UPPSALA
UNIVERSITET

IT 15 025
Examensarbete 15 hp
Mars 2015

NoSQL: Moving from MapReduce Batch Jobs to Event-Driven Data Collection

Lukas Klingsbo

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

NoSQL: Moving from MapReduce Batch Jobs to Event-Driven Data Collection

Lukas Klingsbo

Collecting and analysing data of analytical value is important for many service providers today. Many make use of NoSQL databases for their larger software systems, what is less known is how to effectively analyse and gather business intelligence from the data in these systems. This paper suggests a method of separating the most valuable analytical data from the rest in real time and at the same time providing an effective traditional database for the analyser.
In this paper we analyse our given data sets to decide whether big data tools are required and then traditional databases are compared to see how well they fit the context. A technique that makes use of an asynchronous logging system is used to insert the data from the main system to the dedicated analytical database.
The tests show that our technique can efficiently be used with a traditional database even on large data sets (>1000000 insertions/hour per database node) and still provide both historical data and aggregate functions for the analyser.

Handledare: Bip Thelin
Ämnesgranskare: Kjell Orsborn
Examinator: Olle Gällmo
IT 15 025
Tryckt av: Reprocentralen ITC

Contents

1 Introduction to the "Big Data/NoSQL problem"

In times when big data collection is growing rapidly [1] companies and researchers alike face the problem of doing analysis of the data that has been collected. The enormous amounts of data collected are often saved in distributed big data databases or key-value stores with fast insertion operations but with comparatively slow and complex (but highly parallelizable) query functions [2]. This work tries to solve such a problem at Kivra [3], a company working to digitalise and minimise the use and need of conventional paper mail. To gather analytical data from Kivra's distributed database today requires heavy Map/Reduce queries. To do day to day analysis of the data slowly becomes unbearable and non-scalable as it takes several hours or even days to run a Map/Reduced [4] batch job to fetch the data that is going to be analysed. The solution proposed and examined by this paper is to structure and asynchronously collect data after interesting events occur, that are of a high analytical value, into a smaller and more easily queryable database.

2 Related Terminology

2.1 Big data

Big data [5] is a loose term for ways of collecting and storing unstructured complex data of such large volumes that normal methods of data manipulation and querying can not be used.

2.2 NoSQL

NoSQL [6] is a term that have not fully settled its meaning, it started by meaning no relational databases [6], but now it has moved on to commonly be referred to as Not Only SQL. If a database calls itself NoSQL it usually means that is is non-relational, distributed, horizontally scalable, schema-free, eventually consistent and support large amounts of data [7].

2.3 MapReduce

MapReduce [4] is referring to a programming model that makes parallel processing of large data sets possible. In simplified terms it consists of two functions, one called Map and one called Reduce. The Map function is sent out to each node from a master node and the function is separately executed on each nodes data set and then each node assembles the result back together with the other nodes so that all input values with the same results are grouped together. Then the reduce function is executed, either in parallel or on a master node, and this function does calculations upon each separate result list with the data from all nodes that ran the Map function and it then combines the input from them to the answer that was originally inquired.

2.4 Eventual consistency

The expression eventual consistency is a term used for describing how data is processed in database systems. Eventual consistency means that the data may not be consistent in any one moment, but will eventually converge to being consistent [8]. This makes it possible for distributed databases to operate faster as data does not have to be locked during insertions, updates and deletion and thus does not have to be propagated to all nodes before more actions on the same data can be sent to a node. If one or more nodes operate on the same data as another node whose changes have not propagated through the system yet, the system will have to handle in which order the operations should be performed, this is usually called sibling or conflict resolution [9].

2.5 Riak

Riak is a key-value datastore that implements NoSQL concepts such as eventual consistency. It is usually configured to run with at least 5 nodes to be as efficient as possible. To query the datastore you either use the REST-ful API or MapReduce functionality that it provides.

Riak [10] provides fault tolerance and high availability by replicating data to several nodes, the default is to have all data replicated to three nodes.

2.6 Lager

Basho [11], the company that developed Riak [10], has developed a modular logging framework called Lager [12]. Lager is modular in the sense that it is possible to write separate backends which all can separately do asynchronous data collection. As the system performance can not be noticeably affected by adding a subsystem to collect analytically valued events it is of highest priority that they can be sent asynchronously so that the implication of implementing such a subsystem wont result in delays for the main system.

A problem that these form of logging systems can experience is that the events that they have to handle queue up faster than they can be processed. To solve this Lager [12] has a built-in overload protection to make sure that its message queue will not overflow. It does this by swapping synchronous and asynchronous messaging for the separate backend.

2.7 Abbreviations

2.7.1 RDBMS

RDBMS [13] is an abbreviation for relational database management system. This is the most commonly used database management system and is usually referred to as a traditional database. Data is stored within tables and data can be in relation to each other across tables and such databases are therefore called relational.

2.7.2 DDBMS

DDBMS [14] is an abbreviation for distributed database management system. It is used to centrally manage a distributed database system and to give the user the impression and functionality as though the database system would run on only one node.

3 Background

3.1 About Kivra

Kivra [3] is a company which aims to digitalise and eliminate the use of superfluous analogue paper mail. They do this by making it possible for companies and governments to send documents digitally to customers or citizens. By tying the users' accounts to their social security numbers in the service the sender can be sure that the document reaches the correct recipient.

Today Kivra has a continuously growing user base of more than 500.000 users, according to their website [3].

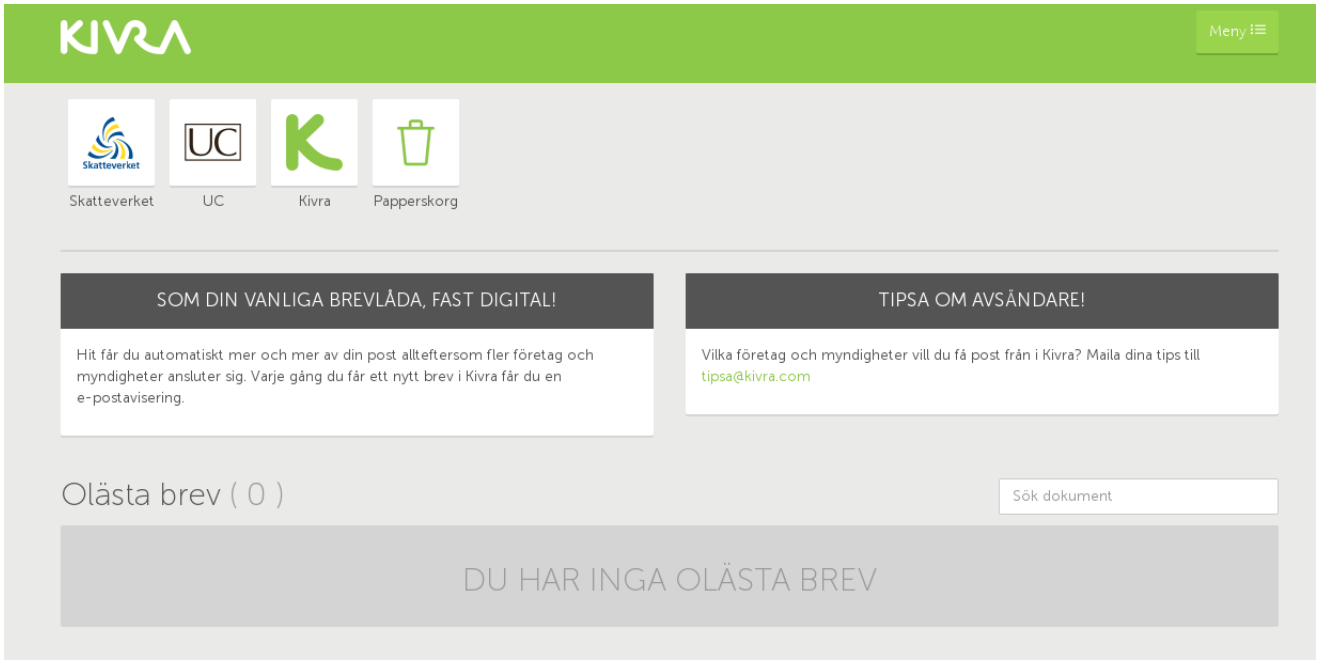

Figure 1: The user interface of Kivra

As can be seen in figure 1, the interface reminds of the interface of any email provider, however the documents sent to the users can be interactive and have a more advanced layout and the main advantage over normal email is that every users identity is established in the registration process. The user can validate its identity either by using mobile BankID [15] or by receiving an activation code to its registered place of residence. This advantage makes it possible for companies and organizations that otherwise would have had to send documents to a registered postal address to being able to send them digitally through Kivra.

3.2 The current system

Kivra's users interact with the backend system (this is the backend box represented in Figure 2) either through Kivra's web page or their app, where they can receive documents, pay bills, set whom they accept documents from and also set personal details. Users sign up either though mobile bankID or requests an activation code to be sent to the address registered to their social security number, this is to ensure the identity of the user. When a user receives a new document it is notified through email or SMS and can then log in to read the document.

The data which is of analytical interest is information about the users, meta data of the documents sent out to the users and whether a user accepts or does not accept (internally called *send requests*) documents from a company or organization (internally called a *tenants*).

3.3 Problem description

The part of the current system that is being examined consists of Riak, a distributed key-value datastore, and a batch script containing MapReduce queries which are running each night to collect data to a PostgreSQL database which is later used for deeper analysis. This system architecture can be seen in Figure 2. The batch script is emptying all of the analytically interesting buckets in the Riak cluster, currently there are two buckets being emptied every night. One of them contains send requests which is meta data telling the system if a user accepts documents for a specific tenant and the other bucket just contains plain user information. In the future meta data regarding documents sent to users will be collected as well.

This solution does not scale and is also inefficient as you have to go through all the data in the growing buckets that are of interest every night, instead of accumulating the data to a separate database dedicated for analysis at the same time as you insert or change data in Riak.

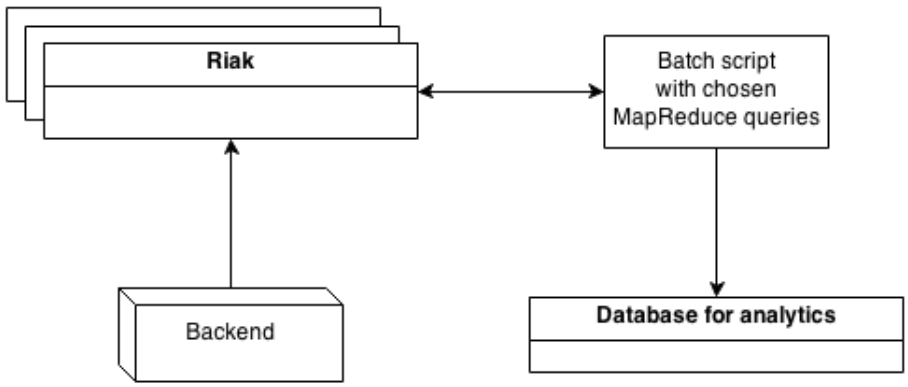

Figure 2: The analytical system at Kivra

4 Methods for determining implementation details

This chapter introduces the different methods used to determine how the new system should be implemented, which DBMS it should use and how the estimation of long term scaling was done.

4.1 Data size and growth estimation

The data size and growth was estimated with statistics collected from the current Riak cluster in Kivra's system. What the current insertion rate would be like with an event driven implementation was estimated by doing SQL queries measuring the number of insertions that could be made within certain time limits. To estimate the size and growth of the future database implementation both past estimations and the current internal estimated growth of the company were used. This, together with the scalability measurements, was required to make sure that the solution would be durable in the long run.

4.2 Scalability measurement

The scalability was estimated for the typical server hardware of Kivra with the different databases and with statistics of growth estimations from the current database. This was necessary to make sure that the system would scale with the expected data growth, within a reasonable time frame.

4.3 DBMS comparison

The database management systems were compared by testing required abilities and examining load and speed by running insertion jobs of different sizes. The DBMSs architecture and suitability were also compared to see which one suited the context best. These comparisons were vital to make sure that the chosen DBMS could handle the expected load of the system.

5 Solution for storing events

When an event of analytical interest occurs, an asynchronous call with information about the event is sent to Lager. Lager then directs it to an in-house tailored Lager backend, which is responsible for making sure that the data is stored consistently and correctly. The database in which the data is stored can then be queried at any time to perform near to real time analysis of the system. After analysing internal database statistics an approach of this form is estimated to cause approximately 100-10000 inserts/hour depending on if it is during the peak hours or not. The estimation can not be averaged as the chosen database is required to be able to handle the peak hours without a problem. This approach will lead to a system with no updates and only reads during analytical calculations. This is due to the fact that the backend will handle each event with a strict progression of inserting the relevant metadata of the event with a timestamp and never have a need of updating, nor removing old data, as all of these chosen events are valuable to store to enable historical queries. This is similar to how Lager [12] logs events to files, except that the analysis of the data will be easier due to the built-in aggregate functions of modern databases. In comparison to the current system it will be possible to perform near to instant analysis of the data already collected instead of having to re-collect all the data of analytical interest from the Riak cluster each time an up-to-date analysis of the data has to be performed.

5.1 Data sets

5.1.1 Determining the type of the data sets

Big data is one of the popular buzzwords in the computer industry today and a lot of revolutionising technology has been built around it. But it is not always efficient to use technology built for these data sets if the data set that is being operated on does not fit the definition of big data. But as there is a thriving hype of higher yield with existing data some companies and institutions use it just for the sake of being part of the trend without actually having a data set that objectively would be classified as big data [16].

To simplify the abstract definition of Big Data to determine whether a data set can be defined as such you can classify the data set by concluding the answers of three questions [5], namely the following:

- Is the data set large?
- Is the data set complex (maybe even unstructured)?
- Are methods optimized for other data sets inefficient?

If the data set satisfies these properties it can be a good idea to store the data in a Hadoop cluster or similar technology developed specifically for big data. If, however, the data set does not satisfy these criteria it is usually better to look for another storage solution [16].

At Kivra there are a few different data sets with information that is valuable for ongoing analysis. The three most important data sets are:

- User information - each users personal details.
- Send requests - connects a user to a tenant and specifies whether the user wants or does not want content from that specific tenant.
- Content metadata - contains meta data of content that has been sent to users.

The tables representing this data would be quite simple and would not store too much information as analysis of the system would mostly regard metadata of some of the information stored in the Riak cluster. See an example of how such tables could look like in listing 1.

Listing 1: Example tables

```
Example fields of the tables
+--------------+  +----------------+  +----------------------+
| User         |  | Send request   |  | Content Metadata     |
+--------------+  +----------------+  +----------------------+
| id           |  | user_id        |  | content_id           |
| name         |  | tenant_id      |  | tenant_id            |
| address      |  | accepted       |  | user_id              |
| city         |  | timestamp      |  | timestamp            |
| email        |  |                |  |                      |
| timestamp    |  |                |  |                      |
+--------------+  +----------------+  +----------------------+
```

5.1.2 Size of the data sets

As each record in the data sets is fairly small (under 200 bytes) it is enough to count the number of records plus the expected growth for the future. Today there are about 500 000 users with information [3]. The send requests records adds up to a little bit less than the number of users times the number of tenants and then there is also around 2 million content records. This results in a total of about 10 million records. All of the data sets have an expected growth of about 5% per

month according to internal statistics of Kivra collected from their Riak cluster. This is well within the limits of traditional relational databases [17].

5.1.3 Complexity of the data sets

None of the data sets have any complex structures, the data types are not varying and there are no big documents that requires to be stored in the analytical database as only the meta data of the content being sent to users has to be saved and not any attachments, images etc.

5.1.4 Inefficiency of other methods

There is no need for extremely fast reads for specific values which key-value stores [18] offer or for extremely fast insertion rates which big data databases [5] can provide. As concluded in the section about the size of the data sets (5.1.2) and the section called Choosing SQL database (5.3.3) the insertion rate of traditional databases will be well above the demands. As there is no complex data the normal table structure of a RDBMS will be suitable for storage of the data.

5.2 Protocols for data transfer

The requirements of the protocol and data structure used to transfer and translate data to the database differs highly on what type of data and from where the data is being sent, some data are required to have absolute transport security and reliability meanwhile other is of lower importance and rather trades transport reliability for speed. To maximize the speed and flexibility of the system as a whole, one or several APIs and web services that can accept several protocols and differently structured data being used towards it before translating it in order to store it in the database should be used, as can be seen in Figure 3.

The exact APIs, Web Services and data structures used for the implementation of this system is outside the scope for this work and will be left to a future implementer to decide. But examples of such protocols could be HTTP through a RESTful [19] service, cURL [20] with one of its many supported protocols or simply whatever protocol that will be built in to the customised Lager [12] backend.

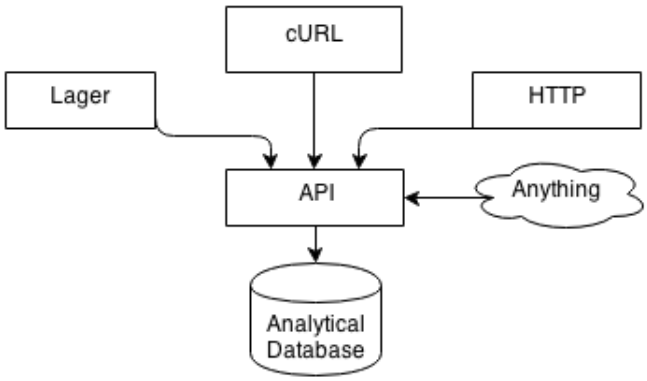

Figure 3: API explanation

5.3 Database

5.3.1 Requirements

As the database will mostly be used for insertions, the insert operations will have to be as fast as possible with the chosen way to store the data. Read operations will not have to be very fast as they are not directly used by any response time critical system but rather by in-house users, a business intelligence system or a system providing an interface of generated graphs and tables for overviewing the information.

The requirements on the speed of updates and deletes are non essential as they will only occur when manually editing the database. The database will not have to be portable, it is not going to be replicated or moved around. For future scaling of the database it is needed to be easily configurable and preferably have wide support for distribution of the data over several nodes. The data is required to be accessible and easy to aggregate upon for future analysis.

5.3.2 Why SQL over NoSQL?

A traditional relational database system was chosen over a NoSQL document store or key-value store for a few different reasons. The first reason was that the high insertion rate trade off that results in eventual consistency [21] given by NoSQL databases would simply not be needed as the amount of data insertions and updates would be well within the limits of a traditional RDBMS [17] as can be seen in Section 6.1.

The second reason was that the query speed of specific values, offered by key-value stores [18], was not required and the demand was rather for a full set of aggregate functions and easy access to complete subsets of the stored data. As many NoSQL databases use MapReduce for fetching data it would per definition be hard to write efficient joins and no built in aggregate functions would be provided [22].

What was also considered was putting a layer on top of a regular NoSQL database that seamlessly provided this type of functionality. The layers considered were HIVE [23], PIG [24] and Scalding [25] for HADOOP [26]. But as the implementation did not need the special functionalities and speed trade-offs, which is deeper explained in Section 6.1 and Section 5.3.1, that were offered by such layers the conclusion was to simply incorporate a traditional SQL database.

5.3.3 Choosing SQL database

There are a large number of different traditional database systems being actively developed today. For this project the focus has been on popular open source DBMSs. The most popular open source relational database management systems [27] today are MySQL, PostgreSQL and SQLite so the tests compare those towards each other and the requirements.

MySQL

MySQL [28] is the most popular open source database and has several popular forks as well, with the biggest one [27] called MariaDB [29]. MySQL does not aim at implementing the full SQL standard, but rather a subset of it with some extra features [27]. MySQL is built in layers with one or several storage engines in the bottom and an optimizer, parser, query cache and connection handling layers built on top of that [30], see Figure 4.

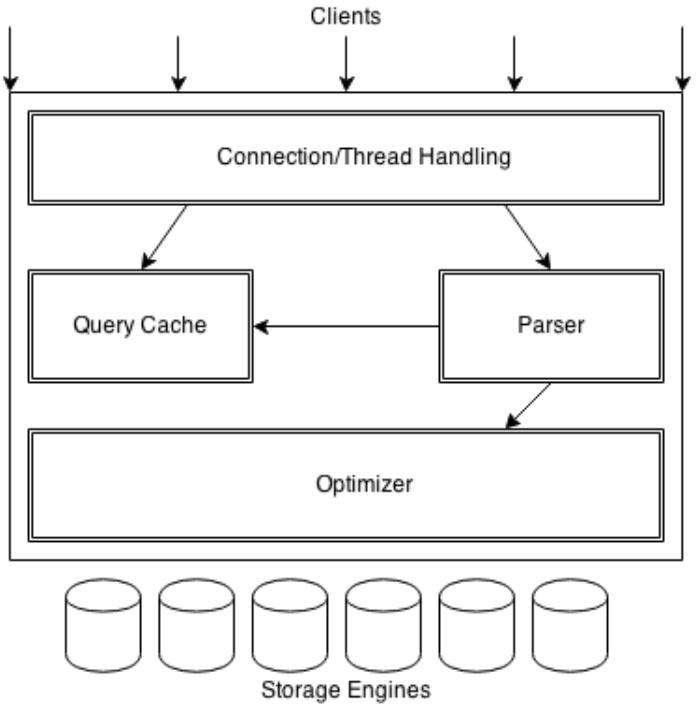

Figure 4: MySQL Architecture [30]

MySQL has a rich set of aggregate functions which are very useful when doing statistical analysis of the stored data. In total, MySQL has 17 aggregate functions in which the most common SQL standard aggregate functions AVG, COUNT, MAX, MIN, STD and SUM are included [31].

PostgreSQL

PostgreSQL is one of the few modern SQL databases that implements the whole SQL standard (ANSI-SQL:2008) [32]. The architecture of PostgreSQL is quite straight forward, as can be seen in Figure 5. The postmaster handles new connections and spawns a PostgreSQL backend process that the client can directly communicate with. All communication between client and PostgreSQL goes through that process after it spawns. The postmaster stays active and awaits new incoming connections. Each backend that a client is connected to has its own separate parser, rewrite system, optimiser and executor. All of the active backends then uses the same storage engine [33]. PostgreSQL, compared to MySQL, only has one very tightly integrated storage engine that in theory could be replaced but that has not been done in practice yet [34].

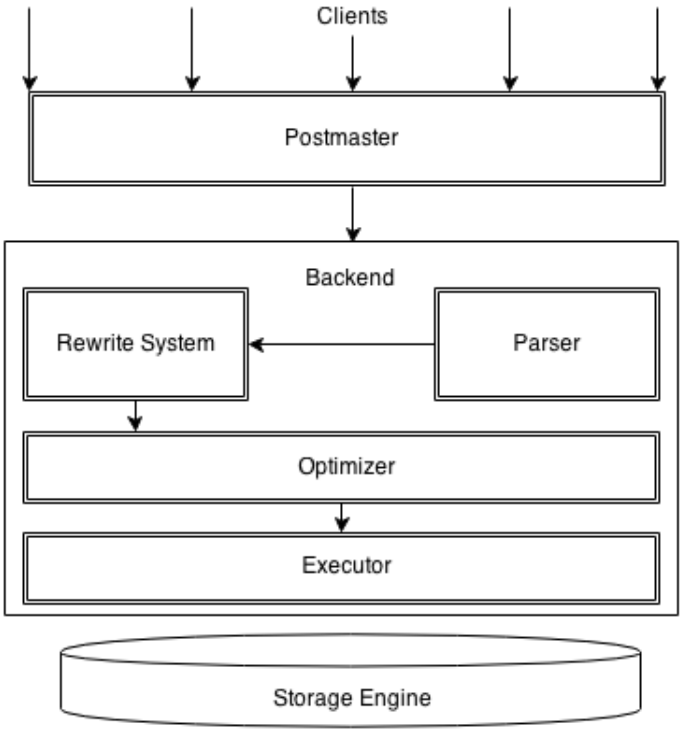

Figure 5: PostgreSQL Architecture [33]

PostgreSQL have a comparatively large set of aggregate functions including 14 general purpose functions and 18 functions mainly used for statistics, totalling in 32 functions [35].

SQLite

SQLite as opposed to MySQL and PostgreSQL is not built as a traditional two tier architecture which can be seen in Figure 6. This leads to that each call to it is done through function calls to a linked SQLite library [36]. SQLite is quite small (658KiB) and has therefore become widely used as an embedded database for local storage [37]. SQLite is bundled with two storage engines, one of them is built with a log-structured merge-tree and optimised for persistent databases and the other one is built on a binary tree which is kept in memory which makes it suitable for temporary databases. The architecture of SQLite has made it possible to design and interchange new storage engines with the built-in ones [38].

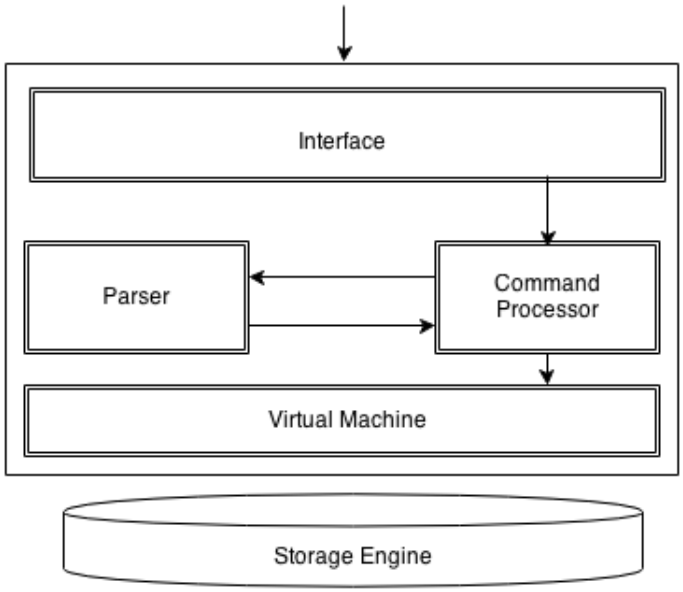

Figure 6: SQLite Architecture [37]

SQLite has 9 of the most basic aggregate functions and lacks the more statistical functionality such as for example standard deviation and covariance that PostgreSQL and MySQL implement [39]. SQLite's architecture is very focused on portability [37] and have made some trade-offs in order to achieve that, as can be seen in its simplistic architecture in Figure 6. One result of these trade-offs is that insertions become very slow, ~ 60 insertions per minute with transactions enabled [40].

As portability is not required for the implementation of the system regarded in this paper, because it is missing many important aggregate functions and because of its slow insertion rate, SQLite will be excluded from the tests in the next section.

5.3.4 Performance tests

As explained in section 5.3.3, SQLite was excluded from the tests as its architecture was deemed unfitting for the task at hand and for the lack of many aggregate functions. The statistics were generated on an average consumer computer with the following specifications:

CPU: Intel Core2 Duo T7700 2.40GHz
RAM: SODIMM DDR2 Synchronous 667 MHz 2x2GB
Disk: KINGSTON SV300S3 120GB SSD partitioned with EXT4
OS: Debian Jessie (Testing)

PostgreSQL version 9.4 and MySQL version 5.6 were used during the tests and the packages were installed from the default Debian testing repositories.

The database and table were created in the same manner on both systems as can be seen in listing 2.

Listing 2: Database and table creation

```
create database testdb
create table test_table
    (data varchar(100),
    moddate timestamp default current_timestamp);
```

The tests were conducted with a series of simple insertions containing random data, as shown in listing 3.

Listing 3: Example insertion

```
insert into test_table (data)
values ('0101001001010100...');
```

The execution times were measured by using the time [41] utility of the GNU project, it measures the time of execution and resource usage of the specified operation. The different files that were used in the tests contained 1000, 10 000, 100 000 and
1 000 000 insertions of random data. When deploying this system the production servers will most likely be a lot more powerful than the consumer laptop that these test were run on. These tests non the less gives a correct representation of how well the different databases will scale. To reproduce or verify the results, see listings 3 and 4 in Appendix A.

As can be seen in Figure 7 both of the systems have a very close to linear performance and any small disturbances in the pattern can be concluded to be from other processes using system resources. Both of the systems perform over the preconditions for an implementation with requirements similar to the ones in this context. But as insertion rate and aggregate functions are of high concern for such an implementation and both systems comply to the rest of the requirements, PostgreSQL will be recommended to use for a future implementation.

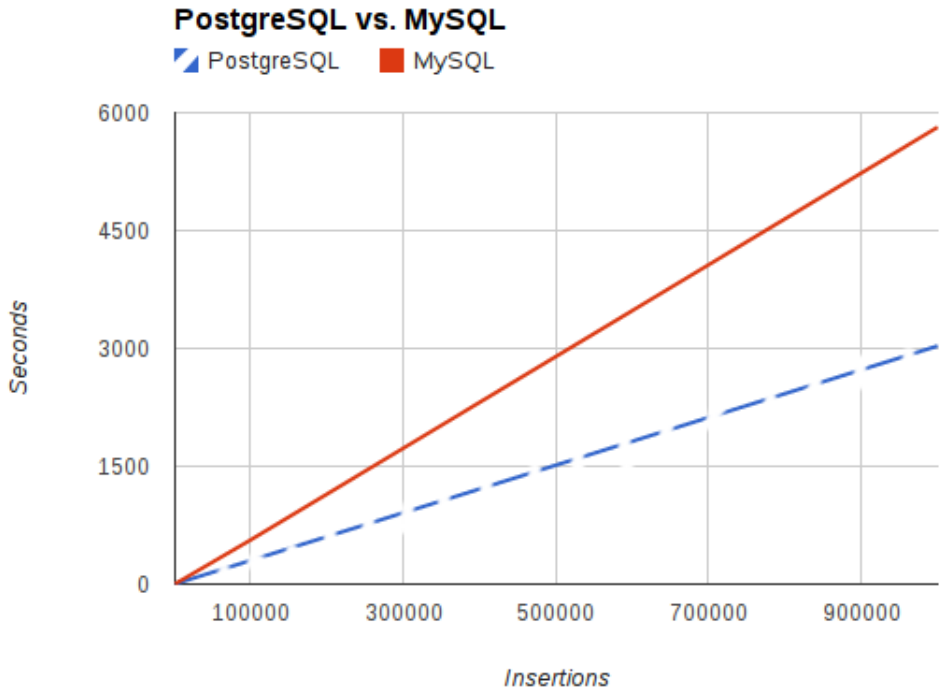

Figure 7: Performance graph

5.4 Storage requirements

As the data comes in as events, there is never a need to update old records in the database, as can be seen in Figure 8. This structure is slightly harder to do complicated queries (Listing 4) towards, as you always have to take the timestamp of the event into consideration. But it also makes it possible to configure the DBMS not to lock tables during inserts, which will increase the speed of insertion of the database. The lack of locks will be especially useful if it is later decided to use a DDBMS for an eventual requirement of scaling up the system in the future. As everything is logged with timestamps it is particularly suitable for business intelligence systems and the likes of such, as you can query time ranges in the past as well as the current state of the system to gather analytics and make comparisons.

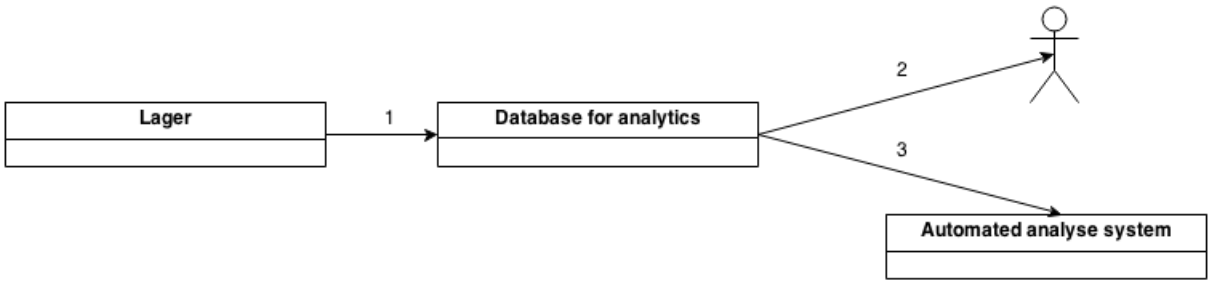

Figure 8: Flow of data

The only time inserts occur is during the time when the customised Lager [12] backend or later added data sources sends events (1, see Figure 8) and the only times reads are performed is when a manual query (2, see Figure 8) is executed or when an eventual business intelligence system or an automated analyse system (3, see Figure 8) that automatically queries the system is used. It will never be necessary to perform updates and no regular deletes will have to be performed, as explained in Section 5.3.1.

The actual database size should not be a problem as the servers are usually equipped with a few terabytes of disk space. An average insertion of Kivra is estimated to around 200 bytes by looking at meta data formed by data currently in the Riak cluster. This means that $5 * 10^9$ insertions would fit on a disk with 1TB of space. A rough estimation of an average of 5000 insertions per hour means that a 1TB would last for more than 113 years, see Figure 9.

1. $1TB/200B \approx 5 * 10^9$ insertions
2. $5 * 10^9/5000/24/365 \approx 113.5$ years

Figure 9: Estimation of disk space usage

6 Resulting system

The result of an implementation of the suggested solution of this paper would lead to a core system that would have a data flow as described in Figure 10. In this flow, when data of analytical interest would be inserted into the Riak cluster it would also be sent to Lager which would insert it into a RDBMS dedicated for analytical data, instead of fetching data in Riak after it has been inserted as was it done in the previous systems data flow that can be seen in Figure 2 and explained in Section 3.2.

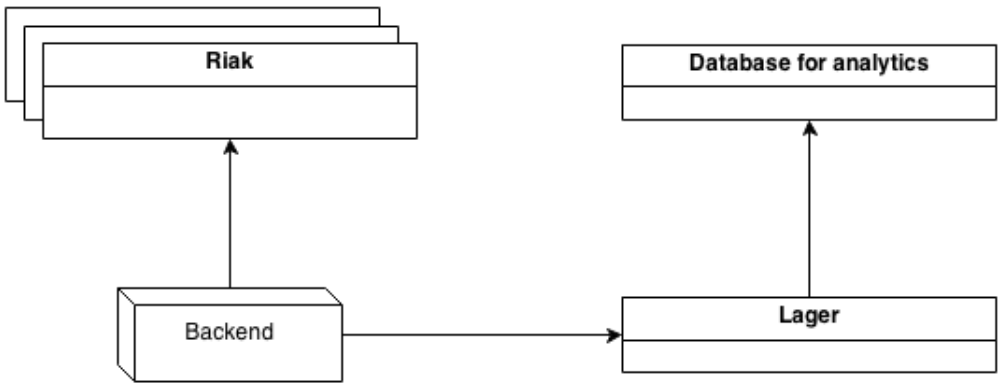

Figure 10: Resulting system

What is not shown in Figure 10, is that separate data sources for the analytical database can be used. These can connect with different protocols and with differently structured data to an API that will be receiving data for the database as described in Section 5.2.

6.1 Scalability

If an unoptimised PostgreSQL database on an average consumer computer is used and with the assumption that little to no data transfer latency will occur during insertion of data the insertion rate can be increased by a factor of at least 100 if following the upper boundary of the estimation (Section 5) of 10000 insertions per hour. As can be seen in the load test in 5.3.4, PostgreSQL can handle a little bit over 1 million insertions per hour. The expected data growth rate is 5% per month, with this growth the same setup could be used for almost 8 years, see Figure 11.

1. $10000 * 1.05^{94} \approx 981282$ insertions/hour
2. $94/12 \approx 7.83$ years

Figure 11: Estimation of data transfer scaling

In the unlikelihood that the same setup with the same hardware and PostgreSQL version would be used in 8 years the PostgreSQL database could easily be sharded, clustered or optimised to reach a lot higher insertion rates. As all he data is independent and has no defined relations to other data, clustering with a pool of independent databases could be used without any requirement of the databases to synchronise with each other, this would mean that you can linearly add 1000000 insertions/hour for each database node that you add to the cluster.

6.2 Practical Example

6.2.1 Storing an Event

After an implementation of a solution like this at Kivra the following would be the flow of a typical change triggered by a user. The system would update the send request bucket of Riak in Figure 12, but changes to other buckets would look exactly the same.

Figure 12 is a practical example of what would happen if a user updated his or her preferences of which tenants it wishes to receive documents from.

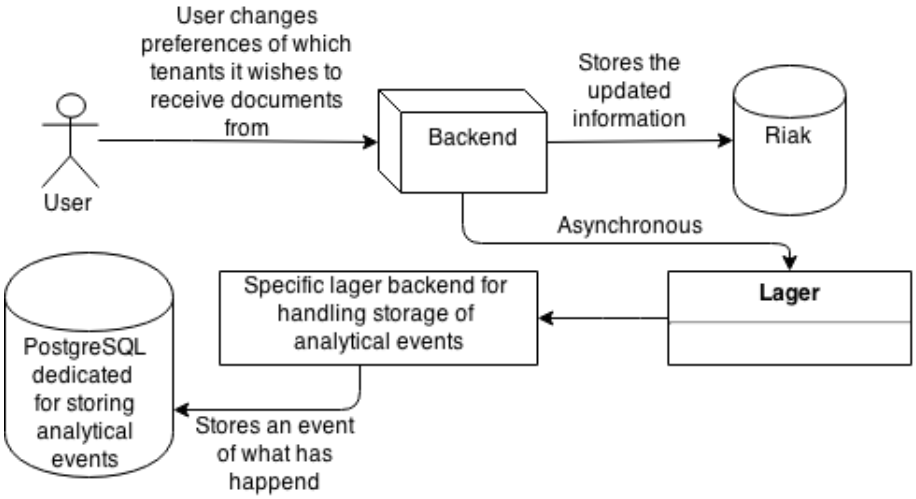

Figure 12: Practical example of a send request changing

6.2.2 Querying Data

To query data stored in the analytical database a query that takes the timestamps into consideration has to be performed. The query in listing 4 is an example of how to query which tenants all users wish and do not wish to receive documents from. Queries for other data sets would look similar.

Listing 4: Example query

```
SELECT SRQ.user_id, SRQ.tenant_id, SRQ.timestamp, SRQ.accepted
FROM sendrequest SRQ
INNER JOIN
   (SELECT user_id, tenant_id, max(timestamp) as latest
    FROM sendrequest GROUP BY user_id, tenant_id) SRQ1
ON SRQ.user_id = SRQ1.user_id
AND SRQ.tenant_id = SRQ1.tenant_id
AND SRQ.timestamp = SRQ1.latest;
```

The query in listing 4 doesn't take into consideration if the same name and tenant combination exists with the same timestamp. It is however highly unlikely that it would happen as the timestamps are stored with a micro second precision. To fully avoid that event, add an auto incremented ID column to the table and only consider the highest ID in the case of a clash.

7 Discussion

7.1 Event-driven data collection

The suggested solution introduced by this paper was named event-driven data collection. The name has been used before for slightly similar purposes with the first recorded case being in 1974 [42]. It has also been used in a different manner in quite a few papers on the subject wireless sensor networks [43] and some regarding the collection of natural ionosphere scintillations [44].

7.2 Flat file database

A flat file database was not compared in this paper because of the precondition of having pre-built APIs and to give easier access to querying and aggregation of data. A flat file database can also be complicated to scale out to several servers for an eventual future requirement.

8 Summary

8.1 Conclusions

The system that has been designed in this paper can be used as an effective tool for logging and analysing events. This form of approach is especially valuable if the main system is using some form of MapReduce database which can be hard to do ongoing analysis of aggregated data on. If the main system is already using a SQL database as its primary form of information storage this type of solution would be quite inefficient due to duplication of data and possibly having to serve another DBMS. In the benchmarking of the different database management systems the standard version without modifications have been used, with modifications and optimisations the outcome would have been very different.

The unorthodox design of the database schema in this paper provides a full historical log of events which makes it possible to compare data from different time ranges, which can be very valuable when analysing data from business systems to for example detect trends and tendencies within the user base.

Another contribution of this work is the use of a logging system to effectively put in an analytical systems to an already existing backend with minimal impact on the performance of the existing system.

8.2 Future work

To finalise the architecture of this system, APIs and web service structures will have to be researched or developed to be put on top of the database. For larger flows of data other solutions would have to be considered or alternatively the used DBMS has to be heavily optimised by for example removing insertion locks. Scaling out to a clustered database could also be an alternative.

Appendices

A Benchmarks

Listing 5: PostgreSQL insertion test

```
1 000 000 insertions
$ time psql --quiet -f test4.sql
psql --quiet -f test4.sql
40.42s user 23.08s system 2% cpu 31:05.03 total

100 000 insertions
$ time psql --quiet -f test.sql
psql --quiet -f test.sql
4.32s user 2.38s system 2% cpu 5:02.94 total

10 000 insertions
$ time psql --quiet -f test2.sql
psql --quiet -f test2.sql
0.51s user 0.22s system 2% cpu 30.172 total

1000 insertions
$ time psql --quiet -f test3.sql
psql --quiet -f test3.sql
0.11s user 0.02s system 4% cpu 3.077 total
```

Listing 6: MySQL insertion test

```
1 000 000 insertions
$ time mysql -u root -p********** testdb < test4.sql
mysql -u root -p********** testdb <
39.78s user 23.36s system 1% cpu 1:36:56.31 total

100 000 insertions
$ time mysql -u root -p********** testdb < test.sql
mysql -u root -p********** testdb <
3.88s user 2.83s system 1% cpu 9:22.67 total

10 000 insertions
$ time mysql -u root -p********** testdb < test2.sql
mysql -u root -p********** testdb <
0.31s user 0.39s system 1% cpu 53.174 total

1000 insertions
$ time mysql -u root -p********** testdb < test3.sql
mysql -u root -p********** testdb <
0.08s user 0.00s system 1% cpu 5.318 total
```

References

[1] "Data, data everywhere," The Economist, February 2010. http://www.economist.com/node/15557443, accessed 2014-06-22.

[2] Y. Liu, P. Dube, and S. C. Gray, "Run-time performance optimization of a bigdata query language," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, (New York, NY, USA), pp. 239–246, ACM, 2014.

[3] "Om kivra," Kivra, June 2014. https://www.kivra.com/about/, accessed 2014-06-26.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[5] C. Snijders, U. Matzat, and U.-D. Reips, "Big data: Big gaps of knowledge in the field of internet science," *International Journal of Internet Science*, vol. 7, no. 1, pp. 1–5, 2012.

[6] S. Tiwari, *Professional NoSQL*. John Wiley and Sons, 2011.

[7] "Nosql databases explained," MongoDB, Inc., February 2015. http://nosql-database.org/, accessed 2015-03-05.

[8] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, pp. 40–44, Jan. 2009.

[9] "Sibling resolution in riak," Richárd Jónás, November 2013. http://www.jonasrichard.com/2013/11/sibling-resolution-in-riak.html, accessed 2014-08-05.

[10] "What is riak?," Basho, June 2014. http://docs.basho.com/riak/latest/theory/why-riak/, accessed 2014-06-25.

[11] "Basho," Basho, June 2014. http://basho.com/, accessed 2014-06-28.

[12] "Introducting lager a new logging framework for erlang otp," Basho, July 2011. http://basho.com/introducing-lager-a-new-logging-framework-for-erlangotp/, accessed 2014-06-28.

[13] T. Bakuya and M. Matsui, "Relational database management system," Oct. 21 1997. US Patent 5,680,614.

[14] B. Beach and D. Platt, "Distributed database management system," Apr. 27 2004. US Patent 6,728,713.

[15] "Mobilt bankid," Finansiell ID-Teknik BID AB, April 2015. http://support.bankid.com/mobil, accessed 2015-04-06.

[16] "Don't use hadoop - your data isn't that big," Chris Stucchio, September 2013. http://www.chrisstucchio.com/blog/2013/hadoop_hatred.html, accessed 2014-08-17.

[17] "Mysql performance study," Hyperic, April 2011. http://download.hyperic.com/pdf/Hyperic-WP-MySQL-Benchmark.pdf, accessed 2014-07-29.

[18] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 53–64, June 2012.

[19] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[20] "Sql as understood by sqlite - aggregate functions," SQLite.org, September 2014. http://curl.haxx.se/docs/faq.html#What_is_cURL, accessed 2014-10-08.

[21] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, p. 1, ACM, 2011.

[22] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an sql db," in *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, (New York, NY, USA), pp. 5:1–5:6, ACM, 2013.

[23] "Apache hive tm," Apache, July 2014. https://hive.apache.org/, accessed 2014-08-05.

[24] "Apache pig," Apache, June 2014. http://pig.apache.org/, accessed 2014-08-05.

[25] A. Chalkiopoulos, "Programming mapreduce with scalding," Packt Publishing, June 2014.

[26] "What is apache hadoop?," Apache, August 2014. http://hadoop.apache.org/, accessed 2014-08-05.

[27] "Sqlite vs mysql vs postgresql: A comparison of relational database management systems," Digital Ocean, Inc., February 2014. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems, accessed 2014-08-18.

[28] "About mysql," Oracle Corporation and/or its affiliates, January 2015. `http://www.mysql.com/about`, accessed 2015-03-05.

[29] "About mariadb," MariaDB Foundation, January 2015. `https://mariadb.org/en/about/`, accessed 2015-03-05.

[30] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High performance MySQL: Optimization, backups, and replication*. "O'Reilly Media, Inc.", 2012.

[31] "Mysql 5.7 reference manual :: 12.16.1 group by (aggregate) functions," Oracle Corporation, March 2014. `http://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html`, accessed 2014-08-30.

[32] "Postgresql - about," The PostgreSQL Global Development Group, August 2014. `http://www.postgresql.org/about/`, accessed 2014-12-14.

[33] "Postgresql 9.3.5 documentation - 46.1. the path of a query," The PostgreSQL Global Development Group, February 2014. `http://www.postgresql.org/docs/9.3/static/query-path.html`, accessed 2014-08-19.

[34] B. Momjian, *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.

[35] "Postgresql 9.3.5 documentation - chapter 9," Oracle Corporation, July 2014. `http://www.postgresql.org/docs/9.3/static/functions-aggregate.html`, accessed 2014-08-30.

[36] C. Newman, *SQLite (Developer's Library)*. Sams, 2004.

[37] "The architecture of sqlite," SQLite.org, July 2014. `http://www.sqlite.org/arch.html`, accessed 2014-08-19.

[38] "Sqlite4 - pluggable storage engine," SQLite.org, July 2014. `http://sqlite.org/src4/doc/trunk/www/storage.wiki`, accessed 2014-08-30.

[39] "Sql as understood by sqlite - aggregate functions," SQLite.org, June 2014. `http://www.sqlite.org/lang_aggfunc.html`, accessed 2014-08-30.

[40] "The architecture of sqlite," SQLite.org, July 2014. http://www.sqlite.org/faq.html#q19, accessed 2015-03-10.

[41] "Time - linux programmer's manual," Linux man-pages project, September 2011. http://man7.org/linux/man-pages/man2/time.2.html, accessed 2014-08-28.

[42] J. D. Foley and J. W. McInroy, "An event-driven data collection and analysis facility for a two-computer network," *SIGMETRICS Perform. Eval. Rev.*, vol. 3, pp. 106–120, Jan. 1974.

[43] C. Townsend and S. Arms, "Wireless sensor networks," *MicroStrain, Inc*, 2005.

[44] W. Pelgrum, Y. Morton, F. van Graas, P. Vikram, and S. Peng, "Multi-domain analysis of the impact on natural and man-made ionosphere scintillations on gnss signal propagation," in *Proc. ION GNSS*, pp. 617–625, 2011.