious which data storage approach (c.f. Chapter 4) fits the best to this scenario. It thus certainly is not obvious which particular product will yield the best experience. Chapter 8 deals with the question of how to create an environment to test the databases we already introduced in section 5.2 in this respect. Chapter 9 introduces the implementation of the resulting tests. Finally Chapter 10 and Chapter 11 discuss the results of our observations.

7.3.2 Event Processing

There are two use cases where a data store instance needs to process events. By processing events we mean the derivation of a new event, based on one or more existing events. In the first case it has to transform specific events in to generic events. Raw events which are gathered on the user's phone are difficult to analyze because they contain device (or OS) specific information. We do not want to resolve those events on the phone, to keep the app simple and to avoid wasting phone resources. The second case is the derivation of complex events. This means forming a new generic event out of one or more generic events. An example would be to calculate an `AppSession` event out of an `AppStart` and an `AppEnd` event. The difference between the two kinds of event derivations is that in the first case we transform raw events to generic events, while in the second case we operate solely on generic events. This means when a new device is added, the system only needs the transformation rules to the basic generic events, while the complex event rules can stay the same.

In contrast to aggregations, event processing has to be extensible. We want to introduce new derivation rules, without having to change the code of the data store. There are two approaches for defining derivation rules in a dynamic fashion. Either we define them in a domain specific language (DSL) or we process program code. The former approach consists of specifying a custom language like SQL in syntax and semantics. A rule in this language could then either be interpreted by the data store, or complied to program code and executed when necessary. The latter approach is letting the designer of a derivation rule write his own program code.

DSL

The creation of a DSL is more complex than simply executing a given piece of code. For the user it is however easier to use a language which does exactly one thing (defining rules) than a general purpose language, which can compute anything. As it is more restrictive, it may limit the user in expressing some rules depending on what he wants to achieve. When we allow the user to write his own code, we first of all need to provide some framework in which the code is executed. He needs some well defined interfaces for accessing and manipulating data.

Running some external code poses several dangers. The execution of the code could do anything, even performing malicious actions. It could for example compromise the privacy of the data by accessing it and storing it to another location. It could harm the integrity of the data by performing some manipulations it is not supposed to do. It could also negatively affect the the performance of the system by exhausting the machines resources. A typical approach to this problem is to execute the code within a sandbox that limits the execution and isolates it from the rest of the system. For the remainder of this chapter we consider the implementation of event processing via custom code only. Code signing is a way to ensure that the code comes from a trustworthy source. This avoids incorporating harmful algorithms into the system by securing these code fragments against unforeseeable exchange. Furthermore the algorithms should only have sufficient rights that are necessary to execute the task.

Sandbox

Code Signing

A further consideration is what happens if a new derivation is specified. It raises the question if we have to apply the rule on past data. If yes, the semantic would be that the derivation applies to all data. If we however do not do it, the validity of a derivation rules starts with its insertion to the system. Then only new data has to be processed.

Last but not least event processing is subject to time constraints. It has to be fast enough to cope with the stream of incoming events. Otherwise it means that more events are received than the system is able to process, leading to an ever growing amount of unprocessed data. This can for example be solved by limiting the complexity and the number of event derivations. Another approach is adding more machines to the system and distributing the load accordingly.

7.3.3 Aggregations

Aggregations provide a higher level view on data. In the data store 1 we want to aggregate all generic events for different granularities (5 min, 15 min, 30 min, 1 hr, 1 d, 30 d). The aggregation types are count, sum, minimum, maximum and average of a given time period. We assume the set of aggregation functions to be fixed. In contrast to derivations rules we do not require it to be dynamically extensible. The implementation however has to be generic enough to cope with not yet known event types and their schemata.

Counting events is the easiest part. But even in this case we need to answer the question of what exactly we want to count. All events of a given type, e.g. all phone calls? Events of a type that have a certain value, e.g. all phone calls that lasted exactly two minutes? Events of a type that have value in a given range, e.g. all phone calls that lasted between two and three minutes? All events with a certain value combination, e.g. all two minute phone calls with Erica? It does not seem reasonable to count all these combinations. In the end we want to compute what the user will find interesting. How many events of a given type occurred during a period will fulfill this. How many phone calls of exactly 2 minutes and 23.2 seconds happened however yields no useful information as this number will most likely be zero or one. How often a particular app was used on the other hand will provide additional value. Our proposal is to count all events of a given type. Additionally the system should allow to define which attribute values should be counted as part of an event types schema definition. The system only calculates those aggregates. This way we can avoid the calculation of uninteresting aggregates. The same considerations hold for the other aggregation functions.

Strategies

There are different strategies to perform the actual computation of the aggregations. One can be called *pay as you go* and the other one is regular scheduling. Pay as you go means we aggregate events when they are delivered. Regular scheduling means we just take the events and postpone the calculation to later. We can think of different types of regularity. We could go by time, e.g. execute the aggregation every 10 minutes. We could go by the amount of events, e.g. start the aggregation once 1 million events have been accumulated. Or we could combine the two and start it every x events or y minutes. When the aggregation is started we take all new events since the last computation into account. The amount of work is basically the same as in the pay as you go approach, as we process the same amount of events. Processing them in a batch however has less overhead and can make e.g. better use of sequential reading from disk or caching.

Avoiding Redundancy

A final consideration in the calculation of aggregates is the avoidance of redundant computations. They are computed continuously like it is the case with event processing. We thus have to meet certain criteria in the execution time. Bigger aggregates can be built upon smaller aggregates by combining them. For some aggregations we have to store some additional information though. To compute the average of a 10 minute range from two 5 minute ranges, we need to know the amount of events, and the sum of the observed value. Only the two end results do not suffice. We also have to keep track on which data we already aggregated such that we do so only once.

7.4 Management [CB]

After describing the data store instances it is necessary to provide means for managing the resulting cluster. This includes interfaces for synchronizing all instances. This is obligatory because it is not sufficient to change one instance and leave another instance behind. The result is a centralized management console for distributing all the changes.

There are two types of information the administrators have to care about, namely the meta data and the cluster control data. The meta data includes on the one hand all information about event types which are either device dependent or generic. On the other hand it includes the derivation rules which are present in form of custom code (algorithms). The cluster control data contains the assignment between the user data and the cluster nodes. This means also configuration parameters about the nodes in the system, the used disk space and errors. Managing the addition of new cluster nodes is also part of this interface.

7.4.1 Meta Data Management

The meta data management controls all shared resources between the data store instances. This includes information about the event types as well as the custom algorithms for derivations. The management interface has to provide components for incorporating changes and synchronizing all instances according to this.

Event Types

The management component provides means for defining the schema of the event types. The maintenance is straightforward as there are only a few parameters to set. This includes an identifier which is a numeric value describing a distinct type. It also includes information about the different attributes of the event. We can describe attributes by determining data type and certain constraints, e.g. a specific range of numeric value. According to Section 7.3 we distinguish between generic and device dependent events. Each device dependent event links to device information. This information includes attributes e.g. about the operating system or the mobile phone model. There can be categories or even hierarchies for events. This supports the semantic ordering of events or dependencies between events.

Schema evolvement is an issue in this case. It is generally possible that event types change over time. This would include that we have to adjust already stored data to avoid inconsistencies. At the same time it is possible though that the old data does not fit to the new schema. While it is possible to store an integer value in a float field, it is not possible to store a string in a numeric field without additional hints. Because this process of transferring old data to a new incompatible schema is complex, we do not allow to change already existing event types. In case it is necessary that there exists a practicable workaround for this. The administrator can introduce a new event type and introduce an event derivation in a way that it maps the old event type to the new type.

Custom Code for Derivations

Besides specifying event types it is also possible to define rules for deriving new event types out of old ones. We realize this by allowing custom algorithms from domain experts. A centralized code repository manages these algorithms and the individual data stores execute them during the event processing phase. As mentioned previously, executing custom algorithms invokes security concerns.

Publishing

Data Store instances need to incorporate the meta data of the management server. This meta data has to be current and synchronized for all instances. The synchronization does not need to be instantaneous. At data store one it is not that critical that users do not get the newest kinds of derivations at the millisecond they appear in the meta data. For achieving the synchronization we thus propose a subscriber model. This means all Data Store instances subscribe at the meta data server and get notifications on any changes. Instances can then retrieve the new meta data and cache them locally. Newly recovered instances can use the same technique for restoring all meta data.

There is a general restriction that if the meta data server crashes there are no changes possible during this interval. This is because we cannot ensure that the changes propagate to all instances anymore. Fortunately this is not critical for the operation of the Data Store instances because they cache the meta data. One minor downside is that the management interface must not allow any changes. Another downside is that it is not possible to incorporate any changes to the cluster. This includes adding new machines or performing the recovering mechanisms. This is because the subscriber model will not work during this time span.

7.4.2 Cluster Management

Cluster management provides means for controlling the current data nodes and maintaining the whole cluster. This includes two strategies. The first one is the user assignment which comes along with node distribution of the overall data in the cluster. And the second one is a strategy for redundancy and data backup. Both types concerns the availability of the cluster.

User Assignment

A decent user assignment is necessary to achieve a good overall performance. This is because load distribution avoids having bottlenecks or congestion within the cluster. The Dispatcher executes the assignments and the management server provides the lookup tables for this. This component first of all allows to monitor all the individual instances. This requires a service on each cluster node that informs the management server about all properties. It also notifies it on any failures which helps to react on errors faster and makes the overall processes transparent.

Load Distribution

Section 7.2 introduced strategies for assigning users to instances. The management console contains parameters for controlling this assignment and reassignment. Furthermore it is possible for adding a new machine to the cluster. As the reason for new nodes is often a high overall load of the cluster, it is useful to initiate a reassignment of users. A goal is to transfer as few data as possible through the cluster network and avoid to impair the overall performance. We can achieve this by transferring a small but noticeable amount of load to the new machine and assign new users rather to this node than to the older ones. Thereby the load distributes eventually and much more smoothly by taking care of the available resources.

Backup and Recovery

We need a backup strategy for two kinds of data. First of all for the event data and also for the meta data. For standard backup mechanisms we could fall back to already existing features of the corresponding database system. Alternatively there is the strategy of having data replicated on multiple instances. This means we can use a different strategy other than recovering an external backup for restoring the current data. By this it is possible to use a hot recovery mechanism, i.e. recovering the system without shutting it down first.

The instance recovery mechanism for the event data takes place when e.g. a hard disk was replaced. First it retrieves its assigned users. Second the instance has to get the information which instances contain replications of this data. This is necessary to get the sources for automatically loading all events into this instance. It is necessary that this recovery mechanism must not slow down the system. We can achieve this by monitoring the network load for avoiding an overload. If necessary we can then reduce the speed of recovery.

7.4.3 Rights Management

Rights management is also an important component of the management server. For the event data it is relatively simple to specify rights. We allow users only to insert or read their own data. For controlling the management server there are more options, including roles and rights. We can distinguish the roles between the meta data manager and the cluster manager. For managing meta data we can think of fine-grained roles like an event type manager, who can alter event types, and a code manager, which is responsible for managing the code repository. The cluster manager is responsible for controlling the cluster and its topology.

We distinguish the rights between read and modify rights. This is sufficient as there are administrators who only need to see all information and others who need to insert, modify or delete data. For the right assignment we assign both roles and rights to an admin user. One user for example should have rights to modify the code repository. We would assign the role code manager to him and additionally the right to modify. A superuser is responsible for the assignment. This is sufficient as we do not expect many admin users within the system. Thereby there is no need for introducing the right to grant certain permissions.

There are not only permissions for users, but also for the event processing algorithms. We use policies to specify what operations we allow them to perform. In an exemplary scenario a scientist hands over a code fragment to a code manager. The code manager knows the task of this code, but not the implementation. He still has to ensure that it performs within its authority. For this there are a few restrictions for the algorithm. On the one hand this includes restrictions to read and write only certain event types. A code manager can derive this out of the task of the code fragment. On the other hand the algorithms should not be able to perform modifications or even deletions at all. This prevents doing malicious changes to the system other than inserting new erroneous data. Old data stays unaffected at any time.

7.5 Big Picture [CB]

The previous sections presented the components of the Menthal backend. To get an overview of the interaction, Figure 7.3 illustrates one possible layout of the overall system. The left upper corner shows how data gets into the system, the right hand side shows how it transfers out of the system. The center part contains maintenance and processing components. In the lower left corner there is the connection to the second data store.

Data Flow

The data collection describes the user's mobile phone usage and the application that collects this data. It is responsible for transferring the data to the backend by sending it to the Dispatcher. The data flows to both data stores, i.e. unmodified to the cluster in Data Store 2 and the EventEngine in Data Store 1. As Data Store 2 is not within the focus of this thesis, we do not go further into detail here. We just consider that it receives the data and processes it further. Furthermore, we regard it as being independent from the remaining system described in this section.

The Dispatcher selects one of the Data Store instances of the cluster based on the cluster management information. The responsible Data Store 1 instance receives the user data. It then performs the processing and aggregations by using the meta data of the management component. Finally it persists all information in the database.

Users access the web server in the Timeseries Access Provider node to review their data and the aggregations. This web server retrieves the data from the corresponding Data Store instance. For this it communicates also via the Dispatcher. Additionally it caches the results in order to reduce the load of the data store instances for subsequent user requests. The access and the graphical rendering is possible by using different devices like a computer, a tablet or again a mobile phone.

The last part of the overview is the management component. Administrators contact the management console in order to change settings and maintain the cluster. There is the cluster management that allows changing rules for the Dispatcher and add or replace Data Store 1 instances. The administrator can also modify the meta data consisting of event definitions, processing rules and algorithms.

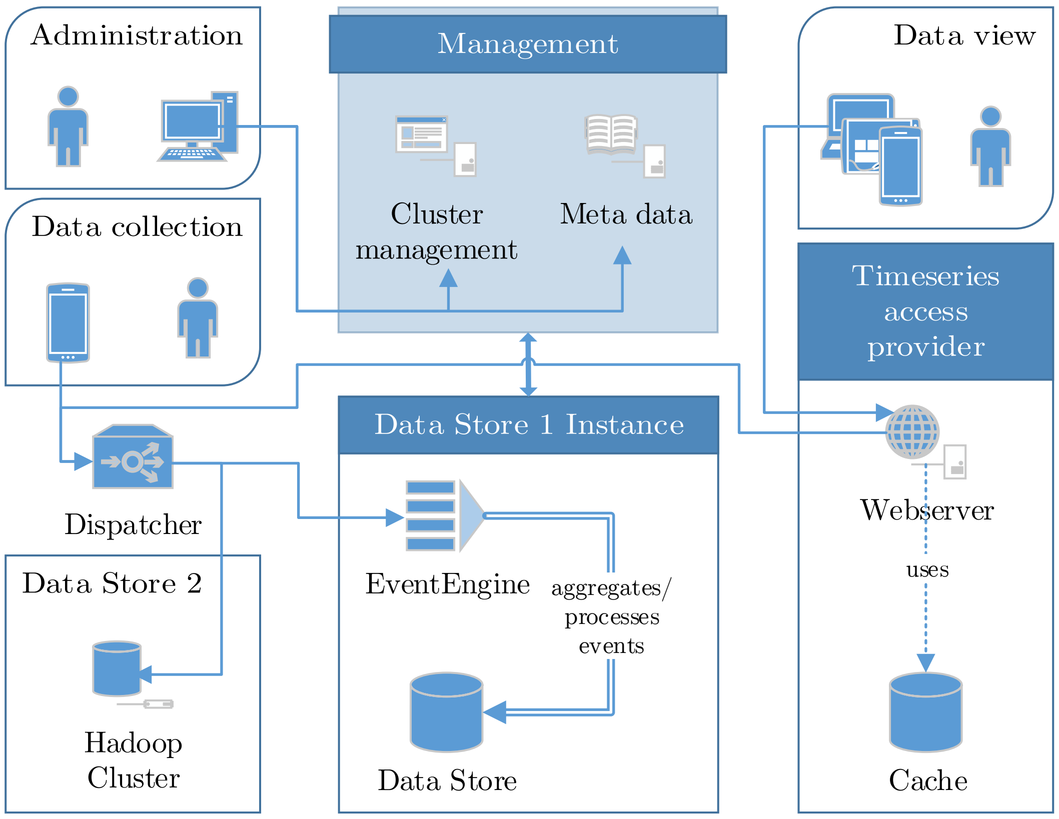

Figure 7.3: Big picture of the Menthal backend

8 Benchmark

Processing large amounts of mobile phone usage events requires first of all the ability to persist and access them efficiently. Modern database systems provide the necessary tools for this. However there is a multitude of database systems and each of them promises to be the best one. One important attribute of a database system is the read and write performance. Benchmarks are the standard way to compare different databases in this regards. While there are many standardized benchmarks like TPC-H [1], most of them follow a generic approach. They test many common scenarios and give a combined impression of the capabilities of the data store. This general impression can be misleading if the use case is very specific. Time series in general and therefore especially mobile phone events fulfill this characteristic.

We decided thereon to design and implement a new benchmark for different database systems to test their ability to cope with our requirements. This chapter deals with the conception of this benchmark, while Chapter 9 introduces the implementation. Chapter 10 and 11 discuss the results of our observations. We begin by defining the objective of the benchmark. Then we discuss the data that builds the foundation of the benchmark. We briefly introduce the existing data set and proceed to demonstrate a way of generating meaningful artificial data. Afterwards we define the test scenarios and based on this informal queries. This provides the necessary tools for a reasonable comparison of different database systems.

8.1 Objective of the Benchmark [MM]

The benchmark shall provide information to choose a database system for Data Store 1. For this we want to compare a defined set of modern database systems with respect to our requirements. These requirements emerge from the characteristics of Data Store 1. We only consider performing simple aggregations, meaning sums

[1]http://www.tpc.org/tpch/

or averages on non-joined data and assume that we can store all data of a single user on a single machine. This is crucial as multiple events from a particular user correspond to each other. A collection of data from multiple database system instances is thus not necessary for answering user centric queries. Rather we can distribute multiple users across different servers to achieve scaling. The term for this approach is sharding. Accordingly we can focus the benchmark on testing a single instance of a database system. This significantly reduces the complexity of the benchmark as we do not have to deal with the interaction of multiple systems.

There are two crucial aspects in the operation of Data Store 1. First of all it must handle the large-scale mass of incoming events. These events occur typically in bursts as the user's phone commits the most recent activity to the server irregularly. The events themselves consist partly of periodical and partly of usage dependent data. The other aspect is the execution of aggregation queries. We process for instance every 15 minutes the most recent data. This also implicates a constraint on the runtime of the queries. It means that they have to finish before the next scheduled queries can follow. An aspect which we decided to neglect is the interference of both write and read operations. While the database system processes the newest data, there will also be even newer data that the users have inserted meanwhile. This will impact the execution time of the running queries. This is however difficult to test and even harder to repeat.

Benchmarks typically do not cover non-performance aspects of the test subjects. Nevertheless the development time also plays a significant role in the realization of a project. Ever since the software crisis the impact of the expenditure of software development is widely known. During the implementation of the benchmark it was necessary to get familiar with the different systems and their APIs. The final comparison therefore takes the experiences we gained during this process into account as well. The performance and feasibility of the different products will be the two evaluation criteria. We are going to weight them differently as the performance is they key factor in our comparison. Finally a mixture of both values leads to the decision which system is the best choice. We will propose this database system to be part of the project's server architecture.

8.2 Test Data [MM]

The benchmark should be meaningful for choosing the right database management system for the use cases of the project. Therefore the test data set should reflect reality as good as possible. Incorporating real data from actual users is the best choice and yields the most realistic results. Unfortunately there was an insufficient amount of data available at the beginning of this thesis. The test system at this point consisted of a few users and less than 50MB of event data. In order to see significant differences between the systems we needed more data.

Data duplication of the existing events would be faulty because it is very unlikely that such a small data set represents the actual smartphone usage distributions. To our knowledge there are also no free available data sets we could incorporate. Data generation based on empirical studies regarding mobile phone usage is a more robust way to gather test data. We can use the empirical observations to extract distributions of common behaviour patterns. Based on these distributions we can sample an unlimited amount of data which still yields reasonable characteristics.

This section first discusses our assumptions and requirements of the generated data. Those ensure on the one hand that data is meaningful and on the other hand provide simplifications which make the generation feasible. It proceeds with the approximation of probability distributions that base on several empirical studies. Finally it describes the algorithm for data generation. We call this the event simulation.

8.2.1 Assumptions and Requirements

Independence of Users

A fundamental assumption is that the data of different users is independent from each other. This eases the generation of events significantly as we can generate each user's data in isolation. This however does not reflect the real world where other people heavily influence a user's behaviour. For instance a user sends and receives messages from other users. Possibly he also installs and uses apps based on recommendations from friends. Nevertheless in the scope of this benchmark the data should be meaningful for each individual user but not across the whole user basis. The data does not have to reflect interactions with other users. For example an outgoing message from one user does not have to be received by another observed user. We can interpret this with the idea that the user's interaction partners are not part of our generated data set. The analysis of data is user centered (cf. Chapter 6) which also makes the independent data generation reasonable.

Single Device

We also assume a user to only have a single device. Multiple devices will not yield additional insights in the tested database systems. This is because we would just get the same usage behaviour more than once for a user. Adding another user to the data generation makes it easier to increase the data size. On the other hand multi-device support poses additional difficulties in the generation and processing of events. Users are unlikely to use multiple devices at the same time, but it could nevertheless occur. The simulation would have to model this which complicates the process. Events may happen in parallel and we would have to distinguish them. The delivery of data is problematic with respect to the storage. Assuming a single device, data is append-only for a single user. With multiple devices this is no longer necessarily true. When one device submits its data, the other devices may still contain unsubmitted data within the same time frame.

Uniqueness of Timestamps

The timestamp of an event is assumed to be unique given the user. Duplicates over all users are possible though. A fine granularity of the measured time makes this reasonable. Devices record events as activities at a certain moment. A high time resolution of the recordings ensures a very small probability that a time duplicate exists. Given this assumption we can uniquely identify events by the combination of a user and a timestamp.

Types of Events

The event generation shall produce two kinds of events: app usage and periodical sensor data. App usage yields unique characteristics as we will discuss in Subsection 8.2.2. We also claim that it contains already much useful information about a user's behaviour. Periodical events like GPS measurements are the counterpart. They occur regularly over the whole day and contain certain sensor information. They should scatter to reflect reality. This means they should not occur at the same time for all users as this would be unrealistic. We do not handle phone calls and text messages explicitly. The former are no longer the primary usage scenario of a mobile phone. They and text messages are more and more submerged in regular app usage by apps like Skype or WhatsApp. They use the Internet rather than the phone network. The same activity thus occurs in a different way.

App Usage Patterns

The generated app usage events should reflect human behaviour well. This means individual users should be diverse. On average the simulated users should behave like real people did in our incorporated empirical studies. We mainly focus on the research of Böhmer et al. [BHS+11] and Khalaf [Kha13]. We consider the following specifications most important to get an idea of mobile phone usage events:

- How often is a mobile phone used during the day?
- When is the phone used during the day?
- How long are the phone sessions?
- How often is a certain application used on the mobile phone?
- What is the duration of using a single application?
- How many applications does a user use consecutively?

Data Delivery

The simulation has to store the data in batches. The requirements in chapter 6 motivate this. Mobile connections do not necessarily intend or allow a real time delivery of data as discussed there. The event simulation should reflect the insertion patterns in order to be realistic. A reasonable approach is to collect events of a user's mobile phone usage for at least one day and then store them all at once.

Reproducibility

There are two reasons for having a reproducible event generation. The event simulation has to generate data multiple times for different database systems. They should receive the same data such that the runtime of the insertion and the queries are comparable. The experiments in general should be reproducible to satisfy scientific standards. The order in which we insert users may have a small impact on the runtime of the insertion and possibly the physical storage. We consider this to be negligible. The events of a single user have to be identical and we have to insert them in the right order over multiple runs though. The test environment has to be static as well. It should be dedicated to the benchmark and not processing any other tasks at the same time. These computations would most likely interfere with our measurements. We have to test different database systems and queries on the same hardware and operating system in order to be comparable. Testing the

Cloud

system in the cloud, for example Amazon's S3, is thus not easily possible. Clouds are characterized by transparency in almost every aspect. This means it is not apparent where some application is executed and what resources are assigned at each point in time. Moreover the load may vary heavily as the cloud shares physical resources among multiple customers. It could also transfer an instance from one physical server node to another. All those aspects can make measurements very inaccurate. There are however some products which provide access to dedicated resources which would make them feasible for benchmarking purposes.

No Real Time Simulation

While data and query patterns of the benchmark should be realistic, we do not want to create a real time simulation. The generation of event data should simulate real behaviour in order to create meaningful data. We will allow however to insert this data instantly as we generate it. This way we can avoid scheduling events and insert more data over time. A real time simulation would reflect the insertion process more realistically but we claim that this does not help in measuring the write performance of the database systems. The inserted data stays the same and by inserting data as fast as possible we can increase the load of the system and thus reach its limit more easily.

8.2.2 Approximation of the Distributions

Usage Distribution

The distribution of phone usage events across the data most likely has an impact on the runtime of queries. In order to get representative results this distribution should reflect the real world in our artificial data set. By looking at a single event we can see that it is important *when* it occurs. For single app usages we are also interested in *how long* they take. Multiple consecutive app usages form an app session. We have to find delimiters for calling events consecutive though. The variable here is

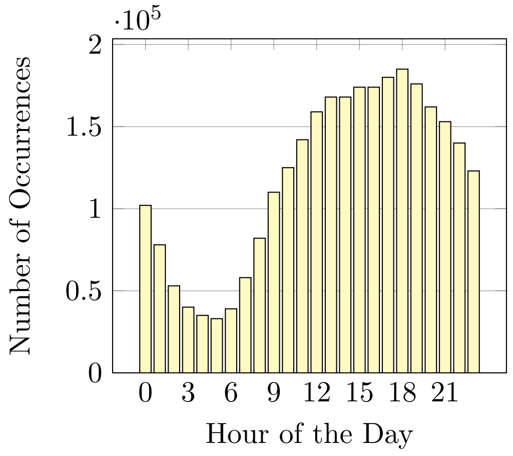

Figure 8.1: Distribution of the number of app usages over the hours of the day [BHS+11]

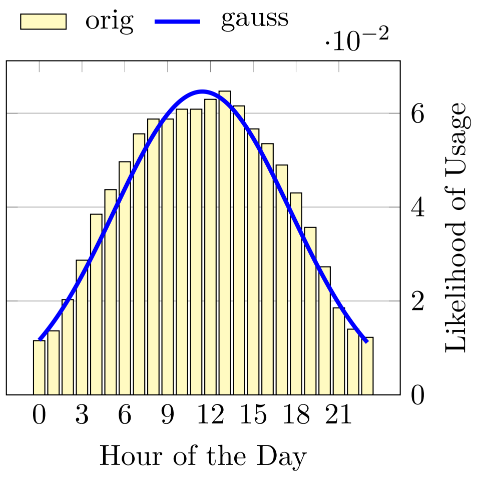

Figure 8.2: Approximation of the 5 hours shifted number of apps distribution

how many apps are used during this app session. We use the empirical results from [BHS+11] and [Kha13] to extrapolate probability distributions that answer these questions. These distributions should be as simplistic as possible.

Day and Night Cycle

Looking at Figure 8.1 it is apparent that human phone usage is not uniformly distributed. We can explain this by considering the day-night cycle in which we live. The participants use their phones mostly during the day, peaking at around 5 pm and being at its lowest at around 4 am. The generation of events should follow this observed pattern. To achieve this we first approximate the empirical results by a fundamental distribution. At the first glance no simple probability distribution seems to fit though. However we have to keep the nature of the data in mind. As the likelihood of usage is shown over the hours of the day, the pattern recurs for the next day as well. If we duplicate the data and append it e.g. on the right hand side, a possible underlying distribution gets visible. Figure 8.2 shows a Gaussian overlaying the app usage distribution. Although it does not completely fit to the empirical results, the error is acceptable for us.

Being able to choose a reasonable point in time for the generation of a event is only the first step. In case of app usage we would now only know the starting time of the application. In order to determine the end time we have to know the duration of the particular app usages. Böhmer et al. found that the average app usage duration depends on the time of the day. We can for example observe in Figure 8.3 that the participants interact with their apps longer in the morning hours. During the day the interactions become shorter, most likely reinforced by other distractions in the daily life of the participants.

Finding Distributions

As in the usage distribution case we want to repeat the reduction of the empirical distribution to a fundamental one. And again the key idea is to keep the periodicity of the data in mind. Shifting the data by 7 hours reveals a single peak, which again looks like a Gaussian distribution. Fitting the data by a Gaussian distribution however leads to a rather big cumulative error. A first hindrance is that the distribution does not aim towards zero. Subtracting an offset is an easy solution for this as we can simply reverse this by adding the offset again later on. Nevertheless the slope of the curve is still too steep for a Gaussian. Selecting a lower standard deviation raises the slope, but also increases the highest value. We could therefore not significantly reduce the overall error in this way.

Of course there is a multitude of other probability distribution functions. We can try to find appropriate distributions by looking closely. There are also tools that automatically try to fit a set of prominent functions to a given data set. We used EasyFit[1] as one representative of this tool range. While it was not able to find a good fit directly, removing the apparent offset brought more success. A logistic function fits the distribution quite well. Adding the offset again, we are able to fit the data as Figure 8.4 shows. Intuitively we can interpret this distribution as a logistic distribution on top of a uniform one.

[1] http://www.mathwave.com/

Phone Sessions

Knowing the start and end time of individual app usages, the next step is to tie them together to phone sessions. While the average user in America has e.g. about 41 apps installed [The12], he does not use all of them each time he uses his phone. In fact Böhmer et al. found that in nearly 90% of the times the smartphone users did not use more than two apps within a session (cf. Figure 8.5). Most of the times they used only a single app before they ended the phone session. Of course there are more extensive app sessions in which a user uses much more applications. Even though they are infrequent, the distribution is still of interest for us. We can find an inverse relationship like the number of apps per session in many scenarios. One example is the number of friends on Facebook or the number of links to a website. There are very few websites with very many other websites linking to them. Most other websites have only a few. In information retrieval we model this as a Zipf distribution [MRS08]. As visible in Figure 8.5 the Zipf distribution is also suitable in our case.

The distribution function approximations discussed in this subsection enable us to create the event simulation. This event simulation produces phone usage events that resemble a real world behavior based on empirical research. Those built the foundation for the execution of queries.

8.2.3 Event Simulation

The Event Simulation is responsible for providing the core test data set of our benchmarks suite. It generates meaningful (cf. Subsection 8.2.1) phone usage data on which we can execute our queries. It consists of three parts: (1) the actual event generation, (2) the sampling of timestamps and durations according to Subsection 8.2.2 and (3) the persistence of the generated data into the supported database

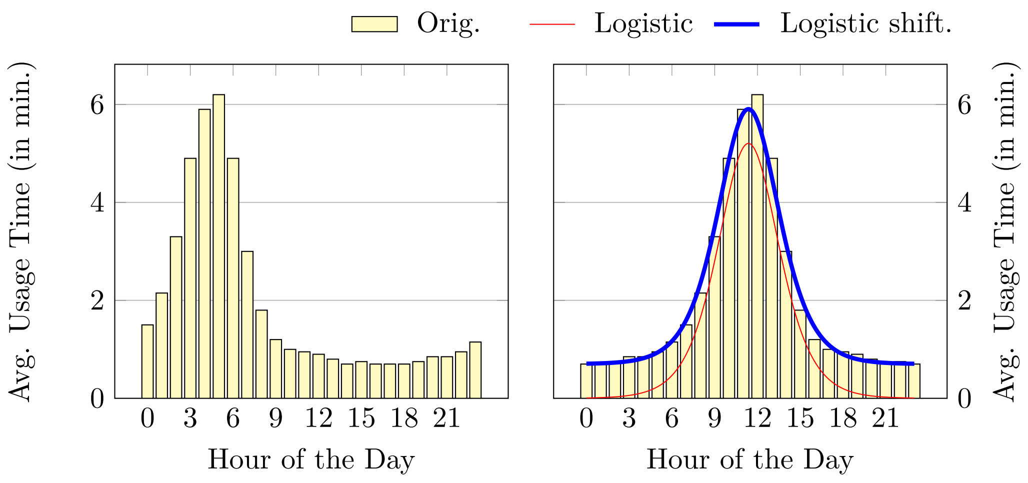

Figure 8.3: Daily average usage duration of opened app per launch in minutes. [BHS+11]

Figure 8.4: Approximation of the shifted distribution with a logistic distribution.

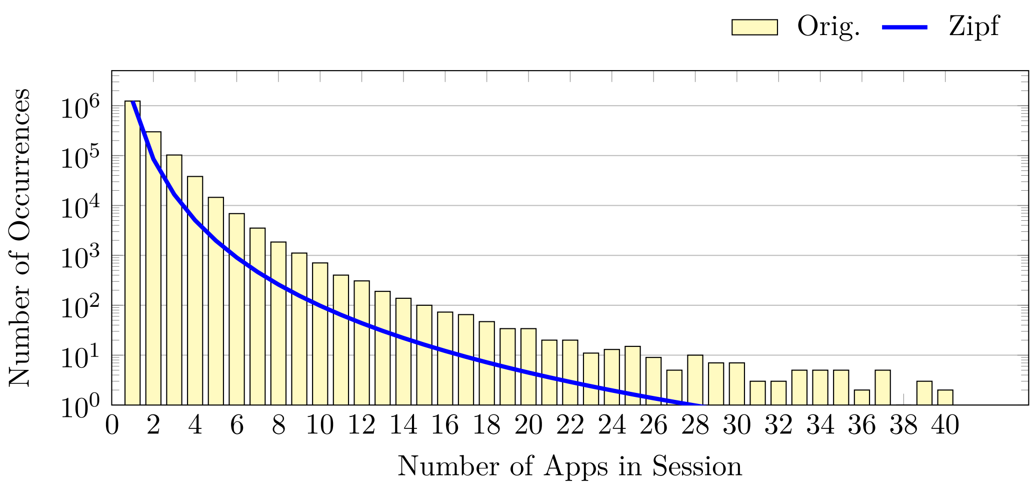

Figure 8.5: Approximation apps in session distribution

systems. The main parameters specify the amount of data it generates. These include a starting date, the number of days and also the number of users it should generate. Additionally there is a simulation seed that determines random sampling processes in the algorithm. In the following we describe the process of the event generation as well as the configuration parameters. This concept is the foundation for the concrete implementation.

Massive Data

The generation of a huge amount of data is the goal of the event simulation. A crucial point for this is the throughput i.e. the amount of events it can create over time. In order to exploit the resources of a modern computer, the program should be multi-threaded. Otherwise the achievable throughput could not scale with the number of processors or cores respectively. The first step is to distribute the workload across multiple threads. We can achieve this by assigning the users to different threads. This is possible as there is no interdependency between two users. The responsible thread creates all events for a single user. Therefore different threads do not need to share their data. The following description thus restricts itself to the work flow within such a worker thread.

Figure 8.6 shows the basic algorithm of the event simulation. First it builds a list of user IDs which it then processes one by one. If there is still a user left to generate, the algorithm fetches a new ID from the list. It then proceeds to generate periodical and app usage events for all the days that are to be simulated. The algorithm stores the data in the currently selected database and it then continues from the start again. We will now describe the user generation as well as the actual event generation in more detail.

User Generation

The generation of a user starts with a unique ID. He gets an a priori app distribution which states how likely he is to use a particular app from the simulation's app pool. This app pool consists currently of a fixed amount (9) of apps and is identical for all users. The user also gets an expected daily phone usage duration in

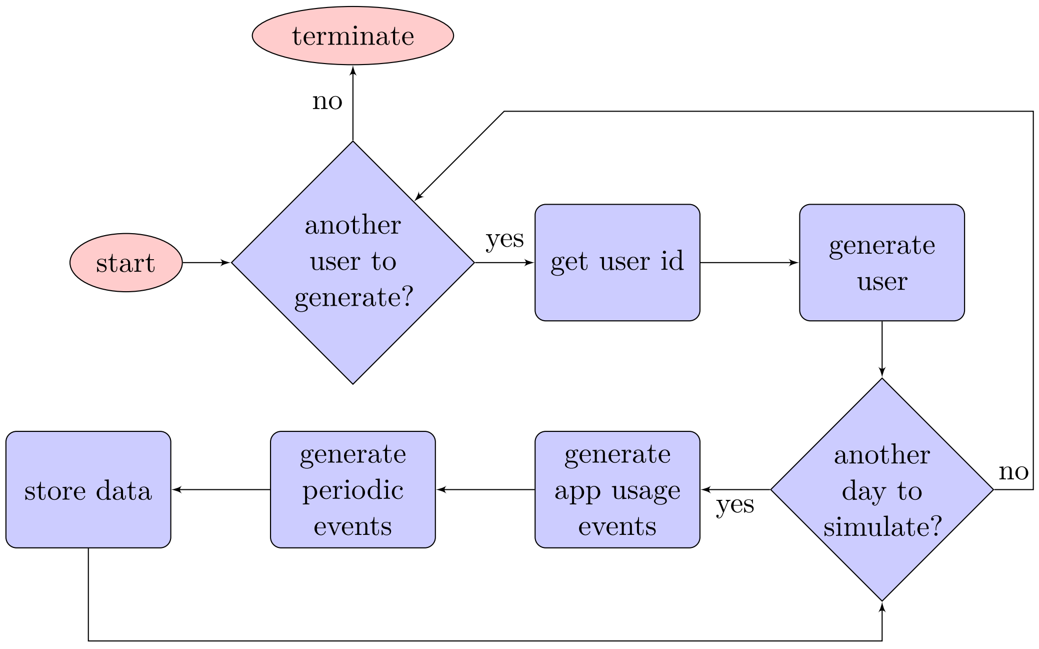

Figure 8.6: Basic algorithm of the event simulation

form of a normal distribution with mean and deviation parameters. The algorithm computes these two parameters by incorporating another normal distribution using simulation specific parameters. It also has to determine the occurrence of the periodic data events. While the period (by default 20 minutes) itself is part of the simulation configuration, the exact points in time are user specific. We are able to compute them from one particular occurrence which represents a starting point. All subsequent occurrences depend on this fixed occurrence plus multiples of the period. The algorithm thus only has to sample the first occurrence. This way the periodical events scatter throughout the hour over all users. Finally the algorithm stores the user´s data by incorporating the selected database system engine (db engine).

PRNG

As mentioned in the requirements the simulation has to be reproducible. On the other hand it should be fast and thus has to be parallelized. This poses difficulties as the order in which we execute operations can differ between two runs. This is especially critical when we make use of pseudo random number generators (PRNG) for random sampling. A PRNG is basically a stateful function that computes numbers that appear to be random[1]. Each call of the function returns a new number and transists it to the next state. We initialize it with a number called seed that determines the number series. Another PRNG with the same seed will generate exactly the same number series. This makes a random process reproducible. If multiple threads however access the same PRNG the outcome might differ in subsequent runs. A single thread can be faster or slower in two executions due to the

[1]These numbers have the same statistical properties as true random numbers

scheduling of the operating system. Thus a thread A might read a random value before a thread B in the first run and the other way around in the next one.

A solution to this problem is the assignment of an own PRNG to each thread. This way each thread will get the same random number series on every execution. However we can not guarantee that the user assignment will be constant by using a thread pool. Thread A might generate user 1 in the first run, but in the second run thread B is responsible for it. The properties of the generated user would thus differ depending on the execution.

A better solution is to assign a PRNG to each user and use it for all sampling operations for this user. This makes the operation independent of the current thread and the order of the user generation. To make the user's PRNG generation itself reproducible, it also has to be independent of the execution. The algorithm thus has to generate the seed deterministically out of the simulation seed and the user's ID. An example would be to take the sum of both values. This way it is unique among all users and deterministically computable.

App Usage Event Generation

The app usage generation begins with calculating the targeted phone usage for the current day. A normal distribution with the user's mean and deviation determines the outcome. Then the algorithm proceeds creating phone sessions until the usage time is exceeded. A phone session consists of an unlock event, one or more app sessions and a lock event. An app session contains one or more `WindowStateChangeEvent`s which indicate that the user is switching to a new application.

At first the algorithm determines the starting point of the phone session. It does this by sampling a random timestamp according to the distribution of the total number of app utilizations visible in Figure 8.2. This makes it more likely for a phone session to happen during the daytime. Next, a Zipf distribution (cf. Figure 8.5) determines the number of apps used in this session. According to this number the algorithm then creates app sessions. For this it first chooses a random app according to the user's a priori app distribution. Then it samples the usage duration using the daily average for the current hour of the day (cf. Figure 8.4) as a mean and deviation for a normal distribution. Finally it ensures that the generated phone session does not overlap with any other session of this day and uses the selected database system engine to store the events.

Periodical Event Generation

The periodical data generation for the current day starts at the given user specific offset in the first hour of the day and recurs every 20 minutes (configuration dependent). We perturb the specific time slightly using a normal distribution with simulation specific deviation. This reflects uncertainties and processing limits on real devices. It is unlikely that measurements that we schedule to be 20 minutes apart really satisfy this with an accuracy of less than a microsecond. Each periodical event contains a float value which we sample from a normal distribution with a configuration dependent default mean and deviation. We can interpret this value e.g. to be the cellular coverage of a mobile phone (signal strength).

Storage

The final step of the event simulation is the storage of the events. While the other steps are generic, the storage implementation depends on the tested database system. Chapter 9 describes the strategy for the different candidate systems. This includes the definition of a schema as well as the implementation of the data insertion.

8.3 Test Scenarios [CB]

The event simulation enables us to create as much event data as we need for our database system comparison. Having this data is the first step, posing meaningful queries on this data set is the second one. Posing a meaningful query is twofold. First of all the query should make sense in the real world. This means it should answer a question that we would actually ask in the application domain. On the other hand a query should test a certain aspect of the data and the data access respectively. Having this in mind we created a compact query set which covers as many different access patterns as possible. We state the queries in an informal fashion by using natural language. The next step is then to implement them for the different database systems and storage types.

8.3.1 Querying App Usage

Human app usage presents particular patterns that we described in Subsection 8.2.2. We designed two queries on the simulated app usage. Both queries yield distinctions into two specialized types. One type tests the query performance on the whole data set while the other one states a temporal restriction. In this way we want to find out how well the database system supports range queries.

App Usage Duration

The first query computes the average app usage duration over the hours of the day. To calculate this average we first have to find all app usage durations in the data set. We can then group them by the hour of the day and then compute the average for each hour. A single app usage starts with opening an app and ends

Scenario	*Query*
(I) App usage	
(I.1) App usage duration distribution	Show average duration of app usages over all users over the whole dataset (or only last 14 days) for the hours of the day.
(I.2) Number of app usages	Show for each app how often it was used (in the morning) over the whole period of the dataset.
(II) Periodical data	
(II.1) Longest period of bad coverage	Find for each user the longest period (length or interval) with coverage of $< 50\%$

Table 8.1: Overview of queries

when a user either opens another app or locks the screen of the mobile phone. Beginning with a `WINDOW_STATE_CHANGED` event we thus have to look for the next `WINDOWS_STATE_CHANGE` event or a `SCREEN_LOCK` event. The *next* event means the first one with a greater timestamp value from the same user.

A variation of this query is to compute this distribution not over the whole data set but only for the most recent events. An example is computing the app usage duration distribution for the last 14 days when having data for a total of 30 days available. This tests how efficiently a database system can access and provide the most recent data. This is an ability which we will need in many different cases in the future.

The motivation for this query is twofold. On the one hand it serves as a test whether our generated data really reflects the distribution we tried to mimic. On a more technical level it also tests the ability of finding two consecutive events that fulfill certain requirements. It is unclear how well our test systems will cope with this access pattern. This query will yield insights into this.

Number of App Usages

The next query wants to retrieve how often apps are used over a period of time. In the most simple form it should output for every app that is known, how often it was used in total (over the whole data set). To compute this, we have to look at all `WINDOW_STATE_CHANGED` events and retrieve the app name. Then we have to increment the corresponding counter.

As a variation we chose to restrict the query to app usages in the morning. Doing so allows us to get an impression how well a database system supports the retrieval of events within a time range. In contrast to the variation of the app usage duration query it is not only necessary to find a split and return the rest of the data. The events which occur in the morning scatter throughout the whole data set. This poses thereby a greater challenge for the tested database systems. Another difference to the first query is that we now need the associated event data of an event. In the first case knowing the event type suffices to answer the query. This time however we need to know *which* app was used that resulted in the occurrence of the current event.

Another motivation for this query type is to test the ability of the database system to calculate simple aggregations. Most of the recurring queries in Data Store 1 will be of this type. Counting and summing up values already gives a lot of information regarding the user's stress level progression. A database system has to support this well if we want to incorporate it.

8.3.2 Querying Periodical Events

Periodical data shows entirely different characteristics than human app usage. This is because we know (roughly) when the next event will occur. The exact time may diverge slightly because of technical reasons. A database system should cope with this kind of data as well. This provides enough motivation to design a query specifically to this type of events.

In order to get some intuition we continue our example from Subsection 8.2.3 and interpret the periodical events as information about the current cell coverage. The cell coverage is a float value between 0 and 10, 0 being no connectivity and 10 meaning maximal signal strength. A value of below 5 signals a bad cell coverage. Having this information we want to find the duration of the longest interval of bad signal quality for each user.

The motivation of this query is again twofold. On the one hand the query incorporates a new kind data with different characteristics. On the other hand it also tests another important ability of a database system. This is how well it supports finding a contiguous series of events that fulfills a given predicate. Table 8.1 summarizes our query set.

8.4 Metrics [MM]

Performance

To compare the different database systems we have to find meaningful metrics. To have sufficient resources for basing a decision later on there are several metrics regarding the performance. There are also nonperformance metrics regarding the development experience. The most important performance metric is the runtime of the queries. This is a time interval metric that helps easily distinguishing the different systems. Another one is the write throughput of the systems. We can measure this while inserting the generated data set. It denotes on the one hand the amount of events per minute we are able to insert. On the other hand it shows the time for inserting the whole data set. In this case it will be also interesting to measure the progression with minima and maxima than just measuring the average. For each metric we have to consider the amount of data as well and measure the size of the database in the number of events, the number of users and the total size in gigabytes. These numbers help to get a hint about the scalability of the different systems.

Ease of Use

Nonperformance metrics describe how well the systems are to use. Setup complications represent the effort for installing the database systems and launching them for the first time. The codability is the most important factor here. It denotes the complexity of realizing the queries in the specific language or interface. During the testing we can compare the maintenance efforts of the systems. Furthermore it is important to have a proper support for the systems. This means having a good community, which answers questions to unclear features or gives support regarding complex solutions. And it means at the first place having a good documentation of all processes and the data model. We cannot transfer these metrics easily into numbers or a scale. Instead we will give rather rough hints about these parts being well supported or rather complicated. It is also to mention that these metrics are more subjective and thereby inaccurate compared to the performance metrics. Nevertheless they are very important for decision-making regarding which database system provides the best overall benefits for being part of the backend described in Chapter 6.

8.5 Setup [MM]

The setup describes the realization of the benchmark regarding the ideas for data generation and query evaluation. We presented these ideas previously in this chapter. For this purpose we inspect the three components and investigate how we should create the test environment before executing the benchmark.

Data Generation

The data generation provides the data set for the queries. It is on the one hand necessary to create the same data set for each database system. Our generation process ensures this. On the other hand it is important to create a meaningful size of the data. Furthermore it is reasonable to create multiple data sets such that we are able to get measurements about the scaling properties of the systems.

Query Execution

The query execution returns timed results that represent the execution time. It should also ensure a good accuracy of the response times. To achieve this it is useful to re-run the queries several times such that we can take the average of the runtimes as a result. This gives much smoother resulting time interval than just taking one value. By doing this we can reduce the chance of having outliers drastically. Outliers can occur by having unexpected system operations which slow down the overall system remarkably. For this we chose five iterations of query processing to get meaningful results. In fact it is then also possible not only to look at the average but taking the median and variance into account. This is only necessary if the values deviate significantly.

Virtualization

Caching

Before and during the query execution there are a few general considerations to investigate. The hardware should not change during the processing of the queries. This sounds obvious but due to virtualization it is not always simple to ensure this. Having different resources would lead to incomparable results. We need to ensure that no other processes interfere during the query processing. This means avoiding processes which are not essential for keeping the system running. Examples are other database systems or foreground processes. Another aspect is taking care of the cache or the temporary files of the operating system as well as the database system. It is necessary to keep track of them and delete them if necessary. Cached results can be significantly faster such that the resulting time intervals would not be meaningful anymore. By having five query iterations the caching would heavily influence the resulting average.

Test Environment

The test environment has to consist of exactly one server. This is because the sharding strategy allows us to focus on a single machine to get insights about the test systems. There are three ways for locating the server, more precisely getting a real machine, a virtual machine or a cloud instance. The cloud solution would be simple because there are many providers out there like Amazon or Google that could provide us any resources we would need for benchmark. The drawback is that the cloud does not offer dedicated resources in general. This means providers do not guarantee resources and we really do not know what is happening on the parental server exactly. For instance, other users could use the same real machine and claim the resources for their processes. Meanwhile we would have unstable system resources like RAM and computation power. We mentioned the general consideration about cloud database systems in Section 3.4. Another scenario is that the cloud swaps the virtual machine entirely to another physical machine. This is even worse for testing different systems. The reason is that it would lead to completely different hardware conditions such that the results are not comparable at all. The cloud is only a solution if there are guaranteed system resources and the provider ensures additionally the physical location of the instance for the time of the tests. The same applies in general for a virtual machine but it is much more unlikely that these problems occur. If the virtual machine is able to claim a fixed amount of system resources, i.e. a fraction of the total resources, the results are comparable. The standalone real machine would be the best solution as there are no interferences with possible other users. Despite that the virtual machine is more flexible than the real machine because it can allocate the resources more flexible. This is of course only true if a server is available with more resources than necessary.

9 Test Suite Realization

The previous chapter described the basic idea and the process of the benchmark from the flow of the event generation to the type of queries and the defined metrics. This chapter treats the implementation of these concepts which forms our test suite. It starts by describing the implementation of the event generation, called the event simulation. This first section tackles the issues of selecting an appropriate programming language and the parallelization of work. It also discusses the interference of the event generation and the insertion into a database system on the same physical machine.

The second section deals with the connection of our selected database systems to our test suite. For each system it discusses how event data can be stored in it. All systems are flexible enough to allow different approaches to this. The corresponding subsection either reasons why one approach is superior or decides to line up several approaches against each other. Those competing approaches within a system will be compared in Section 10.2. Once we defined the schema for a system, we can start inserting data into it. The next step is to implement a connector for the event simulation that allows the insertion of the generated events into the database system. Finally there needs to be a realization of the queries. The runtime of those queries is the main focus of our benchmark. The implementation of the five queries (three scenarios, two sub scenarios) is thus the final aspect of each subsection.

The third and final section discusses the bundling of the different components of the benchmark. It describes how the event simulation and the different database connectors work together. Furthermore it shows the conception and implementation of a simple query framework which executes the queries. This includes taking measurements of the runtime and repeating the process as desired.

9.1 Event Simulation [CB]

Programming Language

We could implement the event simulation as described in Section 8.2 in any programming language. Java is always a good candidate, as it is the most widespread programming language in the academic world. Using Java ensures that there are many programmers that are able to work on the code in the future. C or C++ are among the most performant programming languages that exist. They would make a reasonable choice as well. Our choice however fell on Python, which is one of the most used scripting languages. Being a scripting language, Python code is interpreted rather than compiled. Interpreting code is inherently slower than executing complied code, making Python a bad choice for performance dependent applications. However a lot of Python libraries are interfaces of compiled C code. This vanishes the slow-down by concept. Later on we discuss the decoupling of event generation and event insertion, which makes the performance of the former one less important.

Codability

The main reason for choosing Python is the ease of the development. Python code is easy to write because of the dynamic type system, the functional programming aspects and the huge support of libraries. A dynamic type system means that the type of a variable is inferred and the programmer does not have to specify it himself. Functional programming among other things allows using functions as parameters. Both properties can make the development process much faster. The greatest benefit of Python however is the vast amount of available libraries. With respect to our work there are two categories of libraries. First of all libraries which provide some general utility, and second libraries which provide database connectivity. General utility covers for example libraries like `numpy` which provides a multitude of numeric and statistical functionality. It also covers libraries like JSON or MsgPack which support the serialization of data which is necessary when we exchange data between different programs. Database connectivity is of course especially important in our context. The popularity of Python ensures the support of most of the available database systems.

Parallelization

Section 8.2 mentions the importance of parallelizing the generation of event data. A single thread of execution cannot exploit modern computers effectively. All work would be done on one processor core while the others would stay idle. Threads and processes can achieve a parallelization of computation. A process is a unit of execution, consisting of code and memory. The operating system manages the scheduling and execution of multiple processes and assigns them to available processor cores. A thread is a light weight process. A process can spawn many threads which all share the process's address space. The common memory makes it easy to share data between the threads. The operating system executes the threads in parallel. In contrast to processes threads are less costly to create.

The general approach of parallelizing the event simulation follows multiple steps. First of all we create a user list or queue. It contains all users, that are to be created, in form of an ascending user ID. A fixed pool of threads or processes is responsible to work off this queue. The size of this pool has to be adjustable. Effectively it should be the amount of (virtual) cores of the computer. This allows the assignment of one thread to each (virtual) core. When a thread or process starts, it chooses a user by popping the queue. It then proceeds to generate the user data and subsequently generates all the events for this user. When this is finished it draws the next user and repeats this process until the queue is empty. The user queue is thus the critical point that all running threads have in common. We have to ensure that we assign each user to exactly one thread.

Using threads or processes for the pool in theory has little impact. Threads are cheaper to create, but the pool size is fixed. We have to create them only once. Threads share the same address space so it is easier to share data between them. But the only sharable data is the user queue. The implementation consists then of multiple processes using message queues. In Python the difference between creating a process and a thread is not significant either. It is either a single call with a function as parameter or a subclassing of the `thread` or `process` class. In practice it is however crucial to use processes in this case. A first implementation using threads could not fully exhaust the resources of a test system. Even though there were enough threads to use all cores some of them were idling. This is due to the implementation of threads within CPython. More precisely the Global Interpreter Lock (GIL) prevents multiple native threads from executing bytecode at once [Pyt]. A multi-process architecture is able to avoid this problem and thus make efficient use of all cores of a computer.

Multiprocessing in python has some additional drawbacks that we did not mention yet. There are platform dependent parts. Python code in general is executable on every operating system that has a Python interpreter. Multiprocessing however is different e.g. on Linux and Windows. Windows has no `fork()` operation which leads to some drawbacks. The general conclusion is that a multiprocessed python program which runs on Linux not necessarily runs on Windows. Additionally debugging a multiprocessed program gets harder as Python debugging tools are not able to cope with it by default.

Decoupling Generation and Insertion

The event simulation algorithm inserts events into a database system right after their generation (cf. Figure 8.6 on page 103). In practice this poses some problems. When we execute the generation and persistence on the same machine, we cannot avoid interference. The event generation should run as fast as possible in order to not be the bottleneck while benchmarking the write throughput of a database system. If we generate the data too slowly, a fast database system cannot reach its full potential. To achieve the highest possible throughput, the event simulation should run on all cores of the computer. The pool size should equal the amount of cores in this case. The tested database system also needs resources to store the data though. It is not clear how the operating system divides the available resources among those two different applications. A possible solution would be to generate the data on a separate machine. This was not possible within the scope of this work.

We can break down the process into two steps though. First we generate all events and store them on disk. Then we read them from the disk and insert them to a database system. It is however important to store the events on one disk and let the database system use another one. Otherwise the write process of the database system would be interrupted again and again by the process that reads the generated events from the same disk. If only one disk is available we can avoid this by loading all generated events into the RAM. We would not necessarily have to store them on disk in the first place then. As the RAM is volatile and we would like to use the same event data for multiple database systems, it makes sense to persist them on disk anyway. One problem then is that the RAM space is very limited in comparison to the disk space (cf. Section 3.2). In order to use the available space efficiently we have to use compression. Fast algorithms like Snappy are predestined for such a use case. When storing generated data, they reduce the amount of necessary space. This also helps persisting the data faster by reducing disk I/O.

When we load data into main memory in the second step we have to ensure to decompress the data piece by piece. Decompressing everything at once would simply cost to much space and defeat the purpose of compressing it in the first place. A reasonable approach is to compress all data of a single user individually. We can then spread the compressed data for multiple users across multiple files or simply append them into a single one. The latter allows sequential reading and is thus more performant. Now we can assign the compressed data of a single user to one process. The process decompresses the data and performs the insertion. Afterwards it can delete the compressed and decompressed data from the RAM. Even when using an in-memory database like Redis the memory load could theoretically be constant. First the data is in the memory of the insertion program and then in the database system.

9.2 Database Connection [MM]

We have to store the events generated in the event simulation in the selected database systems and finally query them. The following subsections show how we can store data within the systems. It also describes the connection to the database systems to actually transfer the data. Finally it shows the data access in the scope of our test scenarios. The subsections thus give a further impression of what is possible to achieve with the systems and where problems arise.

9.2.1 EventStore

Using the EventStore in our scenario reveals an impedance mismatch. We first investigate this problem in detail and describe workarounds for solving this issue. Then we describe the interface and the Py4J connection to our Python environment. Finally we present the queries and look at the implementation in particular.

Schema

Streams

The straight forward idea is to use one stream per event type. Streams have a fixed (relational) schema. If we assume that the attributes of an event type do not change, this approach matches. The problem is that in our scenarios events are not submitted in real time. The concept of the EventStore is to store events as soon as they are produced. This means that once data with a given timestamp arrives it is evident that no older data will arrive later. The EventStore exploits this knowledge in order to optimize writing as well as reading performance. New data packages arrive in the right order and it appends them only such that it can write them to disk accordingly.

In our scenario this is not true. We have multiple sources (users) for a given event type. If one user submits his events, there might as well be another user who has data which is still older than these ones. This is the case because the user's device collects the events and submits them as a batch later on. To store an event we have to be sure that we have already stored all older events in the stream. This is because later on we can add only newer events. We would have to buffer all events until all users committed their events, then sort them and insert them in the right order. We would need to postpone all operations on this data like performing aggregations until then, which is not feasible.

Workarounds

We identified two workarounds which let us use the EventStore nevertheless. The first idea is to use one stream per user instead of one stream per event type. The file system would effectively act as an index on the user as we have one file per stream. The schema of a user's stream has to be capable of storing all possible event types. A generic schema is able to achieve this. This schema consists of an integer field for the event type and a string for the attributes. Attributes are basically a map of keys and values. We can serialize them and store them as a string value. Strings have a fixed size in the current version of the EventStore. It compresses the event data however which compensates unused space in a string. We can serialize the attributes for example using JSON or MsgPack. As the structure of the attribute map is well known to us, we can avoid this overhead. The serialization simply consists of key value pairs where we separate the key and the value by a comma. We separate subsequent key value pairs with a semicolon.

This generic approach has a number of drawbacks. First of all the serialization and deserialization of attributes poses some overhead. We cannot effectively query single attributes using the `queryAttribute` method. The approach also hinders the effective generation of secondary indexes on event attributes. A big drawback is that we create a huge number of streams when the system handles many users. Each stream corresponds to one or more files and the operating system has to handle them. We quickly reached the operating system's limit of open files. For this reason the developers of the EventStore introduced a `suspendStream` method. This method enables the deactivation of streams and thereby closing of currently unneeded files. On the next access the store reactivates the requested stream.

A different approach is to store the event's (occurrence) timestamp as an additional attribute. In this scenario each stream corresponds to a single event type again. The streams schema contains a field for each attribute plus an additional timestamp field. We effectively get two timestamps for each event. The first one is the moment of insertion, called system time. The second one is stored as an additional attribute, called application time. The EventStore thus orders the events by their insertion time rather than the time they actually occurred. The `queryAttribute` method still allows range queries on the application time. This approach has some drawbacks as well. First of all it is not clear if we are ever interested in the insertion time of an event. If we never use this information, we are effectively wasting disk space by *misusing* the event's timestamp in this way. The EventStore organizes the data by the insertion timestamp instead of the actual time of occurrences. This however is negligible. When a user submits a batch of events, the store inserts them one after the other with ascending insertion timestamps. This means we can still perform sequential scans on a user's events efficiently. On the other hand we limit the amount of streams to the number of event types instead of the number of users. This potentially makes this approach superior to the first one. Unfortunately the current version of the EventStore does not provide a proper implementation of this feature. This means we have to stick to the first approach within the scope of this thesis.

Interface

The EventStore is an embedded Java database and provides no inherent networking capability. In order to apply our event insertion of the simulation, we need a bridge from our Python to the Java code. Py4J offers a rather elegant solution. It is a library that consists of two parts, one for Python and one for Java. On the Java side we instantiate a Server object which is capable of receiving remote procedure calls. On the Python side we import the Py4J module and can connect to this server. Using the module we are able to call Java methods from Python code. This approach obviously provides some additional overhead. It uses the network card to deliver the data. This on the other hand is realistic as in our real scenario we actually transfer data from a device to the backend via network.

Queries

The query implementation consists of two parts. First of all we created a simple framework to execute our queries in a parallel fashion. The same considerations as in the generation of the events hold when we query them. In order to exploit a machines resources, we have to execute multiple threads of execution. The second part is the actual implementation of the queries we described in Section 8.3.

We use this general query architecture for the other database system in a similar fashion. Figure 9.1 shows the class hierarchy. The two core concepts are the abstract `Handler` and `Worker` classes. A `Worker` instance solves some part of a problem while a `Handler` organizes the execution of multiple `Workers`. A particular `Handler` first of all creates the `Worker` instances. It also provides a method which the `Workers` can call to get a new piece of work, e.g. the next user it shall process. In the end each `Worker` uses the `submitWork` method of its corresponding `Handler`, which then in turn merges the results. For each query we created a concrete implementation of `Handler` and `Worker`. The main class `QueryRunner` instantiates a concrete `Handler` and runs the query.

For querying the app usage duration, each worker processes one user at a time. It starts by querying all events of the user for the specified time frame (last 30 or 14 days):

```
eventStore.getHistoryRange(user, keyStart, Long.MAX_VALUE);
```

The `keyStart` timestamp thus is either 30 or 14 days in the past and the B$^+$-tree index allows us to find the first data item efficiently. The worker now runs through all events in time order and filters for `WindowsStateChange` and `Lock`/`Unlock` events. It then calculates the length of app sessions defined by those events. When no users are left to process, the worker submits its work to the handler who merges all results.

We divide the calculation of the number of apps usages into two cases. In the first case we are interested in the total number of times people used a particular app. A worker just scans through all `WindowsStateChange` events of a user and counts them. In the second case we are only interested in the appd uses in the morning. To make use of the index structure we process the events day by day. For each

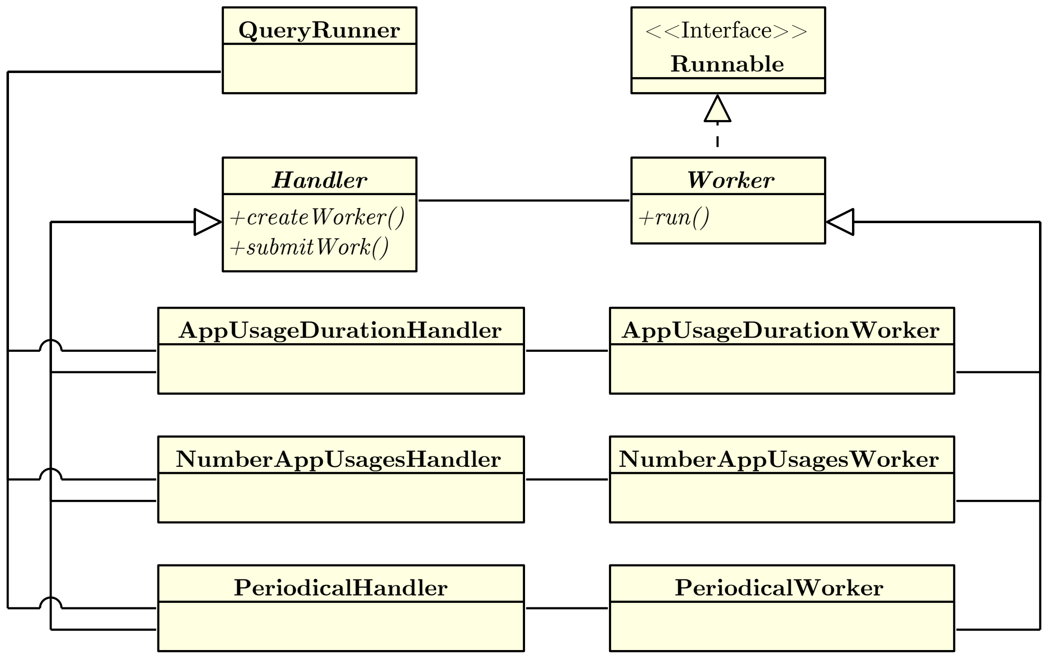

Figure 9.1: Query architecture

day we query the events that happened at this particular day between 5:00 and 12:00. We proceed to count the `WindowsStateChange` events and merge them on submission.

To calculate the longest interval of bad signal quality we only have to look at `Periodical` events. Currently we have to skip all other events as the EventStore lacks the `queryAttribute` implementation. Having these events we run through them in time order. If the assigned value of the current event is below the threshold and we cannot find a start event for the interval yet, we set it as the start event and move to the next event. If the next events value is above the threshold, we calculate the interval length from the start event and set it to `null`. We finally compare the computed interval length to the current maximum and preserve it if it is greater. We continue with this algorithm until all events are processed.

An early observation is that the EventStore does not provide much support in the implementation of these queries. Apart from querying events in a given time range, we have to do a lot of work ourselves. We claim that the implementation of the `queryAttribute` method would improve the query process significantly.

9.2.2 HBase

HBase offers a rich feature set in comparison to the EventStore. This leads to various ways for storing the data and exploiting the implicit indexes. In this subsection we first describe different schema definitions for our use case. Again we have to bridge the gap between the Python and the Java world. After describing how we accomplish this, we turn to the query formulation. This incorporates the query framework we already describes in the previous subsection.

Schema

The first step for using HBase is the definition of a schema in form of tables and column families. In our scenario there are two feasible approaches. The first one would be a single table which stores all information. For each user we would introduce a row, containing the user's information and all of the events as columns. As the set of columns is not fixed, it would be possible to do this. Crucial operations on the events are range queries over the time. These are only possible on rows by defining a row key range. Thus the single table approach is not reasonable.

In the other approach we need two tables: `users` and `events`. For the `users` table a single column family suffices that stores all of the user's information. This table is not critical as it is rather small (a couple thousands entries) in comparison to the `events` table (millions or billions of events). The user's ID suffices as a row key. The design of the `event` table requires more effort.

We can divide an event's data into general information (user id, event type, timestamp) which is common for all events and additional information (event attributes) which differ between event types. There are queries which only operate on the general information, while others incorporate the additional as well (cf. Section 8.3). If a query does not need the additional information, it should be able to leave it out when retrieving the events. A `Scan` object offers this functionality. If we store all information in a single column family HBase would still have to skip the data upon reading. We could only avoid the transmission of the data. The `events` table splits into two column families accordingly. We call them `event` (for common information) and `data` (for event specific attributes).

The design of the row key of the `events` table is the most crucial step of the schema design (cf. Subsection 5.2.2). The row key is the main point for requesting specific pieces of data. In our use case, the most characteristic pieces of information are the user to which an event belongs and the time it occured. We want to be able to query events by those two attributes. Common operations are *give me all events of a user* and *give me all events of a user between x and y* where x and y are two timestamps, y being greater than x. The row key is a compound of the user ID and the event timestamp separated by an underscore, e.g. $\underbrace{000000001}_{\text{user ID}}\_\underbrace{1378989677000}_{\text{timestamp}}$.

Note that we padded the user ID to retain a lexicographical order on the user ID. Using this row key we store the events of a user together and in the right order (lexicographic). We can retrieve all events of a user number 1 for example by defining $(start, stop)$ as $(000000001\_0, 000000001\_INTMAX)$. `INTMAX` is here the biggest number that is storable in an integer variable. It is also the highest timestamp an event can have. If we want to restrict the result to a certain time range we specify $(000000001\_START, 000000001\_STOP)$ with `START` and `STOP` being the beginning and end timestamp of the interval.

It remains the configuration of the column families of the `events` table. We assume events never change. Therefore we can set the number of versions that HBase should store to 1. The parameters `IN_MEMORY` and `BLOCKCACHE` should be set to `true` to enable a prioritization of this tables caching. A bloomfilter (cf. Subsection 3.6.3) on row and column qualifiers allows a faster query execution at only little costs in write throughput and space. A compression of the column families seems to be reasonable as well. The performance impact of this will be a subject of discussion in Chapter 10. The final commands for creating the schema are:

```
create 'users', {NAME => 'info', VERSIONS => 1,
        COMPRESSION => 'NONE', BLOCKCACHE => 'true',
        IN_MEMORY => 'true', BLOOMFILTER => 'ROWCOL'}

create 'events', {NAME => 'event', VERSIONS => 1,
        COMPRESSION => 'NONE', BLOCKCACHE => 'true',
        IN_MEMORY => 'true', BLOOMFILTER => 'ROWCOL'},
        {NAME => 'data', VERSIONS => 1,
        COMPRESSION => 'NONE', BLOCKCACHE => 'true',
        IN_MEMORY => 'true', BLOOMFILTER => 'ROWCOL'}
```

Interface

Now that we fixed the schema, the data insertion is the next task. There exists a library for accessing HBase from Python. It uses HBase's restful (Thrift) interface though. A native approach via Java is more desirable. We introduced the Py4J library in the previous subsection. We used it also in a simple Java Gateway that performs the actual insertion into the database.

Queries

We implemented the queries in pure Java, using the API which Subsection 5.2.2 already described. We built them on top of the small query framework Figure 9.1 showed on page 116. The rest of this subsection shows the implementation of the actual queries. The approach is very similar to the one we used for the EventStore. HBase provides however more features in querying data. This makes the design of the queries more convenient and the execution potentially faster.

We calculate the app usage duration distribution again by processing user by user and scanning through the `WindowsStateChange` and `Lock/Unlock` events. In contrast to the EventStore we can now actually filter for the particular events we need. We create a list of filters for the `Scan` of the `events` table. A row has to *pass one* of those filters in order to be part of the result set of the `Scan` object.

```java
Filter windowStateChangeEventFilter = new SingleColumnValueFilter(
    Bytes.toBytes("event"), Bytes.toBytes("event_type"),
    CompareOp.EQUAL, Bytes.toBytes(32L));
Filter unlockEventFilter = new SingleColumnValueFilter(
    Bytes.toBytes("event"), Bytes.toBytes("event_type"),
    CompareOp.EQUAL, Bytes.toBytes(1005L));
Filter lockEventFilter = new SingleColumnValueFilter(
    Bytes.toBytes("event"), Bytes.toBytes("event_type"),
    CompareOp.EQUAL, Bytes.toBytes(1006L));

return new FilterList(FilterList.Operator.MUST_PASS_ONE,
        windowStateChangeEventFilter, unlockEventFilter,
        lockEventFilter);
```

We do not need additional information like the app name to answer the query. To avoid the transfer of those data, we specify only those columns we actually require:

```java
scan.addColumn(Bytes.toBytes("event"), Bytes.toBytes("event_type"));
scan.addColumn(Bytes.toBytes("event"), Bytes.toBytes("time"));
```

For querying the number of apps used in a time period we are only interested in `WindowStateChange` events. A simple way to query them is to add the `appName` column to a scan object. It will only return rows that contain this particular column. In querying the cell coverage we again restrict the retrieval of events. This time we are only interested in the `Periodic` event type.

Finally we want to mention some general considerations. The queries return byte arrays which we have to parse in order to get the contained information. An important query parameter is the `CacheSize`. This determines how many rows HBase retrieves in a single call. If we set it to 1 this means each call on the scanner results in a remote procedure call that retrieves exactly one row. In our case we always want to process *all* rows of a query result. It is thus reasonable to retrieve the rows in batches. If we query too many rows at once though, we have to wait too long for the result. Empirically a CacheSize of 1000 rows seems to be an efficient value.

9.2.3 PostgreSQL

With PostgreSQL as an object-relational database management system the question is how good this traditional approach fits to the use case of our scenario. There are multiple ways to model the schema and to make use of the flexibility of PostgreSQL. This subsection describes the standard ways to connect to PostgreSQL in our environment for inserting and querying the database. For the queries there are again several approaches. On the one hand they depend on the schema and on the other hand they depend on the SQL features of the system.

Schema

Evolving Schema

The main problem of a relational schema is that it is rather rigid, because in contrast to HBase we have to define it in advance. At the first glance this does not cope with flexible or new event types which will occur in our scenario. Each event type has actually a different schema because specific arguments describe them. Because of this PostgreSQL has to provide means to somehow allow an evolving schema. Despite that we want to avoid adding dynamically new tables like having one table for each event type. A reason for this is that relational systems are not designed for such schema changes.

Normalization

Generally we have two variants for coping with this scenario. The first one is to normalize our events. Diagram 9.2 shows the relational schema of this model. It consists of one table for the users which contains an artificial identifier as the primary key. It also contains additional user information. The Event table stores on one row for each event. It has also an artificial primary key and links to User table via a foreign key. Furthermore it contains all attributes which are common for all event types. This is the timestamp, which denotes when the event occurred on the mobile device, and also the event type, which is a number specifying what other attributes will follow. To retrieve the other attributes of the event the querying application has to know the meta data of this event type.

We locate these other attributes in the Event_data table. One or more rows of this table reference one entry in the Event table, so it describes a n:1 relationship. This table has columns for every data type which is necessary to describe all event types. In more detail this means columns for variable strings, numbers with or without

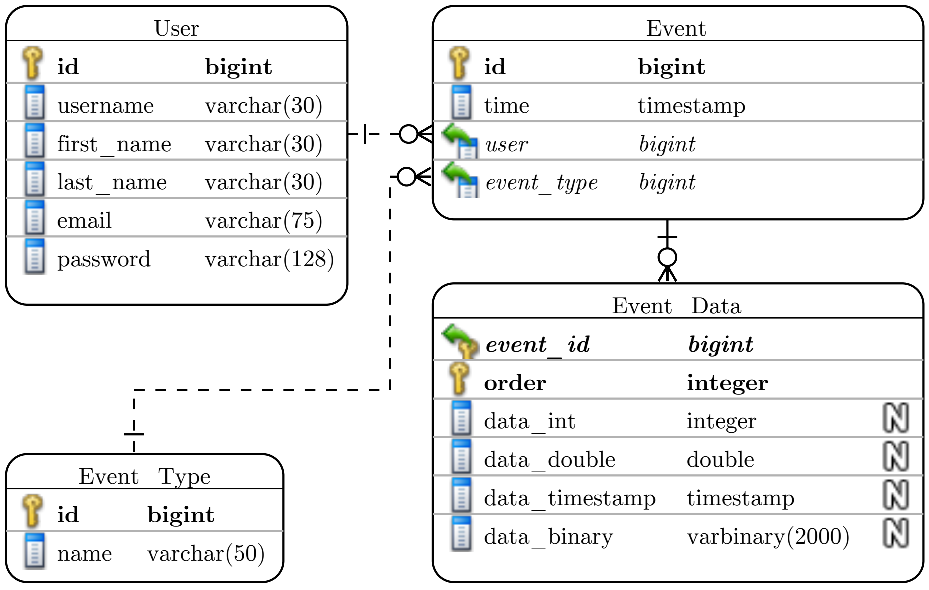

Figure 9.2: Diagram of the PostgreSQL schema

post-decimal digits, floating point numbers, timestamps and so on. This allows on the one hand to flexibly store any possible data type in a row, but it implies on the other hand a lot of NULL values in this table. The table will be very sparse. The major problem is that queries need a lot of joins, because retrieving specific attributes of an event makes it necessary to combine those tables. Unfortunately these are both very large tables such that this join is probably harmful for the performance.

Denormalization

One solution to the join problem is to denormalize the Event and the Event_data tables. This means we integrate the columns of the Event table into the data table. More precisely the Event_data table includes fields for the timestamp, the type and the user. This information is now part of each data entry. An event without data entries pads those columns with null values such that at least one entry exists for this event. The advantage is that we are able to prevent joins of two large tables because everything is now within one table. One disadvantage is that the table size massively increases because we store the event's information redundantly. Another disadvantage is the scattering of one event into multiple entries in that table. This makes it harder to distinguish a single event.

HStore

To wipe out these disadvantages we have to investigate the data types of PostgreSQL that are nonstandard in SQL. PostgreSQL provides two suitable data types for handling this problem, namely HStore and JSON. Both types allows columns to be a key-value map. That means each table cell serves as a distinct key-value store. Unfortunately JSON lacks of query support. It is not yet available in the version 9.2 we are using within the scope of this thesis. This means we would have to customize the retrieval with user-defined functions. Fortunately the HStore type is more mature in PostgreSQL and supports both modifications and queries. It has all means to provide among other things a single field retrieval, which is necessary to implement e.g. filters directly within the SQL query syntax. Using this type we can store the event data in one cell. By having this it is possible to compactly store everything in a single event table without having any redundancy. This avoids having joins on the one hand and on the other hand there is only one row per event. This approach has one minor drawback as the size of one table row in a page in the table space is now larger than before. If the data field is not needed then PostgreSQL has to skip more disk space during the read procedure. To get an impression of the advantages and disadvantages of the different approaches, we will perform tests on both of them in the benchmark.

Interface

Psycopg2

PostgreSQL offers, like most of the relational database management systems, ODBC access. Within a programming language the user can specify any query as long as he knows the native SQL dialect of the system. For newer Python versions the package psycopg2 provides this access by fulfilling the Python DB API 2.0 specifications. The specifications are an interface which makes the access to any DBMS similar or even equal. It is independent which relational database system is underlying. This includes means for working with cursors for iterating over result sets, specifying DML commands and preparing statements. It also provides an uniform error handling.

Furthermore there are features specific to psyopg2. These include thread-safe connections which could be useful in our highly parallel environment. Psycopg2 adapts Python objects to database types. This means e.g. tuples map to records, Python lists to arrays and dictionaries to the HStore column type. For custom Python objects there is also the possibility to write custom adapters and type casters. This means that there is a variety of tools for coping with the impedance mismatch.

Queries

Declarative vs. Procedural

For translating the queries to PostgreSQL there are basically two approaches. One is to write declarative queries which means creating SQL queries in the PostgreSQL dialect. This can also include specific functions or extensions in the PostgreSQL environment. The other approach is creating stored procedures which means investigating the procedural approach to cope with the tasks. Of course there cannot be a pure procedural approach as only SQL provides the access to iterators over the data of the tables. Nevertheless the SQL statements are very puristic as they e.g. provide all data or every event for a specific user. The procedures are responsible for the more complex operations and computations.

Multiprocessing

To make efficient use of the computational resources we have to force PostgreSQL to use all CPU cores. Unfortunately PostgreSQL has the restriction that it only uses one thread at a time. More precisely each connection to PostgreSQL is using only one thread and thereby only using one core. This means it does not use most of the computational resources in a multi-core server environment. In contrast to other relational database systems like Oracle's it is not possible to find any parameter or hint to configure PostgreSQL to alter this. A solution is to use multiple queries and perform them each on a separate connection. This means we have to open up connections in the number of the CPU cores and split our queries to perform a part of it. Finally we have a pool of processes (processes instead of threads because of Python), where each process owns one connection to PostgreSQL. The main process has to merge the results afterwards and can output the final result. This implies that our Python client has to do this part of the work.

App Usage Duration

To find the durations for app usage we have to break down the process into two steps. First of all we have to formulate the query in an efficient way and then we have to prepare this query for parallelization. At this point the normalization that we have discussed before does not play any role. This is because we are not interested in any attribute of the events. It is not important which app was used but only how long any app was used on average during the hours of the day. This means no matter which schema is present we do not have to care for more than the Event table and its columns. Generally this task is how to formulate an SQL query at all. This is because we have to find two consecutive events of the same type (application usage) and calculate the time difference of these. In the relational world this would mean accessing two rows at the same time. This is generally impossible. Nevertheless there are multiple approaches to simulate this behavior.

The first approach to this is the canonical way by using NOT EXIST NOT. Getting two consecutive events is the same as finding two arbitrary events where there is no third event in between them. Additionally all events have to be of the correct type. This leads to the query shown in Listing 9.1. The problem of this query is that it uses even two joins of the whole data set. That should be a major performance loss. Thereby this query tends to be very slow and it is necessary to reformulate it. This is in some way curious as declarative queries specify only the output and not the processing. Nevertheless the optimizer of PostgreSQL cannot find a good and efficient plan for this one.

Listing 9.1: Canonical approach to query app usage duration

```sql
SELECT extract(hour FROM e1.time) AS day_hour,
       avg(e2.time - e1.time) AS avg_duration
  FROM event e1, event e2
 WHERE e1.user = e2.user
   AND e1.time < e2.time
   AND NOT EXISTS (
     SELECT id
       FROM event e3
      WHERE e3.user = e1.user
        AND e3.time between e1.time AND e2.time
        AND e3.id <> e1.id AND e3.id <> e2.id
     )
GROUP BY day_hour
```

Listing 9.2: Improved approach to query app usage duration

```sql
SELECT extract(hour from time) as hour,
       avg(durations.duration) as duration
FROM
  (SELECT event.time as time,
      (SELECT e2.time - event.time
         FROM unisimulate.event e2
       WHERE e2.user = event.user
              AND (e2.event_type = 32 OR e2.event_type = 1006)
              AND e2.time >= event.time
              AND event.id <> e2.id
            ORDER BY e2.time ASC
            LIMIT 1
         ) as duration
  FROM unisimulate.event event
  WHERE event.event_type = 32
  ) durations
GROUP BY hour
```

The second approach uses a *subselect* for eliminating the joins of the prior approach. Listing 9.2 shows the syntax of this query. The *subselect* orders all events via the timestamp in an ascending order. It starts with the entry right after the current row and limits it only to the next one. This is exactly the same as getting consecutive events. Fortunately the PostgreSQL optimizer does not really create a separate select statement for each row in the Event table. It is only sorting the table once and can then put the pointer to the correct row at each time. Additionally there are filters in the `WHERE` part that allow only events of the correct type. As this query replaces two joins by one sort operator it should give a speed up compared to the first approach.

Another (third) approach is to procedurally solve the ask. For this PostgreSQL offers the PL/PgSQL scripting extension to write stored procedures which enhance the standard SQL capabilities. The procedural approach does basically the same that we already discussed in the previous systems. This is because it comes naturally to sequentially traverse through all events if they are in the correct order. That idea makes it easy to find and process two consecutive events. For processing them this way the start procedure has to loop through all users. Within the loop it then has to do a simple SQL query to retrieve all events for this user. It retrieves a cursor for sequentially traversing through all the events, filtering for the correct ones and calculating the intervals. As it is not clear how advantageous the set-based access to the data is in PostgreSQL we have to compare this approach to the previous one. Nevertheless this approach neither uses any join nor has the necessity to retrieve the Event table more than one time per query.

Window Function

However there is a fourth approach that combines the idea of sequentially traversing the Event table and using SQL as query language. PostgreSQL allows using so-called analytical functions as enhancement to the SQL standard. These analytic functions include window functions which basically allow having more than one

Listing 9.3: Incorporating Window Functions to query app usage duration

```sql
SELECT hour, avg(duration)
FROM (
  SELECT extract(hour FROM time) AS hour,
  coalesce((case when event_type <> 32 THEN interval '0' second
            else lead(time) over (PARTITION BY "user"
            ORDER BY time ASC) - time END),
            interval '0' second) duration
  FROM unisimulate.event WHERE event_type IN (1005,1006,32)) temp
WHERE duration <> Interval '0' second
GROUP BY hour
ORDER BY hour
```

cursor on the data at a time. Listing 9.3 shows the window function as we use it in our select statement. It is possible to specify the window besides the currently accessed row during computation. This window is in our case partitioning the Event table by the user and then ordering every partition by the timestamp in ascending order. There are lead and lag functions that allow us selecting the next or the previews row in reference to the current one. In this case we look at the next window for calculating the interval. This approach is very promising as it avoids any join and uses a specialized function instead of a workaround with *subselect*.

Parallelization

After designing query strategies we now have to investigate the second step for the process: to parallelize it. To do this we take the process pool and provide every process a separate connection as mentioned previously in the interface part. We can split all queries by computing the app usage duration only for parts of the user base. For achieving this we can add the split condition into the `WHERE` part of the query by enhancing it with an `AND` and the restriction to a range of users (*user* `BETWEEN` x `AND` y). Additionally for the procedural query we can add new function parameters to specify this range.

Number of App Utilizations

In order to count the number of app utilizations we have to access the data field of an event. From this field we can extract the name of the application. Furthermore counting is easier than computing intervals because only one event is required at a time. This means there is no need for analytical queries in this part. Despite that we can basically use the same algorithm as before for parallelizing this SQL statement. We can split again by user such that there are multiple statements. Each statement gets the additional restriction in the `WHERE` part for specifying a range of users.

Longest Duration of Bad Cell Coverage

The query for computing the longest duration of bad cell coverage per user is the second one that incorporates the event data fields. As it also computes intervals between two events it is actually similar to the computation of the app usage durations. In this case we are not looking for consecutive events. Instead we are looking for the next event which does not have a bad cell coverage anymore. This means we do not know in advance which of the following events is our "next" one respectively we do not know the distance between each pair of events. As a result we cannot make efficient use of the window functions here. This is because we cannot specify a fixed n (predecessor or successor) for the lead or lag function. As a side note it is maybe possible to solve this with SQL recursion and to loop through all possible distances. But this approach does not seem very promising and feasible to be an efficient solution. Nevertheless we can adapt the other approaches of the app usage duration to find the two corresponding events for computing the intervals. We only have to enhance the filtering by specifying the *bad cell coverage* condition. The approach of parallelization fits in this query, too.

Optimizations

There are some optimizations for each of the query variants. One important thing is to have proper statistics. To some degree it is unnecessary to have current statistics but statistics that represent the data well. Another thing is to avoid having too much dead space in the table space pages. As mentioned in the PostgreSQL description in Subsection 5.2.3 this is achievable by using the `VACUUM` and `ANALYZE` commands. Furthermore indexes can support the queries. A valid idea is to use an index that first of all indexes the user and then the timestamp in the Event table. This makes sense as we always first distinguish by the user and then order the events by the timestamp. The B$^+$-tree is a useful option for this index as it is preferable for range querying. By having an index we can also cluster the Event table regarding to this index. This clustering command aligns the data in the table space according to the index structure. By this we can achieve a lot of sequential reads which should enhance the query performance. After enabling the clustering PostgreSQL does not do this automatically in the future (e.g. on insertion). This means we have to manually re-cluster over time.

9.2.4 Redis

Redis is a key-value store and has a rich feature set for specifying the values. This allows a variety of approaches to determine a schema. Along with the schema we discuss the design of the whole setup and the ways to communicate with the Redis server. The way the queries are implemented depends heavily on the schema but also has some additional considerations.

Schema

As Redis stores the data in a key-value fashion, the schema needs to answer two questions: *what is the key* and *what is the value* of an entry that corresponds to an event? This allows to investigate several variants where two ideas arise naturally out of the scenario.

The first variant considers each entry to be one event. This means the key has to be representative for this event to allow a lookup. In this case an event needs either an artificial ID or an identifier consisting of a unique combination of for instance the user's ID and the timestamp of the occurrence of the event. The value has to cover all attributes of an event. This could be a string serialization of a map which maps from the attribute names to its values. A better solution is to make use of the Redis hash value type. This allows saving the map without serialization. The values need a string representation because Redis saves every value as a string. The names within the map already have a string representation.

Unfortunately Redis offers no means to specify range queries over keys. This is an important requirement as most of the time we have to retrieve only a subset of all events, e.g. all events for a specific user or events in a certain time interval. However Redis supports the `KEYS` command. This allows retrieving multiple entries by defining a regular expression for the key. There are possibilities for modeling the key in a way such that these expressions would be able to match a key range. However it would be rather complicated to design the key and the queries this way. Furthermore the Redis documentation states that this command is meant for debugging issues only and should not be used on large data sets. As we have large data sets in our use case this workaround is not feasible. Another idea is to store a list of event IDs for each user and use this to define subsets for using the Redis `MGET` command. This would somehow allow querying ranges as well, but this massive redundancy of event data does not strengthen this idea and rather leads to the next variant.

Aggregate Storage

In the second variant we store all events for one user under one key i.e. in one entry. This key has to represent the user and thereby is his ID. The first idea for the value type is to serialize all events including all attributes for the single events as a string in JSON format. The problem with this is that it is no longer possible to access a single event. This is because it is always necessary to retrieve the whole value, deserialize it and then get the desired event. This is an enormous overhead as the entries grow as the amount of events per user increases. Furthermore it is not possible to formulate range queries with Redis in this variant for the same reason.

Value Types

A solution is to investigate Redis' value types for this variant. The `HASH` type allows saving each event in a separate field, where the events data needs to be a string and is thereby serialized. The field would have to uniquely identify the event and could be the timestamp. This makes sense as the uniqueness of the timestamps for one user is an assumption of our use case. This approach does not support range queries for the events and hence is not desirable. Using the `LIST` type would provide an ordering of the list items. The items are in our case the events and thereby range queries are possible. It is problematic that we cannot specify these range queries via the timestamp. This is due to the serialization of the item and there is no possibility to access the timestamp directly. It is only possible by decomposing and processing the value client-wise. Additionally Redis does not take care of the ordering of the list items automatically such that we have to manually take care of this. A manual implementation of the range query for the desired timestamps is possible using a workaround. But this is not desirable as well. Using `SET`s does not provide any benefits because there is no order in the set. Hence range queries are not possible at all.

Sorted Sets

In contrast the sorted set (`ZSET`) allow an ordering of the items. Each item is assigned a score and the whole set is ordered according to this numeric value. We can use the timestamp as the score because it is unique (set characteristics). This also provides a sophisticated and automatic ordering of the events. And it allows using the timestamp for defining range queries on events for a user. This means we represent each user by exactly one key-value pair, where the value is a sorted set. This set stores all events separately in a serialized string and orders them via their timestamp of occurrence.

JSON/MsgPack

As mentioned before Redis stores everything as a string. A (field) value does not consist of several fields (like columns in HBase). We have to serialize an event to get a string representation of it. The most popular serialization format is JSON, which creates string representations for objects like a JavaScript's dump output (cf. example in Listing 9.4). Having the delimiters as string characters can pose a large overhead when serializing a lot of small fields. Redis supports another serialization format called MsgPack. This is a more efficient representation as it uses small binary delimiters instead of full characters. Using this instead of JSON should yield a more optimal utilization of the RAM and thereby more performant I/O operations. A useful thing is that Redis scripting includes message MsgPack libraries for Lua. This means it is not difficult at all to use it as a replacement for JSON. Furthermore MsgPack extensions exist for a lot of popular programming languages including Python.

Compression

Having the schema we can think about optimizations regarding the Redis server. As mentioned in Subsection 3.6.1 compression can be an important factor to optimize I/O. Redis offers a LZF compression for their set types. This compression only works on sets until a certain amount of entries. The parameters `zset-max-ziplist-entries` and `zset-max-ziplist-value` describe the maximum size of the entries and the maximum size of a value in the entry set. The system defaults are rather small values such that lists in our orders of magnitude have generally no compression. We can specify a sufficiently large amount for these properties such that Redis compresses every set. Unfortunately the compression gets slower and slower as the size of the set grows. The reason is that Redis compresses the whole set and not each entry individually. The first drawback is that for each new inserted item, Redis has to decompress the set for the item and compress the whole set again. This is a very costly operation for large sets and this is why the default configuration enables

Listing 9.4: An event in JSON format

```json
{
    "user": 42,
    "timestamp": 12133734,
    "type": 32,
    "data": {
        "app": "Angry Birds",
        "path": "com.birds.angry.game",
        "description": "Fancy video game"
    }
}
```

compression only for small data sets. Another drawback is that it is impossible to retrieve individual items anymore. This is again a result of the compression of the whole data set.

Multiple Instances

Section 5.2.4 described that a Redis instance only utilizes one core of the CPUs in a server due to its design. So the available processing power can be a bottleneck if we use large queries and server-side scripting. As our scenario requires this we have to find a workaround to use all spare processor cores. An idea is to start one Redis instance for each available core. This allows each instance to use one core for its own and this would enable a 100% CPU utilization. Prior only $1/n$ of the power is available, where n reflects the number of CPU cores the system. This approach requires a more complex distribution of the data and the workload to the number of instances. In our scenario it is easy to separate the users between Redis instances as sharding is perfectly fine here. The data insertion can randomly use one instance for storing the data of a user. This is not problematic because we store all events from one user in one entry. Thereby it cannot happen that the data from one user spreads over multiple Redis instances. The data generation only has to ensure that the user-instance assignment is fix, i.e. storing it after assignment or using a hash function. For querying it is in our scenario unimportant which user resides on which instance. In general we have to fix the assignment globally.

Interface

py-redis

The Redis community offers several libraries such that almost every programming language is able to connect to a Redis instance. There is not one default connector for a programming language, but in general there are several available. As we chose Python for implementing the data generation and our queries, we took py-redis as the most mature client library. Redis actually offers a DSL for communicating with the server. It is the default way for interaction within the Redis console. However the language is basically a set of distinct commands with a small amount of parameters. Py-redis exposes a method for each available command. This is different to e.g. ODBC, where the connector allows sending queries based on the DSL directly to the server.

Lua Scripting

Redis offers with Lua a scripting language for server-side processing. We can use this feature for communication as well. Py-redis allows us sending scripts to the Redis instance where it executes them and returns the result. However sending large scripts over the network for each command can pose a large overhead. To optimize this the Redis library allows registering scripts in advance and execute them later on. When registering Redis persists the script and answers with a hash value. We can use this hash value to call the function. It is not possible to name functions in Redis. Despite that the hash value for the same script is always identical.

Queries

First of all the query design depends heavily on the selected schema, but there are also other screws for testing and optimizing the performance. As mentioned before there are two considered ways for serializing the events themselves, namely JSON and MsgPack. Our claim is that MsgPack increases the performance as it reduces the overhead for string serialization. To test this we insert the data in both ways into the system and design the queries to work with both formats. This is fairly straightforward as we only have two switch the deserialization method regarding the type.

Server- or client-side Scripting

Despite this distinction every query is formulated in three different ways as there are three different approaches: pure Python code, server-side scripting with Lua and a mixture of both. Pure Python code means retrieving the necessary data from the Redis instance and doing all computations in the client, which is a Python program. This includes utilizing Redis methods e.g. performing range queries without using server-side scripting. After this process finishes its work it can directly output the results. Server-side scripting means the exactly opposite. Lua scripts query the database and perform all the work on the instance. This means we can avoid network traffic with respect to the data because the script only needs to return the results. Our wrapping Python script calls these procedures and has to merge the results. This is because the multi-process environment requires to call several methods in parallel. The drawback of this approach could be the limited amount of library support in the Lua environment. But the basic features were sufficient to implement the queries. The third approach mixes the prior ones and shares the workload between client and server. It does smaller computations on the server in Lua, for instance computing one user instead of all users. On the other side it performs more complex merge operations in Python. This also reduces the amount of data shipping in comparison to the first approach. Furthermore it maybe also reduces the locking of the server-side scripting. As the Lua scripts work like transactions in Redis, this could be a drawback in parallel computation. We can test this later on, if this is really beneficial in comparison to the second approach.

Architecture

The general architecture of the Redis query environment bases on the multi-processing of our general query design. For this we have a process pool which means a fixed amount of processes initialized on start that do the work. This also reduces the overhead of generating processes on-the-fly. As our prior optimizations included setting up a multi-instance architecture with one instance per server core, one process deals with exactly one instance. This means the size of the process pool equal the amount of cores on our test server. A process sets up one connection and can process the queries independent from the user distribution over the instances. At the end the main process has to merge the results from the several connections. The communication is done via message queues. The pool's processes append their results to the queue and the main process can merge the values as they arrive.

App Usage Duration

Computing the app usage durations requires several steps. At first we have to retrieve either all events or events within the time range of 14 days. To achieve this the command ZRANGEBYSCORE allows us to specify an interval. This interval can have all types like close-close, open-open and also close-open (and vice versa). In py-redis the command is:

```
connection.zrangebyscore(userId, startTime, "(" + str(endTime))
```

In the Lua internal library it looks like this:

```
redis.call("ZRANGEBYSCORE", user, startDate, "(" .. endDate)
```

The user ID corresponds to the key we want to query while the start and end date specifies the desired time interval. The open parenthesis indicates an open interval, where it is curious that even at the interval end there is an open parenthesis. After this step the Python script retrieves an iterator over all events. It unpacks the serialized events one by one either with MsgPack or JSON during the loop. Then it filters them for the correct event types and computes the intervals. To do this the script has to temporarily store one value and has to subtract the prior timestamp when another event occurs.

Number of App Utilizations

For counting the number of app utilizations this script basically does the same than for computing the usage durations. But instead of computing an interval it simply has to increase the number corresponding to the specific application every time the filter matches a correct event type. This script lays out all apps in an array which has one position for each application with an zero or positive integer value in it. To get only events from the morning we cannot simply do the range queries like before. Furthermore we have to introduce an outer loop that selects each day and for each day does one range query. This query retrieves all events within a time span of that particular day (morning hours). The Python (or Lua) script computes these interval's start and end timestamps.

Longest Duration of Bad Cell Coverage

To find the longest duration of bad cell coverage we can use again the basic script from the app usage duration. The only difference is here that it is not sufficient to compute an interval as soon as the filter matches an event. It is furthermore necessary to look into the attributes of the event and filter for the cell coverage attribute. This attribute is numeric and has to be below or beyond a threshold. A temporary variable again stores the starting event and by finding the end of bad cell coverage the script can compute the interval. As a result the script outputs the longest interval for each user. This means it can return the value directly after it has computed the data of one user.

9.3 Bundling [CB]

The previous section described the realization of the test cases for each individual database system. Now we can merge those together to a test suit. As described at the beginning of this chapter we decided to use Python for this purpose. For this reason it allows adding the individual system's implementation as modules. Alternatively for systems like HBase it connects via Py4J. In this case the modules are Java classes. We created a virtual Python environment that contains all necessary packages to execute the test suit in. We describe the concrete setup of our benchmark in Section 10.1.

Measuring In order to achieve benchmark results we have to add a measurement functionality to all programs. This includes on the one hand a timestamp which we set up before executing a certain query. On the other hand we have to log the output after each execution to verify the results. Furthermore we introduce additional log output that saves the intermediate progress during the processing of queries. This is helpful in investigating long-running queries and optimizing them. It is necessary to detect failures during the test. One program part detects system or run-time errors and propagates them to the result. Another part transmits the output of the query to an additional log file. Both parts help verifying the correctness of the benchmark.

Repetition For achieving accountable results it is not sufficient to execute each query once. In this test suit we apply the number of five repetitions for each test. This is a trade off between vanishing outliers and having an acceptable overall performance time. The result is then the average of the five distinct results. To have more inside into the numbers we can consider the standard deviation and also the median. In this way we can avoid results that are inaccurate due to strongly deviating numbers. If the standard deviation is insignificantly small we can simply ignore these additional values.

Caching Caching of results or parts of them is a major problem in benchmarks in general. This is because the repetition of queries does not reflect the truth. In this case the database systems tend to answer the same query faster because of simply recalling stored results. This would lead to incorrect time measurements. We can detect caching mechanisms by looking at different measurements of the benchmark results. If the first query was significantly slower than its repetitions, then we can estimate a caching mechanism by this. There are several possibilities to eliminate the caching. This depends if the database system caches the data or the operating system does this process. For caching at the database system's level there are most of the time commands or configuration parameters that allow to switch off caching or clear the cache during runtime. By using Ubuntu Linux it is generally also possible to use a single command for clearing all cache files. As we are using the virtual machine this is not always possible due to the restriction of rights regarding the host system. In this case we have to find a workaround. One possibility is to overwrite the RAM with random data before starting the next measurement. Independent of which mechanism we are using we have to do this procedure after each execution of a query.

10 Performance Evaluation

We performed the benchmark on a test machine to gain some insights on the performance of the database systems. This chapter presents the results of these benchmark measurements. First of all it describes the test environment which includes the properties of the used machine as well as the workload. Then it shows the results of the optimization process described in Chapter 9. This leads to the individual best configuration for each system which forms the foundation for further comparison of the systems against each other. The third section looks at the write throughput, the fourth at the query performance. The final section concludes the chapter with a discussion of the presented results.

10.1 Test System [CB]

In order to gain reproducible results we have to specify all parameters of the benchmark. The process of generating data and the query implementation are fixed. In contrast the amount of data and the test system are variable. This means we have to specify meaningful values with respect to available resources. We thus first of all take a look at the test machine.

Machine Properties

Table 10.1 shows the properties of the test machine we used for our measurements. It has to be noted that all tests were performed on a virtual machine. Hardware-wise the most characteristic feature of the host system is that it has 12 cores of relatively low frequency compared to today's Intel Core i7 processors for example. Within our virtual machine we were able to use 10 of these cores. As each core supports Hyper-Threading, this means 20 CPU threads of execution. Thus parallel execution of the algorithms is crucial in order to use the machine effectively. Consequently we used 20 threads or processes respectively for our algorithms. Furthermore the virtual machine offered 120GB of RAM. We used current stable versions of Ubuntu, Java and Python as well as the database systems.

Component	***Specification***
Hardware	
CPU	10x 2GHz Intel Xeon Processor Core with HT
Memory	120GB DDR3 RAM, 1333MHz
HDD	2TB, SATA-600, 7200RPM, 64MB buffer
Software	
Operating System	Ubuntu 12.04 LTS
Virtualization[1]	KVM / OpenVZ
Java VM	Oracle Java SE 7 Update 25
Python environment	Python 3.2
Python packages	distribute (0.6.34) hiredis (0.1.1) msgpack-python (0.3.0) numpy (1.7.1) psycopg2 (2.5) py4j (0.8) python-snappy (0.5) redis (2.7.5) wsgiref (0.1.2)
Database systems	
EventStore	Current beta as of 09/13
HBase	0.94.9
Redis	2.6.14
PostgreSQL	9.2

Table 10.1: Test system

Data Sets

It is reasonable to have more than one test data set size in order to observe the scalability of the systems. We decided to take one small and one big data set. The big data set should be maximal given the system resources. The most scarce resource is the RAM. Redis as an in-memory data store sets the upper limit of what we can stored on the system.

In order to have enough resources left for other operations the data should not use the RAM up completely. We thus decided to limit the amount of generated data to 3/4 of the available memory. There are two parameters which affect the size of the generated data set: the amount of users and the amount of days we generate. We consider the amount of days to be fixed at 30. This means we can scale the amount of users in order to control the size of the data set. By projecting the data size within Redis we can generate up to 60,000 users within these bounds. Our big data set therefore includes this number of users.

The size of the small data set should be significantly smaller. It should however not be too small, such that measurements become blurred because they are very

[1]For technical reasons we had to use different virtualization techniques. KVM was used for measurements regarding the optimization, Open VirtualiZation for the large data sets.

small as well. We empirically found a set of 1,000 users to fulfill this requirement. To get an impression, each user has around 8,000 events on average in a time span of one month. This means the small data set consists of 8 million events, the large one of around half a billion.

10.2 Optimization Results [MM]

When we discussed the implementation of the benchmark in Section 9.2, we mentioned several potential optimizations. This section presents the results of our tuning efforts. We consider the small data set for reviewing the differences in the approaches. According to the measurements we choose for each system the best configuration. We then use this for performing the benchmark on the bigger data set. Note that the EventStore does not occur in this section as we did not perform any major optimization on this system.

10.2.1 HBase

The query implementation is fixed, as the possibilities to formulate them in HBase are rather limited. We can however change the configuration of the schema of the two tables which are involved. These are the `users` and `events` tables. We performed some optimizations which are guaranteed to bring positive results. For example to limit the number of versions HBase stores for a given cell. As we do not update the cells anyway and thus are never interested in the history of a cell, we can avoid unnecessary overhead. Enabling the compression however is doubtful performance-wise.

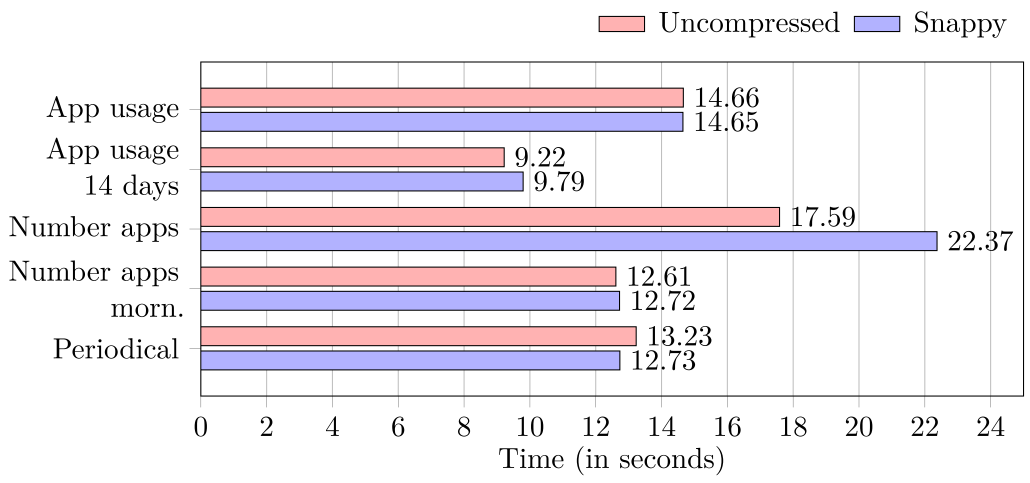

Figure 10.1: HBase query runtime comparison with and without compression on 1,000 users.

Figure 10.1 shows a comparison of queries on the uncompressed and Snappy compressed data set. Overall there is no significant difference between the execution times of the queries. There is one query which takes slightly longer on the compressed data set. The other queries alternate in what type is more performant. We were able to achieve a compression ratio of less than 20% (from 2GB down to 400MB). We expect the impact on the I/O on the larger data set to cause a significant speed up. Because of this we use the Snappy compression for further experiments.

10.2.2 PostgreSQL

Configuration File

In PostgreSQL we found three optimization approaches. The first is the configuration file `postgresql.conf`. The most important parameters are the buffers `shared_buffers` and the working memory `work_mem`. An article in the PostgreSQL official wiki [STB] explains the former one to determine the amount of memory PostgreSQL uses for caching data. The latter one specifies how much memory is dedicated for complex operations like sorting. The default values of PostgreSQL are very inefficient for a machine like the one we used for testing. We set the buffers to be 1/4 of the RAM and the `work_mem` to be 256MB.

Index

The next step in optimizing the query performance is the use of additional indexes. All queries access the data first by the user and then select a time or a time range. PostgreSQL automatically creates an index on the primary of a table. In our case

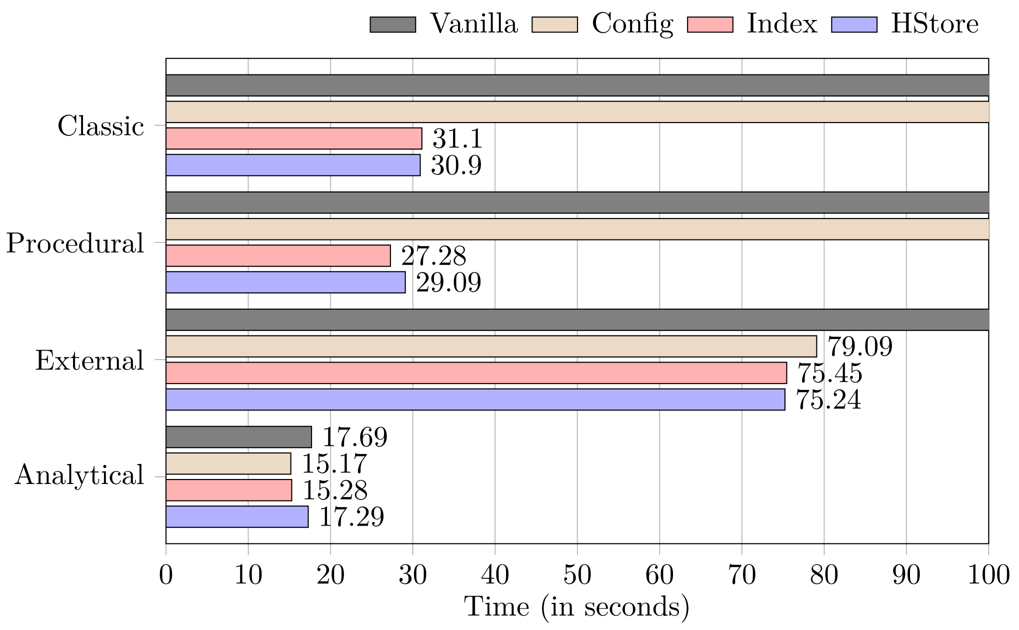

Figure 10.2: Performance of different approaches to query the app usage duration in PostgreSQL on 1,000 users.

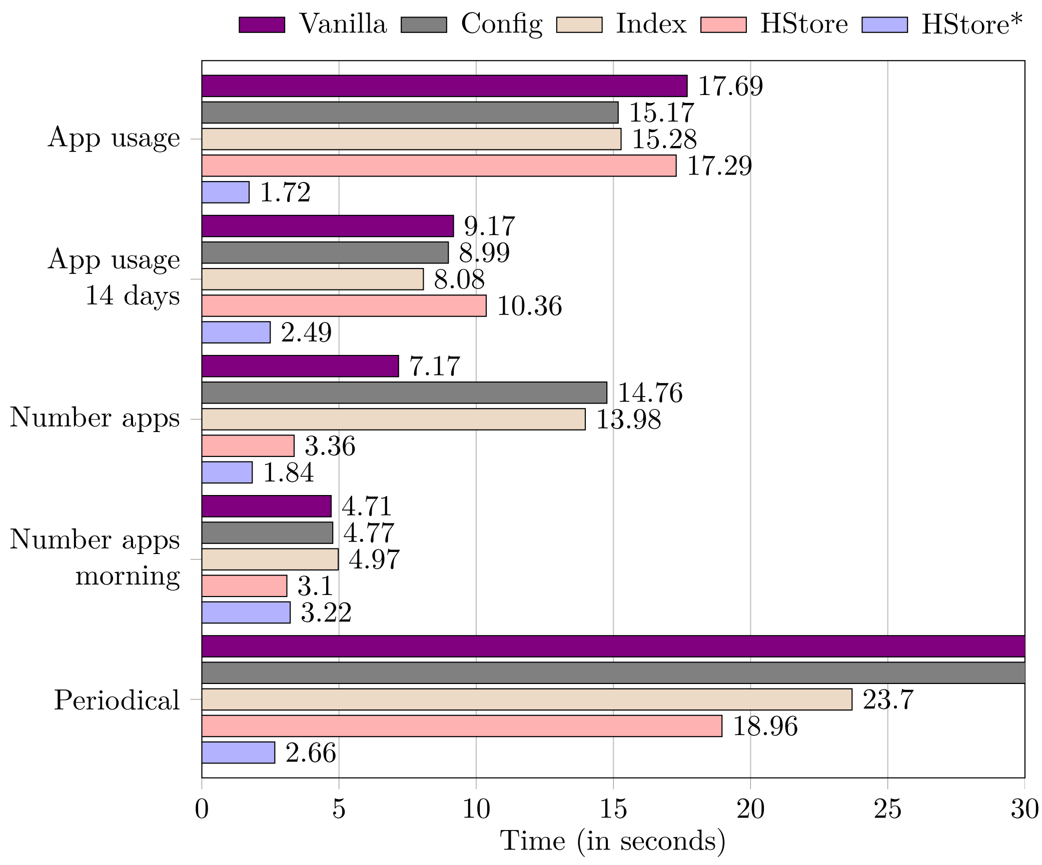

Figure 10.3: Optimization results of PostgreSQL queries on 1,000 users

this is an artificial key in form of an ascending number which makes it easy to join the event's data. To support the access on user and time we thus created an additional index in form of a B^+-tree on (user, time). Additionally we clustered the table using this index. This causes PostgreSQL to arrange the table data according to the index.

HStore

Using the HStore data type is the third optimization and enables us to store all of the event data in a single table. We can thus avoid joins when we incorporate the event's attributes in a query like the number of app usages. On the other hand queries which do not incorporate the event's attributes have to scan more data as they have to skip more data inbetween two events.

The app usage duration query is a special case. It needs to find consecutive events of a given type, which is rather hard to formulate in SQL. We found different ways to achieve this (cf. Subsection 9.2.3). The classical approach states the condition as a NOT EXISTS statements. The procedural approach uses PL/PgSQL to query the information. In the external approach we query multiple stored procedures at the same time. The analytical query incorporates window functions.

Figure 10.2 shows the performance of the different ways to query the app usage duration and how fast they are in the different configurations. Without optimizing the configuration and building an index, the classical and the procedural approach do not terminate within 1000 seconds. The external approach is an improvement when using no index. If an index exists it seems to be only additional overhead. The overall winner is the analytical query as it performs well in all configurations. We can see an improvement by using a better configuration but not by using an index. Using HStore has an additional overhead in this scenario. The query does not include event attributes and as discussed previously it thus has to skip more data.

After identifying the analytical query as the best approach of querying app usage, we can now compare the performance of all queries across the different optimization stages. Figure 10.3 shows the query execution time for all queries by using a single connection. The `hstore*` bars show the result of distributing the queries across multiple connections. This works around the one process per connection limitations we found in PostgreSQL. By looking at the number apps query we can see that the modified configuration not always improves the execution speed. The additional index is also not always useful, but never harms the execution significantly. HStore significantly improves queries that incorporate event attributes like the number of apps or the periodical query. Distributing the query load across multiple connections almost always shows a massive speed up. An exception is the number apps morning query which shows no improvement. This is most like due to the already short execution time. These results lead to our decision to use the multi-connection approach on HStore for further testings.

10.2.3 Redis

For optimizing the query runtime on Redis we identified three main paths. These are adjusting the access type, the storage type and the number of concurrent instances. Running multiple instances was the first idea after we noticed the comparatively bad performance of the system. It helps us to overcome the architectural limitations to a single processor core. Figure 10.4 shows how well Redis' performance scales when we enlarge the pool of instances to the number of CPU cores. We see a massive difference in the runtime on a single core and multiple cores. This leads us to the conclusion of using an instance pool for further measurements.

The second idea is to vary the serialization of our data. JSON is a rather space inefficient representation in our case. MsgPack offers a more compact format of an event. This not only saves memory space, but also allows us to read an event faster as it smaller in size. We can see this by looking at the bars of JSON and MsgPack. Our further investigation thus sticks to MsgPack.

Our final optimization regards the interface on which we access the individual Redis instances. We compared three different possibilities as discussed in Subsection 9.2.4. Figure 10.4 shows that in fact a combination of Python and Lua significantly improves the performance over pure python. However using pure Lua does not

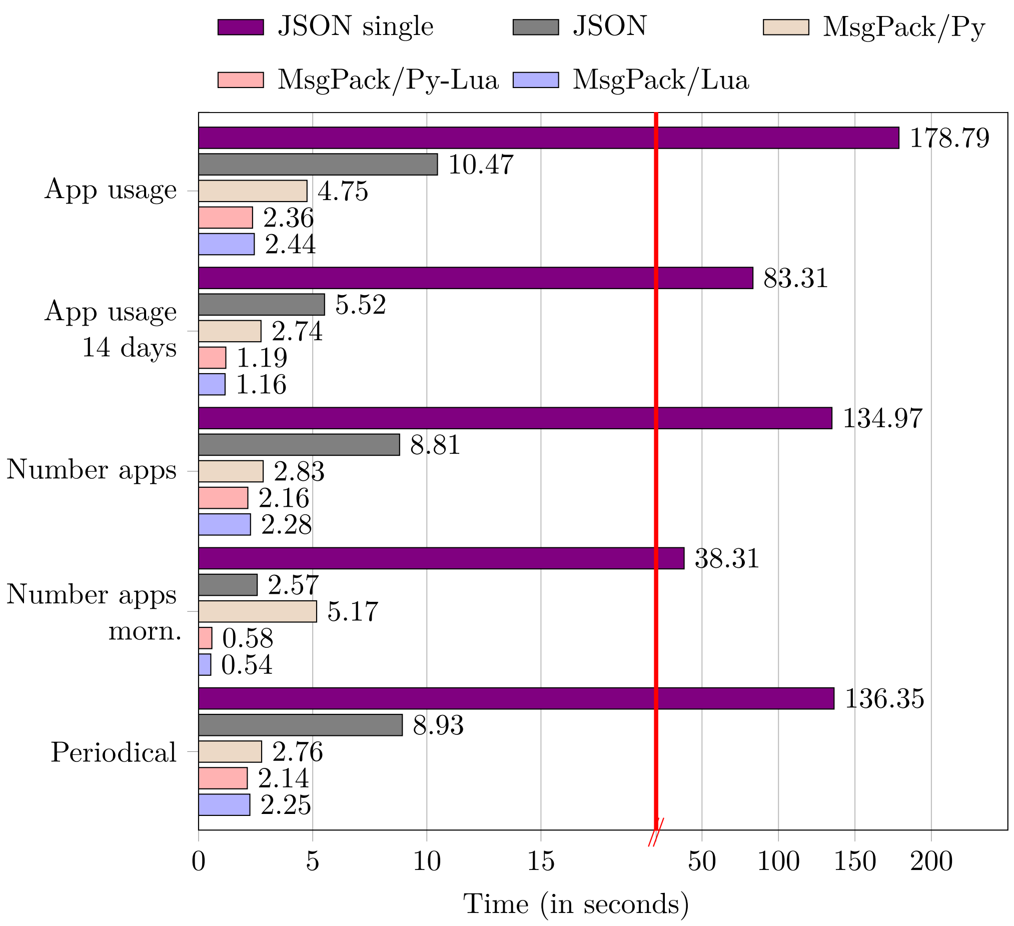

Figure 10.4: Redis comparison: Execution time of the queries in different configurations on 1,000 users.

significantly improve the performance any further. We thus consider the combination of multiple Redis instances with MsgPack and Python plus Lua to be the best configuration.

10.3 Write Throughput [MM]

While the main focus of our benchmark is the query performance, we first want to take a look at the write process. For this we use the best configuration we found in the previous section. The main metrics of interest are the insertion time and the used disk space. Additionally we compare the write throughput over time.

Figure 10.5 shows the write throughput in users per minute for the first 100 minutes. It shows significant differences in the amplitude and also in the fluctuation of the values. We now take a closer look at how they differ exactly and try to reason why they do so.

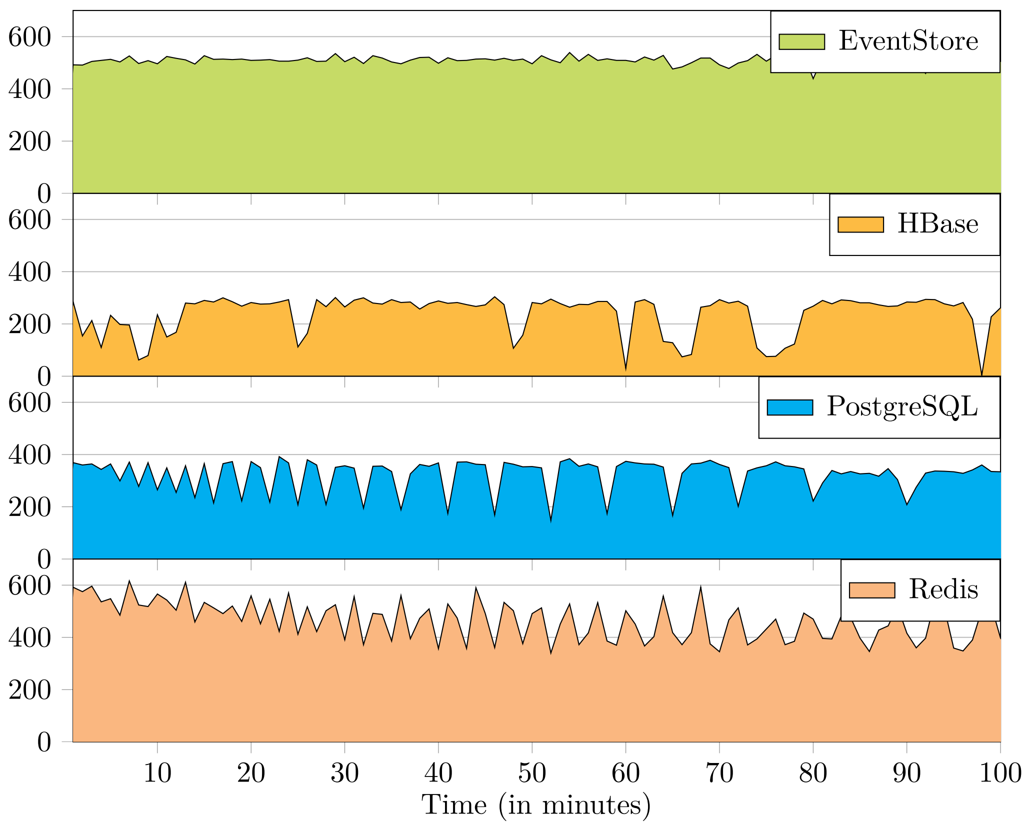

Figure 10.5: Write throughput in users per minute

EventStore The EventStore shows the most steady insertion behaviour of all the systems. It also remains on the top level for throughput. As the system has been developed to be write-optimized, this poses no surprise. The small fluctuations may be caused by the difference in the amount of events for users.

HBase HBase's throughput is generally on a lower level as the EventStore's. Additionally there are frequent drops in the throughput. They are induced by the compaction mechanism which regularly locks certain column families for insertion. It can be optimized by changing the size of the MemStore but not be avoided completely. They are crucial for the efficient processing of queries and therefore needed in a realistic scenario. The compression with Snappy improves the write throughput. It is no source of slow down. We did not optimize the write process further though.

PostgreSQL PostgreSQL shows some fluctuations in the write throughput as well. They are more regular than that of HBase and do not drop as far down. The overall throughput is thus greater than HBase's. The reason for the oscillations are most likely the indexing mechanisms. One possibility is the rebalancing of B-trees.

Redis The throughput of Redis reaches the highest values of all observed systems. It also fluctuates the most heavily. This is most likely caused by the multitude of different instances which work concurrently. There is no central system that coordinates them, as it is the case for all other systems. The reason why Redis does not heavily outperform all other systems is because of durability. That is while Redis stores all data in memory it also persists them on disk regularly. Its throughput trend thus goes downwards as it has to manage more and more data on disk.

Figure 10.6 shows the total time until the write process of 60,000 users for one month of mobile phone usage was completed on the different systems. The EventStore is the winner, Redis is close though being 10% slower. PostgreSQL already needs an additional hour and HBase takes twice as long as the EventStore. The insertion speed ranges from 35,000 events per second (HBase) to 70,000 events per second (EventStore).

Last but not least Figure 10.7 shows a comparison of the actual database sizes. Redis is rather inefficient in the storage of the data. This is also due to the lack of compression on the sorted sets. PostgreSQL's large size results from the additional indexing structure on the user and the time. It makes up 40% of the total size and could be avoided if it allowed storing the data as an index organized table. The EventStore does exactly this and therefore only takes up 27 GB. HBase with Snappy compression enabled also outperforms PostgreSQL and Redis. It plays in the same league as the EventStore, as they both use LZ77 based compression algorithms (Snappy and LZ4 respectively).

10.4 Query Performance [CB]

This section presents the performance of the systems on the queries we presented previously. We described them informally in Section 8.3 and the implementation in Section 9.2. The sections consists of two parts. It first discusses the results of the two app usage queries and their two variants. Then it treats the periodical data queries regarding the cell coverage. Each time it considers all systems at once for a single query.

As a remark we have to mention that we experienced some internal caching behavior of the EventStore. Unfortunately we were not able to completely resolve it due

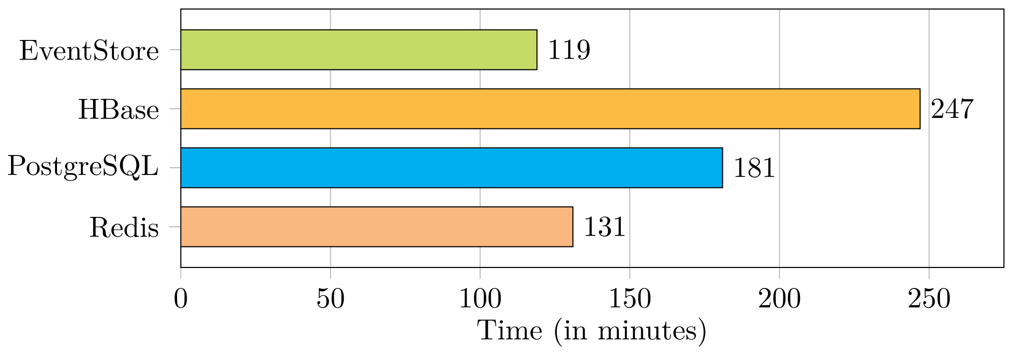

Figure 10.6: Comparison of the total time until write completion for 60,000 users.

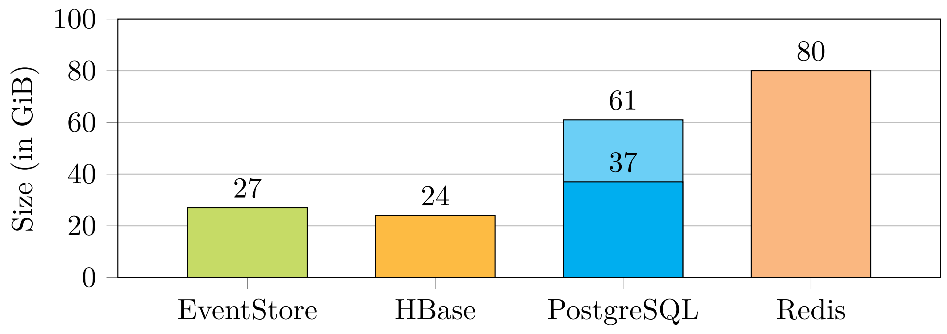

Figure 10.7: Database sizes

to the early development status of the software. This resulted in a rather slow processing speed of the first attempt and a significant speeds up afterwards. We only present the stable results which we have to consider carefully as they may not be achievable in a real world scenario.

10.4.1 App Usage

Figure 10.8 shows the query execution time for the app usage duration query. As a reminder: It computes the average app session length over the hours of the day. The figure shows two bars for each system. The uniform colored one represents the runtime over the whole time span of 30 days. The dotted one reflects the runtime over the part of the data in the last 14 days.

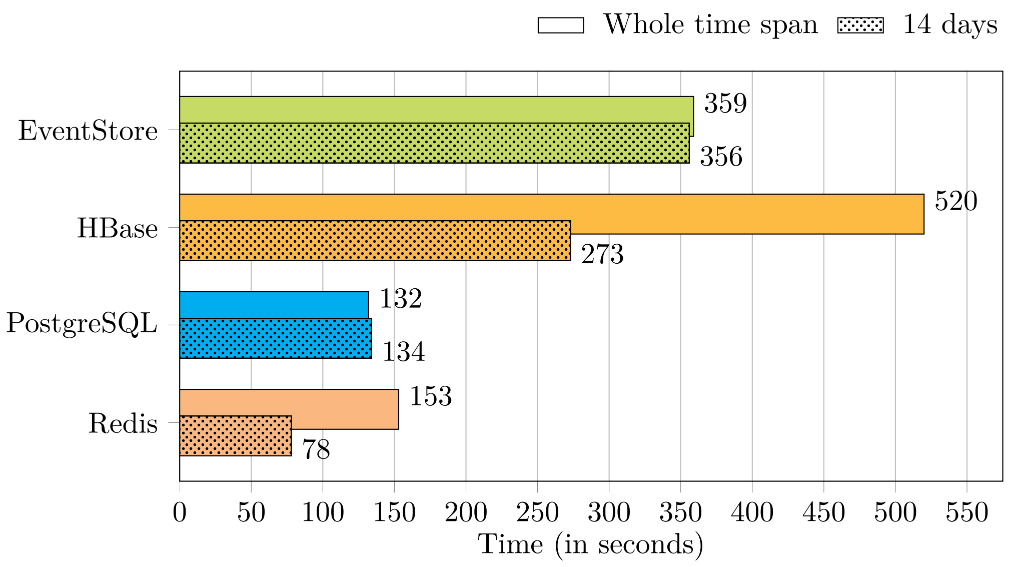

Figure 10.8: Comparison of app usage query execution speed.

The first thing to notice is that the restriction to a limited time span not always means that the query is faster. While this is true for HBase and Redis it does not hold for PostgreSQL and the EventStore. The reason for this is that in HBase and Redis we follow a procedural approach for calculating the intervals. Operating on fewer data thus means less I/O and fewer computations. For the EventStore this also holds, but in this case querying a smaller data set does not have an equally large impact on the performance. In PostgreSQL however the restriction to a time period means the calculation of an additional predicate. In this case the execution is even slightly slower for the smaller data set. PostgreSQL is the winner for the whole data set. Redis is superior when it comes to the computation on a range of the data.

Figure 10.9 shows the query execution time for the number of apps query. It computes how often apps are used within the data set. The figure again shows two bars for each system. The uniform colored one represents the runtime over the whole time span of 30 days. The dotted one only takes the events which occurred in the morning hours (from 5 to 12) into account.

PostgreSQL really sticks out in this use case. Both variants are the fastest among the contestants. Calculating aggregations is a classical task of relational database systems and PostgreSQL shows how well they support this. Redis is also quite fast but is only able to keep up with PostgreSQL on the smaller data set. HBase on the other hand did not perform too well. In this case, especially the query on the large set is an order of magnitude slower than PostgreSQL. The EventStore again ranks in the middle of the performance of the systems.

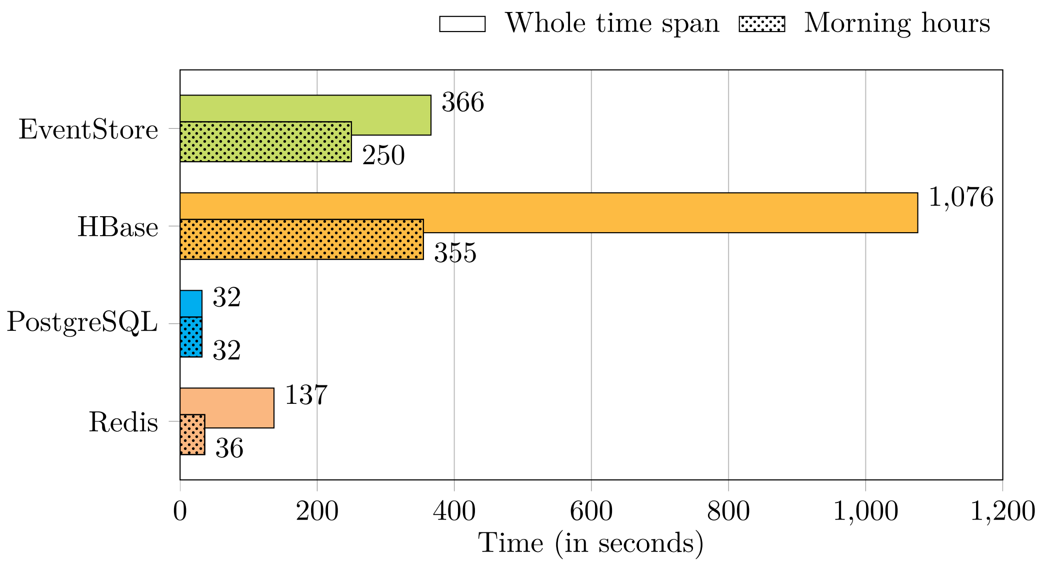

Figure 10.9: Comparison of number of apps query execution speed.

10.4.2 Periodical Data

Our second scenario deals with the calculation of the longest interval of bad signal coverage for each individual user. Figure 10.10 shows the performance of our contestant database systems. It paints a similar picture as the other scenarios did previously. Redis and PostgreSQL are the leaders, Redis being only insignificantly faster. HBase once again falls behind, yet not so severe. The EventStore is this time closer to the performance of HBase than to PostgreSQL and Redis.

10.5 Discussion [MM]

Figures 10.11 and 10.12 illustrate the overall performance of all systems across all queries. While Figure 10.11 shows the results for the small data set, Figure 10.12 relates to the larger set of 60,000 users. For the small data set we can see that Redis and PostgreSQL are the most performant systems. In contrast HBase is significantly slower and the EventStore is somewhere in between. Figure 10.12 paints a similar picture. This shows that all systems scale in a similar fashion. There are differences regarding the type of the query, e.g. PosgreSQL scales very well for the number of app usages. On the other hand the app usage duration query seems to be more demanding for PostgreSQL. The same holds for Redis.

Overall there is no clear winner across all aspects, i.e. queries and inserts. Figures 10.11 and 10.12 illustrate that PostgreSQL and Redis perform in the same league. HBase and the EventStore can not keep up with their overall performance. They are however more efficient in storing the data. The EventStore is especially good in inserting new data. Disk space however is cheap and thereby it is not necessary to save as much disk space as possible if this means sacrificing performance. We expect the write throughput to be not as crucial as the query performance. The data load stays roughly the same for a fixed amount of users, the query set however will grow over time. This makes it important to process them as fast as possible.

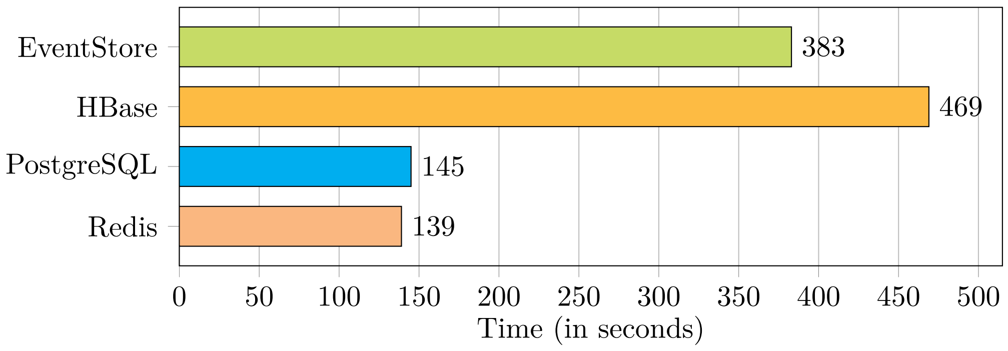

Figure 10.10: Comparison of longest time bad cell coverage query execution speed on 60,000 users.

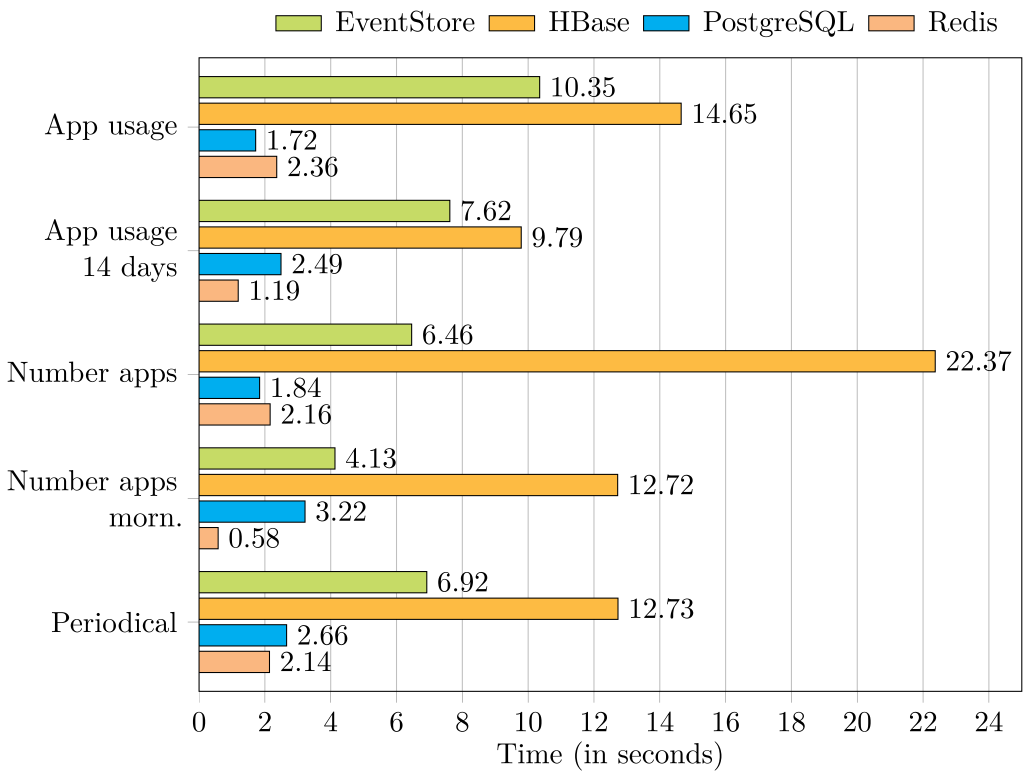

Figure 10.11: Overall comparison of query runtime on 1,000 users

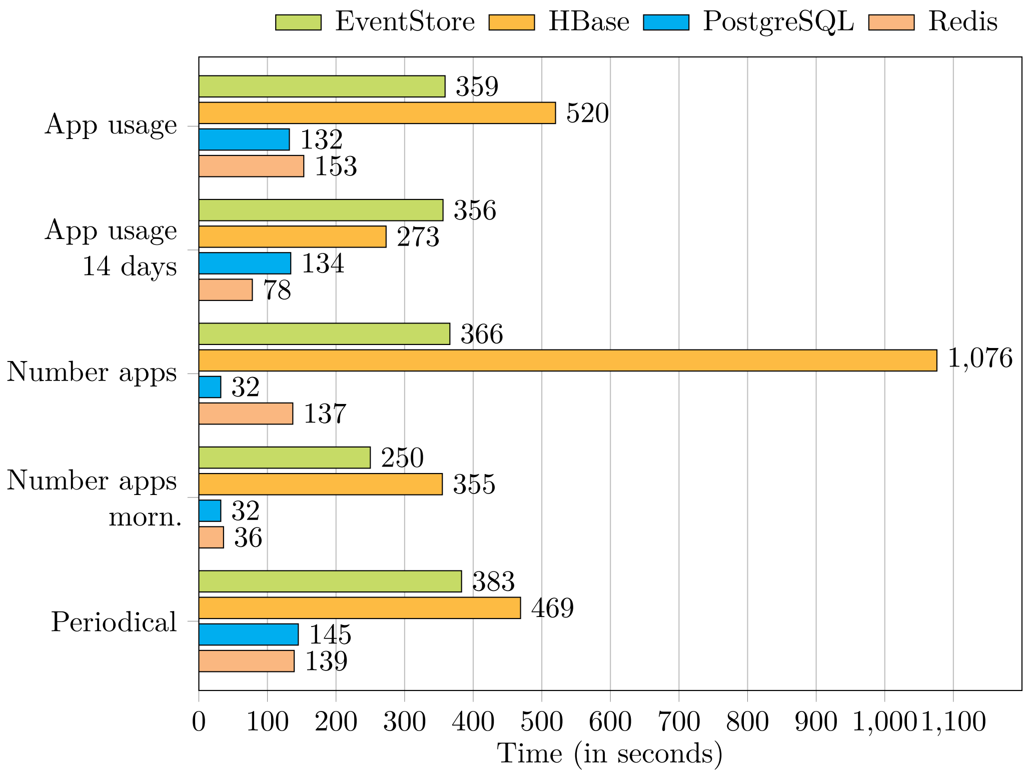

Figure 10.12: Overall comparison of query runtime on 60,000 users

A central question is whether we can determine a clear winner between PostgreSQL and Redis. While they show a similar query performance, PostgreSQL is slower in insertion. Redis on the other hand produces a lot of overhead in memory consumption. We have to press on the fact that Redis has to keep all data in main memory. The fact that it is wasteful in this respect is a big disadvantage. If we favor write performance over RAM disposal we will tend to Redis. If RAM is a limiting factor we will prefer PostgreSQL.

A final note has to address the optimization we performed on the tested systems. Most systems did not require too much attention. PostgreSQL on the other hand we had to tune heavily in order to retrieve good results. It is questionable whether some optimizations like the collection of statistical information and clustering of the data are practical in a real life scenario. Especially the latter method requires quite a lot of computation time and disk I/O which impairs the overall system performance. It would have been fair to incorporate those operations in the comparison of the execution times. This is however not feasible to arrange.

11 Codability Comparison

For our benchmark we selected only a few database systems, but they already cover a wide spectrum. They follow completely different approaches regarding their model, architecture and typical use cases. Chapter 10 already evaluated the performance of these systems with respect to our specific use case. This chapter looks at other factors which influence the choice of a database system for our project. This includes everything but the performance and is maybe even more important. Hardware is cheap nowadays. In many cases it is beneficial to just buy an additional machine and run a slower software than investing more time and money in developing a faster one. The ease of development thus becomes a deciding factor.

This chapter consists of six sections. It begins by looking into the installation process of a single instance of each database system. The next step is the setup or configuration to enable using it effectively. This may be either straight forward or quite hard depending if there are many configuration parameters. In the end a system should not just perform its task but also do this efficiently. The third section discusses the development experience with the different database systems. It shows pitfalls and advantages we experienced when implementing the individual parts of our test suite. Section 11.4 shows how well the approach of the systems fit our use case and where it was necessary to find workarounds. The fifth section discusses the documentation and community aspect of the systems. The chapter sums up all aspects in a final comparison.

11.1 Installation [MM]

We worked with different types of database systems. They range from small (embedded) systems like the EventStore up to big (clustered) solutions like HBase. This fact also reflects in the installation process which heavily varies between the systems. While we install the software typically only once for a machine it is still a noteworthy factor. A successful scalable system constantly grows. This yields in new machines that we have to add to the overall system. The installation of the database system is therefore a continuous process and we should expect a good support.

EventStore The EventStore is an embedded database which resides in a single jar file. The installation process thus only consists of including the contained library into a Java project. On the other hand there is the limit that it is only able to work within a Java project itself. If we want to use it externally we have to define a networked interface ourselves. We also have to take care of the dependencies of the EventStore like the XXL library or the compression library LZ4.

HBase Installing HBase does not reveal any highlights but there are some pitfalls. There is no default software repository included for Ubuntu Linux. We could for example install it via the Cloudera repository but we have to include this one manually. The standard way of installing consists of downloading and extracting an archive file from the HBase website. HBase uses IPv6 by default. We have to deactivate it on the system in order to be able to use IPv4. Additions like Snappy are rather difficult to install. The process consists of installing the library via `apt-get` and copying the native libraries into the correct HBase folders. It also requires the native Hadoop libraries which we have to copy as well. HBase needs multiple components to function. Apache Zookeeper as well as Master and Region Servers come along with it by default. If we want to use HDFS as reliable storage layer we have to install this and configure it separately. The other components work together out of the box.

Redis Redis is not available in the default software repositories of Linux either. In contrast to HBase we have to compile it first and install it then on the machine. As discussed in Subsection 9.2.4 we need multiple Redis instances in order to fully exploit a machine with multiple processor cores. Fortunately it easy to install concurrent versions on the same system. The installation process only needs to configure a different network port for the instance. This port number uniquely identifies an instance as `redis_portno`. Unfortunately there is no decent solution to communicate with those instances as a whole. The Redis cluster application is not yet in a stable version. Solutions like Twemproxy or redis-b pose too much overhead in comparison to a custom implementation.

PostgreSQL The PostgreSQL installation process appears to be the easiest among our tested systems. Due to its popularity it is part of default software repositories of Ubuntu Linux. This means the we can perform the complete installation by using a single `apt-get install` command. We have to install extensions like HStore manually afterwards. Fortunately this is also possible by using the `apt-get` command. There is no maintenance interface or admin console of any kind to achieve the installation directly from PostgreSQL.

11.2 Configuration [MM]

The configuration of the database systems serves two purposes. First of all it ensures that the systems operate *correctly*. This means they store their data in an appropriate place in the file system and process insertions and queries we desire it. It also covers the *efficient* operation of the systems. This should ensure that the physical resources limit their performance and not some artificial constraints of a misconfiguration.

The EventStore and Redis provide a rather clear configuration with only a few parameters. For the EventStore its only a handful, for Redis about 40. This means the developer does not need to worry about optimizing to many adjusting screws. On the other hand they lack some freedom which would be beneficial. The EventStore for instance does not provide an option to enable or disable the compression.

HBase and PostgreSQL in comparison offer more extensive possibilities for configuration. PostgreSQL has a well documented configuration and the community suggests default values for the most important settings. For HBase it is rather unclear what options really exist and how we should set them. It additionally uses a XML structure for configuration which is even harder to maintain compared to a simpler key-value structure of the other systems.

Initial Configuration

The quality of the initial configuration differs between the systems. The EventStore and also Redis already work efficiently without the necessity of adjusting any settings. Despite that the EventStore needs some approximation of the maximum size of the streams. We can however determine this generously as it has no impact to the size of the streams themselves. HBase's default configuration does not even persist the tables and data by default. To achieve durability we have to set a target folder in the `hbase-site.xml`. PostgreSQL does not make very efficient use of the RAM. We manually have to specify the amount it is allowed to use. The community suggests this value to be related to the total available RAM size. PostgreSQL is the only database systems which provides its own rights management system. In our case it is (for now) overhead because we have to be configure it in order to gain access to the system from our applications.

11.3 Usage and Development [CB]

The usage of the systems and the ease of development is an important factor for this comparison. Generally we can distinguish between two different approaches. One is developing with puristic systems like the EventStore and Redis that offer a limited and very specialized functionality. The other is using sophisticated database systems like HBase or PostgreSQL. They offer a full-fledged feature set and more options for finding a solution. Furthermore we look at possibilities to run the systems in cluster mode.

EventStore

The EventStore is the most puristic system among all tested database systems. The advantages are in that way a clear interface with only a few methods. This makes it unambiguous which one we have to choose for retrieving the data. For missing features the XXL library offers a lot of methods that work on the EventStore iterators. The EventStore depends on this libraries and thereby must include it. This means it is available by default. We get receive iterators by using the standard operators and can achieve e.g. a different sorting, filters or even join operations on them. As the operators are very fundamental there is no overhead in using the EventStore. For example it does not have to translate a domain specific language or do some execution plan generation dynamically in comparison to SQL-based relational database systems.

A disadvantage is the strict append-only format which restricts the possibilities of using a stream. There exists no complex query language. This is on the one hand because of the simplicity but on the other hand it does not allow us to specify complex operations without additional programming effort. The schema is relational and very rigid because the lacking of extended data types. The EventStore is a beta version yet and hence is still under development. This means there are still missing features. This additionally means that it has an even more restricted the feature support by now because of unimplemented methods. Furthermore the handling of streams comes along with the handling of files in the file system. We have to care about the number of open files manually by closing them on demand. Automatic opening of closed streams is supported though. As the EventStore is an embedded database system in Java it only supports this programming language. We can only access it from Java programs. This means we have to manually create ways to connect to it from other languages. In order to make this system network-compatible we also have to wrap a network gateway application manually.

Redis

Redis is also some kind of puristic because it is basically a key-value store. But it offers a variety of possibilities because of the rich feature set for the value types. This includes hashes and sorted sets which allow multiple ways of finding an appropriate schema. Furthermore the server-side scripting by using the programming language Lua is a good enhancement of the standard feature set. It also allows transactions in some way which can be useful depending on the use case. If transactions are necessary other systems like HBase and the EventStore leave no choice than implementing them in a custom way. Redis also has client support for a lot of programming languages such that it is very flexible and applicable in many scenarios.

Contrary to this is the necessity for maintaining multiple instances to exploit a machine's resources. There is no other way for utilizing more than one CPU core with Redis. Unfortunately there is no range access on the keys such that we have to swap these operations to the values. Furthermore Redis allows no complex queries because of the puristic and simple operator set. This provides e.g. no means for performing server-side calculation without using scripting. We can upload Lua scripts and registered them in the system before using them. This saves network traffic for sending large scripts for every query. A major drawback is the rather big memory footprint of Redis. Redis takes up quite a lot of space to store a given set of data compared to the other systems. This is especially critical as Redis is an in-memory database system where the amount of RAM sets the limit of storing entries.

HBase

HBase is one of the most sophisticated systems in this benchmark. It offers three types of APIs for communicating with it. One is a shell which allows specifying the commands by scripts or by hand. There is a direct connection via a Java library that allows specifying all commands by using Java methods and objects. We can also use any other programming languages to connect with HBase by using a restful API. This is very flexible but on the other hand poses additional overhead in comparison to the direct connection with the Java library. HBase offers rather extensive query capabilities. This includes for instance filters or a column selection for reducing the network and disk I/O. It also allows to do these operations server-side. The system offers also a very flexible schema design. It has the notion of a table as a very tangible foundation but with extremely flexible capabilities of specifying columns separately for each row. The schema additionally allows several other useful extensions like compression and bloom filters (indexes). Composite indexes are also possible by using the row key in an intelligent way.

There is no proper support for ad-hoc queries within HBase. This is because custom features and other operators are possible, but the distribution process is unfavorable as it requires adding compiled jar files to the Region Servers. It is not possible to perform more complex operations within the database system because HBase provides, aside from e.g. filtering, no means for complex operators. HBase has an automatic schedule for doing its compactions which optimize the physical storage layout of the stored data. This disturbs the write process and is rather complicated to control. There are possibilities to change the process e.g. by configuring the amount of data which HBase first stores in memory and then writes to disk at once. To our knowledge it supports no configuration for manually performing this operation though.

PostgreSQL

PostgreSQL offers the largest feature set among all tested database systems. The SQL support is a big advantage for formulating complex queries in an understandable and rather compact way. Its vast SQL extensions allow a lot of possibilities in writing the SQL queries. The server-side scripting is very advanced. This includes the own language PL/PgSQL but also scripting extensions for a lot of other languages like Python or Java. PostgreSQL uses ODBC for allowing external programs connecting to it. As nearly every programming language has a library for ODBC access, PostgreSQL is not restricted to some specific language. Other features like transactions in the ACID paradigm can be useful in certain scenarios. Specifying the schema is even more sophisticated for instance in comparison to the EventStore. Having e.g. with HStore the possibility to define very complex common types allows us to investigate several solutions to one problem. Additionally PostgreSQL allows several means for optimizing its processes. This includes a variety of index possibilities as well as statistics which support the way to automatically generating the execution plans. This again leads to very performant solutions for distinct queries.

A downside of PostgreSQL using SQL is that there are queries which we cannot easily express in this syntax. There is also a discrepancy between *what* we want and *how* we can retrieve it. In SQL we specify only the result and give hints in some way. By this PostgreSQL creates a more or less efficient execution plan. This leads to the point where PostgreSQL could actually execute some queries efficiently but we cannot give enough hints on how to achieve this. Chapter 10 showed that different queries which retrieve the same information show significant differences in their performance. We have to classify the declarative nature of SQL correctly. Often it is unclear which is the best way of querying and what leads to the most efficient execution plan. Optimizing queries thus becomes a tedious and error-prone task. Additionally the restriction of one process for one query (or connection) is very inefficient for analytical queries. It unnecessarily restricts the physical resources that PostgreSQL is actually able to use to perform a given query. In practice we thus have to split queries based on some criteria (in our case e.g. by the user) to run them efficiently. Unfortunately the scripting languages PL/PgSQL does not provide means for multi-processing either. Extensions for Python and Java seem to lack the libraries for setting up multi-threading.

11.3.1 Clustering

Clustering, i.e. the ability to distribute the database system, is also important factor. This is because for the realization of the backend it is necessary to have means for sharding and replicating the data. Sharding is necessary because of the big data fashion of our scenario. The replication can prevent data loss.

In this way the EventStore does not provide any features allowing a network application at all. So it is necessary on the first hand to wrap the application in a program covering the network capabilities and on the other hand to write the sharding and replication processes manually. Redis provides replication in a very easy way. It is possible to specify a master-slave system by attaching slaves to existing instances. After this Redis applies the synchronization automatically. We would have to realize more complex scenarios in a custom way. For sharding there is no proxy extension available yet, but the development team announced it for later versions. So we need a proxy application in front of the Redis cluster which applies some table or a hash function to distribute the commands.

As HBase is an implementation of BigTable its design allows it to run as a distributed system. This means we basically do not have to add any additional component to the system. It is only necessary to configure the HDFS properly to initialize sharding and replication. PostgreSQL provides also means for both replication and sharding. For replication there exist several extensions in the PostgreSQL environment. Two examples are PgCluster and pg-pool-II. Sharding has no completely automatic support in PostgreSQL. But fortunately there is PL/proxy which is an extension for a straight forward composing and distributing process of PostgreSQL commands. It first enables use to execute our queries on the numerous instances of the cluster. It then also supports us in gathering the results.

11.4 Use Case Compatibility [CB]

Besides looking at general considerations about development it is indispensable to take our specific use case into consideration. For this we partly investigate the features that we have mentioned above and evaluate them in this regard. Additionally we take a look at specific problems during the implementation.

EventStore

As our scenario is rather fundamental and far away from complex the EventStore does not lack of any really important features. This means the puristic approach fits well to our use case especially if we rate it as reducing overhead. The LZ4 compression in the EventStore provides very good compression rates at basically no cost. This also leads to a very space efficient storage which is one of the most efficient among all systems. Unfortunately the specific use case of the EventStore for real-time data scenario does not completely fit to our use case of asynchronous bulk inserts from a lot of sources. But fortunately it is possible to find a workaround that includes using one stream per user. Additionally via serialization it is possible to make the schema rather generic. On the other hand this leads to a lot of (open) files in the file system. We have to miss-use in some way the file system is an additional index structure for accessing our data. As we want to perform calculations on the newest data it is beneficial that we are able to query events by time. On the other hand it would be more efficient to query events also based on other attributes like the event type. This is not yet possible as there is no implementation available for the corresponding methods.

HBase

As we are expecting a huge amount of data it is useful that HBase is by design suitable for it. HBase can handle basically any amount of data if enough resources in the form of cluster nodes are available. Its flexible table design is beneficial because we have on the one hand a very tangible way of working with tables but on the other hand very flexible way to cope with the different event types very well. As we have in our test scenarios only one way of accessing the data, namely first over the user and then over the timestamp, the structure of a user-time index is very suitable for this problem. If we were would have different ways for accessing the data we would have to insert redundancy to the system to perform the queries efficiently. The index automatically sorts all data of a user by the timestamp which allows us to perform range queries. We modeled it by exploiting the lexicographical ordering in an intelligent way. Using Snappy allows us to achieve a very efficient storage of the data. After the installation the application of compression is very easy in HBase. It shows a significant positive effect in storing and also querying data.

Redis

Redis allows an easy and fast access to the data due to the in-memory management. The option of using sorted set fits very well to the design of the event data. Redis takes up more space than the other systems for the same data and as we are dealing with a huge amount of data this is problematic due to the scarcity of memory. It is thus even worse that the compression of ordered sets does not fit to the way we have to model the data in Redis. If we could make use of compression we could reduce the memory problem. The problem begins with the fact that is not possible to perform range queries on the key in Redis. This leads to the necessity to use large sorted sets (ZSETs) and store many events under one key. And this again leads to the point where the Redis compression does not work well because of the huge sets. Read and write access to these sets gets slower to a point where it becomes unusable.

PostgreSQL

PostgreSQL has the advantage that we have historically the most experience with relational database management systems. This includes having good capabilities in formulating even complex SQL queries in comparison to the provided languages or APIs of the NoSQL systems. PostgreSQL adapts to the queries very well because of the extended SQL support with e.g. HStore. The analytical functions (window functions with lead and lag operators) allow formulating queries in such a way that even sequential processing is possible in SQL. In our use case SQL is also very handy because experts can rather easily write new operators for the data store. This is because it is sufficient to write basically only one sentence for specifying the whole query instead of having multiple API accesses.

A drawback of the system is that there are a lot of features which are unnecessary for our particular use case. These features are in some way an overhead for which we have to pay by means of performance and space. Despite that there is the TOAST for compression in PostgreSQL and we cannot really control this in detail. This means we cannot specify additional compression methods that would yield compression rates in comparison to HBase or the EventStore. Th single process execution does not fit to analytical queries because they do not afford a multi-user system. Instead we aim for serialized executions of computations with a huge amount of data. Because of this we have to manually create a parallelized environment for executing the queries.

11.5 Documentation and Community [MM]

A well documented software is easy to use. A good documentation provides an introduction to the approach of the software and covers every aspect in depth. It is especially helpful when learning to use a new piece of software. More experienced users will dive further into the usage of the software and thus need a good documentation of more sophisticated aspects as well. A documentation will never cover everything though. Additionally there are questions which go beyond what a documentation can provide. For instance how we could solve a certain problem best when using the software. Then it is beneficial to ask other users or look up answers to questions which other users have solved previously. A large and well organized community helps achieving this.

Redis

Redis has a relatively small community. On the other hand the feature set of the software is rather limited compared to PostgreSQL. The website `www.redis.io` contains a lot of useful information. It explains each command and even lists the time complexity of the operation. A nice feature is an online command line interface.

It lets one use a virtual Redis database system right in the web browser without any installation.

HBase HBase has a bigger community than Redis. We can assess this already by the fact that it is an Apache project. One of the requirements of being an Apache project is having a significant amount of developers and a large user base. The HBase website seems to be rather unstructured and it is often hard to find the desired information. Questions e.g. posted to StackOverflow often remain unanswered which is not the case e.g. for PostgreSQL. There are however good textbooks which provide a very hands-on approach. Most notably is the book *HBase in action* [DK12]. Despite all that there are distributions like Cloudera which include HBase. They form an additional source of knowledge and support for the system.

PostgreSQL PostgreSQL clearly has the biggest community of our test set. It has been around for a long time and is along with MySQL the most successful open source database system. Consequently there is a multitude of sources one can make use of. Of course there are also a lot of books dedicated to PostgreSQL. But even the standard documentation is exceptionally well done. It explains commands syntactically and semantically and most pages contain appropriate examples. The PostgreSQL Wiki[1] contains many useful hints and suggestions. It also features discussions about the future development of the database system.

EventStore The EventStore is different than the other systems. First of all it is still in a beta status and is not used anywhere except in Marburg University and at the University of Bonn with respect to this thesis. Accordingly there is no community one could fall back to. On the other hand we were able to have a direct contact with the developers who are the most knowledgeable people for this particular system. A documentation apart from Javadoc[2] was not present though. This makes it hard to understand certain parts of the interface. As the interface itself is very puristic this is not as crucial as it would be with a more complex product.

11.6 Conclusion [CB]

The final section of this chapter draws a conclusion of the experiences we made with the investigated systems. It has to be noted that the results are rather subjective in the focus as well as the assessment we conclude. In contrast to the objective results of the performance evaluation in the previous chapter this chapter thus represents our own opinions. A classic way of quantifying personal opinion is the Likert scale which assigns sentences a rating on a scale. This scale consists of five steps of agreement with a statement. "$--$" means we strongly disagree, "$-$" means we disagree, "◯" means we are neutral, "$+$" means we agree and "$++$" means we strongly agree with the statement.

[1] https://wiki.postgresql.org/wiki/Main_Page
[2] A documentation tool for Java projects

	Installation	*Configuration*	*Development*	*Use case*	*Documentation*
EventStore	◯	+	◯	−	−
HBase	◯	−	+	++	+
Redis	+	+	+	+	+
PostgreSQL	++	+	++	+	++

Table 11.1: Codeability comparison

Table 11.1 shows how well well we found the different systems to fulfill the following statements:

- It is straightforward to install the system.
- It is easy to configure the system.
- The development process is well supported and allows a good workflow.
- The system fits to our use case.
- The system is well documented and has a sufficiently large community.

It thus gives a rough and simplified impression of the systems in those aspects.

PostgreSQL gave the overall best impression. The best fit for our use case seems to be HBase though as it naturally works well with large data sets. Redis made a good impression as well but could not stand out in any particular aspect. The ups and downs of the EventStore's rating are mostly due to its beta status. In our opinion HBase and PostgreSQL made the best impression within the scope of developing our benchmark suite.

12
Conclusion and Outlook

The task of this thesis was to investigate the development of a server side backend within the Menthal project. This included the management and processing of mobile phone usage data in order to allow users to analyze their behavior. The major challenges were the Big Data scenario, the high data throughput and the execution of continuous queries. The backend had to cope with large quantities of data and scale with the amount of users. Numerous users providing their usage data to the system leads to a high rate of data insertions. The backend thus had to be able to store and process this data in a near real time fashion.

The thesis began by reviewing mobile sensor data and data storage in general. After introducing the most common data storage approaches it gave a brief overview of the database system market. It also selected four systems for further inspection. These were the EventStore from Marburg University, the column-family store HBase, the RDBMS PostgreSQL and the in-memory key-value store Redis. A closer investigation of the use case led to the formulation of specific requirements of the backend. Based on this the thesis proposed an architecture and sketched solutions for emerging problems. This left the concrete data storage technology open. The remainder of the thesis thus dealt with finding the best solution. The first step for this was the definition of a benchmark which tested different aspects with respect to our specific use case. The next step was the implementation and execution of this benchmark. The thesis finished with a discussion of the experimental results.

The benchmark consisted of two parts: the generation of data and the specification of queries. First, we invested significant effort in generating meaningful data. To get as close to reality as possible we incorporated results of empirical studies. We then simulated phone usage in a way that reflects the observed distributions. Second, we focused on finding queries that covered a great variety of aspects of the database systems. The queries also tested scenarios which were likely to occur in the context of the Menthal project. This led to a query set that allowed us to gain insight into the differences of the systems.

Although we could not determine a clear winner in all aspects, we were able to create some objective criteria for the selection. With PostgreSQL and Redis we found two highly performant solutions for our use case. HBase fell behind performance-wise but provides a flexible environment for large scale data management. The EventStore, although still under heavy development, was able to keep up with the more mature systems and was convincing in its write throughput.

Concluding this thesis, we take a look at open research questions. A verification of the validity of the benchmark would be reasonable in both aspects: data and queries. First, we should look at how our artificial data compares to real data in more detail. We restricted ourselves to some specific distributions. There are however numerous other characteristics which first have to be identified and then investigated. This however requires access to a sufficiently large amount of real data which will only be available at a later stage of the Menthal project. Once this data is available, these comparisons are to be made. The other part is to check our proposed query set against the real world usage of the data. If there are significant deviations, we would have to extend the benchmark in this respect. The presence of enough data in the future does not make the benchmark obsolete. In contrast to real data which is subject to privacy restrictions, we can freely share the benchmark's data generation.

This thesis only considered open source and free to use database system products. It would be valuable to include commercial systems into the benchmark. An interesting product is SAP HANA. It combines the relational model of PostgreSQL with the column-oriented storage in HBase and the in-memory technology as in Redis. It is also built for distribution which fits to our use case. Incorporating it requires licensing the product and implementing a connector to our benchmark.

We observe that special purpose applications require specific solutions. Core database system functionality like ACID becomes questionable. There are cases where we need to disable such elements completely in order to gain advantages in other aspects like performance. Modern systems have to allow such flexibility. Users on the other hand need the knowledge to decide what is beneficial for the use case. They also need sufficient insight into the systems to configure them in this manner.

In terms of speed, individual optimization can outperform the choice of a database system. We saw this during our work where we were able to improve the execution time of queries on a system by orders of magnitude. In general we are no longer able to argue about the performance of a system, but have to tune it individually for a given task. This makes benchmarking compelling as it allows us to compare different systems in a controlled environment. It is however only as good as we optimize the system. At the bottom line, if companies are given the choice of investing either into more skilled people or more sophisticated software, they should always favor the former.

Bibliography

[And] Android Developers Guide. Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html. Accessed 29.08.2013. (cited on page 6)

[BHS$^+$11] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47–56, New York, NY, USA, 2011. ACM. (cited on pages 98, 99, 100, and 101)

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. (cited on pages 56 and 57)

[Dat] Database Research Group, University of Marburg. JEPC - Java Event Processing Connectivity. http://dbs.mathematik.uni-marburg.de/research/projects/jepc/index.html. Accessed 22.08.2013. (cited on page 63)

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. (cited on page 59)

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. (cited on page 52)

[DK12] Nick Dimiduk and Amandeep Khurana. *HBase in Action*. Manning Publications, pap/psc edition, 11 2012. (cited on pages 66 and 155)

[EN99] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999. (cited on pages 11, 15, and 16)

[Fowa] Martin Fowler. Domain Specific Languages. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html. Accesses 17.06.2013. (cited on page 15)

[Fowb] Martin Fowler. Polyglot Persistence. http://www.martinfowler.com/bliki/PolyglotPersistence.html. Accessed 11.07.2013. (cited on page 58)

[Gar] Gartner Inc. Press release: Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent. http://www.gartner.com/newsroom/id/1848514. Accessed 19.08.2013. (cited on page 5)

[Geo13] Lars George. HBase Schema Design. Presented at NoSQL Matters Conference, Cologne, 2013. (cited on page 67)

[GMUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book (2. ed.)*. Pearson Education, 2009. (cited on page 37)

[Ham94] James Douglas Hamilton. *Time Series Analysis*. Princeton University Press, 1 edition, 1 1994. (cited on page 8)

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA, 2007. (cited on page 37)

[Inc] Apple Inc. Mac Developer Library: Local and Push Notifications in depth. https://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/WhatAreRemoteNotif.html#//apple_ref/doc/uid/TP40008194-CH102-SW7. Accessed 19.08.2013. (cited on page 5)

[KE06] A. Kemper and A. Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg, 2006. (cited on page 31)

[Kha13] Simon Khalaf. Flurry Five-Year Report: It's an App World. The Web Just Lives in It. http://blog.flurry.com/default.aspx?Tag=Usage2013. Accessed 17.06.2013. (cited on pages 98 and 100)

[Kom] Matthew Komorowski. A History of Storage Cost. http://www.mkomo.com/cost-per-gigabyte. Accessed 14.06.2013. (cited on page 13)

[Lam10] Chuck Lam. *Hadoop in Action*. Manning Publications, 1 edition, 12 2010. (cited on page 59)

[MB11] Erik Meijer and Gavin Bierman. A co-Relational Model of Data for Large Shared Data Banks. *Queue*, 9(3):30:30–30:48, March 2011. (cited on page 51)

[MGL$^+$10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010. (cited on page 58)

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, 7 2008. (cited on pages 30 and 101)

[Nic07] Lara Nichols. A comparison of object-relational and relational databases. Master's thesis, California Polytechnic State University, 2007. (cited on page 46)

[Pyt] Python Wiki. GlobalInterpreterLock. https://wiki.python.org/moin/GlobalInterpreterLock. Accessed 12.09.2013. (cited on page 112)

[PZ12] Hasso Plattner and Alexander Zeier. *In-Memory Data Management: Technology and Applications*. Springer, 2nd ed. 2012 edition, 4 2012. (cited on page 62)

[RSRS98] Chandramouli Ramaswamy, Ravi Sandhu, Ramouli Ramaswamy, and Ravi S. Role-Based Access Control Features in Commercial Database Management Systems. In *In Proceedings of 21st NIST-NCSC National Information Systems Security Conference*, pages 503–511, 1998. (cited on page 28)

[SC96] Adrian Spalka and Armin B Cremers. An Axiomatic Interpretation of Confidentiality Demands in Logic-Based Relational Databases. In *Logic in Databases*, pages 303–319. Springer, 1996. (cited on page 28)

[SF12] P.J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012. (cited on pages 11 and 50)

[SKS06] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006. (cited on pages 11, 15, 16, and 31)

[Sna] Snappy Project. Website. https://code.google.com/p/snappy/. Accessed 24.09.2013. (cited on page 30)

[sol] solid IT. DB-Engines Ranking. http://db-engines.com/de/ranking. Accessed 19.08.2013. (cited on page 61)

[STB] Greg Smith, Robert Treat, and Christopher Browne. Tuning Your PostgreSQL Server. http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server. Accessed 12.05.2013. (cited on page 136)

[The12] The Nielsen Company. State Of the Appnation - A Year of Change and Growth in U.S. Smartphones. http://www.nielsen.com/us/en/newswire/2012/state-of-the-appnation-%C3%A2%C2%80%C2%93-a-year-of-change-and-growth-in-u-s-smartphones.html, 5 2012. Accessed 5.10.2013. (cited on page 101)

[vdBDS00] Jochen van den Bercken, Jens-Peter Dittrich, and Bernhard Seeger. javax.XXL: A Prototype for a Library of Query Processing Algorithms. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 588–, New York, NY, USA, 2000. ACM. (cited on page 64)

[Vog09] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. (cited on page 20)

[Wika] Wikipedia. Hierarchical Database Model. https://en.wikipedia.org/wiki/Hierarchical_database_model. Accessed 05.07.2013. (cited on page 47)

[Wikb] Wikipedia. Network Model. https://en.wikipedia.org/wiki/Network_model. Accessed 18.07.2013. (cited on page 47)

[Wor13] Jeffrey Word. *SAP Hana Essentials*. Epistemy Press. Galileo Press, Incorporated, 2013. (cited on page 42)

List of Figures

List of Tables

Listings