Throughput rates of Simple Operations when Scaling-out with RDBMS and NoSQL Databases

Peter Christiaan Dijkshoorn

Throughput rates of Simple Operations when Scaling-out with RDBMS and NoSQL Databases

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Peter Christiaan Dijkshoorn
born in Vlaardingen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen B.V.
Simon Carmiggeltstraat 6-50
Amsterdam, the Netherlands
www.adyen.com

Throughput rates of Simple Operations when Scaling-out with RDBMS and NoSQL Databases

Author:	Peter Christiaan Dijkshoorn
Student id:	1217437
Email:	`p.c.dijkshoorn@gmail.com`

Abstract

RDBMS databases typically increase read operation throughput and decrease write operations throughput when more machines are involved. NoSQL databases increase at both throughput rates when more machines are involved. However, NoSQL databases are typically build to run with tens, hundreds or even thousands of machines while little is known about their performance on a smaller scale. This thesis describes the measured maximum throughput rates of read and write operations at RDBMS and NoSQL databases with one to four machines. These measurements show a potential for future databases.

Preface

In 2004, the little version of me stepped into the highest building of the Delft University of Technology, the EWI faculty. And (hopefully) the taller version of me steps out these days with a Masters degree. Taking a little bit of the size of the EWI building with me.

It seems a long journey, eight years for a five-years study but after my Bachelor degree I left to gain some experience in other fields, so I like to say that it does not count (politics thinks so too). During the overall period I grew in multiple ways which, at face, might not have contributed to the short-term goals. The same goes with the this last research period. However, I am confident that this period of my life brought me enough to make make it worth it.

Of course, I want to thank all the people who made this happen:

I want to thank Andy Zaidman and Maikel Lobbezoo for reading and commenting on all previous versions of this thesis. Both responded very fast and to the point, I could not wish myself better supervisors. Thanks!

Furthermore, I want to thank the other members of the committee, Arie van Deursen and Stefan Dulman, for their interest in my research project.

Many thanks go to my colleagues at Adyen who supported me with my project and always were nice sparring partners. Thank you guys for the talks and drinks to get things going.

I am also thankful to the management of Adyen for giving me the opportunity to perform this research in a very interesting environment.

Thanks to all my friends and family for hearing my complaints and over-enthusiastic shouts. You might not realise it but without you I would not have been here, thanks!

So, this should end my preparation time, let's see what I will do with it :)

Peter Christiaan Dijkshoorn
Delft, the Netherlands
November 6, 2012

Contents

List of Figures

Chapter 1

Introduction

Some webservices have such a high request throughput rate that the database, at the back-end of the webservice, is either a performance bottleneck or a feature bottleneck. An example[1] can be illustrated with the system of Adyen, a fast growing and globally operating payment service provider company located in Amsterdam, the Netherlands. Adyen has designed their system to be modular and replicated so they can spread the requests over the replicas. For instance, the webservice where shoppers make a payment runs on many machines simultaneously and all requests are spread among them. This way a user is directed to any of the running replicas of the webservice and together the replicated webservices can handle much more requests than a single webservice instance could do (see Figure 1.1a and 1.1b). At the end of the interaction with the shopper, the payment should be stored in the Adyen database, indicated by the arrow to the cylinder in Figure 1.1. However, due to the high rate at which the webservice is used, the single database becomes overloaded, thus a *performance bottleneck*. To fix this, the data can be buffered into a local database and only once in a while the data is flushed to the central database (see Figure 1.1c). This increases the total requests the webservices of Adyen can process even more.

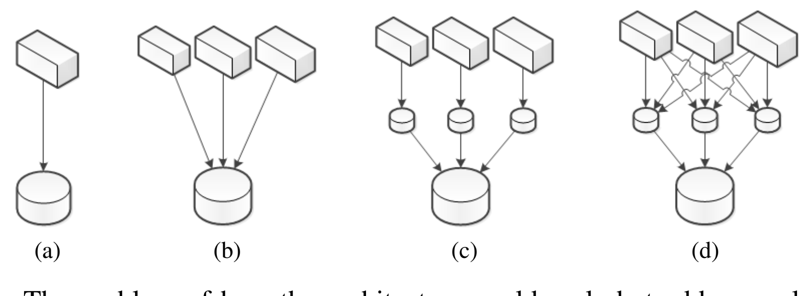


Figure 1.1: The problem of how the architecture could scale but adds complexity. (a) a single service using a single database, (b) multiple services using the same database, (c) services using local databases as write buffer, (d) new feature needs to read most recent data from all buffers.

[1]The example is used for illustration only, it does not reflect the system of Adyen entirely. However, other use cases are identified and will be discussed but are too complex for this introduction to the problem.

However, the buffering technique also limits the implementation of new features. For example, to know the status of a payment made only seconds ago, all the local buffers of each replica of the webservice should be queried (see Figure 1.1d). If this would happen for every payment request then the performance of the webservice declines dramatically. One can apply additional solutions to overcome this new problem but the fact is that the buffering technique created a solution for the performance bottleneck but also a problem as a *feature bottleneck*. Ideally, there would be simply one database instance which could just handle the higher request rate without intermediate write buffers. This can be done in two ways[6].

Scaling-up versus Scaling-out

One way to let a database process more requests is to give the machine more powerful hardware (See Figure 1.2a). For instance, install multiple disks, more space, more memory, faster memory, faster processor, more cores, multiple processors, and so on. This concept is known as *scaling-up*. The other way is to let the database work in a distributed manner (See Figure 1.2b). For instance, letting multiple machines work together to deliver a single database instance which processes requests at the required rate. This is known as *scaling-out*[6].

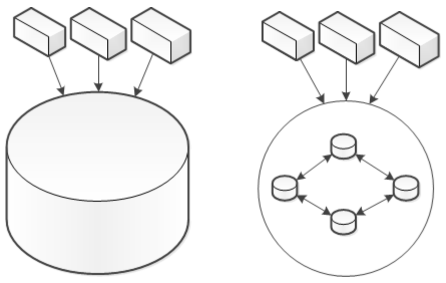

(a) scaling-up (b) scaling-out

Figure 1.2: Possible solutions: (a) higher performance database machine with better cpu, more memory and bigger storage, (b) letting multiple machines work together and act as a single database.

With the preference for one database which is able to handle the throughput rate and could scale along when the throughput rate gets larger, one has to choose between either scaling-up or scaling-out. Unfortunately, scaling-up is limited by the hardware developments[31], thus the only real option seems to be a system which can scale-out. However, the ability to scale-out differs for each type of database.

RDBMS databases

The most commonly known type of databases are relational databases[28]. Relational databases are based on theories developed in the seventies and eighties, such as the *Relational Model* and the *Transactional Model*. These concepts are discussed further in Chapter

2 but briefly said the *Relational Model* is to provide general access to all the data stored in the database and the *Transactional Model* is to execute operations on the data in a reliable manner. We will refer to this type of databases with the acronym *RDBMS* (Relational Data Base Management System).

RDBMS systems are generally built to *scale-up*[28]. The hardware of the machine should be replaced with better hardware if the user wants to process more requests. Scaling-out is also possible but in a limited manner. RDBMS databases can be set up with replication to multiple machines, either with only one writeable copy or with multiple writable copies. However, it is shown that in either case the maximum possible throughput rate of write operation decreases dramatically when more machines are added[24]. Thus RDBMS databases only scale-out to a limited number of machines.

NoSQL databases

During the last decade, some Internet services experienced such a high growth in usage that these services needed a database which could scale-out far beyond the limit of RDBMS databases, think of the services of Google, Amazon and Facebook. They developed a new kind of database with the ability to scale-out to hundreds of machines[8]. But they do not support the Relational Model and the Transactional Model because the services did not need them. Also, these databases were designed to run with hundreds of machines, which might make them less suitable when running at lower numbers of machines. We will refer to these systems with the term *NoSQL*, as the authors of these systems themselves tend to name their kind and because it is short and easy to remember. However, there is no consensus about the naming of these systems because the term NoSQL would not be adequately descriptive[33], we just use it as a label to refer to these systems.

The problem

A database which is able to scale along with the growth of requests is needed in the example of this introduction. We found that scaling-up is limited by hardware developments[28] and that the widely known RDBMS databases can not scale-out to an unlimited number of machines[24]. The NoSQL systems, however, easily scale-out to hundreds of machines[8] but might not run optimally when only a few machines are involved. Now, our question focusses on what maximum throughput rates the NoSQL systems are able to reach with only a few, say one to four, machines and if they, on that scale already, perform better than RDBMS systems. This way, we hope to find with what throughput rate it is beneficial to switch from a RDBMS database to a NoSQL database. Let us turn this into a proper research question in the following section.

1.1 Research Questions

In the former section we gradually introduced the reader to the problem area of our research, in this section we want to further narrow down this area to come up with our research question. As mentioned, we want to know how the maximum throughput rates of operations with NoSQL systems relate to those of RDBMS systems when scaling out to only a few machines. Now, we need to modify this to explain what kind of NoSQL systems we want to take into account (because a wide variety exists) and what operations we will put to the test. These two aspects are described below and afterwards we can state our research question.

To start with, the NoSQL systems can be divided into three families: 1) key-value stores, 2) document stores and 3) wide-column stores[34]. Each of these families has a number of implementations with each also being very different. Testing all systems would be a too large research, so we have to focus on one family. We will explain the differences in Chapter 2, for now it is important to note that we focus on the wide-column NoSQL systems only.

Also, we need to define which operations we want to perform during our measurements. There are many operations one can do with data, not only to a single data item but also with multiple data items. In our use cases we need to read and write single data items at high throughput rates. Thus in our research we will focus on these types of operations, also referred to as Simple Operations[34]. We will focus on reading an item based on its identifier, update an item based on its identifier and create a new item with a specified identifier.

With these aspects made clear we state our main Research Question as:

Main Research Question

„How do RDBMS databases relate to wide-column NoSQL databases in terms of maximum read and write operation throughput rate in a scaling-out context?"

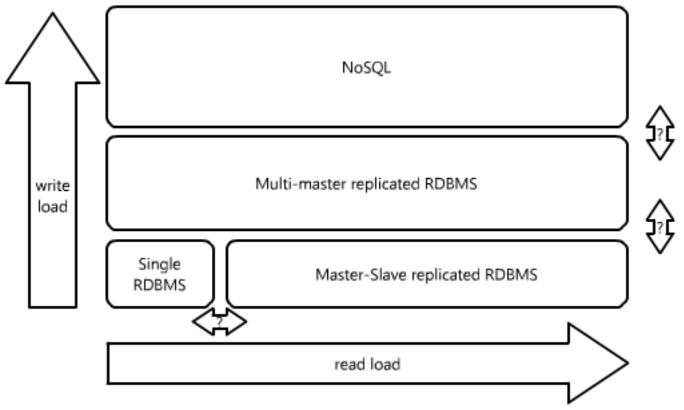


Figure 1.3: Schematic display of the solution space, the boundaries are to be defined in this research.

By answering this question we will get throughput rate numbers of both read and write operations. If we would scale those two on each axis of a two-dimensional chart, we could draw a dot in that chart for each system with its found maximum throughput rates. For instance, the RDBMS systems are expected to have a lower maximum throughput for write operations than NoSQL systems, thus we expect the dots for these systems to stay on the lower part of the chart. The same goes for the different RDBMS scaling possibilities and for NoSQL systems. We drew a sketch of how this chart could look like in Figure 1.3. At the end of our research we want to update this sketch with the actual results found.

To get the maximum throughput rates of read and write operations on both RDBMS systems and NoSQL systems, we first have to select systems from each family for our experiment. Then we need to identify which numbers indicate the performance of the systems under test. For instance, next to the maximum throughput rate we reach, we also want to know if this maximum is reached because the hard drive cannot do better or that the CPU is fully used. This is necessary to interpret the results of each system and to relate them to each other. From the results we then hope to derive general statements about which system might be favourable in what situation. We formulate this sequence of actions into four sub research questions:

> **Sub Research Question 1**
>
> *„What RDBMS and wide-column NoSQL database systems are available for high throughput in a scaling-out context?"*

> **Sub Research Question 2**
>
> *„What are the key performance indicators to measure the maximum throughput rate for each database system?"*

> **Sub Research Question 3**
>
> *„How do the key performance indicators relate from each system to another?"*

> **Sub Research Question 4**
>
> *„How to decide which system would perform better in which environment?"*

By answering these sub research questions we have a structured way to address our main research question. Also, the different parts can easily be used when making future comparisons or further elaboration on the performance of these systems. And thus makes this research more re-usable.

We introduced this research specifically via a certain use case at Adyen, let us now consider whether this research could also be relevant for others as well.

1.2 Relevance in Industry

Quite often one can find news articles about websites or webservices failing when they receive a lot of requests. This could be the intended result of a malicious action, as with Distributed Denial of Service (DDoS) attacks, but it also happens when the website or webservice suddenly gets more real visitors or a sub-system fails. However, it happens quite often:

In January 2012, ING had trouble serving the many request from all their customers' desktops and mobile phones when part of their database infrastructure was broken[2]; In January 2011, the Dutch government started an internet portal with a personalized overview of the pension of their citizens, it was not accessible due to overload of interest in the first few days[3]; Websites selling tickets to concerts were often overloaded at moment of opening sale, for instance Pinkpop 2011[4]; When weather conditions forced the Dutch railways in February 2012 to cancel most trains, travellers in the need of the new timetable often found the website inaccessible[5]; In January 2011 a fire at a chemical company in Moerdijk, The Nethetlands, occured and the crisis website of the Dutch government went down immediately by the huge amount of people in the need of information[6]; The same happened to the website of Amsterdam Schiphol Airport when an airplane crashed nearby in February 2009 and all travelers wanted to know the status of their flights[7].

We do not claim to know the specifics of these events but they do reveal that websites or webservices still have a hard time in processing a high load of requests. Adyen finds itself in an environment where it has to process a continuously growing high load of requests and they are ahead with making their webservices scalable. However, Adyen sees that the database is either a performance bottleneck or a feature bottleneck. This brings us to believe that a solution for Adyen would also be useful for making the above mentioned websites or webservices more resilient to an high load of requests. And thus is our research towards the relation between RDBMS and NoSQL databases in terms of maximum read and write operation throughput rate relevant for the industry.

1.3 Relevance in Academia

For Academia, we see three items of interest in our research. One is in flexible schema databases in multi-tenant service oriented architectures. Another is found in a general trend of webservices which is followed by Academia. And the third is in logging in service oriented architectures.

[2]http://www.rtl.nl/(/actueel/rtlnieuws/economie/)/components/actueel/rtlnieuws/2012/01_januari/24/economie/internetbankieren-ing-ligt-weer-plat.xml

[3]http://www.trouw.nl/tr/nl/4324/Nieuws/article/detail/1841598/2011/01/07/Pensioensite-ligt-op-dag-een-vaak-plat.dhtml

[4]http://www.l1.nl/nieuws/site-pinkpop-plat-na-presentatie

[5]http://nos.nl/artikel/337035-live-overlast-door-sneeuw.html

[6]http://webwereld.nl/nieuws/107416/falend-crisis-nl-krijgt-crisismanager.html

[7]http://www.nu.nl/internet/1923164/website-schiphol-weer-in-de-lucht.html

First, the problem of handling the high load on the database in a service oriented system is surfaced from research into multi-tenant service oriented architectures. A multi-tenant service offers the same service to multiple tenants (i.e. organisations) with a slightly different schema. In our payment example one could think of a merchant who wants to record an invoice address and a delivery address, while another merchant does not need to record an address at all. Research is done to find a database solution with a flexible schema but they often conclude that the possible solutions have such a low performance that they are hardly usable[3, 17]. We do it the other way around, we do a performance measure between the NoSQL and RDBMS systems of which the NoSQL systems often support a flexible schema. From this perspective, our work could be interesting to the multi-tenant services research field. A follow up research project is needed to investigate the suitability of NoSQL systems for multi-tenant webservices.

Next to that, general ideas in Academia about webservices indicate a strong direction into designing scaling-out systems instead of scaling-up systems[6, 2]. It seems unavoidable that the databases used with webservices are also needed to scale-out in the future[35]. Industry has already build databases which do that but they focus on their own, huge scale, use cases[8, 15, 30]. With our research we draw a picture of how these systems can be used at a smaller scale and we hope to find gaps which could be addressed in the future development of databases.

Third, a more tangible result can be gained into the research of logging in services oriented architectures. These architectures typically have so many services which output so many log messages per invocation that the log messages become so numerous that it is difficult to get useful information from them. This can be seen as a use case with a high write operation throughput rate for which our research question compares two database families. We actually implemented parts of such log messages collecting system with the use of the results of our research which revealed great information about the running webservices of Adyen.

1.4 Actors

The research is performed at Adyen in Amsterdam and Delft University of Technology (DUT) in Delft. For completeness, we briefly describe in this section how they are related to this research.

Adyen is a globally operating payment service provider, which means that webshops (and recently also physical shops) can connect to the system of Adyen and Adyen connects to hundreds of payment solutions all over the world. This way, the shops can offer their shoppers the payment solutions they want. Adyen has a service oriented system which processes hundreds of thousands payments each day. Most of their systems are replicated to share load and to have backup systems. The systems of Adyen should deliver high throughput and their experience is that for some subsystems, the throughput is limited by the database. With several design techniques they manage to process more load every day but these techniques also limit the functionality of some sub-systems. The ideal database would be able to process this high throughput of operations and is able to scale-out in the

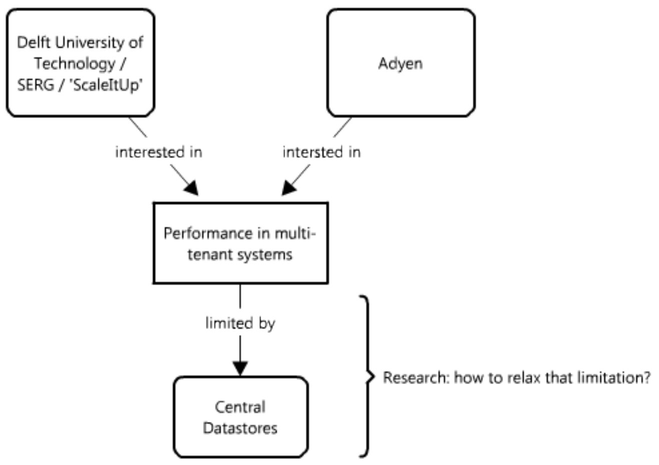

Delft University of Technology / SERG / 'ScaleItUp'
Adyen
interested in
intersted in
Performance in multi-tenant systems
limited by
Central Datastores
Research: how to relax that limitation?

Figure 1.4: Relation between actors and research domain

future. This is how they are interested in this research.

The DUT Software Engineering Research Group has a research project about multi-tenancy in service oriented architectures, called 'ScaleItUp'[8]. One of its challenges is to investigate the performance of such systems. As described earlier, due to the flexible schema requirement of these systems it is hard to find a database with sufficient performance. We think our work can give a hint towards NoSQL systems for this problem.

Next to that, to investigate the performance of webservices it would be useful to have insight in the log messages. We described earlier that this easily becomes a problem because of the amount of log messages. Our work contributes to the ScaleItUp project by investigating which database family can be used to process so many log messages.

To conclude, Adyen is also a Business Sponsor of the ScaleItUp project and contributes with this research to that. Figure 1.4 shows the relation between the actors and the subject of this thesis. And last but not least, this Master thesis is made for the fulfilment of the Master degree of the author at Delft University of Technology.

1.5 Document Structure

The document is divided in the following chapters. In Chapter 2 we describe the Background of the techniques used in RDBMS databases and NoSQL databases which are relevant for our high throughput analysis. Chapter 3 describes the environment of Adyen further and describes some projects at Adyen for which this research forms the basis. Chapter 4 describes the method we use for the experiment. The results of the experiment are presented in Chapter 5. In Chapter 6 we compare our results with related research. Threats we see in our experiment are listed in Chapter 7. In Chapter 8 we describe our contributions as

[8]ScaleItUp website: `http://swerl.tudelft.nl/bin/view/ScaleItUp/WebHome`

answers to our sub research questions and in Chapter 9 we draw our conclusions based on the results and thereby answer our main research question.

The reader is of course invited to read everything in a linear manner but we made enough references to guide the reader back from the Conclusions as deep as his or her interest goes.

Chapter 2

Background

In the Introduction we pointed out that the NoSQL databases are fundamentally different from RDBMS databases, in this chapter we dive into the internals to understand the differences relevant to our research topic. The reader is expected to have a general understanding of how RDBMS databases work. The NoSQL systems are relatively new and the reader is not expected to be familiar with it. We briefly recap the relevant techniques of RDBMS systems and discuss the techniques of NoSQL systems in more detail. The background information enables us to interpret the results of our experiment in Chapter 5.

2.1 RDBMS systems

The popular RDBMS databases we know today, like IBM DB2, Oracle, Microsoft SQL Server, MySQL, PostgreSQL and SQLite, are all generally based on fundamental research performed in the eighties[35]. They provide Atomic, Consistent, Isolated and Durable transactions with an Ordering, Indexing and Access Path Independent interface. These terms play an important role in the performance comparison between RDBMS and NoSQL systems. In this section we describe the background of these terms, readers already familiar with these terms can skip to the NoSQL databases in Section 2.2.

There are two major concepts surrounding RDBMS which are relevant for our research: the concept of the *relational model* and the concept of *transactions*. The relational model describes how data should be stored to maximise its usability. The concept of transactions defines how data should be manipulated in a concurrent environment to keep the results predictable. Both concepts were not followed during the development of the NoSQL systems which enabled them, they claim, to get better performance. That is why it is important to know how these two concepts were so important in the development RDBMS databases. The following two sections give an overview of these concepts. Also, we point out some remarks which were made during the development of these concepts which are worth mentioning with the recent development of NoSQL systems in mind.

2.1.1 Relational Model

The first concept is the relational model, which was introduced by Codd[9] in 1970 because he saw difficulties with the way information systems stored their data. The information systems provided an interface which highly reflected the way the data was stored. He introduced three independencies that should be honoured to provide *Symmetric Exploitation* and proposed the *Relational Model* as a solution to implement it. We will describe these in the following text.

Symmetric exploitation

Often information systems reflect the way the data is stored internally. Lets explain this with an example of an information system for employee administration, all employees in a company are grouped per department. A typical information system would let you specify the department to see a list of employees in that department. Internally it has a tree-like structure with each employee attached to a department and all the departments attached to the root, e.g. the company. Now, if a user wants a list of all employees at the Finance department, the system can easily provide that list.

A different user however, might want to know at which department John Doe is working because Jane, a client of John, has just arrived. The user knows the information is in the system, but our naive system does not provide any other means then going through all the departments to find the record of John Doe. This is generally the observation of Codd, the systems only provided a predefined way to access the data. The systems were not **access path independent**.

A secondary access path to the data could be easily made. For instance, by an alphabetical ordered list of employee names. But then again, if another user wants to know which employees live near his new home-town or the manager wants to know which employees earn more than a certain amount of money, the system needs to be redesigned. Codd argued to make it possible to query on any data item of a relation, even when no index is available. This can be done by ensuring that the system is **index independent**.

In the record of John Doe, many systems would have ordered the data and just remembered that the name is on the first line and the telephone number is on the second line in the file. However, a new version of the application could split the first-name and surname on the first line into two separate lines. This puts the telephone number on the third line and the whole application has to be revised to update all the pointers. Also, a migration program should convert the old data format to the new one. Codd stated the third property: the systems were not **ordering independent**.

These three independencies, when honoured, would result in a system which does not put limitations on the way the user can interact with it. It should enable the user to query on any data item that is in the system. Codd called this **symmetric exploitation** and argued that this is more important than offering **symmetric performance**. Symmetric performance would mean for instance that a user can only query on data-items of a relation which are indexed. This would ensure a quick retrieval of data but limit, as described above, the exploit-ability of the data.

The RDBMS databases implement the symmetric exploitation via the relational model, which is described next.

Relational model

The concept of symmetric exploitation was directly translated by Codd into the relational model as a way to implement the three properties[9]. This model describes information as being a relation between certain domains. For instance, the employee is a relation between the set of domains: name, telephone number and salary. An instance of this relation for John Doe would be: *<"John Doe","06-12345678","3000.00">*.

The Information system holds a list of instances of this employee-relation and a user should be able to query for employees having a salary of exactly 3000.00 or with a specified telephone number as easy as when the user specified the name. The Information system might implement a short-cut to find the specified values faster, but it does not need to. The symmetric exploitation is more important than symmetric performance[9].

This is an important concept in RDBMS systems and we will see that NoSQL systems do not support this concept as within their big-scale scenarios, symmetric performance is more important than symmetric exploitation.

Another concept very important to RDBMS systems but not supported by NoSQL is the concept of Transactions. The next section describes how the concept of transactions makes sure the data can be modified concurrently and why NoSQL databases do not support it.

2.1.2 Transactions

If multiple users can change the data in the database at the same time, the dataset as a whole can easily become corrupted. Lets assume in our example that everyone of the Finance department gets a pay-raise and that meanwhile the name of the "Finance" department is changed to "Treasury" department. These are an example of two transactions, both a set of multiple operations. Because both run at the same time, it could occur that half of the employees is already transferred to the Treasury department without the pay-raise while the transaction executing the pay-raise can not find any more employees working at the "Finance" department. When both transactions are finished, half of the employees at Treasury has the pay-raise and the other half has not, while both transactions argue to be successful.

This is unacceptable and the concept of transactions takes care of that. It lets both operations run as were they running in isolation, so the example of above could only have two outcomes: 1) All Finance employees have the pay-raise and are transferred to the Treasury department; 2) All Finance employees are transferred to the Treasury department and none have a pay-raise. The latter case can be identified rather easily and the operation of the pay-raise can executed again with the new department name in it. This is a very brief example of what the concept of transactions brought to RDBMS systems.

A more formal list of properties which a database should guarantee to transactions is known by the acronym *ACID*. ACID stands for Atomic, Consistent, Isolated and Durable[23, 26], which means:

Atomic: each transaction either executes completely or not at all;

Consistent: a transaction, when started from a consistent state, brings the data to another consistent state;

Isolated: concurrent running transactions run as they would do alone;

Durable: the results of a committed transaction survive any subsequent malfunction.

Together these four properties ensure a reliable execution of transactions. For our discussion on the maximum operations throughput rate it is important to discuss the *Durable* property further.

A durable transaction is a transaction which, once committed, will never be undone[23]. Take, for example, a user who inserts a pay-raise for our John Doe and the system reports back to the user that it has successfully done so. If the database system then shortly after crashes, it should still recover with the correct salary for John Doe.

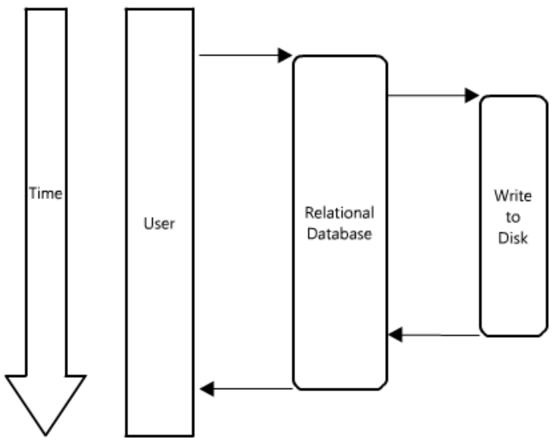


Figure 2.1: Schematic display of how a write operation is handled in a RDBMS.

The paper introducing the property of Durable transactions suggests to implement this by writing the modified data to disk before reporting back to the user (see Figure 2.1)[23]. This way, if the user is informed of a successful transaction, the data is on disk and would survive any software crashes and power outages in the future. If the crash happens before the data is written to disk, then the user is also not informed of a successful transaction. The paper also suggests to use disk mirroring to protect the data from disk failures, the chance that one disk will survive rises as more disks are involved[23].

RDBMS systems still make their transactions durable by writing to disk first before returning to the user[28]. This is important for our discussion about the maximum write operations throughput rate of RDBMS systems. As we will see later on, NoSQL systems do not use the disk for every operation (or transaction) but implement another solution to make data changes durable.

We described some of the main concepts behind RDBMS's which are relevant in our discussion of the performance of both families of databases, RDBMS and NoSQL systems. Let us continue by describing how a RDBMS can be deployed in a distributed manner.

2.1.3 Distributedness

The techniques described earlier explain the internals of a database deployed at a single machine, on top of this solution several possibilities exist to let multiple machines work together. In this section we will elaborate on these setup possibilities.

Relational databases have many implementations, ranging from proprietary products like Microsoft SQL Server, Oracle Database 11g and IBM DB2 to open source products like Oracle MySQL and PostgreSQL. The proprietary products are sometimes offered with specialized hardware to let the user *scale-up* the performance of the database. But both proprietary and open source products often have the ability, in one way or another, to run with several machines in a co-operating manner. Typical setups are master-slave and multi-master setups. The first setup has one machine to which write operations are send, and this one machine propagates the write operation to multiple slave machines. The slave machines can only be queried to read data.

In the multi-master setup writes can be send to all machines and these all propagate the writes to the others. This setup needs a conflict resolution algorithm to handle multiple write operations to the same data object. In the following section we discuss the details of both setups further.

Master-Slave setup

If a database is mostly used to read data and the load to read the data on a single machine is more than it could handle then one can use a master-slave setup (also termed master-standby)[25]. In such a setup, a single master machine is used to serve the write requests and multiple slave machines can handle the read requests. When new data is written to the master this is copied to the slaves, see Figure 2.2. This way, the system could scale out his read performance. Because write operations are still handled at a single machine, it can not scale in write load [20, 24].

Copying the data from the master to the slave can be done in two ways: synchronously and asynchronously. Synchronously means that a write operation on the master is directly copied to the slaves before returning to the user. Asynchronously means that the write operation is directly returned to the user by the master and only at arbitrary moments the master copies the new data to the slaves. Obviously, a write request from a user returns to the user faster when copying asynchronously. A drawback is that the overall database system, the combination of the master and all the slaves, is not consistent anymore. It easily follows from the asynchronous concept that a user may read an old value from one of the slaves while it had just written a new value at the master [25]

At Adyen, PostgreSQL is combined with Slony-I[1] which makes an asynchronous master-slave setup. This is mainly because in a synchronous setup the write operation takes too

[1] `http://slony.info/`

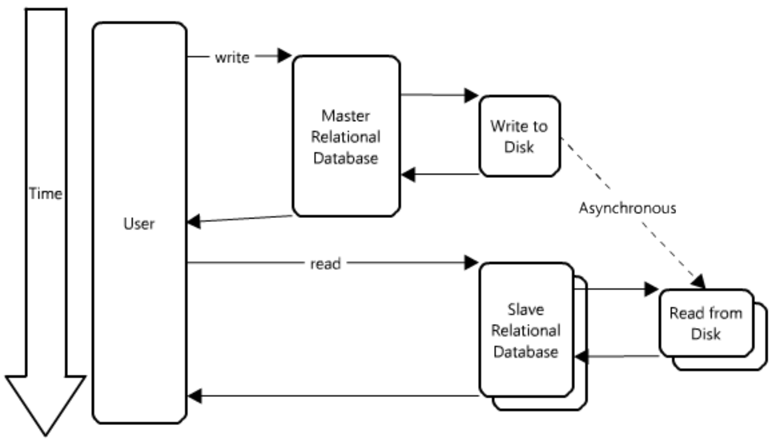


Figure 2.2: Schematic display of write and read operation in a Master-Slave setup.

much time. The application of Adyen is designed with the possible inconsistency of an asynchronous setup in mind. It is important to note from this, that industry chooses performance over consistency.

Multi-Master setup

In a Multi-Master setup each machine can process write operations and copies the new data to all the other machines, see Figure 2.3. Such a setup could, intuitively, be able to process more write operations per second than a single machine would. However, as more machines are involved, the time needed to contact all the master machines rises and thus less write operations per second can be handled per machine[24].

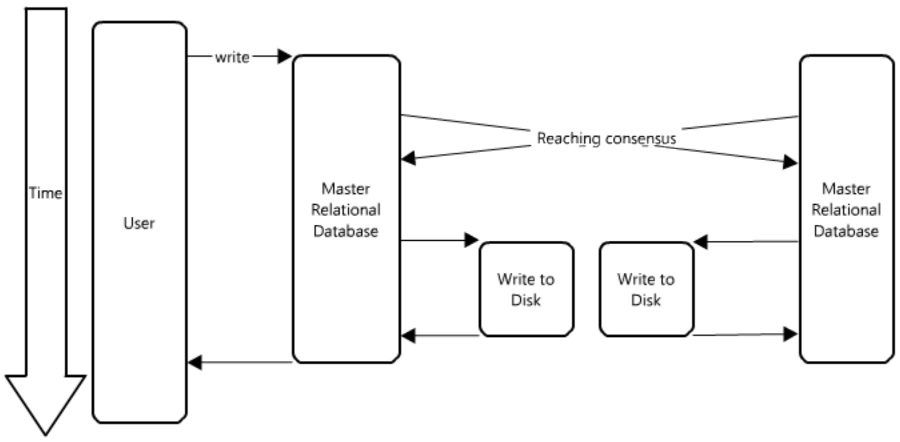


Figure 2.3: Schematic display of write operation in a Multi-Master setup.

Just like the master-slave setup, the multi-master setup can copy the new data synchronously and asynchronously. However, in this setup the machine to which the data is copied could also have modified its data. Both machines have received a write operation on the same data item and now have to decide which value they both take for that data item. This is called a conflict and a *conflict resolution* method is needed. The conflict could be resolved by the user (e.g. the user gets an overview of conflicts to decide which value is appropriate), or an algorithm is implemented which, for instance, selects the most recent value. These conflicts, and thus the application of the conflict resolution methods, are more likely to occur when more machines are involved. If data can be written at three machines, there can be three different versions while on a single machine only one version can exist. In fact, the occurrence of conflicts seems to increase a thousand-fold if the number of machines and workload increases a ten-fold[24]. This behaviour makes the database as a whole unresponsive if more machines are involved. We expect to see this reflected in our research.

A multi-master setup is interesting for our research because it has characteristics which NoSQL databases also have. In NoSQL databases, one can also send its write operations to any machine involved. These machines then also propagate the write operations to other machines, which makes it very similar to the multi-master setup of traditional databases. In the next section we will describe the techniques which NoSQL uses to overcome the drawbacks we saw with the distributed setups of traditional databases.

2.2 NoSQL systems

Companies like Google, Amazon and Facebook work with a much bigger amount of data than could fit on a single machine. They experienced that none of the setups of the relational databases could cope with the amount of data and the operation load they wanted to handle. Engineers from these companies had to come up with a solution for this and Google made the first real move in this area: BigTable[7] was presented in 2006. Amazon followed in 2007 with the Dynamo system[15] and Facebook published Cassandra in 2010[30]. We will look into the specifics of each system later on in this section but for now we will focus on the main differences in relation to the traditional relational databases.

2.2.1 Naming and the Relational model

The term *NoSQL* describes a broad area of data-storing systems, finding the common denominator is difficult thus let us first explain on which type of systems we will focus. The NoSQL systems are roughly divided in three families[34]. There are *key-value* systems, which map a single key to a single value. There are *document-store* systems, which map a specific key to a serialized object. And there are *wide-column* systems, which map a chosen key to multiple columns. In our thesis we want to compare the throughput rates of NoSQL systems with traditional RDBMS systems to seek for a possible replacement for the RDBMS system, currently used. We want a system closest to the functionality of RDBMS systems, we think that the wide-column systems are closest because of the column concept.

From the names of the three different types of NoSQL systems, one can already suspect that the storing model is not the same as with the RDBMS systems. As we explained carefully how the relational model brings *symmetric exploitation* by discarding symmetric performance (see Section 2.1), one could easily say that NoSQL systems do it the other way around. NoSQL systems are mostly build to deliver symmetric performance and dropped the symmetric exploitation concept. Data can only be retrieved by providing its unique key which obviously ignores the *access path- and indexing independencies*.

This way, the NoSQL systems do not support the *relational model* on which the RDBMS systems are based (Section 2.1). Also, the language designed to query the relational model, *SQL*, is also not supported by these systems from the start of development. It easily follows how the names for this group of systems, *NoSQL* and *Non-Relational*, came up.

2.2.2 Sharding of data

The wide-column NoSQL systems are designed from the ground up with the idea of multiple cooperating machines[7, 15, 30]. These cooperating machines were not supposed to be exact copies of each other, like most distributed setups of relational databases do. The idea is to increase the amount of data that could be stored. This let to automated *sharding* methods which means that the total amount of data is divided in parts and each part is placed on a different machine. Sharding is also sometimes seen in Relational databases, but then the application code should figure out which machine to query for a specific data item. In NoSQL systems selecting the appropriate machine happens as part of the system. Although, each system does this differently, some systems select the appropriate machine in the client driver, others select the appropriate machine internally[7, 30]. We will discuss this in Section 2.2.5. Let us first look at replication.

2.2.3 Replication for Durability

While having such a sharding system, some NoSQL systems also offer replication. This allows a machine to be taken out of the cluster without interruption of service because the data is still available on another machine. Replication happens only to a subset of all machines in the cluster. This design guarantees that the data is still available when a machine is taken out (by maintenance or failure) and limits the replication overhead. This can be seen as how NoSQL delivers the durability property.

Durability with RDBMS systems was delivered by writing to disk while NoSQL systems can reach it with replication. This could save time in processing a write operation because doing a network call to another machine and letting it store new data in memory is much faster than writing to the local disk[12] [2]

Now that it seems that NoSQL systems can reach the Durability property which was important in developing traditional RDBMS. Let us discuss another key property of RDBMS, Consistency, in the following section.

[2]note that SSD's (SSD: Solid-State Drive, physical storage without mechanical moving parts) are not considered yet in that research, as they are too small and expensive at time of writing).

2.2.4 Eventual Consistency

If a data item resides at multiple machines, as happens with replication, a new value written on one machine is not automatically reflected at the other machine. If then that same data item is read from the other machine, that read returns an older value than was written. This is inconsistent behaviour and in most applications this is unwanted. To overcome this, a method should be implemented to keep the system act consistently.

In distributed setups of traditional RDBMS systems there is a choice between distributing the writes synchronously or asynchronously. In the synchronous model all the copies of the data would be adapted before the user is informed that the data is committed. This would be a strictly consistent system. However, Gray showed in 1996 that this model leads to enormous conflicts and deadlock situations [24]. In the asynchronous model the copies of the data are adapted after the user is informed that the operation succeeded. An inconsistent situation is almost always present just after a write. However, *eventually* the copies are adapted and after a while the data is consistent again. This is called Eventual Consistency.

Most NoSQL systems support by default only Eventual Consistency, this often brings people in an alerted state as Consistency is very important in RDBMS databases through the acronym ACID (Section 2.1.2). Let us elaborate further on consistency in NoSQL databases to take away the scepticism.

Gray showed that a distributed database system with replication implementing strict consistency would experience dramatic deadlock rates. This lead to lowering the requirements to BASE[19], as a replacement of ACID:

Basically Available: 24x7 available, even with partial failure or with adding nodes;

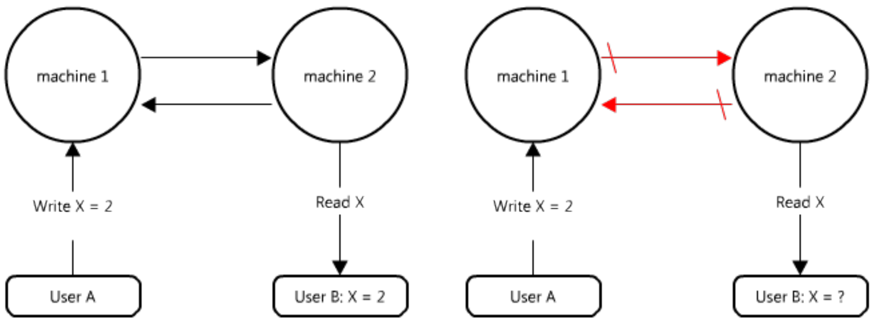
Soft state: a global state may be calculated afterwards, it is not needed at runtime;

Eventual consistency: if no updates follow, eventually all copies of data are synchronized at some point in time.

Note that Basically Available actually describes two things, the system should be *Available* and the system should be able to cope with failing nodes. Other nodes can not communicate with a failing node thus this can be modelled as network *Partitioning* [5]. The distributed database system should be able to cope with network Partitioning. This is important because hardware failure happens too often to be ignored in system design[12].

Now we have seen that the three properties Consistency, Availability and Partitioning are important, lets for explanatory reasons model a distributed system of two nodes.

In Figure 2.4a we have a two-node system and we write a value 2 for data item X, the receiving machine communicates with the other and a user reading data item X from the other machine gets the value 2. We say, our system is Consistent, Available and Partition tolerant. Let us introduce a network partition. In Figure 2.4b we have a cut between both nodes, not a node-failure but just a dead link. Now the system clearly is available, because users can interact with the nodes. The user is again writing the value 2 to data item X, there is however no way that the other machine can be informed of this write operation. The other user can thus not read the new value from the system. Our system actually is not Consistent while it is Available and Partition tolerant.



(a) (b)

Figure 2.4: Summary of the proof of CAP theorem, 2.4a is a Consistent, Available and Partition tolerant system but when a partition is made 2.4b it is also not consistent any more.

This is only one case of the proof of the CAP-theorem (also known as Brewers-theorem): “a system can only provide two out of the three properties: Consistency, Availability and Partition tolerance” [5, 22]. For a future user (e.g. Adyen) it is important to consider which two properties are most important for him or her to select a proper product. For instance, Apache Cassandra, a NoSQL database, initially only supported Availability and Partition tolerance. Later on, it lets the user configure the *Consistency Level* per operation, although this impacts the Availability and Partition tolerance[30].

2.2.5 Products

As mentioned earlier, there is a wide collection of systems identifying itself as NoSQL system and we focus on wide-column NoSQL databases, we do not consider document-store or key-value NoSQL systems. In particular we focus on Apache HBase and Apache Cassandra. We will refer to the techniques described earlier and will show the differences relevant for our research.

Apache HBase

Apache HBase is an open source implementation based on the paper about Google’s BigTable[8]. BigTable is a distributed storage system for structured data and was developed “to reliably scale to petabytes of data and thousands of machines”[8]. It is build to run on top of Google’s GFS[3], a distributed file storage system part of the Google MapReduce framework[13, 14]. The Apache HBase implementation runs on their Apache counterparts, Apache HDFS[4] as part of Apache Hadoop. In this section we give a summary of relevant parts from the BigTable paper[8] to give the reader an understanding of how the system

[3]GFS: Google File System

[4]HDFS: Hadoop Distributed File System

operates. This is important to know to be able to correctly interpret the results of our performance tests later on.

BigTable is much like a key-value store with the key split up into a *rowkey* and a *columnkey* part. When a new value is assigned to a key, the appropriate machine writes it in an appending manner to its underlying data-structures. Because the data is only written in an appending manner, multiple entries for the same key can exist. To distinguish these entries, a timestamp is added to each entry. BigTable uses the timestamp to select the appropriate value for a data item.

Ranges of rows are stored into *tablets*, this can be compared to *sharding* in RDBMS. The tablets form the unit to distribute over the machines. Each machine is responsible for a range of tablets. A master server keeps track of what tablet is where located. This information is stored again in tablets, called *metatablets*, and is distributed over the tablet servers. The location of the *metatablets* is stored in the *roottablet* which is also kept up to date by the master server but again stored at the tablet servers.

The location of this single roottablet is stored in a system called Chubby (or the Apache variant: ZooKeeper). Chubby is also a distributed system which operates like a master-slave setup and ensures a single version of the information it holds. BigTable also uses Chubby for locking.

A client application typically 1) reads the location of the *roottablet* from the Chubby cluster; 2) reads the location of the *metatablets* from the *roottablet* which it retrieved from a tabletserver; 3) reads the location of the required *tablets* from the *metatablets* from the appropriate tabletserver and 4) retrieves the data value for the required key from the appropriate tabletserver.

This is a complex process, compared to how RDBMS and Cassandra deal with data retrieval, and we expect to see this reflected in our tests.

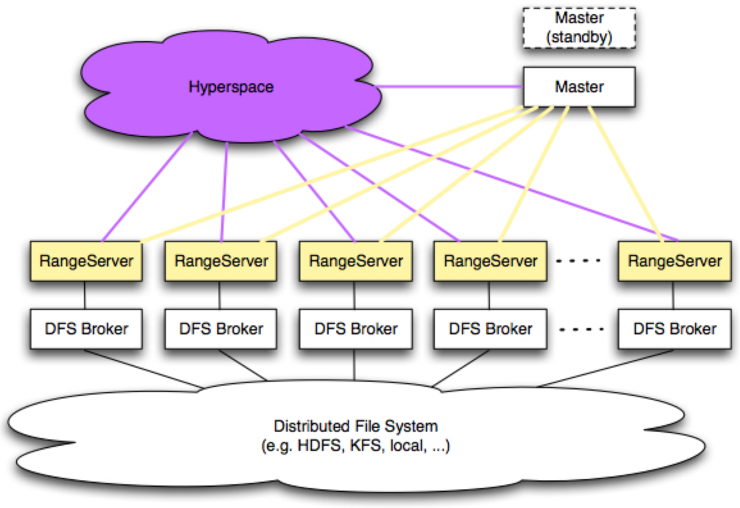


Figure 2.5: HBase system architecture (http://hbase.apache.org/).

Once the client found the appropriate tablet server and sends it a write operation for a particular key, the tablet server stores this in a memory table. As the memory table gets too big, a new one is created for future operations and the old one is written to the file system in the *SSTable* format. *SSTable* (Sorted String Table) is a file format with entries sorted on the key and includes an index to quickly look up the appropriate key.

The flushing of the memory table to a file on disk is called *minor compaction* and is initiated by the size of the memory table. Multiple SSTable files again are merged during a *merging compaction* which is initiated at random. Multiple versions of a data item can still exist in multiple SSTable files which uses unnecessary space. A *major compaction* reclaims this space by combining all SSTable files into one. This model of storing data enables a fast processing of write operations because it never has to look up the location of a particular data item when writing a new value to it.

Reading a data item is a different story, the data is written in an appending manner and multiple versions exist in the data files. To select the appropriate file a *Bloomfilter* is used. A *Bloomfilter* has a collection of non-unique hash functions and generates a bit-array[4]. For a key to look up, it generates the specific, but not unique, bitarray for that key and does a bitwise AND with the bitarrays of the SSTables. Each SSTable has a bitarray which is the bitwise OR of all the bitarrays of the entries it contains. This way, the Bloomfilter can tell whether the key is certainly not in the SSTable file or that it might be in the file. The system loads that file and looks up the specific entry using the index in the file. The loading of the file is a timely operation thus the hash-functions of the Bloomfilter need to be chosen carefully.

Important to note is that the SSTable files of BigTable are actually stored on the GFS, a distributed file system. GFS can be configured to replicate the SSTable files, which makes BigTable *Partition tolerant* while still being *Consistent*. BigTable is designed to support these in favour of being *Available* (recall CAP-theorem mentioned earlier in this chapter). The inner-workings of GFS[21] are much like BigTable but it goes beyond the scope of this thesis to give a detailed overview of this system. Relevant for our case is that this GFS might also impact the performance.

As discussed in the beginning of this section, Google BigTable is not publicly available. However, an open source implementation exists which closely follows the information available about BigTable. For our research we use Apache HBase, combined with Apache HDFS, all from the Hadoop family.

Apache Cassandra

Apache Cassandra was initially developed by Facebook and is available under an open source license. Cassandra is based on the papers of Google BigTable and Amazon Dynamo[30]. Dynamo was developed as a high-available key-value storage system while trading-in consistency in some circumstances[15]. Cassandra, like Dynamo, values *Availability* over *Consistency* and follows closely the paper of Dynamo but with the wide-column approach and the SSTable format from BigTable. In this section we discuss the relevant techniques of Cassandra further, based on the paper[30].

Unlike BigTable, Cassandra does not have master servers, all servers communicate with each other on an equal basis. This is a big advantage compared with BigTable as it implicitly means that Cassandra does not have a single point of failure. Also, the client is not aware of which data is on which machine, a client can connect to any machine and send it operations for any key. The receiving machine connects to the machines responsible for the data items and collects the results (See Figure 2.6). The receiving machine is called the *coordinator node* for that operation.

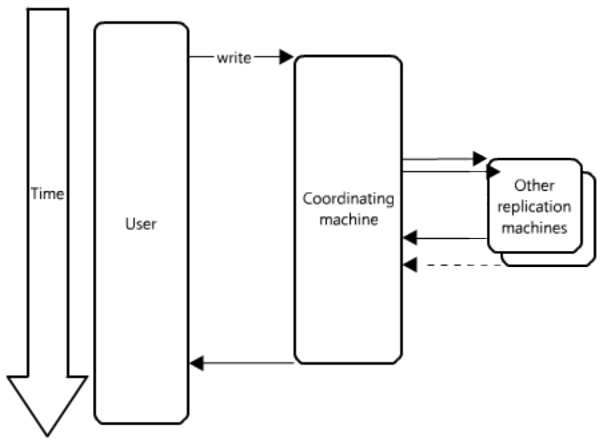


Figure 2.6: Schematic display of write operation in Cassandra.

The coordinator node is also responsible for detecting multiple versions of a data item when replication is active. It simply requests the data items from all responsible machines and compares the timestamps of each. When an old copy exist, that machine is updated by the coordinator node. This is called *Read Repair* and takes place after detecting the different versions. That does not mean that the user also receives the most recent version of the data item. By default, Cassandra returns the first value that the coordinator receives from the machines having a copy of the data item. This makes Cassandra *Available* and *Partition tolerant* and not *Consistent*.

Similar behaviour occurs when a write operation is issued and a machine having a copy of the data item does not respond to the coordinator node. The coordinator node then simply keeps retrying the write operation every now and then, a process called *Hinted Hand-off*. The user, however, is unaware of this and always receives a successful reply from the coordinator node as long as a replica is available. This behaviour also reflects the choice of Cassandra for *Availability* and *Partition tolerance* instead of *Consistency*.

However, a user can specify a *consistency level* per operation with which Cassandra can behave more consistent. Per operation, a user can specify how many copies must respond to the coordinator node before the operation is considered successful. For instance, a read operation with consistency level *All* means that all copies should reply to the coordinator before the coordinator replies to the user. The coordinator is then implicitly forced to consider all copies and return the most recent version of the requested data item. This also means that the coordinator waits for the response of all machines which should have a copy of the data item and thus the overall system could be less tolerant to network *Partitioning* and less *Available*[29].

Another consistency level option is *Quorum*, which forces the coordinator to wait only for a majority of responds from the machines having a copy of a data item. For instance, when a data item is replicated at five machines and a write operation is received with a consistency level *Quorum*, the coordinator will successfully reply to the user after receiving only three responds from the five copies. This method is more *Available* than when operations with consistency level *All* are issued because it can still operate successfully if one or two machines are unavailable. By configuring the consistency level for both read and write operations to *Quorum*, Cassandra should be able to reach strict consistency by giving in on its *Partition tolerance*[1].

Together with the same *SSTable* format and compaction strategies as Google BigTable, all techniques described above are relevant when comparing operation throughput. As just described, Cassandra can behave consistent with particular operation parameters (consistency level is a per operation parameter). In our tests we took this into account and measured the performance difference of each of the consistent settings.

2.3 Summary

In this chapter we described the relevant techniques and ideas found in RDBMS and NoSQL databases to understand the operation throughput in our tests.

We described, in Section 2.1, how the *relational model* ensures *symmetric exploitation* (access path, indexing and ordering independency) in contrast to symmetric performance and how a transaction should ensure the properties: *Atomicy*, *Consistency*, *Isolation* and *Durability* (acronym: *ACID*). Durability is mostly implemented by writing to disk in RDBMS. We described *synchronous* and *asynchronous* replication in *master-slave* and *multi-master* setups as possibilities to *scale-out*.

We described, in Section 2.2, how NoSQL databases were designed to scale-out to cope with greater datasets and how replication is used instead of a disk write to make data durable. We mentioned the *CAP-theorem* which states that only two out of the three properties *Consistency*, *Availability* and *Partition tolerance* can be supported by a distributed system. This lead to *Basically Available*, *Soft state* and *Eventual consistent* (acronym: *BASE*) systems instead of *ACID* systems. We described the relevant parts of how Google BigTable works and thereby the parts of the open source implementation Apache HBase. These systems prefer *Consistency* over *Availability*. Apache Cassandra is another NoSQL system which by default, does not provide *Consistency* but can be configured to do so with giving in on *Partition tolerance* and *Availability*.

With the techniques of both families of systems made clear, one could understand that the NoSQL systems are build to scale-out to tens, hundreds or even thousands machines, as discussed in Section 2.2. We also made clear that RDBMS systems have scaling-out possibilities, although only for a small number of machines, in Section 2.1. Our research focuses on the question when the performance of RDBMS while scaling-out ends and when the performance of NoSQL systems becomes beneficial while scaling-out.

Chapter 3

Case: Adyen

This chapter describes the environment where Adyen operates in and discusses some particular projects where our research contributed to. This might help the reader in having a context for this research and it describes the activities of the author at Adyen. First, an introduction to Adyen:

Adyen is a Payment Service Provider, it supplies a one-stop shop for (online) merchants to let their shoppers pay via various payment methods. Shoppers are offered for example to pay via PayPal, iDeal or Credit Card while the merchant only needs to connect to Adyen. This way, Adyen processes hundreds of thousands payments per day from all over the world. Their systems are highly optimised to process these payments fast and correctly while adhering to the strict security regulations of the business.

The system design reflects this by using various tricks to minimise the impact of a single payment. The services interacting with the shopper are made *stateless* in such a way that only the payment itself gets through to the central components, no other interaction with the shopper is collected centrally. The services, which run in a replicated manner, *buffer* the payments which are now and then flushed to the central database. However, not every piece of data that makes up a payment is recorded at the same machine. Some elements are stored at other machines, a form of *vertical partitioning*. This worked well and has enabled Adyen to grow considerably in the past, as it will also do in the future.

However, a few projects exist with a certain feature set which can not be met if above techniques are used for growth. These projects are the main incentive to look around for other solutions. The NoSQL databases seemed like to offer such a solution and this research was done to investigate that further. In the following sections we point out three of the projects we worked on: statistics, graph data structures and log analysis.

3.1 Statistics

During the year that the author was embedded at Adyen, all kind of statistics were calculated from data generated by the systems. Either for public press releases[1], or for stating numbers for internal business cases. An example of the last, is the frequent downtime of the iDeal

[1]For example: `http://www.adyen.com/wp-content/uploads/2012/03/MobileFigures.pdf`

systems (Dutch payment method of Currence[2]). Having such statistics gives the company a very powerful position in business negotiations[27].

The statistics projects are often about very much data and the answer is required in relatively short time (end of the day). Often, it is about finding irregularities, so beforehand it is not known what statistics are useful and we need to *browse* through the data to discover meaningful things. To gain the flexibility in querying we usually loaded the data into a RDBMS database with which we could explore the data. The data is, however, of such big size, that loading it into the database is often a time-consuming task (even with parallel import execution, fsync off, large transactions and using most memory).

Executing queries to *browse* around and find relevant numbers, also revealed that the database has a hard time with the enormous amount of data. The queries took a relatively long time (even without joins or sub-queries, just aggregating) to complete and made the whole system quite unresponsive for other queries. When the business has already put a time-limit on such a project, then you have almost no option to increase the speed of the database other than taking less data in. The drawback of less data is that the statistics become less reliable.

A database which could get faster by adding more machines to it, would in these cases be an ideal database. The NoSQL databases we have tested support this *scaling-out* property and our research shows how they perform with a small number of machines in relation to the RDBMS database configurations. Getting more machines in short time is another problem, but for these short term projects one could also think of setting up the workstations of all team-members into one single joint database system for such a day. Work is in progress to arrange such a setting.

3.2 Graph data structures

Most data structures at Adyen are simple lists, payments for example are continuously added to a 'list' and only the last added payments are retrieved to get the details of it. This is great for system design, the system can phase out the old payments to less accessible but bigger storage. This way it can control the amount of data to which fast access is needed.

However, another project needs to link payments to each other to make up a graph data structure as in Figure 3.1. Such a graph can be useful for Risk analysis (Fraud detection) which has been researched, and proofed to be a significantly better solution than current methods, by Bert Wolters(Adyen)[38]. Another benefit is that such graphs would provide insight into the Shopper profile. However, the applications with the graph data structure do not have the property that only a popular, or latest, set of payments is retrieved but instead it needs to read payments in random order.

For system design this is a very unpleasant property, once the dataset runs out of memory and starts to be only available on the hard disk, the overall performance of the system will drop by a factor ten (at least, with the machines of our research). Since we plan to use such a system during a live interaction cycle with the shopper where the overall response time is very important, this sudden decrease in performance is undesirable.

[2]`http://www.ideal.nl/`

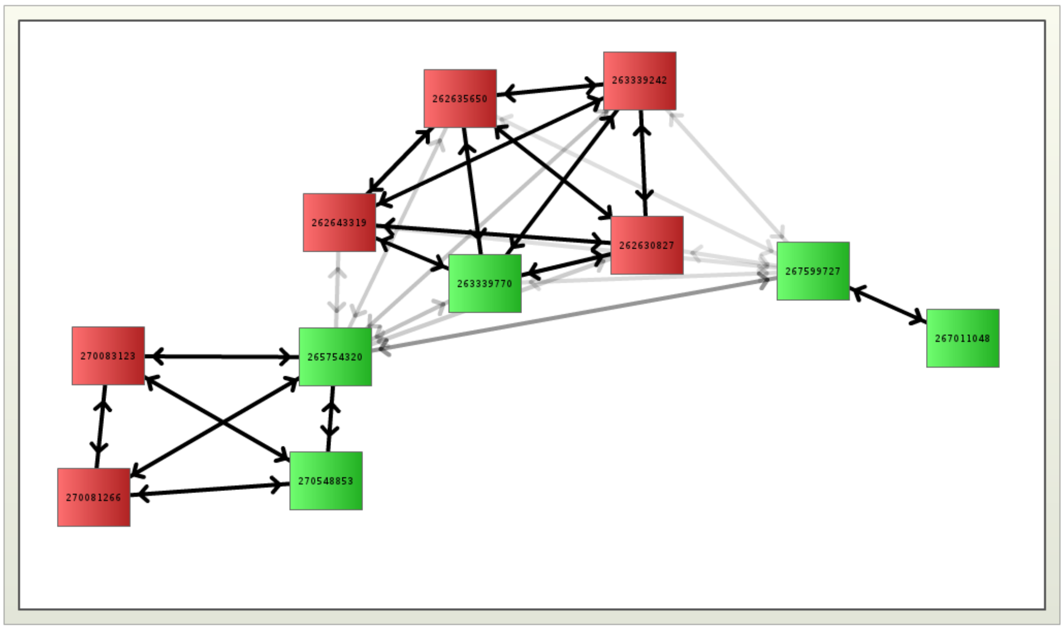


Figure 3.1: Linking of Payments in a proof of concept system running with Cassandra, the green, validated, payments might be interesting to review because they are directly or indirectly linked with quite some red, refused/fraudulent, payments. Each edge represents that both payments share a property, such as IP-address or Credit Card number. They density of the link represents the strength of the relationship.

In fact, we need a database system that can grow in dataset size and provide the same constant performance with arbitrary dataset size. Our research shows that NoSQL systems can do that by scaling out. However, our research also shows that reading data from these systems is a lot slower than RDBMS systems do (if the dataset fits in memory). Currently, we are testing Proof of Concept systems to decide with what configuration this project proceeds.

3.3 Log analysis

The systems of Adyen consists of many machines, some of them are dedicated for a particular module of the system and others run replicated services. When a Shopper sends a HTTP request to one of our services, a lot of other machines are somewhere in the chain to process the request and to make up the response. Each of the machines generate data which is not directly needed by any other system at that moment. Such data is written to log files and makes the system to focus on what it should do: processing payments.

The log files contain a lot of information which is needed for several secondary services we offer. Several techniques are in place to get the most needed information from the log files. However, most techniques involve inspecting the log files at each machine separately. In the ideal situation we would like to have all log messages at a single data repository or database. With such a solution, we could gather an entire trace of all log messages generated by different machines with a specific shopper request. With such traces we could deploy Spectrum-based Fault Localization (acronym: SFL) techniques, researched last year at Adyen by Joël van den Berg[37]. SFL, in short, is a method to identify the most likely faulty component from a collection of traces which are marked as either successful or not[37].

Currently, different solutions are tested to deliver a single database for all the log messages. A solution with a tuned RDBMS (PostgreSQL) did prove not to be able to deliver at our requirements. At the moment of writing, a Lucene solution is tested but did also proof that our load of log messages might be to big for this solution. Again, our research shows that a NoSQL solution might be a perfect fit and work is in progress to test such a solution too. The research of another master student, Joey Siadis, will built upon this system and develop it further to include traces.

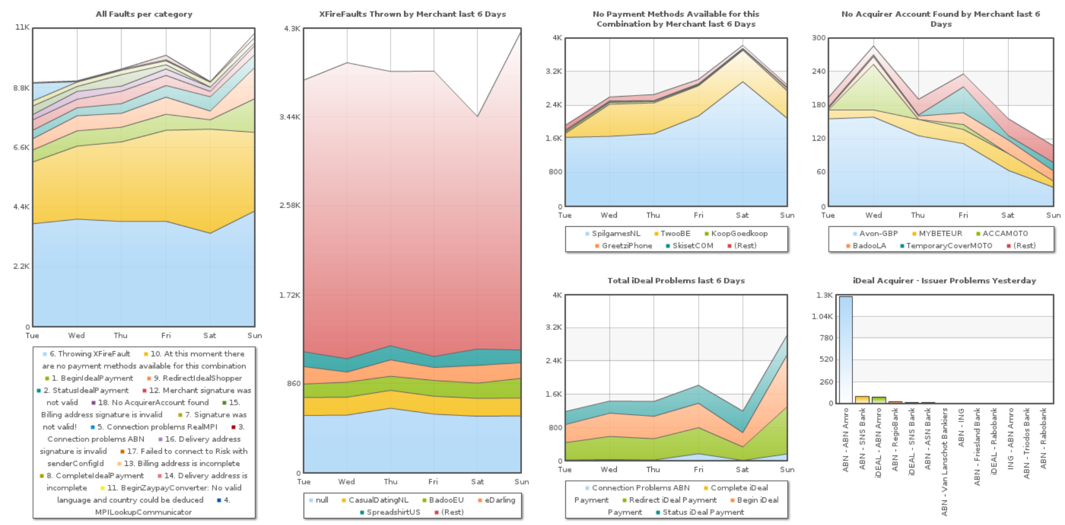


Figure 3.2: Webinterface of charts about the number of errors with merchant integrations over the past six days.

Intermediate steps are also taken: as part of this research we also developed an interface with charts showing the counts of different types of error messages as in Figure 3.2. From this project we gained understanding of the enormous amount of errors in the systems which are mostly errors made by the implementations of the merchants. Some practical situations were solved by this tool, including but not limited to:

- unannounced security audit, not allowed by our policy.
- accumulating requests from a merchants misusing the recurring payment method.
- several misconfigured payment methods at the merchant side (leaving shoppers at a blank page).

- defaulted merchant trying to get through as many payments as possible, leading to rising load on our systems.

For our research it gave us insight in the amount and characteristics of log messages and thus to understanding what characteristics a database should have.

Chapter 4

Experiment description

In this Chapter we describe how we set up our experiment to answer our research question. To recall, our reserch question is:

Main Research Question

„How do RDBMS databases relate to wide-column NoSQL databases in terms of maximum read and write operation throughput rate in a scaling-out context?"

Following from this question we defined four sub questions. Two of them are relevant for the set up of our experiment:

Sub Research Question 1

„What RDBMS and wide-column NoSQL database systems are available for high throughput in a scaling-out context?"

Sub Research Question 2

„What are the key performance indicators to measure the maximum throughput rate for each database system?"

Concerning Sub Research Question 1, we saw in the Background Chapter (Chapter 2) that multiple products exist for both RDBMS and NoSQL systems. We need to select proper representatives of both families and will do that in section 4.1. We deploy the systems at certain machines which need to be described to interpret the absolute outcome values correctly. For instance, we could get a throughput rate of 30 operations per second, which could be interpreted rather disappointing when reached on a fully geared server box, while it is fairly high when reached on a machine available in the eighties. We describe the machines in Section 4.2.

Sub Research Question 2 aims at multiple angles. First, we need to have a *Stress tool*, to execute operations on the database systems and measure how many operations pass per second. Secondly, we need a *System Monitoring tool* to investigate the resources of the machines to verify by what resource the maximum throughput rate is bounded. This is important because with this we can validate that the machine issuing the operations towards the database is not the limiting factor but the actual database itself is at its maximum throughput rate. This also gives us insight in how each system is bounded to which system resource, and thus tells us how the system can eventually scale-up when needed. Both tools are discussed in respectively Section 4.3 and 4.4.

4.1 Systems

For our experiment we need to select RDBMS and NoSQL database systems. The RDBMS systems have a longer development history and we have more products to choose from but with less varying features. The NoSQL systems are still heavily in development and each product differs more in the technology behind it. In this section we explain our product choices.

4.1.1 RDBMS systems

There are many (well-known) RDBMS database products: IBM DB2, Oracle, Microsoft SQL Server, MySQL, PostgreSQL and SQLite, to name a few. It is assumed that selecting one reasonably popular product will cover the performance achievable with RDBMS technology sufficiently. We need a database server which we can also deploy in the distributed configurations, like master-slave and multi-master. SQLite is not able to do that, as it is merely a locally executing library than a listening server.

Next to that, we do not have budget for a commercial product so we are limited to choose between MySQL and PostgreSQL. We choose PostgreSQL as it is used throughout the systems of Adyen.

This is an important choice as it will represent the entire family of RDBMS systems. It could very well be that other RDBMS systems perform better than PostgreSQL does in our use cases. In that case, the absolute throughput rates we get from PostgreSQL could be lower than other RDBMS systems could achieve. However, the technology behind each RDBMS product is roughly the same. Our choice for PostgreSQL could thus result in a faster declining rate or slower inclining rate than possibly better performing RDBMS products when they scale out. This should be taken into account when evaluating the results of both RDBMS and NoSQL systems. If the results lie close enough that another RDBMS product could likely achieve the same throughput rate as NoSQL then we should be very careful with the statements we make.

We will test PostgreSQL running on a single machine and on multiple machines using scaling-out configurations: master-slave and multi-master.

The master-slave configuration is set up using PostgreSQL with Slony-I[1]. Slony-I is a

[1] `http://slony.info/`

set of scripts which distribute just-written data from the master instance to the slave instances. As discussed in Section 2.1, the master-slave replication can work synchronous or asynchronous. The master-slave configuration we use works with asynchronous replication which, as discussed, trades in consistency for performance.

The multi-master configuration is set up using PostgreSQL with Bucardo[2]. Bucardo is also a set of scripts which replicate the data asynchronously. Because conflicts can occur when two or more master servers have edited the same data item, one has to select a conflict resolution scheme. We selected the conflict resolution method ‘latest’, which should select the newest value as the proper value. The first machine is configured to *swap* all the data between each master instance. This puts most extra work for the multi-master configuration at the same machine and more efficient methods could exist. However, the results of our experiment did not show that selecting a more efficient multi-master configuration could make much difference in the conclusions we could draw.

EMC Greenplum

A co-worker at Adyen wanted to see how EMC Greenplum[3] would do in our experiment and that is why it is included. Greenplum is a special configuration of PostgreSQL servers, with one master server and multiple data servers. The master server accepts the operation requests and addresses the appropriate data servers as itself does not hold any data. Each data server holds a specific range of data. Greenplum is proprietary but has a community version available. We did our best to configure it correctly with the manual[16].

4.1.2 NoSQL systems

The NoSQL products vary more in features and techniques than RDBMS products do, which makes selecting proper products more difficult. However, we already narrowed the research question down to only incorporate wide-column NoSQL systems. In Chapter 2 we explained that these systems better mimic the data model of the RDBMS systems than the *key-value* and *document-store* NoSQL systems. Also do these systems support scaling-out for write operations.

Within the wide-column NoSQL systems there are multiple products to choose from but many being a re-implementation of the same techniques. For example, Accumulo, Hypertable and HBase are all implementations based on the Google BigTable paper. We choose to include HBase in our experiment as it is the most popular at the moment of writing (mainly because of the entire Hadoop platform). As described in Chapter 2, Cassandra uses a whole different technique to be able to scale-out write operations. Using no master servers, or any other special node, Cassandra is a very interesting product to have a look at. We think these systems cover well enough the capabilities of NoSQL systems to have a reliable comparison with RDBMS systems to answer our research question.

In both HBase and Cassandra, one can configure how many replicas will be made of each data item. Replication is used to deal with node crashes. Because data is not directly

[2] `http://bucardo.org/`

[3] `http://www.greenplum.com`

written to disk, the replicas make sure the data will survive a crash of an individual node. This made us to also include different configurations for this into our experiment. However, HBase appeared to already have difficulties without any replication on our machines, so we were not able to test HBase with replication. Cassandra, however, is included with replication configured to three. It was advised on the Cassandra mailing-list to use an odd number of replicas and we have only four machines (See Section 4.2), thus three was the only option.

As discussed in Chapter 2, operations send to Cassandra can carry a consistency level specification. Specifying whether for this operation only ONE, QUORUM or ALL replicas should agree. We have included all three consistent options in our experiment: both read and write operations using QUORUM; write using ALL and read using ONE; and write using ONE and read using ALL. This way, it should give us insight of how this influences the throughput of operations in relation to the throughput reached by RDBMS systems.

As a summery, a list of the systems we use in the experiment with their version numbers and configurations:

- PostgreSQL 9.1.3 as Single instance RDBMS
- PostgreSQL 9.1.3 with Slony-I 2.1.1 as Master-Slave RDBMS
- PostgreSQL 9.1.3 with Bucardo 4.4.8 as Multi-Master RDBMS
- EMC Greenplum 4.2.1.0 build 3 Community edition
- Apache HBase 0.92.1 upon Apache Hadoop 1.0.1 as NoSQL without replication
- Apache Cassandra 1.0.8 as NoSQL without replication
- Apache Cassandra 1.0.8 as NoSQL with 3 replicas and read QUORUM, write QUORUM
- Apache Cassandra 1.0.8 as NoSQL with 3 replicas and read ONE, write ALL
- Apache Cassandra 1.0.8 as NoSQL with 3 replicas and read ALL, write ONE

4.2 Machines

The details of the machines we used in the experiment are important to mention to interpret the resulting absolute values correctly. To not have hardware differences influence the results we wanted machines with equal hardware. We needed more than one machine to show the expected decline of RDBMS systems and we needed a sufficient number of machines to show the incline of NoSQL systems. Our guess was to start off with four machines and see how it goes. Later on, it showed to be a choice with sufficient results, more on this in Chapter 5.

Company policy, the amount of time required to set up all systems and the time needed to perform the tests for each system prohibited further testing to more and/or real machines.

The virtual machines run on a physical machine via the KVM virtualization platform with VirtIO as storage driver[4]. The virtual machines are only used to run the database systems on. Together they make up the database server. A notebook was used as the client

[4]`http://www.linux-kvm.org`

to generate all the operations and send them to the database systems. In Table 4.1 one can see the hardware specifications of all the machines involved. In between the notebook and virtual machines there is the normal office network with only one gigabit switch to pass.

Physical machine:	CPU: 4-core, 2.4 GHz, 64 bit
	RAM: 36 GiB
	HDD: 250 GiB 15000 RPM 64MB Cache SAS 3.0 Gb/s
	NIC: 2x1 Gbit/s combined
Virtual machines (4x):	CPU: 1-core 64 bit
	RAM: 1 GiB
	HDD: 15 GiB
	NIC: VirtIO
	OS: CentOS, kernel 2.6.32-220.2.1.el6.x86_64
Notebook:	CPU: 2-core, 2.4 GHz 64 bit
	RAM: 4 GiB
	HDD: 320GiB 5400 RPM 8MB Cache SATA 1.5 Gb/s
	NIC: 1 Gbit/s
	OS: Ubuntu, kernel 3.2.18-030218-generic

Table 4.1: Specification of machines involved in the tests

4.3 Stress tool

To determine the maximum throughput of write and read operations on each different database system, we need a tool which could generate operations for each system. There are tools available for each database system separately but we also found a tool able to connect to each database system in our experiment.

For RDBMS systems there is a benchmark called TPC-C which is developed to test transactional behaviour with relational databases[11]. As mentioned earlier, NoSQL systems explicitly do not support the relational model nor transactions. The TPC-C is thus not a suitable testing tool for our purpose.

There is, however, a tool from Yahoo! which was specifically designed for generating simple operations to drive any RDBMS or NoSQL database system to the maximum possible. The tool is called Yahoo! Cloud Serving Benchmark[10] (abbreviated YCSB) and has many configuration options. For instance, one can define the number of operations to execute and whether it should do INSERT, UPDATE or SELECT operations. The data is generated randomly but is always of equal size, by default it is 10 columns wide of each 100 bytes of random alphanumeric data. We made four specific configuration choices for our experiment.

First, YCSB can be configured in how often a specific data-item is accessed in comparison to the others. For example, to simulate that certain data-items are much more popular than others (See [10] for more details). We use the Uniform distribution because in our use cases we do not have items more popular than others.

Second, the workload of a run with YCSB can be configured to be a mix of INSERT, SELECT, UPDATE, DELETE and SCAN operations. This way, one can closer simulate a real life situation. However, in our experiment we want to know how a certain type of operation performs thus we want to test each type of operation separately. Moreover, we focus in our research on read and write operations, see Chapter 1, which we translated to SELECT, INSERT and UPDATE operations.

Third, the connector for Cassandra did only support *Consistency Level ONE*. As mentioned earlier in this chapter we wanted to test the other consistency levels as well. We modified the code of YCSB to also have the consistency level configurable.

Fourth, per run of YCSB one need to define how many threads to use. After some small experiments we defined a rule of thumb of 6 threads per machine involved. That means that if we have set up Cassandra with four machines, we configured YCSB to use 4 times 6 threads at the notebook.

Together, this results in a particular set of tests from which we think we can draw conclusions to answer our research question. However, to make our results less dependent on accidental circumstances we performed each test five times. The results are then averaged to minimise the impact of accidental outliers.

4.4 Measurement tool

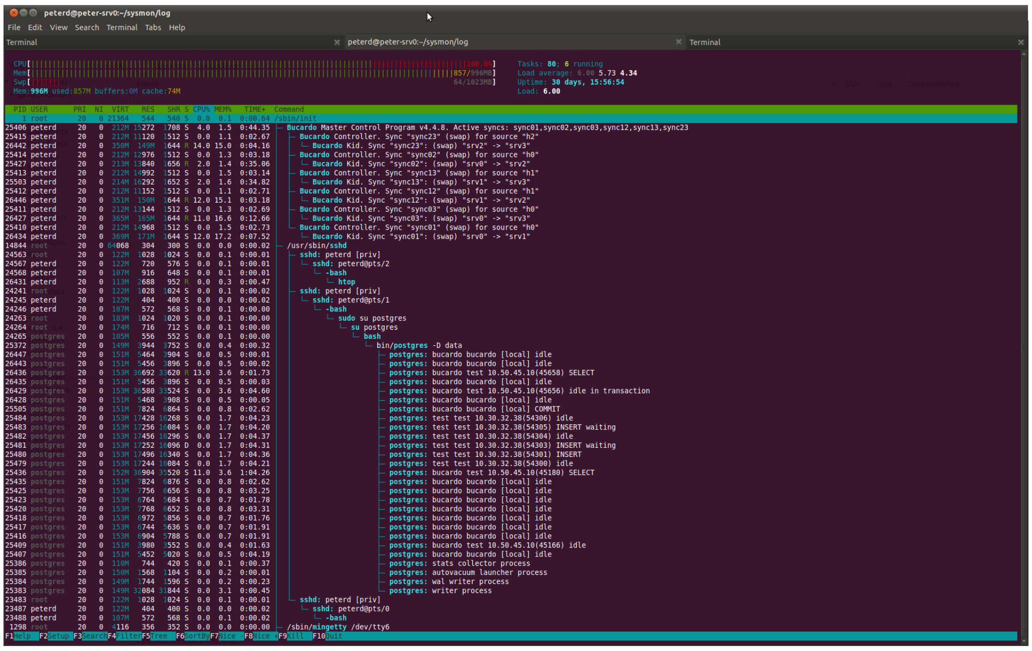
The stress tool tries to put through as many operations as possible but we need to know why it is bounded to that maximum. Is it because the notebook can not generate more operations per second? Or is it because one of the servers can not do better? Is it because the network interface is fully used? Or because the CPU is fully occupied? With the answers of these questions we can prove that the database can not reach higher throughput numbers with the supplied hardware. Next to that, we get insight in how the database systems use their resources and thus what hardware improvements are best for each system.

To address these questions, we need to monitor specific resources of each machine. Together with the throughput number reached and latency of the operations, these values tell us about the performance of the database system. We call these Key Performance Indicators (acronym KPI) and are listed below:

- Throughput: how many operations per second are processed
- Latency: time between sending the request and receiving the response at the client machine
- CPU usage: how much stress is the CPU experiencing
- IO-Wait: how much time is the CPU blocking for an IO (typically harddisk) operation to complete
- Memory usage: how much memory is used

- Swap usage: how much swap space is used
- Bandwidth in/out usage: how much is sent over the network

Most of these values can be watched with System Monitoring tools like *top*, *htop* (See Figure 4.1), *gnome-system-monitor*, *jvisualvm* or *jconsole*. Most of these applications offer a live inspection of the KPI's. We need to inspect the KPI's of four server machines and one notebook and we want to record the values to analyse it later. The live monitoring applications are thus not practical for our purposes.


peterd@peter-srv0:~/sysmon/log
File Edit View Search Terminal Tabs Help
Terminal
peterd@peter-srv0:~/sysmon/log
Terminal

```
  CPU[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]     Tasks: 80; 6 running
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||857/996MB]     Load average: 6.00 5.73 4.34
  Swp[||||||                                                                  64/1023MB]     Uptime: 30 days, 15:56:54
  Mem:996M used:857M buffers:0M cache:74M                                                     Load: 6.00

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    1 root       20   0 21364   544   540 S  0.0  0.1  0:00.64 /sbin/init
25406 peterd     20   0  212M 15272  1708 S  4.0  1.5  0:44.35 ├─ Bucardo Master Control Program v4.4.8. Active syncs: sync01,sync02,sync03,sync12,sync13,sync23
25415 peterd     20   0  212M 11120  1512 S  0.0  1.1  0:02.67 │  ├─ Bucardo Controller. Sync "sync23" (swap) for source "h2"
26442 peterd     20   0  350M  149M  1644 R 14.0 15.0  0:04.16 │  │  └─ Bucardo Kid. Sync "sync23": (swap) "srv2" -> "srv3"
25414 peterd     20   0  212M 12976  1512 S  0.0  1.3  0:03.18 │  ├─ Bucardo Controller. Sync "sync02" (swap) for source "h0"
25427 peterd     20   0  213M 13840  1656 R  2.0  1.4  0:35.06 │  │  └─ Bucardo Kid. Sync "sync02": (swap) "srv0" -> "srv2"
25413 peterd     20   0  212M 14992  1512 S  0.0  1.5  0:03.14 │  ├─ Bucardo Controller. Sync "sync13" (swap) for source "h1"
25503 peterd     20   0  214M 16292  1652 S  2.0  1.6  0:34.82 │  │  └─ Bucardo Kid. Sync "sync13": (swap) "srv1" -> "srv3"
25412 peterd     20   0  212M 11152  1512 S  0.0  1.1  0:02.71 │  ├─ Bucardo Controller. Sync "sync12" (swap) for source "h1"
26446 peterd     20   0  351M  150M  1644 R 12.0 15.1  0:03.18 │  │  └─ Bucardo Kid. Sync "sync12": (swap) "srv1" -> "srv2"
25411 peterd     20   0  212M 13144  1512 S  0.0  1.3  0:02.69 │  ├─ Bucardo Controller. Sync "sync03" (swap) for source "h0"
26427 peterd     20   0  365M  165M  1644 R 11.0 16.6  0:12.66 │  │  └─ Bucardo Kid. Sync "sync03": (swap) "srv0" -> "srv3"
25410 peterd     20   0  212M 14968  1512 S  0.0  1.5  0:02.73 │  └─ Bucardo Controller. Sync "sync01" (swap) for source "h0"
26434 peterd     20   0  369M  171M  1644 S 12.0 17.2  0:07.52 │     └─ Bucardo Kid. Sync "sync01": (swap) "srv0" -> "srv1"
14844 root       20   0 64068   304   300 S  0.0  0.0  0:00.02 ├─ /usr/sbin/sshd
24563 root       20   0  122M  1028  1024 S  0.0  0.1  0:00.01 │  ├─ sshd: peterd [priv]
24567 peterd     20   0  122M   720   576 S  0.0  0.1  0:00.01 │  │  └─ sshd: peterd@pts/2
24568 peterd     20   0  107M   916   648 S  0.0  0.1  0:00.01 │  │     └─ -bash
26431 peterd     20   0  113M  2688   952 R  0.0  0.3  0:00.47 │  │        └─ htop
24241 root       20   0  122M  1028  1024 S  0.0  0.1  0:00.02 │  ├─ sshd: peterd [priv]
24245 peterd     20   0  122M   404   400 S  0.0  0.0  0:00.02 │  │  └─ sshd: peterd@pts/1
24246 peterd     20   0  107M   572   568 S  0.0  0.1  0:00.00 │  │     └─ -bash
24263 root       20   0  183M  1024  1020 S  0.0  0.1  0:00.00 │  │        └─ sudo su postgres
24264 root       20   0  174M   716   712 S  0.0  0.1  0:00.00 │  │           └─ su postgres
24265 postgres   20   0  105M   556   552 S  0.0  0.1  0:00.00 │  │              └─ bash
25372 postgres   20   0  149M  3944  3752 S  0.0  0.4  0:00.32 │  │                 └─ bin/postgres -D data
26447 postgres   20   0  151M  5464  3904 S  0.0  0.5  0:00.01 │  │                    ├─ postgres: bucardo bucardo [local] idle
26443 postgres   20   0  151M  5456  3896 S  0.0  0.5  0:00.02 │  │                    ├─ postgres: bucardo bucardo [local] idle
26436 postgres   20   0  153M 36692 33620 R 13.0  3.6  0:01.73 │  │                    ├─ postgres: bucardo test 10.50.45.10(45658) SELECT
26435 postgres   20   0  151M  5456  3896 S  0.0  0.5  0:00.03 │  │                    ├─ postgres: bucardo bucardo [local] idle
26429 postgres   20   0  153M 36580 33524 S  0.0  3.6  0:04.60 │  │                    ├─ postgres: bucardo test 10.50.45.10(45656) idle in transaction
26428 postgres   20   0  151M  5468  3908 S  0.0  0.5  0:00.05 │  │                    ├─ postgres: bucardo bucardo [local] idle
25505 postgres   20   0  151M  7824  6864 S  0.0  0.8  0:02.62 │  │                    ├─ postgres: bucardo bucardo [local] COMMIT
25484 postgres   20   0  153M 17428 16268 S  0.0  1.7  0:04.23 │  │                    ├─ postgres: test test 10.30.32.38(54306) idle
25483 postgres   20   0  153M 17256 16084 S  0.0  1.7  0:04.20 │  │                    ├─ postgres: test test 10.30.32.38(54305) INSERT waiting
25482 postgres   20   0  153M 17456 16296 S  0.0  1.7  0:04.37 │  │                    ├─ postgres: test test 10.30.32.38(54304) idle
25481 postgres   20   0  153M 17252 16096 D  0.0  1.7  0:04.31 │  │                    ├─ postgres: test test 10.30.32.38(54303) INSERT waiting
25480 postgres   20   0  153M 17496 16340 S  0.0  1.7  0:04.36 │  │                    ├─ postgres: test test 10.30.32.38(54301) INSERT
25479 postgres   20   0  153M 17244 16084 S  0.0  1.7  0:04.21 │  │                    ├─ postgres: test test 10.30.32.38(54300) idle
25436 postgres   20   0  152M 36904 35520 S 11.0  3.6  1:04.26 │  │                    ├─ postgres: bucardo test 10.50.45.10(45180) SELECT
25435 postgres   20   0  151M  7824  6876 S  0.0  0.8  0:02.62 │  │                    ├─ postgres: bucardo bucardo [local] idle
25425 postgres   20   0  153M  7756  6656 S  0.0  0.8  0:03.25 │  │                    ├─ postgres: bucardo bucardo [local] idle
25423 postgres   20   0  153M  6764  5684 S  0.0  0.7  0:01.78 │  │                    ├─ postgres: bucardo bucardo [local] idle
25420 postgres   20   0  153M  7768  6652 S  0.0  0.8  0:03.31 │  │                    ├─ postgres: bucardo bucardo [local] idle
25418 postgres   20   0  153M  6972  5856 S  0.0  0.7  0:01.76 │  │                    ├─ postgres: bucardo bucardo [local] idle
25417 postgres   20   0  153M  6744  5636 S  0.0  0.7  0:01.91 │  │                    ├─ postgres: bucardo bucardo [local] idle
25416 postgres   20   0  153M  6904  5788 S  0.0  0.7  0:01.91 │  │                    ├─ postgres: bucardo bucardo [local] idle
25409 postgres   20   0  151M  3980  3552 S  0.0  0.4  0:01.63 │  │                    ├─ postgres: bucardo test 10.50.45.10(45166) idle
25407 postgres   20   0  151M  5452  5020 S  0.0  0.5  0:04.19 │  │                    ├─ postgres: bucardo bucardo [local] idle
25386 postgres   20   0  110M   744   420 S  0.0  0.1  0:00.37 │  │                    ├─ postgres: stats collector process
25385 postgres   20   0  150M  1568  1104 S  0.0  0.2  0:00.01 │  │                    ├─ postgres: autovacuum launcher process
25384 postgres   20   0  149M  1744  1596 S  0.0  0.2  0:00.23 │  │                    ├─ postgres: wal writer process
25383 postgres   20   0  149M 32084 31844 S  0.0  3.1  0:00.45 │  │                    └─ postgres: writer process
23483 root       20   0  122M  1028  1024 S  0.0  0.1  0:00.01 │  └─ sshd: peterd [priv]
23487 peterd     20   0  122M   404   400 S  0.0  0.0  0:00.02 │     └─ sshd: peterd@pts/0
23488 peterd     20   0  107M   572   568 S  0.0  0.1  0:00.02 │        └─ -bash
 1298 root       20   0  4116   356   352 S  0.0  0.0  0:00.00 ├─ /sbin/mingetty /dev/tty6
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 4.1: screenshot of htop, live monitoring of cpu and memory usage

Next to that, the application measuring the values at a machine is also using the resources thus influences the measurement (called *in-band* measuring[39]). This need to be avoided or at least brought to a minimum. Otherwise, it could influence the values in a way that we conclude the wrong things.

There is a product, called YCSB++[36], that combines the recording of most of the KPI's we listed with the execution of a test with YCSB. It is based on an old version of YCSB with little support for the versions of database systems we use and merging it with the current version of YCSB seemed impossible because YCSB had undergo some big re-factoring cycles. Most troublesome important however is that it sends the values for the KPI's during execution to a Apache Zookeeper cluster (see Section 2.2.5). We thus had to arrange more machines to run a separate Zookeeper cluster on and then still it had much processing while it did in-band measuring. We found this too much overhead for our relatively small cluster we wanted to experiment with.

To meet all our requirements, we made the *Measurement Tool* ourselves[5]. The tool collects every second the values for all the KPI's from the `/proc` filesystem and writes them to a file via the Log4J library. The Log4J library makes sure that the tool is not constantly writing to disk but also not keeping too much data in memory. At the end of the execution of a test, the measurement tool at all the machines were stopped by the notebook. The notebook then collects all the files of each machine and processes them to a file for each KPI with the values for all machines in a separate column, all part of the *Measurement Tool*. With using the chart features of *Microsoft Excel*, *OpenOffice Calc* or *R*[32], a chart displaying the KPI values over time with each machine represented by a different line could provide us insight into our results.

4.5 Summary

In this chapter we described how we set up our experiment. The *Measurement Tool* will start recording at all *Machines*, the *Stress Tool* will generate operations to send to the database *Systems* running on all *Machines*.

The type of tests that will be executed is defined in the configuration of the *Stress Tool*, the input variables of the experiment. Most important input variables of the experiment are the database system which is tested and the type of operation that is executed. Each test is executed five times to average the resulting throughput rates. The values of the KPI's are collected by the *Measurement Tool* and these values are the output of the experiment. In one overview, the experiment could be defined by Equation 4.1.

The output values are discussed in the next Chapter and conclusions are drawn in Chapter 9.

$$\left\{\begin{array}{l}\text{PostgreSQL Single}\\ \text{PostgreSQL Master-Slave}\\ \text{PostgreSQL Multi-Master}\\ \text{EMC Greenplum}\\ \text{HBase}\\ \text{Cassandra No-Replication}\\ \text{Cass 3-Rep, Read, Write Quorum}\\ \text{Cass 3-Rep, Read One, Write All}\\ \text{Cass 3-Rep, Read All, Write One}\end{array}\right\} \times \left\{\begin{array}{c}\texttt{INSERT}\\ \texttt{UPDATE}\\ \texttt{SELECT}\end{array}\right\} \times 5 \quad \Rightarrow \quad \left\{\begin{array}{l}\text{Throughput}\\ \text{Latency}\\ \text{CPU usage}\\ \text{IO-Wait}\\ \text{Memory usage}\\ \text{Swap usage}\\ \text{In Bandwith}\\ \text{Out Bandwidth}\end{array}\right\} \tag{4.1}$$

[5]The Measurement Tool can be found at: `http://pcdijkshoorn.nl/sysmon`

Chapter 5

Results

As can be derived from our Equation at the end of the former Chapter, our experiment generates quite some data. For each database system and each type of operation we ran five tests. From such a set of five tests, the average of operation throughput is calculated. This value is then less influenced by accidental processes running in the background, disturbing network usage or other variants we were not able to isolate our experiment from. These values will be plotted for each database system to find how these are related between the RDBMS and NoSQL database families. The average operation throughput numbers are important to answer our main research question:

> **Main Research Question**
>
> *„How do RDBMS databases relate to wide-column NoSQL databases in terms of maximum read and write operation throughput rate in a scaling-out context?"*

To prove that the test indeed did reach a maximum value, the other KPI's, such as CPU usage, are recorded. If the results show that the CPU is used at its maximum during the processing of operations at a certain throughput rate then we have reason to believe that the database can not reach higher throughput rates. The measurement of an exhausted resource indicates that the throughput rate was at a maximum for that machine.

When measuring the KPI's of multiple machines when running a distributed system with different roles for some machines (eg. client role or master role), we could have an indication with resource of which role is bounding the maximum throughput rate of the distributed system as a whole. We also discussed this in Section 4.4. In this chapter we list the bounding resource with each measured maximum throughput rate. We call this the *'bounded by'* value, as the throughput seems to be bounded by the listed resource.

All the KPI's are collected by the *measurement tool* (See Section 4.4) which generates a file per KPI with a column for each machine. Such files can be imported into applications like *Microsoft Excel*, *OpenOffice Calc* or *R*[32] to draw charts of it.

As an example for the charts we made, we will now describe a set of charts generated from an `INSERT` operation test at a single instance of PostgreSQL. Figure 5.1 shows these charts. Each chart displays the value of a specific KPI over time for each machine involved in the experiment. The PostgreSQL single machine configuration is installed at `srv0` and the test is executed from the `client`. This can also be seen from the bandwidth charts, the incoming bandwidth is high for `srv0`(Figure 5.1e) and the outgoing bandwidth is high for the `client` machine (Figure 5.1f). The other charts of this test execution show that both memory and swap usage is relatively low, that CPU usage at the `client` is about half and minimal at `srv0`. The IO-wait usage, however, is nearly its maximum for the `srv0` machine. This backs the assumption that the database system has reached its maximum throughput rate for that operation with the current hardware configuration. With the knowledge that the reached throughput rate is a maximum, we can compare each system with each other and answer our main research question.

In this chapter we will discuss the average maximum throughput rate of RDBMS and NoSQL systems separately. Per family, we discuss the throughput rates per operation type, thus INSERT, UPDATE and SELECT separately. At the end of this chapter we relate the maximum throughput rates of the INSERT and UPDATE operations to the maximum throughput rates of the SELECT operations. In Chapter 9, conclusions are drawn from the data that is presented here.

5.1 RDBMS systems

To define maximum operation throughput numbers for a RDBMS system and a few scaling-out configurations, the PostgreSQL system is used. PostgreSQL is set up to run as a single server, in a master-slave configuration and in a multi-master configuration.

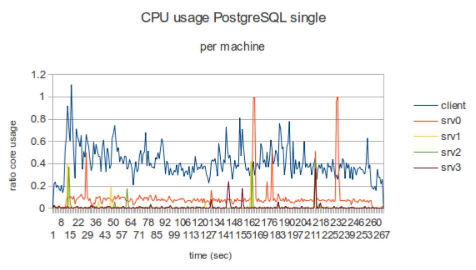
The single server PostgreSQL system is tested twice. First, it is tested with enough data to keep the system busy for a few minutes. Second, it is tested with much more data because there was the suspicion that the RDBMS system would show different results when the data could no longer be kept in fast accessible memory (1 Gigabyte, see Section 4.2). We took 2 Gigabytes of data for the *postgres_single_big* case.

The Multi-Master configuration is also tested in two ways. First, with two servers and second, with four servers. This is done because the four server results were so surprising that we wanted a measurement point in between.

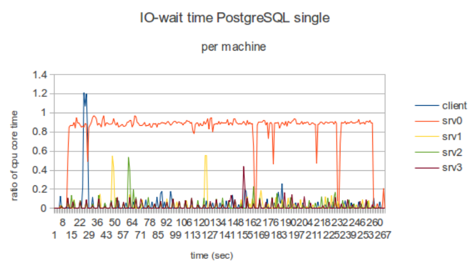
The results are described for each operation separately:

5.1.1 INSERT operations

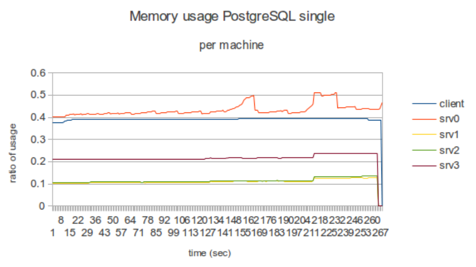
The INSERT operation throughput rates of the RDBMS configurations show the expected behaviour. From Gray[24] we know that RDBMS systems do not scale-out well when it comes to write operations (as discussed in Section 2.1). In Table 5.1 we see that the single machine PostgreSQL set up has the highest throughput rate, 414.3. The four-machines master-slave configuration gets to 280.8 operations per second. The multi-master configurations go from 108.6 (two-machines) to 47.3 (four-machines) operations per second. The last column shows what resource is most, if not fully, used at the time the system processes operations at the determined operation throughput rate.



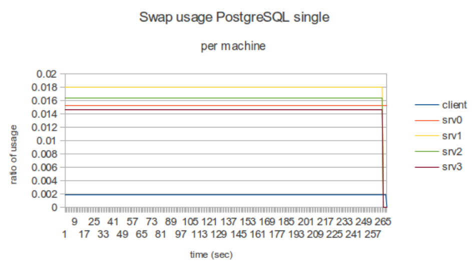
(a) CPU usage



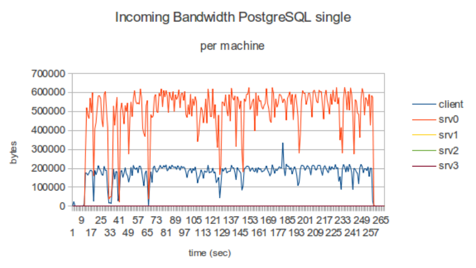
(b) IO-wait time



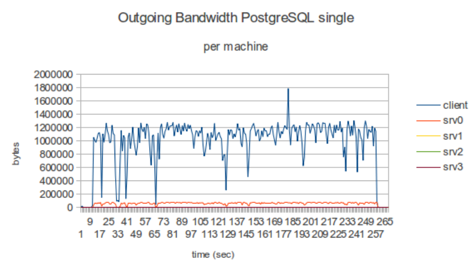
(c) Memory usage



(d) Swap usage



(e) Incoming Bandwidth



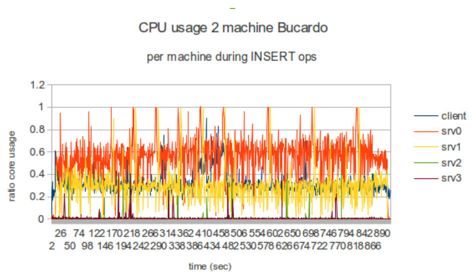
(f) Outgoing Bandwidth

Figure 5.1: Typical graphs displaying the resource usage during an `INSERT` test execution on PostgreSQL single instance. PostgreSQL is running at `srv0` and the client generates the operations

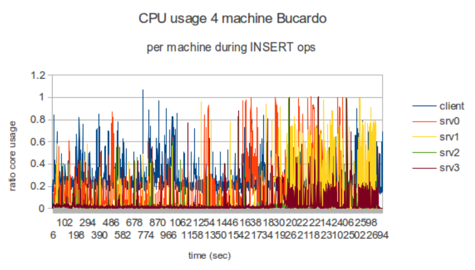
	Throughput Rate	(bounded by)
postgres_single	414.3	(IO-wait)
postgres_single_big	412.5	(IO-wait)
postgres_slony	280.8	(IO-wait master)
postgres_bucardo_2machines	108.6	(IO-wait and CPU servers)
postgres_bucardo_4machines	47.3	(IO-wait)
greenplum	23.9	(IO-wait)

Table 5.1: Average maximum INSERT operation throughput rate of RDBMS systems.

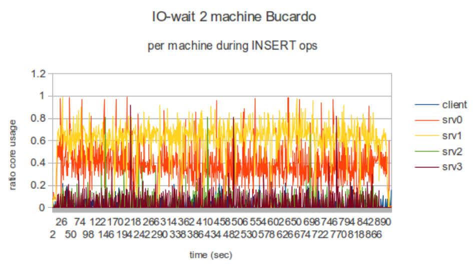
The multi-master configurations are interesting to have a closer look at. In Figure 5.2 we see that the two-machine configuration has a higher CPU usage value than the four-machine configuration. We see that the more machines are added the more the system is waiting for IO to complete.



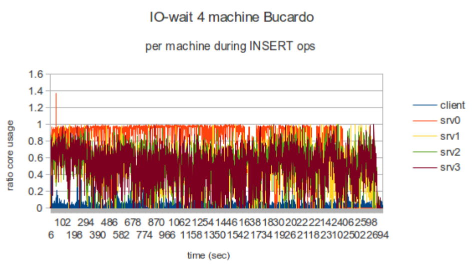
(a) CPU 2-machines



(b) CPU 4-machines



(c) IO-wait 2-machines



(d) IO-wait 4-machines

Figure 5.2: Charts displaying the CPU usage and IO-wait time during an `INSERT` test execution on PostgreSQL+Bucardo on a 2- and 4-machines instances. The 2-machine configuration performance is more dependent on the CPU than the 4-machine configuration.

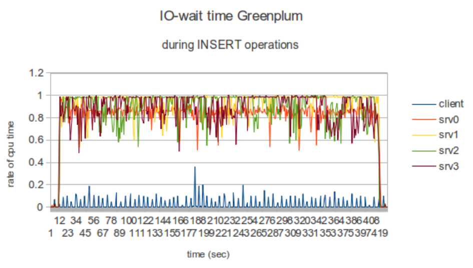


Figure 5.3: IO-wait per machine during INSERT operations with Greenplum

The throughput value of the Greenplum system was most disappointing, only 23.9 operations per second. The architecture suggests a master server which only works as a relay to the data machines, the master itself would not write the data on its own hard disk. The KPI graphs show a different picture, see Figure 5.3, the IO-wait of the master server is at around 80% during the INSERT operations. The master server is thus apparently heavily relying on hard disk access and the overall database can not benefit from the architecture to reach higher throughput rates.

5.1.2 UPDATE operations

The UPDATE operation is slightly different compared to an INSERT operation. It is about changing a value of an already known data-item. The RDBMS system reflects this difference in decreased throughput numbers. Table 5.2 shows lower values than the ones in Table 5.1. This could be explained by the way the data is stored in a RDBMS. The system first needs to look up the location of the data-item and then overwrite the value in-place. However, we do not claim to know exactly why this is, as it is outside the scope of this research.

Table 5.2 also shows the same decreasing throughput trend when more machines are involved as we saw with the INSERT operation throughput numbers.

While all tests clearly showed a resource going to a maximum usage, in the tests with Greenplum we did not find one. Why this system only reached a 8.9 maximum throughput for UPDATE operations remained a mystery for us. The results of Greenplum were not promising enough to start a research into it.

	Throughput Rate	(bounded by)
postgres_single	321.5	(IO-wait)
postgres_single_big	95.2	(IO-wait)
postgres_slony	70.9	(IO-wait all servers)
postgres_bucardo_2machines	66.1	(IO-wait and CPU servers)
postgres_bucardo_4machines	35.0	(IO-wait)
greenplum	8.9	(?)

Table 5.2: Average maximum UPDATE operation throughput rate of RDBMS systems.

5.1.3 SELECT operations

When a RBDMS system receives a write operation, it writes the new data to disk before responding to the client to ensure durability (See Chapter 2). This made the IO-wait time used to its maximum when executing write operations as seen in the former sections. With SELECT or read operations, the RDBMS system, in this case PostgreSQL, uses memory cache to retrieve the data from. It kept a copy of at least part of the data in memory while it also wrote it to disk. The maximum operation throughput rate for SELECT operations is thus much higher, see Table 5.3.

Also, with the scaling-out configurations all machines have a local-copy of the data. All machines can process SELECT operations independent of the other machines. In our tests the CPU of the client machine could not generate more operations than the scaling-out configurations could handle. A maximum value was thus not defined. This shows the importance of recording all the KPI's at all machines when evaluating distributed systems.

	Throughput Rate	(bounded by)
postgres_single	5434.1	(CPU)
postgres_single_big	275.8	(IO-wait)
postgres_slony	6626.5	(CPU client)
postgres_bucardo_2machines	6643.2	(CPU client)
postgres_bucardo_4machines	6683.8	(CPU client)
greenplum	88.0	(CPU)

Table 5.3: Average maximum SELECT operation throughput rate of RDBMS systems.

To be able to generate even more load, one can, again, *scale-up* or *scale-out* the stress tool. We had only this specific machine to use as a client and could not do either actions to be able to generate more operations per second. Also, as discussed, YCSB does not support this yet, for this reason YCSB++ was developed. YCSB++, however, seems to be build for much larger tests, already needing multiple machines for the simplest cases, see Section 4.3. For our purposes, it is not even needed. In our conclusions we use the fact that this value is larger than another value and a proper measurement of this value will only make this value larger. Our conclusions will be stronger, not falsified, by a proper determination.

Back to Table 5.3, the single machine RDBMS configuration shows a strong decline in maximum operation throughput rate when it has much more data in store. Such a configuration was loaded with about two gigabytes of data while the machine has only one gigabyte of RAM. The tests were re-executed and this resulted in a much lower throughput rate. Instead of the CPU, which is fully used in the tests with less data, the IO-wait is now fully used. That reflects that the data has to be read from disk. These two values are interesting to compare with the ones of the NoSQL systems later on, which are assumed to also read from disk, we discuss this later on.

5.1.4 Remarks

A significant detail to note among the RDBMS configurations, is that both master-slaves and multi-masters configurations use asynchronous replication. This means that not all replicas are equal when the client is already told that the write is successfully committed. Due to this fact, two (or more) versions can exist in the system. By this design, it cannot guarantee consistency. Consistency could be guaranteed when a synchronous method was used for replication. However, such a system is hardly used in industry as the consensus is that it imposes even more computational overhead to the system than the asynchronous method[24]. This in contrast with the NoSQL systems which do synchronous replication and still show a reasonable performance, as we will see in the next section.

5.2 NoSQL systems

The results in the former section describe how the throughput rate of write operations on a RDBMS system decreases as more machines are involved. This is as expected but with absolute numbers we could compare how other systems perform compared to the RDBMS systems. In our research we compare two NoSQL systems with the RDBMS systems. In this section we describe the results.

In the Experiment Description (Chapter 4) we already mentioned and explained our choice for HBase and Cassandra to include in the tests. While both are reported to work well with tens or hundreds of machines, in our tests both systems have to work with only one to four machines. HBase is only tested in one configuration, with four machines and no replication. Cassandra is tested in five configurations, one with one machine and the others with four machines but different replication and consistency settings. There is one configuration with four machines and no replication. The other configurations have four machines and keep three replicas of each data item. The different consistency configurations are tested to use either QUORUM, read ALL/write ONE or read ONE/write ALL, as described in Chapter 2 and 4.

To compare the results of the RDBMS with the NoSQL systems we have to compare configurations which are most equal to each other. That means we will compare the results of the single machine Cassandra configuration with the single machine RDBMS configurations (the small dataset and big dataset). Likewise, we can compare the four machine configurations of HBase and Cassandra with the four machine multi-master RDBMS configuration.

Let us now turn to the results, for each operation separately:

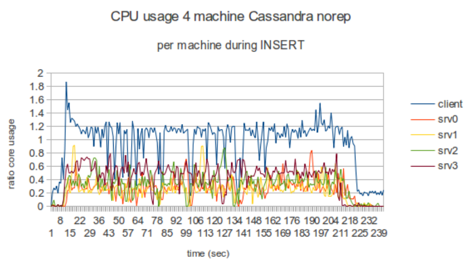
5.2.1 INSERT operations

The NoSQL systems differ from RDBMS systems in the way the write operations (INSERT and UPDATE) are processed. We discussed the details in Chapter 2 but in short, RDBMS systems write to disk and NoSQL to memory. The found throughput rates also reflect this as can be seen in Table 5.4. Recall that the highest INSERT operations throughput RDBMS systems reached in our tests is 414.3 INSERT operations per second. That was on a single machine RDBMS configuration. The Cassandra single machine reached a throughput rate of 3884.8 INSERT operations per second, which illustrates the difference in processing these operations.

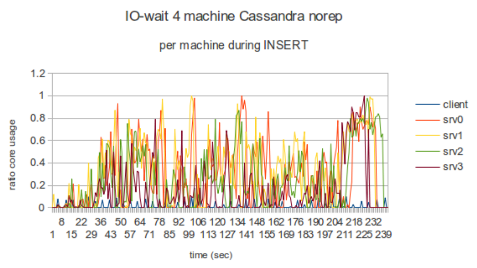
	Throughput Rate	(bounded by)
hbase	3797.8	(CPU and IO-wait)
cassandra_norep_1machine	3884.8	(CPU)
cassandra_norep_4machines	4671.8	(CPU and IO-wait and Client CPU)
cassandra_3rep_consistent_quorum	2222.4	(CPU and IO-wait)
cassandra_3rep_consistent_rall_wone	2605.7	(CPU and IO-wait)
cassandra_3rep_consistent_wall_rone	1928.3	(CPU and IO-wait)

Table 5.4: Average maximum INSERT operation throughput rate of NoSQL systems.

The difference in processing is also reflected in the resource usage during the tests, take for example the 4 machine Cassandra configuration, the CPU usage and IO-wait graphs during the INSERT test are displayed in Figure 5.4. With the RDBMS configurations we mostly saw the IO-wait time as the limiting factor. With the NoSQL systems we see a mix of CPU usage and IO-wait time as the limiting factor. However, we also see that there is a strong indication that the CPU of the client machine is also bounding further progression of the throughput rate. The overall picture shows us that we are close to the maximum throughput rate but not yet at it.



(a) CPU



(b) IO-wait

Figure 5.4: Charts displaying the CPU usage and IO-wait time during an `INSERT` test execution on a 4 machine Cassandra configuration without replication. Both resources seem to mix towards.

With the RDBMS configurations, we saw a decline in throughput of INSERT operations when more machines are involved. When we look at the results of the single machine and four machine Cassandra configurations we see an incline, from 3884.8 to 4671.8. This is of course exactly the point why the NoSQL systems were developed, to be able to scale-out to increase the write operations throughput.

In our experiment we also included Cassandra configurations with replication and tested different consistency schemes. Table 5.4 also shows the impact of these schemes to the throughput rate. The *cassandra_3rep_consistent_wall_rone* configuration, write ALL and read ONE, comes to a lower throughput number than the configuration with the QUORUM scheme. The *cassandra_3rep_consistent_rall_wone* configuration, read ALL and write ONE, shows a higher throughput rate than the *cassandra_3rep_consistent_quorum* configuration. This is expected as the write operation only needs to finish on one machine while with the QUORUM scheme it should finish on two machines, as we learned in Chapter 2. If data should only be written on one machine then it could suggest that it should perform as fast a no-replication configuration. However, it does not, Cassandra sends each machine holding a replica of the data the write operation but it returns to the client as soon as the number of machines defined by the scheme have processed the write operation. The process of initially informing all the replicas should thus be accounted for the lower throughput rate than the 4 machine, no-replication configuration.

The HBase configuration, with four machines, reaches a bit lower throughput rate than the one-machine Cassandra configuration. The next section about the UPDATE operations shows a bigger difference.

5.2.2 UPDATE operations

An UPDATE operation differs from an INSERT operation in that it is about changing an already known data-item. We already mentioned this in the discussion of the UPDATE operation throughput numbers of the RDBMS systems. With the RDBMS systems we saw lower UPDATE operation throughput numbers than INSERT operation throughput numbers. With the NoSQL systems it is the other way around. Table 5.5 shows the UPDATE operation throughput numbers of the NoSQL configurations. The UPDATE operation throughput rates are almost twice the INSERT operation throughput rates. For HBase it is even a tenfold increase.

	Throughput Rate	(bounded by)
hbase	30551.6	(CPU client)
cassandra_norep_1machine	7561.4	(CPU client and server)
cassandra_norep_4machines	9157.2	(CPU client)
cassandra_3rep_consistent_quorum	6601.9	(CPU client and servers)
cassandra_3rep_consistent_rall_wone	7173.2	(CPU client and servers)
cassandra_3rep_consistent_wall_rone	4652.3	(CPU client and servers)

Table 5.5: Average maximum UPDATE operation throughput rate of NoSQL systems.

A possible explanation for this behaviour differs per product. HBase has a client connector which is highly coupled with the internals of HBase, it could be that it caches the UPDATE operations locally and only propagate them in groups to the HBase machines, thereby gaining speed. The behaviour of Cassandra could be explained with that an UPDATE operation does not change any indexes. With an INSERT it should change the indexes to keep track of this data-item, while for an UPDATE these indexes are already set.

Important to note is that each of the Cassandra configurations seems to behave in the same relation to each other as they did with the INSERT operations.

For our research it is important to note that with RDBMS systems UPDATE operation throughput rate is lower than INSERT operation throughput; and with NoSQL systems the UPDATE operation throughput rate is higher than the INSERT operation throughput rate.

However, the throughput numbers of Table 5.5 were recorded while also the client runs out of resources. This shows us that the reached throughput number are not the maximum throughput numbers that could be reached on the machines. Although, we continue to use these results as further increasing of these throughput rates make our conclusions even stronger (See Chapter 9).

5.2.3 SELECT operations

We now turn to the SELECT operations throughput rate on NoSQL systems. For SELECT operations on RDBMS we found a throughput rate of 5434.1 (See Table 5.3) on a single machine. The numbers NoSQL systems achieved in our tests are much lower, a comparable configuration (Cassandra, single machine) only got to 430.7 SELECT operations per second, visible in Table 5.6.

	Throughput Rate	(bounded by)
hbase	198.2	(IO-wait)
cassandra_norep_1machine	430.7	(IO-wait)
cassandra_norep_4machines	516.7	(IO-wait)
cassandra_3rep_consistent_quorum	138.2	(IO-wait)
cassandra_3rep_consistent_rall_wone	98.6	(IO-wait)
cassandra_3rep_consistent_wall_rone	224.8	(IO-wait)

Table 5.6: Average maximum SELECT operation throughput rate of NoSQL systems.

The lower throughput can be explained by the internals described in Chapter 2 and by the resource that seems to limit the throughput rate. With the NoSQL systems, we see that read operations highly depend on the IO-wait time, which indicates that the data has to be read from disk. This is comparable with the tests on RDBMS with more data than could fit in memory. The RDBMS with more data than could fit in memory also had a much lower throughput rate, namely 275.8.

It is interesting to see that the single machine NoSQL system did reach a bit higher throughput rate than its RDBMS-big-dataset counterpart. With the knowledge that data is stored much less structured in NoSQL than with RDBMS systems (the SSTable format), this could indicate that the indexing mechanism are more advanced in NoSQL systems than in the RDBMS systems.

The differences among each NoSQL configuration also show some interesting details. The HBase configuration was set up with four machines but shows a much lower throughput rate than a single machine Cassandra configuration.

The four machines Cassandra configurations show that having replicas drops the throughput rate considerably, from 516.7 to 138.2. Among the configuration with replication we see a record of the impact of the different consistency schemes. The scheme with read ONE (last row) has indeed a higher throughput rate than the read QUORUM, which again has a higher throughput rate than the read ALL scheme.

Having replicas in the system seems to have a great impact on the throughput rate. With the INSERT and UPDATE operations we already saw that, and now with the SELECT operations it again shows a strong degradation of performance. This is remarkable because only for consistency check all the replicas could be queried but in the read ONE scheme that is not needed. Why the throughput rate for SELECT operations in a replication configuration with read ONE scheme is lower than with the configuration without replication remains unknown to us.

5.2.4 Remarks

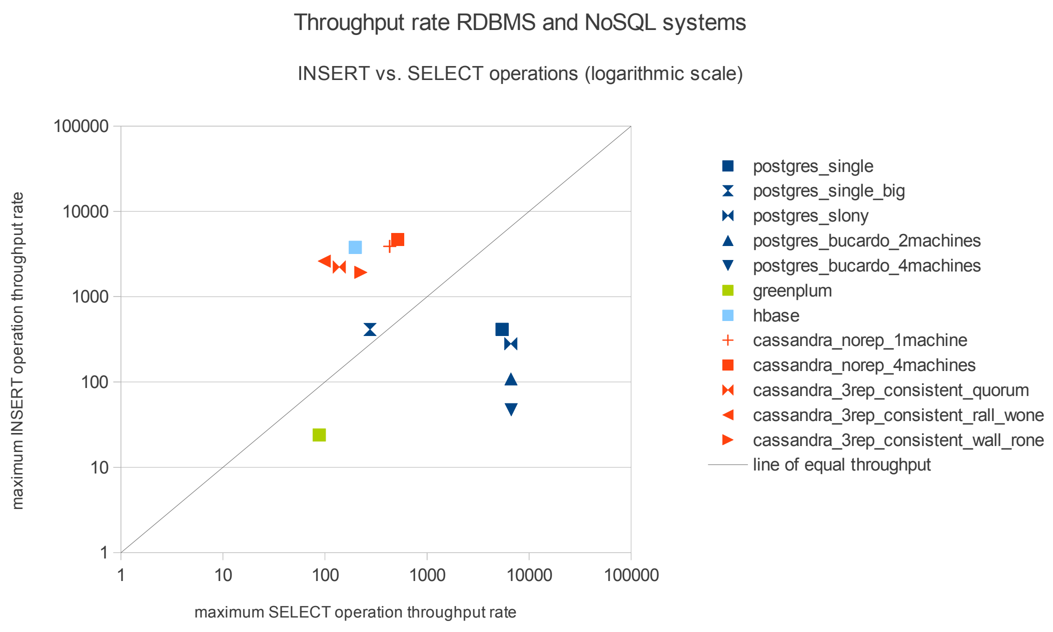
The differences within the selection of NoSQL configurations are an important contribution of this research. To our knowledge no research exist in which the impact of these different replication schemes in Cassandra are compared. As discussed in Chapter 4, we modified the YCSB stress tool to support these kind of tests, and we believe that replication is a key-factor in a production environment because it is the only mechanism NoSQL systems use to ensure durability of data, which means that the latest written data would survive typical system crashes.

5.3 Overview

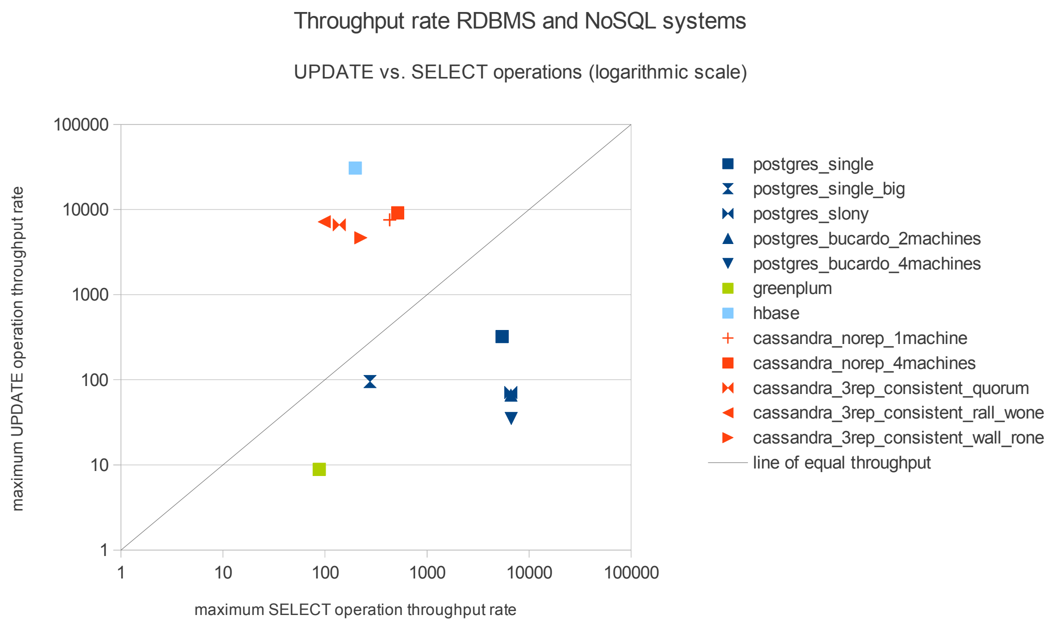
In this Chapter we showed the throughput rate values we measured for each configuration under a load of three types of operations. Two of these operations, INSERT and UPDATE, involve writing data and the other, SELECT, involves reading data. We showed the results in tables and discussed them in the text. To have a more practical overview of how the configurations relate to each other, we plotted the results into two graphs. Both have the throughput rate of the SELECT operations on the x-axis but on the y-axis the throughput rate of respectively INSERT operations and UPDATE operations is put. This makes two graphs where each configuration can be drawn with a dot to indicate where in the performance space it is.

Figure 5.5 shows such plots. Please note that the plots use a logarithmic scale. There is a diagonal plotted to help to interpret the plots. What we see is that both families, RDBMS and NoSQL, have their own specific operation in which they reach the highest operation throughput rate. The RDBMS seems optimised for read operation throughput and NoSQL for write operation throughput.

The discussion of these plots belongs to the Conclusions, Chapter 9.



(a)



(b)

Figure 5.5: Plot of each configuration with their maximum `SELECT` operation throughput rate against their maximum `INSERT` or `UPDATE` operation throughput rate.

Chapter 6

Related Work

Our research is a comparison of specific configurations from the RDBMS and NoSQL database families, in this chapter we discuss results from related research papers. Our research compared the configurations based on the maximum throughput rate of three types of operations, `INSERT`, `UPDATE` and `SELECT` operations. We found a research paper which also describes a performance measurement of similar systems, although they used older versions. Also, two papers focussed on flexible schema databases are discussed. They both conclude that the proposed solutions come with a major performance drop. The following sections describe both types of related research papers.

6.1 Throughput rates

The paper of Cooper et al., which also introduced the YCSB tool (See Chapter 4), discusses the throughput rates of combined workloads at various systems[10].

They put MySQL, HBase, Cassandra and PNUTS under test and defined how the latency changed when the throughput increases in a controlled manner. The throughput rates in their paper are for a combined set of operations. For example, a test consists of 50% SELECT and 50% UPDATE operations. The throughput rates reported for such a workload are 11978 operations/sec for Cassandra, 7904 operations/sec for HBase, 7283 operations/sec for sharded MySQL and 7448 operations/sec for PNUTS.

There are a few difference with our experiment set up. First, they use earlier versions of HBase and Cassandra. Second, their research uses six machines with better hardware than in our experiment. Third, the combined workload could highly skew the combined throughput rates, which makes it hard to compare to our research.

As Future Research Cooper et al. mention research to the costs and benefits of replication in terms of performance should be investigated. In our research we included the Cassandra configurations with replication and different consistency schemes, with which we have built into the direction which Cooper et al. showed us.

6.2 Flexible schema databases

Research into multi-tenancy within the Software as a Service (SaaS) domain has an angle in finding a database with a flexible schema. A flexible schema means that the schema can easily be *extended* and *modified* while the system is online[3]. This is useful for multi-tenancy systems which often need only a slightly different schema for each tenant.

Solutions for this problem range from making a new table for each tenant to extending tables, to XML usage[3] . XML is a format in which the format is defined implicitly by using named tags, this way providing a flexible schema. We found two papers, Aulbach et al.[3] and Fang et al.[17], which compare several flexible schema solutions.

Aulbach et al.[3] tests Microsoft SQL Server 2008 (RDBMS product), pureXML in IBM DB2 and HBase (NoSQL product). They selected a few implementations which are more suitable to implement a flexible schema but discover that the performance loss in relation to the conventional solution is considerably high. They mention that the ideal database would, among other things, need to do replication by itself and not rely on a distributed file-system. Their conclusion is that „the ideal database system for SaaS has not yet been developed"[3].

In our research we think we have build into this direction by comparing HBase and Cassandra, the latter does not have a distributed file system (see Chapter 2).

Fang et al.[17] did a similar test but than with a Native XML Database, namely Berkeley DB XML. They compare their results to Aulbach et al. to discover that the Berkeley DB XML is also not an ideal database for flexible schema in SaaS because of a major performance difference with the conventional solution, the RDBMS[17].

6.3 Remarks

With respect to these two related research topics we think we have built into a direction showed by them and returned valuable insight into the throughput differences of NoSQL systems compared to RDBMS systems.

Chapter 7

Threats to Validity

The results of the experiment look promising enough to us to draw useful conclusions from them, however, some notes should be taken into consideration. This chapter describes properties of our experiment which we see as threats to the validity of our results. A subsequent research into this direction might use this section to design their experiment, as we did not see or did not have the ability to design them into ours.

7.1 Virtual Machines

The databases were installed on virtual machines running on the same real machine. This means that the virtual machines had to share the same real hardware. This could lead to unpredictable behaviour as multiple virtual machines want to access the same real hardware resource, a kind of racing behaviour could show up.

However the sure impact, we do not see that the sign of the growth rate when adding machines changes if real machines were used. This means that we expect that the absolute results could be shifted, but relative to each other they will be the same.

7.2 Number of Machines

In our experiment we had access to four machines to deploy the databases on, to exactly measure the size of the gap we needed more machines. Currently, we have the idea that around ten machines running a Cassandra database would equal the read operation throughput rate of a single machine RDBMS database but that is an idea. Because it makes a difference if it would be six machines or sixteen, it would be very useful if we could have determined that number exactly.

7.3 Client as Limiting Factor

The maximum read operation throughput number of the RDBMS databases with the master-slave and multi-master configurations were not determined because the Client machine running the Stress tool could not generate more operations (the CPU of the client was fully stressed). A proper determination of this maximum value was the idea in the beginning and it still would be good for the overall picture to have that value. This can be achieved by (developing and) using a distributed Stress tool.

However, the strong decreasing maximum write operation throughput number of the RDBMS when scaling-out also shows that real maximum read operation throughput number would not make any difference to our conclusions.

7.4 Low Machine Specifications

The machine specifications were a variable we could not change and we saw that especially Apache HBase had difficulties to get running with these, relatively, low specifications. We could easily say that HBase was not designed to run in this limited environment, not only with such a low number of machines (which was our goal to determine), but also not with such low hardware specifications. This could have influenced our experiment much. It could very well be that the throughput numbers would relate much different to each other if neither system would have to fit into specifications it was not designed for.

7.5 Only Measured for a Short Period

Another consideration which should be kept in mind with the conclusions, is that the experiment only drove the databases to maximum throughput for a short period of time. The datasets used with the tests where chosen to put the databases at maximum throughput rate for several minutes. This is enough for our purpose, we wanted to compare the maximum throughput rates of multiple databases. However, a database can be designed to postpone certain tasks when experiencing a high throughput rate of operations. Examples of this are the *major compaction* and *hinted handoff* techniques mentioned in the Background chapter (Chapter 2). It could easily be that the maximum throughput rate declines as the system is put under stress for a longer period of time.

Chapter 8

Contributions

While working to the answer of our main research question, we believe we solved a few intermediate steps which could be useful to others. In this chapter, we present the contributions of our research in three parts, each part answers one or more sub research questions. First we present our Theoretical contributions, we continue with the Practical contributions and end with the Business contributions.

8.1 Theoretical Contributions

Our research makes three theoretical contributions. First, we introduced the problem which industry faces into academics(Chapter 1). Secondly, we composed a current overview of RDBMS and wide-column NoSQL products with their technologies, available to address the problem (Chapter 2). Third, we listed a set of Key Performance Indicators which are needed to record during analysis of the operation of the proposed products (Chapter 4).

The remainder of this subsection will briefly discuss each of the theoretical contributions.

8.1.1 The Problem

A database which is able to scale along with the growth of write operation throughput is needed in the example we used in the introduction. We found that scaling-up is limited by hardware developments[28] and that the widely known RDBMS databases can not scale-out unlimited[24]. Our experiment showed this too. The NoSQL systems, however, easily scale-out to hundreds of machines[8] but might not run optimal when only a few machines are involved.

The problem is that RDBMS offers no scaling-out option to address rising write operation throughput rates and a replacement should be found. We tested NoSQL systems as a replacement but found out that there is room for improvement, we discuss this further with the conclusions, Chapter 9.

8.1.2 List of Scaling-out Databases

In the Background chapter (Chapter 2) we discussed both the RDBMS and NoSQL databases and how these scale-out. That answers our first sub research question:

> **Sub Research Question 1**
>
> *„What RDBMS and wide-column NoSQL database systems are available for a high throughput of read and write operations in a scaling-out context?"*

We showed that for RDBMS databases there are generally three options:

1. Single RDBMS instance
2. Master-slaves setup
3. Multi-master setup

We discussed that it is already known in literature that these solutions can not scale-out to any number of machines. The write operation throughput rate declines dramatically when more machines are added and our experiment reflected this too.

We also discussed the NoSQL databases and showed that the wide-column family is the one out of three which is closest related to RDBMS systems. Of these wide-column NoSQL databases we listed the two publicly available systems:

1. Apache HBase
2. Apache Cassandra

We discussed the techniques used in these systems which influence the throughput rate of the read and write operations. This showed that HBase has a more complex internal structure than Cassandra and that Cassandra does only support Eventual Consistency by default. We discussed how Cassandra can get more Consistent but pointed out that the system is not designed to be really strictly Consistent.

The list of databases contributes to future research by showing which systems are most relevant currently for high throughput read and write operations in a scaling-out context.

8.1.3 List of Key Performance Indicators

For answering our main research question we of course need to record the throughput rate of the read and write operations we execute at the databases. As we defined in sub research question 2:

> **Sub Research Question 2**
>
> *„What are the key performance indicators to measure the maximum throughput rate for each database system?"*

In the description of our experiment, in Chapter 4, we also described which other Key Performance Indicators we think are needed to measure. These values would help us justify that the throughput rate we measure is indeed the maximum the database could reach.

The Key Performance Indicators we listed:

1. Throughput: how many operations per second are processed
2. Latency: time between sending the request and receiving the response at the client machine
3. CPU usage: how much stress is the CPU experiencing
4. IO-Wait: how much time is the CPU blocking for an IO (typically harddisk) operation to complete
5. Memory usage: how much memory is used
6. Swap usage: how much swap space is used
7. Bandwidth in/out usage: how much is sent over the network

The list of Key Performance Indicators contributes to future research by showing what needs to be measured to determine whether the maximum throughput rate is reached. We found this missing in related research papers.

8.2 Practical contributions

During our research we also met several practical challenges which we want to share with others. First, we developed a tool to measure the Key Performance Indicators. Secondly, we worked with very new systems and we will discuss briefly the manageability of those systems. Third, we also described some use cases at Adyen where we used these new systems and we think one of them, the Log Analysis use case, can be of great benefit for others.

These three topics are described in the remainder of this subsection.

8.2.1 Measurement Tool

We developed a tool (presented in Chapter 4) to measure the Key Performance Indicators. The tool is designed to have minimal impact on the hardware resources, because it is a so-called *in-band* measurement tool. Measuring in-band means that the measurement influences the measured values itself. The tool runs at every machine and records all needed values locally. Afterwards the values are collected centrally and an overview is created for each KPI with the values for each machine next to each other.

The tool gives insight in the resource usage by the processes running in the meantime. In our research we used this tool to show that the database reached its maximum throughput rate because either the CPU had to wait for disk IO or that the CPU was fully used.

This tool can be used by any research which needs to inspect the hardware resource usage during a certain period of time at multiple machines with minimal impact on that same hardware resources.

8.2.2 A Word on Manageability

In the Background chapter (Chapter 2) we described how both HBase and Cassandra are organised internally.

We already had the impression that HBase would be complex to set up because the description of it in the BigTable paper[8]. During our experiment we experienced the same. HBase does not have a direct interface for a user, such as a Command Line Interface. A user should build little programs to administer the Column Families in HBase. Also, HBase and its underlying HDFS seem to be very resource intensive and it was difficult to get it working at our machines with limited hardware.

Cassandra was better manageable in that respect. However, Cassandra is under active development and new versions are released at an impressive pace (6 subsequent versions in 7 months). This also means changing API's which makes it hard to adapt your system to the newest version.

8.2.3 Log Analysis Tool

During the research several practical use cases have been worked out at Adyen. One of them is the Log Analysis tool which is designed to give a summarized view on all the log messages generated by the systems of Adyen. This system can scale-out if one detects that it can not keep pace with the rate of the log messages. Each machine processes a set of log messages, parses them and increments the proper counters for each. A web interface interprets the counters to generate charts which show the human operator in one view the current state of all the systems. At Adyen we use this to find Merchants which need help with the integration process.

The Log Analysis tool is still in development but the idea is clear. With using the scaling-out possibility of NoSQL systems, it should be able to process any rate of log messages. When finished, this tool contributes by providing insight in high-throughput distributed webservices.

8.3 Business Contributions

The results of our experiment are useful for making business decisions. First, we showed how each database is limited by each hardware resource. Secondly, we showed how each database excels at different operations. Let us discuss this further.

8.3.1 Relation of databases and KPI's

Sub Research Question 3

„How do the key performance indicators relate from each system to another?"

We used our measurement tool to record the hardware resource usage at all machines when we ran our experiment. The results showed clearly what we expected from the literature.

The RDBMS databases had reached their maximum IO-wait time for the CPU when it had to process write operations at maximum speed. Literature already suggested that by the *Durability* property of *ACID* (See Chapter 2). The read operation throughput seems to be limited by the CPU. However, we saw that with a bigger dataset, one that does not fit in memory, the IO-wait also becomes the limiting factor.

With the scaling-out setups (master-slave and multi-master) of RDBMS we saw that the CPU needs more time to process a write operation and the maximum throughput rate is bound by a combination of CPU time and IO-wait time. We did not see any improvement in maximum write operation throughput with either scaling-out solution. However, for read operations the scaling-out works. We were unable to determine the maximum throughput rate for read operations because the CPU of the client machine firing the operations was the limiting factor.

The maximum throughput rate of write operations with NoSQL databases is limited by a combination of CPU time and IO-wait time. For read operations the IO-wait time is the limiting factor. When scaling-out these systems the same factors are limiting both throughput rates. With RDBMS databases this differed much.

The relation between the Key Performance Indicators and the systems in the experiment show which factor is limiting the maximum throughput rate for each type of operation. This contributes to businesses in that it enables one to discover which hardware is useful for what system. It can also help in *scale-up* the current system to increase the maximum throughput rate, although limited.

8.3.2 Decision making

Sub Research Question 4

„How to decide which system would perform better in which environment?"

Our experiment shows how the maximum throughput rate of read and write operation relate to each other with both families of databases. With our results one can select the proper database system if one knows how frequently the read and write operations will take place in his or her use case. For instance, we showed how dramatically the write operation throughput dropped when the RDBMS systems are scaling-out. A user should keep this in mind when selecting these type of databases when he or she expects to process a growing number of read operations.

We have seen that NoSQL databases start at a lower read operation throughput but they are able to scale-out without declining throughput rates of either type of operations. The user in the above example might thus be better of to start with a NoSQL database of several machines than with a single machine RDBMS. Simply because he or she will get stuck with the write operation throughput when scaling-out for the read operation throughput at a RDBMS database. Although, the number of machines needed for a NoSQL database to equal the throughput rate of read operations of a single machine RDBMS database might be challenging.

However, we have also seen that there is a difference between HBase and Cassandra. HBase has a very high throughput rate of `UPDATE` operations but a disappointing throughput of `INSERT` and `SELECT` operations. Cassandra seems to perform more equal on all three tested operations.

This difference in qualities between both families of databases contributes to the decision making process when choosing which database to use for particular applications. We have presented a few use cases in which the application could greatly benefit from the specific behaviour of the NoSQL databases.

Chapter 9

Conclusions and Future Work

Our research started with a main research question and four sub research questions, in Chapter 8 we answered the latter, in this chapter we present the answers to the main research question. Also, in the Further Work section we propose directions in which we see future research should focus next.

9.1 Conclusions

In the Introduction Chapter we made a sketch of how we expected that the different databases relate to each other in terms of the maximum throughput rate for read and write operations, see Figure 9.1 to recall. This sketch would represent the answer to our main research question.

> **Main Research Question**
>
> *„How do RDBMS databases relate to wide-column NoSQL databases in terms of maximum read and write operation throughput rate in a scaling-out context?"*

With the results of the experiment plotted on the same axes we used in the sketch, we get a picture of the actual situation. We made this plot in Figure 9.2a and draw a few lines to help interpret the data.

The blue lines indicate the read and write throughput rates reachable by the RDBMS database in various scaling-out configurations. The red lines indicate the read and write throughput numbers reachable by NoSQL databases (Cassandra in this case). The black lines labelled '1 machine' and '4 machine' cross the datapoints of both RDBMS and NoSQL database configurations with one and four machines, respectively.

We easily see that the NoSQL database increases on both read and write operation throughput rate when scaling-out while the RDBMS database only increases its read operation throughput rate and decreases meanwhile its write operation throughput rate. Also, we see that the two black lines, indicating equal machine configurations, cross each other. This shows that both families indeed differ in how more machines impact their performance.

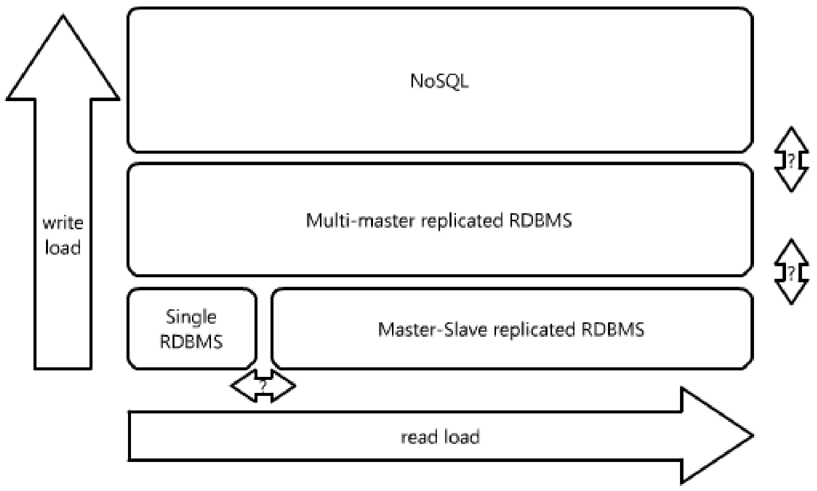


Figure 9.1: Sketch of expectations, made in the beginning of the research.

The following subsections describe conclusions drawn from these figures.

9.1.1 A NoSQL database needs more than four machines to equal the maximum throughput rate of read operations on a single RDBMS database

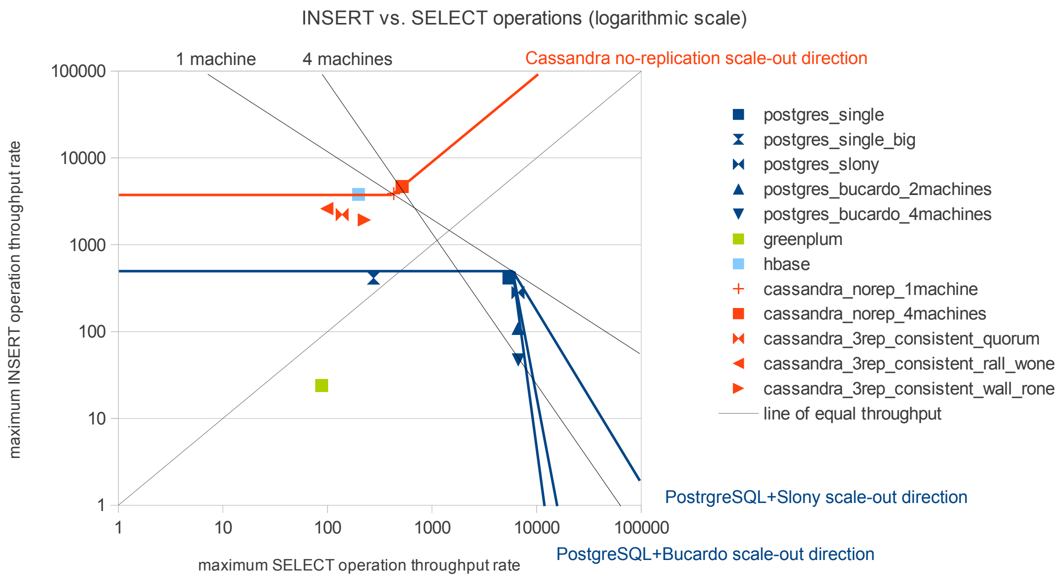
If we look at the maximum throughput rate of read operations, we clearly see two different systems. The RDBMS systems start at a high throughput rate and can scale out to bring this maximum even further. The NoSQL databases start at a much lower throughput rate. Our results show that more than four machines are needed for a NoSQL database to process the same rate of read operations as a single machine RDBMS database can process. This is a conclusion to take into account when replacing the one for the other.

9.1.2 NoSQL databases effectively increase the maximum write operation throughput rate with scaling-out.

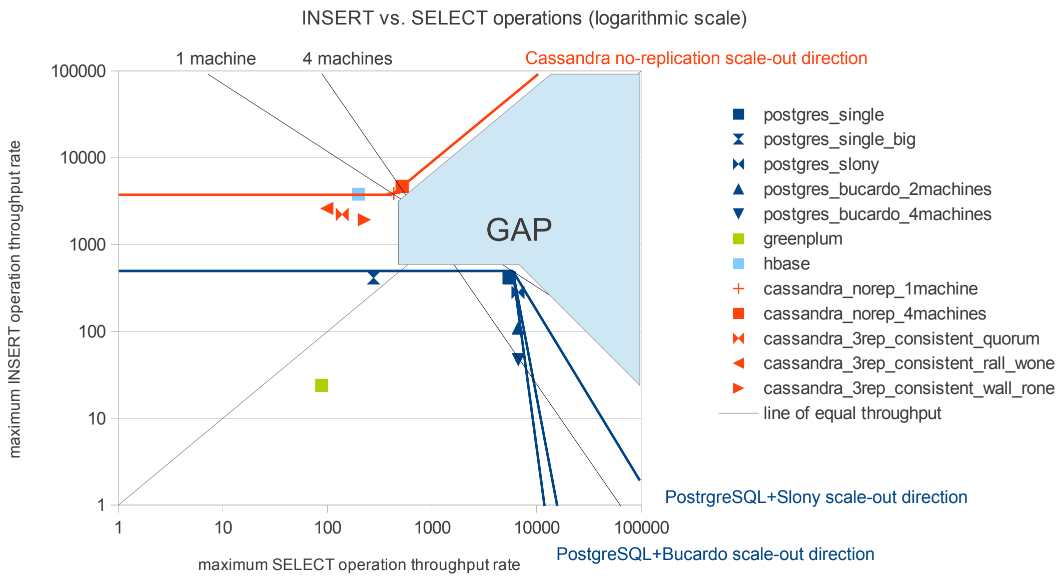
We have seen that the RDBMS databases dramatically decrease their maximum throughput rate for write operations when scaling-out. This was already shown by Gray in 1996[24]. NoSQL databases are designed specifically for scaling-out to increase the write throughput rate. This is also shown in the results of our experiment.

9.1.3 Cassandra has a higher maximum throughput rate for both read and write operations than HBase

If we look to the differences within the wide-column NoSQL family, we see slight differences. The Cassandra database without replication reached a higher maximum throughput



(a) Analysis



(b) The gap

Figure 9.2: Lines drawn (a) upon Figure 5.5a of the results of the experiment, show that there is a space (b) where the combination of read and write operation throughput rate is difficult to meet with the database products in our experiment.

rate for both read and write operations than HBase. When we studied the internals of both systems, we already remarked a complexity difference and it seems that this pays out in the performance as well. However, our machines where not sufficient to test HBase with replication enabled. So, we can not tell how both systems relate to each other with replication.

9.1.4 There is a Gap

Our main research question is to find out how both database families relate to each other. We have seen that the NoSQL databases with four machines only reach a small portion of the read operation throughput rate of RDBMS databases (See Figure 9.2a). We have seen that with write operations it is just the other way around. RDBMS databases have a declining throughput rate for write operations when scaling out. From this we conclude that both database families have their own specific use cases.

The RDBMS databases excel in read operation throughput and NoSQL databases excel in write operation throughput. A use case in which read and write operations occur with an equal frequency will not find its ideal database within either these two families of databases. Choosing a NoSQL database will at least enable the possibility to scale-out in the future but it also means that initially more than four machines are needed for the same read operation throughput rate which RDBMS databases reach with a single machine, if one wants to have the same durability as with RDBMS databases and enables replication in NoSQL.

We translate this to the statement that there is a gap in the operation throughput capacity between the RDBMS and NoSQL databases. This gap is seen in the upper-right part of the chart in Figure 9.2b. The importance of this gap is to be defined, as it could very well be that use cases tend to need either write throughput or read throughput. We think we have shown use cases in which this is not true. A manager at Adyen asked intuitively: ”Can we combine both?”. That question summarises the need from Industry, because they need databases able to process more write requests than a single RDBMS database can. However, from an academic perspective it is interesting to investigate whether both database families can be combined because they use fundamentally different techniques to reach these particular throughput rates.

9.2 Future Work

We concluded that there is a gap between RDBMS and NoSQL systems in terms of maximum throughput rate for read and write operations. We drew this conclusion because we found that a NoSQL database needs more than four machines to get to the same throughput rate of a single machine RDBMS database. We think future work should focus on this gap in two ways. First, we need to define how this gap evolves when more than four machines are involved. Secondly, it should be investigated whether it is possible to merge both families together. In the following text we describe this further with sample research questions.

9.2.1 More Machines

> **Future Research Question 1**
> *„How many machines are needed for a NoSQL database to equal the read operation throughput rate of a single machine RDBMS database?”*

We determined that a single machine RDBMS can achieve a maximum read operation throughput rate of 5434.1 operations per second. We found out that a four machine Cassandra database, without replication, can achieve 516.7 operations per second. Investigation is needed in how many machines Cassandra, or any other NoSQL database, would need to equal the throughput rate of a single machine RDBMS database. The amount of the needed machines can be used as an indication for the size of the gap between both families of databases. For instance, it would make a difference if we need six instead of ten machines to equal a single machine RDBMS.

9.2.2 Mixing Internals

By determining how big the gap is between both database families, we also get to know the importance of the following, more fundamental, question:

> **Future Research Question 2**
> *„How can the fundamental techniques of RDBMS and NoSQL be combined to get the best of both worlds?”*

We described in Chapter 2 that the fundamental techniques differ in both families of databases. Also, the results differ. The RDBMS databases seem to excel in read operations throughput but are hardly writeable when scaling-out. And NoSQL databases excel in write operations throughput but have a very low read operations throughput. A database which could excel in read as well as in write operations throughput when scaling-out is desirable in the described use cases (See Chapter 3).

Also, the relational model, which proved its value by the wide use of RDBMS databases is not supported by the NoSQL databases. However, we did not find any proof (yet) that it is fundamentally impossible to implement a relational model in a scaling-out database. Gray did proof that the way RDBMS implement the relational model can not scale-out but he did not proof that the theoretical concept of the relational model can not be scaled-out[24].

We are very curious to the further development of this field, and especially to the closing of the gap we indicated.

Bibliography

[1] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 9(1):1–20, 1991.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[3] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A comparison of flexible schemas for software as a service. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 881–888. ACM, 2009.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[5] E.A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[6] V. Cardellini, E. Casalicchio, M. Colajanni, and P.S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[7] F. Chang, J. Dean, S. Ghemawat, WC Hsieh, DA Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed structured data storage system. In *7th OSDI*, pages 305–314, 2006.

[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[9] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *SoCC*, pages 143–154. ACM, 2010.

[11] Transaction Processing Performance Council. *TPC Benchmark^{TM} C*, February 2010.

[12] J. Dean. Designs, lessons and advice from building large distributed systems. In *presentation at Third ACM SIGOPS International Workshop, Big Sky, Montana*, 2009.

[13] 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.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[16] EMC. *Greenplum Database 4.2 Installation Guide*. EMC, 2011.

[17] S. Fang and Q. Tong. A comparison of multi-tenant data storage solutions for software-as-a-service. In *Computer Science & Education (ICCSE), 2011 6th International Conference on*, pages 95–98. IEEE, 2011.

[18] Association for Computing Machinery and IEEE Computer Society. *Computer Science Curriculum 2008*. IEEE Computer Society, December 2008.

[19] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 78–91. ACM, 1997.

[20] Hector Garcia-Molina. *Performance of update algorithms for replicated data in a distributed database.* PhD thesis, Stanford, CA, USA, 1979. AAI8001920.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[22] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[23] J. Gray et al. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154. Citeseer, 1981.

[24] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[25] PostgreSQL Global Development Group. *PostgreSQL Manual*, 2012.

[26] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[27] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *Intelligent Systems, IEEE*, 24(2):8–12, 2009.

[28] S. Harizopoulos, D.J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[29] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)*, 28(3):257–294, 2003.

[30] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[31] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[32] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[33] M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[34] M. Stonebraker and R. Cattell. 10 rules for scalable performance in'simple operation'datastores. *Communications of the ACM*, 54(6):72–80, 2011.

[35] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[36] Milo Polte Kai Ren Wittawat Tantisiriroj Lin Xiao Julio Lopez Garth Gibson Adam Fuchs Billie Rinaldi Swapnil Patil. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC '11), October 2011.

[37] Joel van den Berg. Finding faulty components in a dynamic distributed system at runtime, 2012.

[38] Bert Wolters. Managing e-commerce credit card risk: an integral approach, 2012.

[39] J. Zinky, J. Loyall, and R. Shapiro. Runtime performance modeling and measurement of adaptive distributed object applications. *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 755–772, 2002.