Michael Münzer BSc.

Structural combination of MySQL and NoSQL

A use-case based example torwards polyglot persistence

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Software Development and Business Management

submitted to:

Graz University of Technology

Supervisor

Assoc. Prof. Dipl.-Ing. Dr. techn. Denis Helic

Knowledge Technologies Institute

Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Parkside Informationstechnologie GmbH

Supervisor: Christian Werding

Graz, January 2015

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgments

This thesis finishes my master studies of Software Development and Business Management at Graz University of Technology. The project was done in cooperation with the Knowledge Technologies Institute and carried out at the Parkside Informationstechnologie GmbH. At this point, I would like to thank a few people for their support during the last year. I want to thank my supervisor Denis Helic for his guidance and help during this time. He gave me valuable feedback during my project and thanks to him I was able to work on my suggested research project.

I would also like to thank all involved employees at Parkside Informationstechnologie GmbH, who gave me valuable insight into their experiences and pitfalls with selected database solutions. The great working environment and support motivated me in times of failure and helped me moving forward. Finally, I thank my family, girlfriend and friends for their support and understanding in times where I spent most of my time working on this project.

Michael Münzer

Contents

List of Figures

List of Tables

1 Abstract

Recent changes in user behaviour and higher expectations regarding service availability and response time lead to the use of leaner NoSQL data-models. NoSQL solutions are often used and evaluated with large amounts of data in very specific applications.

In this thesis a mid-sized dataset is used to evaluate the benefits of a document-based store in combination with MySQL. This flexible model makes it possible to fulfill multiple application requirements. Based on a real-life example different modelling techniques are compared and evaluated in multiple configurations. The different handling of JSON based data in document-based data-stores like MongoDB or CouchDB is evaluated in terms of performance, maintainability, and complexity.

Änderungen im Benutzungsverhalten bestehender Anwendungen sowie höhere Anforderungen an Serviceverfügbarkeit und Antwortzeit führten zur Entstehung einfacherer Datenmodelle. Diese unter dem Begriff NoSQL bekannten Datenmodelle finden häufig in Kombination mit großen Datenmengen in speziellen Bereichen Anwendung.

Im Gegensatz zu häufig durchgeführten Evaluierungen sehr großer Datensätze untersucht diese Arbeit einen mittelgroßen Datensatz. Dieser wird unter Verwendung einer dokumenten-basierten Datenbank und relationalen MySQL Datenbank evaluiert. Durch die Kombination von zwei unterschiedlichen Ansätzen soll es möglich werden verschiedene Anforderungen in einer Applikation sinnvoll und effizient zu erfüllen. Basierend auf einem realen Anwendungsszenario werden unterschiedliche Datenbankkombinationen untersucht. Der Umgang mit JSON basierten Datenbanksystemen wie MongoDB oder CouchDB wird dabei im Hinblick auf Performanz, Wartbarkeit und Komplexität evaluiert.

2 Introduction

This thesis aims to use an analytical approach in finding the right data solution for different portions of data. By identifying structural properties of an existing relational database, it is evaluated if the combination of different database solutions can lead to more intuitive storage infrastructures. In contrast to most available publications a mid-sized dataset growing at linear speed is used for this evaluation. This represents the situation of a small web startup having increased demand for data storage. User demands include all-time availability and good response times. It should therefore be ensured that platform speed stays high with increased usage.

The evaluated platform currently uses a distributed MySQL cluster solution combined with a memcached key-value based store. This is a very popular combination as memcached provides fast in-memory data access, while complex queries can still be performed in the backend. Database operations are done using standard raw SQL statements or an ORM. The application is based on the Symfony 2 PHP framework[1] which is a MVC-architecture. Database entities are managed and accessed with the Doctrine ORM[2]. Doctrine is operating on the application layer and responsible for all database communication within the application. The application layer can combine data from different data-stores and enables data exchange between them.

According to Parker et al. (2013) NoSQL solutions promise great performance and flexibility attributes for data typically used in web applications. To provide an infrastructure that is ready for future growth performance attributes should become better than they actually are. In terms of flexibility we want to take advantage of lean data models used in NoSQL solutions.

The current MySQL solution provides potential for the use of leaner NoSQL models. Currently some tables within the database have many columns, where most datasets are only using few of them. An example for sparse data with few, but different fields within a single table is shown in Table 2.1. This problem occurred due to mandatory schema definitions of RDBMS. By leaving out large amounts of empty fields during database reads and writes, increased platform speed is expected.

[1] http://symfony.com/
[2] http://www.doctrine-project.org/

As Abadi (2007) noted, NoSQL solutions like column-stores are better suited for this kind of data. To tackle this inflexibility in the relational database, dynamic JSON data was saved into database fields (Table 2.2). The problem with this approach is that MySQL is not optimised for this kind of usage and the ability to perform simple data queries is lost. Within MySQL, queries on stored JSON data can only be performed by using either regular expressions or running multiple queries on the application layer.

Id	**Name**	**ABS**	**Airbag driver**	**Airbag co-driver**	**Alarm**	**Radio**
1	Seat Ibiza classic					
2	Seat Ibiza comfort					Sony X1
3	Seat Ibiza premium	Premium K7	MSP			Sony X2
4	Fiat Punto classic					Blaupunkt K7
5	Audi A4 classic		MSP			
6	Audi A5 premium	Premium K7	LSP	LSP	CS2003	Sony X4

Table 2.1: Sparse data table

2.1 Motivation

In recent years, the web-revolution brought new challenges and changing requirements to existing database solutions. Relational databases were modified and extended to support data distribution at a larger scale. Since the first appearance of relational databases in the paper of Codd (1970), many things changed, but the general approach largely stayed the same (Stonebraker et al. 2007).

Hardware became cheaper, faster and more reliable, but usage-patterns changed as well. In the last decade, dealing with tremendous amounts of users and delivering content to multiple devices became the standard. Because of that, researchers and web-affine companies began to question certain attributes of commonly used RDBMS and tried to find enhanced solutions for their specific needs.

Id	Name	Content
1	Seat Ibiza classic	
2	Seat Ibiza comfort	`{"Radio":"Sony X1"}`
3	Seat Ibiza premium	`{"ABS":"Premium K7", "AirbagDriver":"MSP", "Radio":"Sony X2"}`
4	Fiat Punto classic	`{"Radio":"Blaupunkt K7"}`
5	Audi A4 classic	`{"AirbagDriver":"MSP"}`
6	Audi A5 premium	`{"ABS":"Premium K7", "AirbagDriver":"LSP", "AirbagCoDriver":"LSP", "Alarm":"CS2003", "Radio":"Sony X2"}`

Table 2.2: Relational table including JSON data in its cells

Under the term NoSQL many new products have been developed. They are moving away from traditional systems and explore new approaches. Most of them are able to manage exploring amounts of data, while still providing all-time availability. Often the consistency of RDBMS is traded against additional availability. Their purpose is to provide better solutions for todays problems and getting away from the predominant one-size fits all thinking.

Current research in the area of NoSQL is commonly about evaluating the performance after changing the database infrastructure. Most often the complete infrastructure is exchanged and an existing RDBMS database is replaced by a new NoSQL solution. Performance evaluations are usually done with big-data (tera- or petabyte) within a small cluster.

Many large web companies such as Google, Facebook or Amazon broke down relational models in exchange for scalable lean models. But not all of them mean to completely replace RDBMS, so does Facebook according to Diehl (2011) still use MySQL within its infrastructure. Companies try bringing together the best of both worlds and implement polyglot persistant architectures. Sadalage & Fowler (2012) describe polyglot persistent architectures as a combination of technologies taking advantage of multiple solutions tackling different problems. This means that multiple data-stores are combined in an intelligent way, so that each of them is dealing with different needs and requirements.

2.2 Methodology

Many applications use RDBMS in an unintuitive and overly complicated way. Implementing different, but better suited structures can lead to more statisfying results. An example are graph-based databases which can be used to represent social relationships. They take advantage of optimised graph operations like Breath-First-Search for data crawling. To achieve the same result in RDBMS recursive SQL queries or data crawling within the application layer have to be implemented.

On the way of finding a good storage solution, requirements are summarised and a typical use-case is defined. This shall ensure that performance evaluations are performed on realistic data and the results can be seen as an accurate foresight. Based on current usage statistics upcoming amounts of data are estimated and its evaluated how well the chosen solution can deal with them. The application performance is evaluated with load-testing tools. This shows how many requests the picked solutions can handle. When setting-up a distributed solution, consistency properties have to be considered as well. To evaluate this, different NoSQL solutions are explained in section 5.5 and their attributes are discussed in section 6.1.

As portions of the data shall be moved to a NoSQL solution, existing MySQL data has to be migrated to the new data-store. This can become difficult, depending on the chosen NoSQL-solution and its data structure. As most solutions are just existing since very few years, there are almost no migration tools available. Possible migration strategies for the chosen solution are explained in subsection 6.2.3.

To make sure that adding an additional data-store to the existing infrastructure creates long-term benefits, support and maintainablility attributes are included in the evaluation. Evaluation takes place by answering questions like how well a NoSQL solution is supported or how easy it is to set up and integrate into an existing infrastructure.

2.3 Objectives

This thesis attempts to answer the following key questions in the upcoming chapters, especially within the evaluation of chapter 7:

- How do different NoSQL data models differ from the relational model?
- What data structures are well suited for NoSQL solutions?

- Is it possible to migrate relational data into other data structures? How complicated is this process?
- How does a NoSQL solution perform against MySQL when a mid-sized dataset is used?
- What effect has a NoSQL solution on maintenance and deployment procedures?
- How well can relational databases be combined with non-relational solutions?
- How is dealt with the increased amount of complexity?

2.4 Outline

By going through a real-life scenario, the practical usage of a NoSQL database and the way to pick the right one for given data structures is evaluated. Their strength, weaknesses and position in the currently predominant world of relational databases should be shown. During that process often unconsidered advantages and problems of RDBMS are summarized.

In this chapter, the current situation has been stated and the purpose of this paper has been defined. Chapter 3 focuses on the theoretical background and differences between RDBMS and NoSQL solutions when operating in a distributed environment. This also includes how clusters can look like and what has to be considered to ensure high-availability.

Based on that, relational databases are summarised in chapter 4 and a closer look is taken on their advantages and disadvantages. NoSQL solutions will be introduced in chapter 5. It explains their increasing importance in todays web-infrastructure and what to consider when using them. To find the right solution for the data structures defined in subsection 6.1.1, different types of NoSQL solutions are explained in section 5.5.

After getting in touch with various storage solutions, the currently used MySQL database is examined. After analysing it, all important properties can be found in section 6.1. Based on that, a concrete NoSQL implementation is integrated into the existing infrastructure. To evaluate if the new approach generates additional benefits, it is evaluated in terms of performance, scalability and complexity in chapter 7.

Predefined objectives stated in section 2.3 will be discussed in section 8.1 and a short outlook with topic related questions is given in section 8.2.

3 Theoretical foundations

Web applications often need high availability and throughput combined with low latency. A problem occurs when the application also relies upon persistence and has to be scaled up. There exist two ways for scaling a data storage solution without creating a bottleneck. One is called vertical scaling and the other is horizontal scaling.

Vertical scaling means to extend the database server with additional hardware, making it faster and more reliable. This type of scaling becomes expensive very quickly and has an upper limit set by the current technological progress. These limitations make horizontal scaling much more popular.

It is almost ever cheaper to buy new hardware and extend a running system instead of replacing an old one. Horizontal scaling does so by enabling larger infrastructure to seamlessly add additional servers. Building up infrastructures with multiple servers means to tackle many more problems in terms of concurrency and distribution. This makes horizontal solutions more complex, but also more flexible (Dan 2008).

3.1 CAP theorem

According to Gilbert & Lynch (2002) there are three competing goals in distributed applications. The CAP-theorem states that it is not possible to ensure that all of these goals are met simultaneously. Those three goals are:

- **Consistency:** After each transaction there is a stable state on each instance within the system. This means that after one instance has done some disk-writing, all other instances need to update their data. Distributing changes after each write-operation can lead to lower response times when the cluster is growing.
- **Availability:** Every operation must terminate in an intended response. This means that requesting data from an instance shall return a result with no need to send a second request. To ensure high availability, connection timeouts have to be prevented.

- **Partition tolerance:** The system should always be reachable, even when one or more nodes collapse. An application is considered fault tolerant when the system stays stable without all of its nodes available.

Horizontal-scaling is based on high-level data partitioning. The focus between the remaining two goals can then only be set on either consistency or availability. NoSQL solutions are commonly using the BASE model, where availability is favoured. In contrast, unclustered RDBMS are using the ACID model, where consistency is preferred and partition-tolerance is ignored. Ensuring a high level of consistency leads to higher response times which can become unacceptable. But losing consistency means that different users access the same data at the same time and data corruptions can occur.

The CAP theorem does not mean that each database solution has to pick two out of those three attributes, but it means that some are favoured over others. Figure 3.1 shows some rather popular database solutions and which attributes they favour with respect to CAP.

Brewer (2012) says: "In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related." What he means is that the decision between availability and consistency has to be made only when a timeout occurs. You can either cancel the operation and decrease availability or proceed and take the risk of inconsistencies. This statement leads towards hybrid solutions behaving different according to the circumstances.

3.2 ACID

As mentioned in section 3.1, most RDBMS guarantee ACID transactions. According to Edlich et al. (2011) it is an acronym for four different properties:

- **Atomicity:** Each transaction is atomic. If an error occurs within a single suboperation, the whole transaction has to be reverted and the previous state has to be recovered.
- **Consistency:** After finishing a transaction, the database state has to be consistent. All following operations have to operate on the previous finished state.
- **Isolation:** Transactions running in parallel do not have to interfere each other. Each transaction has to wait until the previous is finished with its operation.
- **Durability:** After one transaction is finished successfully, all data is saved into the database. The effect of a transaction must persist and thus never be lost.

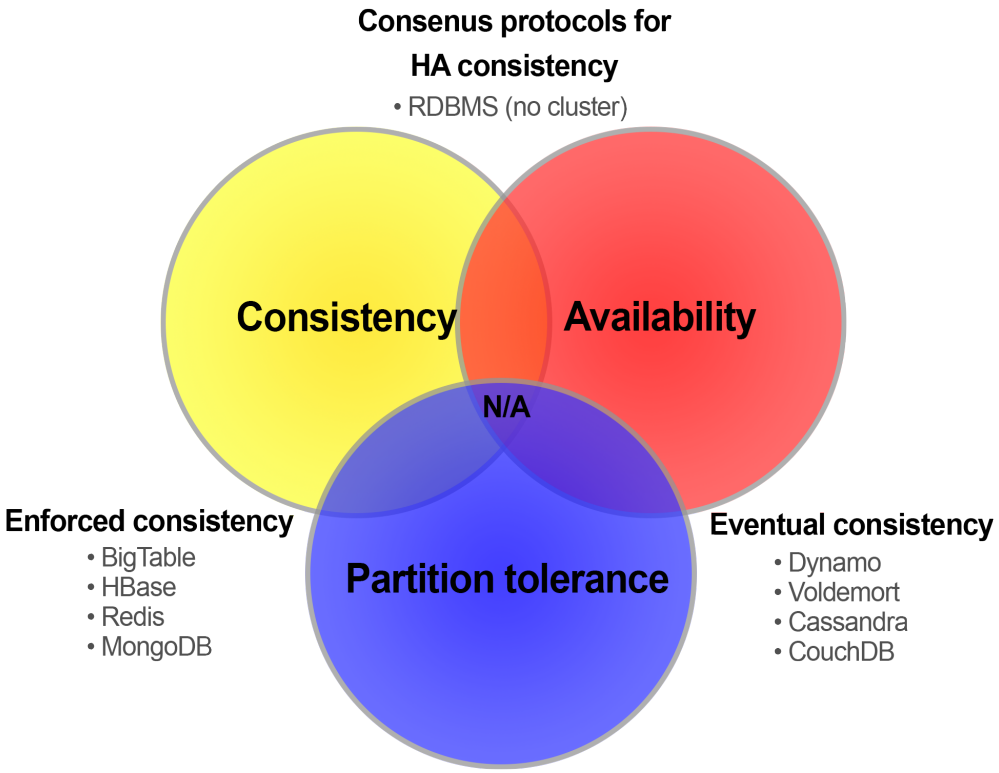


Figure 3.1: Database solutions and their web-scaling properties

When ACID properties shall be ensured across multiple database instances, 2PC (two-phase-commits) can be used. They can be broken down into the following two steps (Dan 2008):

- The transaction coordinator asks all database instances to pre-commit the operation. If all instances can do that, the next step is taken.
- The transaction coordinator asks all database instances to commit the operation. If any instance vetoes the commit, all other instances have to revert their changes and the system starts over with the first step.

Stonebraker et al. (2007) notes that using 2PC should be avoided in distributed transactions. They are using round-trip communication, which can take very long. As they are just delaying the decision, this can have a major performance impact.

3.3 BASE

In contrast to ACID, the focus in BASE is on availability. It allows database nodes to have different data-views and follows an optimistic approach in terms of consistency. The system does not enforce a consistent state after each transaction. Instead it hopes that the made changes are distributed quickly across all other nodes. A consistent state is normally reached some time later. The amount of time this distribution takes depends on the number of nodes and their response times within the system.

BASE is useful for applications where old data or slightly different user content is acceptable. The advantages gained in terms of availability are often more valuable for the application. It can be used for peoples activity-stream messages on the Facebook wall, but should be avoided for banking transactions.

Often it is difficult to identify opportunities where relaxed consistency is OK. As these inconsistencies can not be hidden from the end-user, engineering and product-owners must be involved in picking the right data for this data solution (Dan 2008). Using this kind of architecture allows segments of data being read and processed in parallel. Algorithms such as Map-Reduce explained in subsection 5.2.1 do exactly that. Strauch (2011) put together some differences between ACID and BASE, which are described in Table 3.1.

Systems based on this concept have the following attributes:

- **Basically Available:** There are fast responses even though some nodes are slow or have crashed.

- **Soft State:** The state of the system may change over time and this change is independent from incoming data.
- **Eventually Consistent:** Indicates that the system becomes more consistent over time. To guarantee this, the system must not receive any incoming data transactions during this time. It also means that data sent to clients may be cached or outdated.

ACID	**BASE**
Strict consistency	Weak consistency
Isolated system	Availability first
Focus on commits	Best effort
Nested transactions	Approximate answers are OK
Conservative (pessimistic)	Optimistic and simple
Difficult evolution (e.g. schema)	Fast and easy evolution
Simple code	Harder code

Table 3.1: Differences between ACID and BASE according to Strauch (2011)

3.4 Replication

Web applications often have more reading than writing operations. This ratio is about 10:1 in web applications focusing on content delivery. One way to increase reading capacity is replicating the same data on to multiple servers. Doing so increases the probability that a client acquires a free server for reading. There are two ways setting up such a system:

- **Master-Slave:** One master can have multiple slaves. Writing operations and logging is done by the master and all slaves are just used to process reading operations. Using this setup makes the master a SPF (single point of failure) and the system can not operate correctly after the master breaks down.

  In this case, it is possible to auto-select a new master within the system. Doing so still decreases availability, as there always is a downtime until the new master is up and running. Using this technology in a cluster implies that consistency is preferred over availability.

- **Master-Master:** There are at least two masters handling writing operations within the system. Due to this matter, possible conflicts can occur during writing.

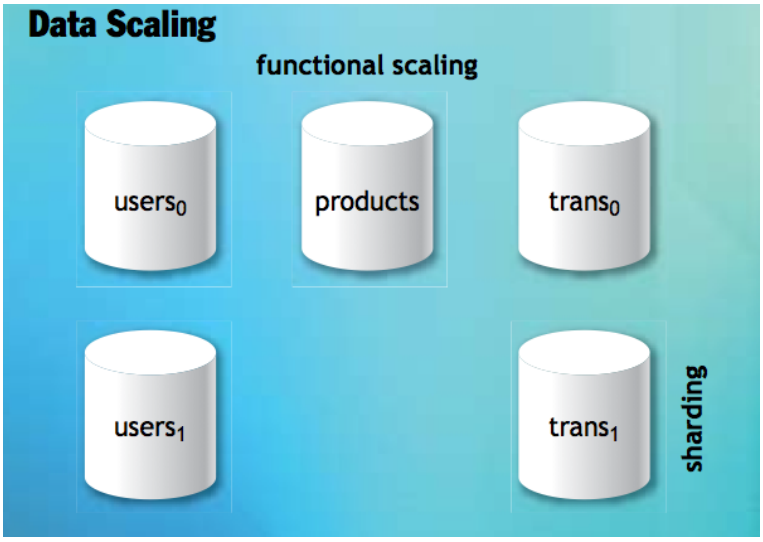


Figure 3.2: Two dimensions of horizontal scaling according to Dan (2008)

When this happens, the system has to decide between synchronous or asynchronous writing. Within a synchronous system all machines have to wait until the replication is finished before a response can be delivered. Sending responses immediately indicates eventual-consistency, where conflicts need to be resolved afterwards. Restoring a consistent state can e.g. be done by finding time differences between dataset changes and keeping the last modification in the database.

3.5 Clustering

Using replication needs the whole database to be locked frequently. To avoid distributed locking of a single data repository, we can split the repository into multiple data storages. A single database can be dividing into functional partitions. This allows e.g. splitting a customer- from a product-dataset. This solves the problem of data bottlenecking and distributed locking. Doing so does of course not come without any new challenges. We now have to deal with data synchronisation between multiple instances.

Additionally to functional partitioning, there is a second dimension which can be used. It is called sharding and a very popular pattern in distributed systems (Dan 2008). A visual representation of these two concepts is shown in Figure 3.2.

3.5.1 Functional partitioning

Distributing functional areas on multiple database servers is very important for higher degrees of scalability. Coupling different tables via key-constrains can only be done on a single server. Operating on multiple servers means that these constraints have to move into the application layer.

A benefit of functional groups is that each of them can follow a different scaling strategy. This enables each group to weighted the three attributes of CAP differently.

3.5.2 Sharding

When data is distributed across multiple nodes within a larger system, it is called sharding. A shard is a server or set of servers containing parts of the whole data. To distribute data across shards, a shard key is used. This key can be seen as an unique field and defines a certain key-space. Every chunk of that key represents a certain range and is managed by its own shard. Figure 3.3 shows how this creates clear responsibilities for all shards, where each shard is assigned to a specific key-interval.

Sharding creates interesting possibilities, e.g. reducing the amount of scanned data during a data request. The request just gets redirected to the shard responsible for the requested data-range. When shards operate in parallel, this helps decreasing response times dramatically (Rèmy Frenoy 2013).

Response times get even faster when data is stored close to the user. Shards enable storing data close to the regions where it is retrieved most frequently. Traffic has to be analysed and shards have to be reorder in a way that their physical server locations are close to the traffic source. Using sharding with an increased number of shards reduces availability. This is because the risk of collapsing nodes increases as well. To reduce these downtimes, replication is used and the same data is stored on several shards.

Architectures which make extensive use of sharding are called shared-nothing architectures. They ensure that no node is dependent on another one for any kind of update or operation.

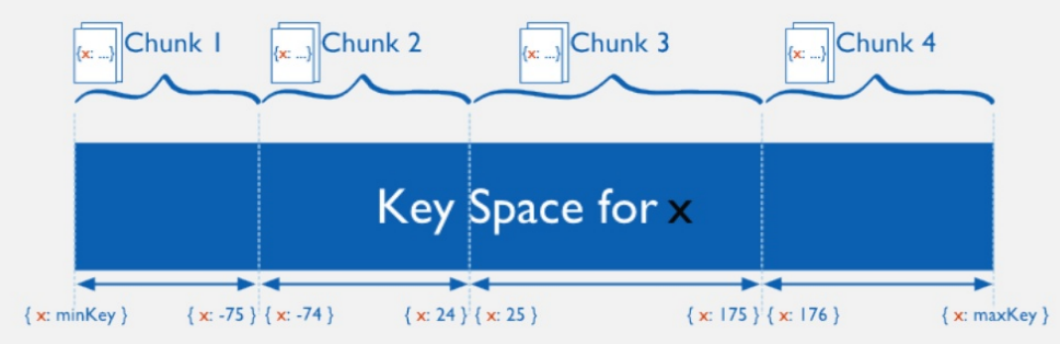


Figure 3.3: Shard keys and their splitting taken from Wilson (2013)

4 RDBMS

4.1 Characteristics

RDBMS are the most commonly used data-stores and have been widely adopted for many different applications. Originally designed for vertical scaling they usually work best for mid-sized datasets that do not need to be distributed. In these use-cases RDBMS provide sufficient performance and there is no need to use less mature solutions.

Based on relational models, data is represented in tuples and grouped into relations. Each relation has a relational schema defining the structure of a table. All data within a table fulfills certain properties defined by the schema. These properties always guarantee a consistent state of the data model. Schema definitions can include type definitions, length limitations for certain fields etc.

Within a relation, each row represents an object and each column an attribute of that table. Relations between tables are represented by three different relationship types using primary and foreign keys. Relations can take different forms, e.g. 1:n, 1:1 and n:m to represent associations between tables (Elmasri & B. 2009). With the help of these three associations data dependencies between tables are defined.

Another design goal of RDBMS is the reduction of redundancies. This is done with normalisation, where duplicated data is extracted and transferred into a new table. Additional key-constrains are introduced to match the relationship of the new table to the other tables. Doing so, makes the database more error resistant and increases maintainability when data is changed (Geisler 2011). The drawbacks are a more complex schema and more expensive data retrieval.

Relational databases rely on relational calculus and have comprehensive ad-hoc querying facilities provided by SQL. This query-language has been widely adopted and offers a single standard interface to access multiple relational data-stores in a consistent way.

Before NoSQL there have already been different approaches to replace RDBMS data-stores. Technologies like object databases or XML stores have never gained the same adoption and market share as RDBMS. Most of them have either been absorbed by RDBMS (e.g. data-stores were extended to store search-optimised XML data-fields) or became niche products (Strauch 2011).

Most RDBMS are built on System-R, which has the following architectural characteristics still shining through (Strauch 2011):

- Disk oriented storage and indexing structures
- Multithreading to hide latency
- Locking-based concurrency control mechanisms
- Log-based recovery

Over the years RDBMS improved and some extensions have been made. Improvements include compression-support, shared-disk architectures, bitmap-indexes and support for user-defined data-types and operators.

4.1.1 SQL

SQL is a declarative programming language for designing and managing data of RDBMS. It builds a consistent, useful abstraction framework on top of the data-storage and allows applications to optimise data-access within the boundaries given by that abstraction. Most relational databases use different dialects of the SQL standard, in which proprietary functions have been added.

The probably most notable benefit of this query-language is its flexibility and ability to create queries on demand (Elmasri & B. 2009). In the beginning, it was created to enable simple database access for the common user. This paradigm is still valid and language basics of SQL are considered as easy to learn and apply.

Within its very easy and proven syntax, it is rather hard to have low-level bugs. High-level bugs like incorrect join-operations or wrong assumptions about the data model in contrast can of course occur.

SQL can help a lot when data is manipulated, but many concepts can still not be expressed. This is because it is a non-turing complete language. The list of limitations includes matrix transformations, bayesian filtering, probability analysis and the inability to work with nested structures like trees or graphs. When writing more complex SQL code, some important features of other programming languages are

missing as well. There is e.g. no debugger support and no high-level concept such as object-orientation.

An often mentioned problem is that large queries take very long and using multiple joins is bad practice when operating in a distributed environment. This can be solved, but many people do not know that SQL provides a small set of optimisation techniques for exactly that. Often it needs a good SQL programmer who understands the inner-works of SQL and can take advantage of that. With the help of these optimisation techniques its performance is even competitive to NoSQL solutions. Using such query optimisations makes code certainly more complex and harder to duplicate by less experienced programmers.

4.1.2 MySQL cluster

MySQL cluster provides a shared-nothing architecture where each data node contains its own portion of the data (Cal Henderson 2006). Tables are automatically partioned horizontally. This is done using auto-sharding, shown in Figure 4.1, to scale-out read and write operations on to multiple nodes. This figure also shows that MySQL cluster automatically creates node groups. In each group, data updates are synchronously replicated to protect against data loss. The system is available as long as one node within a group is accessible. Additional to synchronous replication, asynchronous checkpoints are written to disk (Göbel 2011).

Sharding is done on the database layer eliminating the need to shard anything on the application layer. Less responsibility on the application layer makes application development and maintainability easier (Keep 2011).

MySQL cluster is a multi-master ACID compliant architecture that uses synchronous replication through 2PC. It is designed to have no single-point-of-failure and data should not be lost if a single machine collapses.Wennmark (2012) claims that using Oracles memcached API in combination with MySQL cluster for data reads and writes, scales very good on commodity hardware. Using this distributed memory-based key-value store achieves an up to nine times higher INSERT/SET throughput compared to normal MySQL databases.

MySQL cluster is especially good for applications that perform many simple queries, need low latency and high availability. The underlying NDB-Engine used in the cluster offers just a subset of the query-possibilities a normal MySQL solution provides (Göbel 2011). Joins are even more expensive, especially when they have to work across machine-borders.

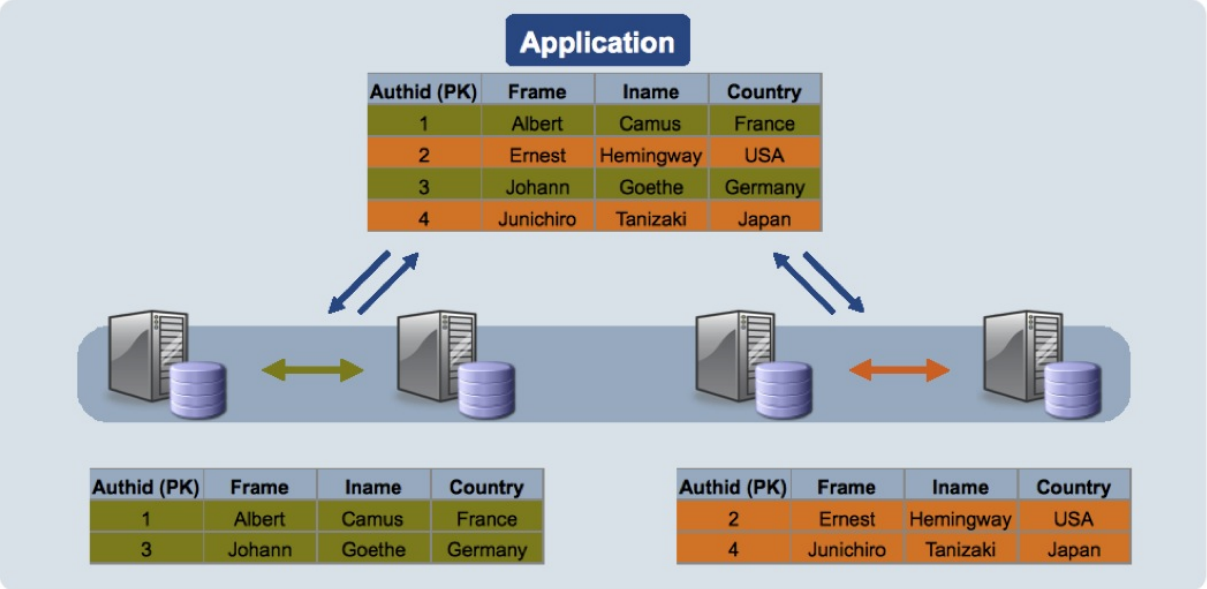


Figure 4.1: An example for auto-sharding in MySQL (Keep 2011)

4.2 Advantages

Relational databases are proven and their use-cases are widespread. For this and many more reasons people often forget about their advantages and want to replace them even when it is not necessary. Core advantages of RDBMS include:

- **SQL-query language:** SQL is a single and easy-to-understand interface to create and change data as needed. Already implemented aggregation-functions or simple SQL queries are very easy compared to other techniques of data gathering.

  A very popular algorithm for data retrieval in NoSQL databases is Map-Reduce explained in subsection 5.2.1. If queries are simple, SQL is easier to implement than Map-Reduce. When in contrast difficult operations are necessary, it takes about the same time until the programming operation is performed.

- **Access Control:** RDBMS have a robust security model, that allows the creation of user accounts, roles and groups. Permissions can be set granular for different users and operations (INSERT, DELETE, etc.). Going without these security features makes databases operations faster, but more error-prone.

- **Standardisation:** Many open-source and commercial tools offer high-degree support for SQL. This includes automated reporting, visualisation, web-based data administration etc. These tools are mature and professionally supported. Some NoSQL databases in contrast use new and experimental languages and tools, where future support is not guaranteed. Most often this are open-source projects only supported by volunteers. No one can tell if there will still be a community behind certain tools in the near future.
- **Data modeling:** An advantage during establishing the design of a RDBMS is that the developer has to consider the nature of the data. He has to think about their relationships, attributes and entities. When no schema is present, this also means that there is no protection against mistakes and misspellings. It assumes an advanced developer skill-set, where less control is traded against better performance.

4.3 Disadvantages

Performance is always a big issue in RDBMS. We already discussed how MySQL cluster can compensate this and compete with the high level of availability NoSQL solutions provide. Still some limitations are not fading away by using a cluster over a single RDBMS:

- **Unstructured data:** Storing data in tables, loosely coupled by key-constrains, makes sense in terms of mathematical relations. A problem occurs when data-sets have highly diverse attributes, that do not easily fit into a relational schema. These data attributes result in lots of empty data-fields stored in the database.

  Getting around this, people started to store unstructured data in large blobs within data-fields. Querying this data with string-matching patterns often performs miserable and is difficult to write in SQL.
- **Data mapping:** In most programming languages data is needed in a different form than SQL delivers it. Most of the time an own application layer unit called Object Relational Mapper (ORM) is needed to convert database tables to usable pieces of code. This slows down the development process and adds additional complexity to the program. It also asks the question, why data is even structured in this kind of way and expensive join-operations are needed to convert it into some usable representation.
- **Difficult optimisation:** Very well known MySQL customers like Google, Yahoo, etc. are delivering products that are working with RDBMS at a very high performance level. The reason why not everybody can do this, is that it is very

difficult and needs lots of expertise to understand the inner works of MySQL. Assuming this knowledge is present, the whole data design has to be done wisely. Structures like buffers, indexes and joins have to be used in a very smart way (MySQL 2014).

- **Costs:** Professionally supported relational database solutions are more expensive than free open-source NoSQL products. Due to the rigid data model behind RDBMS, development costs and time consumption can become higher as well.

5 NoSQL

5.1 Motivation

The NoSQL movements designated mission is it to provide alternative storage solutions to situations where relational models are a bad fit. NoSQL as a term was first used by Strozzi (2010) for a RDBMS which did not support SQL. It was later adopted for a whole movement that develops alternatives to RDBMS and is now mostly interpreted as "Not only SQL". The starting point for this movement was the first implementation of Googles Bigtable (Chang et al. 2008) and Amazons Dynamo (Decandia et al. 2007). Both web companies deal with huge amounts of data and always search for new ways to store and manage it efficiently.

When an increased volume of data is generated, it is called big-data. Being able to analyse, visualise and find trends in this data has become increasingly important. According to studies of Manyika et al. (2011) this will underpin new waves of productivity, growth and innovation. But big-data is not the only reason why NoSQL solutions are becoming more popular. A list of motives based on the research of Lai (2009*a*) tries to explain why this movement grew since the introduction of Web 2.0:

- Advances in cloud-computing and distributed web applications created the need to store large amounts of data (tera-/petabyte). This data needs to be stored in distributed databases to provide high availability and scalability.

- The ability to scale fast and easy has become increasingly important as most web companies use agile development methodologies during production. This type of lean database-models make it easy for many web startups to extend their infrastructure after the first success. According to Strauch (2011) this avoids late and expensive changes in application code.

- Companies began simplifying the database schema of RDBMS by denormalizing it. This relaxes durability and referential integrity. To provide better availability, query-cache layers are used and read-only replicas have been separated from writing ones.

- Cloud computing forced a design approach that makes front- and back-end functionality strongly decoupled. Data storage used in the back-end can have a high impact on the applications scalability. Experience showed that traditional RDBMS can become a bottleneck there. For this reason, nowadays most cloud-providers like Azure, Google or Amazon offer an easy to setup NoSQL service.

- Cass (2009) states that the current one-size fits all databases thinking was and is wrong. ACID properties are not needed when the presentation layer can deal with inconsistent data. More flexibility can be achieved by accepting data loss, even though this is not ideal.

  David Merriman from MongoDB Inc. says that there is no single tool or technology for the purpose of data storage. Following this thought, many diverse custom solutions and segmentations in the database-field are evolving. This leads us to use-case optimised data-stores. Some of these use-cases are e.g. business-intelligence, online transaction processing or persisting large amounts of binary data.

- Data models originally designed with a single database in mind are hard to be partioned and distributed among several database servers. Abstractions are used to hide distribution and partitioning problems from applications. Strauch (2011) claims that performance increases if applications know about latency, distribution failures, etc. Doing so makes applications more responsible, as they have to deal with more complex situations.

- In a discussion within his blog Hoff (2010) says that sharding MySQL to handle high write loads and caching objects in memcached to handle high read loads is insufficient. The problem is that it needs a lot of glue code to make it all work together.

  He says that the clunkiness of the MySQL and the memcached-era is over soon. Large scale applications should use systems built from scratch with scalability, asynchronous database I/O, huge amounts of data and task-automation in mind.

- Back in the 1970s databases have been designed for single high-end machines, where vertical scaling was sufficient. Nowadays, many large applications use commodity hardware that is likely to fail. Todays data is not always structured and many applications do not even need dynamic queries. Instead they use prepared statements and stored procedures.

- Operating in a cluster needs architectural considerations to prevent single-points-of-failure and bottlenecks within the management services. A stable, scalable distributed system needs to tolerate failure. Running tasks in parallel enables a non-disruptive approach for usually disruptive tasks. Application updates should always be done on-cluster, with no need to interrupt a running system.

5.2 Characteristics

Section 3.1 explained that NoSQL solutions are generally less consistent by using the BASE model. But less consistency does not mean none at all. Instead eventual consistency (Service & Cloud 2009) is used very often. This means that the transaction protocol does not guarantee that reading and writing of database entities will always be consistent instantaneously. Instead it is possible that entities are temporarily inconsistant in case of failure or latency. Using less strict data-constrains enables less complex and more flexible data models.

Non-relational databases do not use slow locking mechanisms, but better suited forms of concurrency control. An often used mechanism relying on timestamps or vector-clocks to determine transaction modification-dates is MVCC (Multi-Version Concurrency Control). Using this concept provides point-in-time consistant views by maintaining multiple versions of the data. Having multiple versions in place allows each user to access a timestamp based snapshot of the data.

Database changes will only be seen by other users, when the transaction has been fully completed. Instead of overwriting old data, it will just be marked as obsolete. Users which have started reading operations can finish their transaction with old-data snapshots. In contrast to locks, this kind of optimistic concurrency never blocks reading operations and allows performing them in parallel.

Due to cost-pressure, NoSQL solutions can run on commodity hardware infrastructure. Machines can easily be added and removed without causing operational effort. MySQL cluster and most other solutions provide auto-sharding to make scaling easier and cost-efficient.

Most non-relational solutions are open-source using Map-Reduce jobs for data retrieval and processing. When no relational database schema is present, this most often means that something similar is used in the application layer. Within this layer system testing and validation constrains have to be added. This adds additional complexity to the application layer in favour for higher transaction volumes.

When applications know about distribution, the following eight assumptions can and will prove false. They are better known as the eight fallacies of distributed computing (Rotem-Gal-Oz 2006):

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology does not change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

5.2.1 Map-Reduce

Map-Reduce is a programming model for processing large data-sets within a distributed environment. Introduced by Dean & Ghemawat (2008) it has been included within the Apache Hadoop framework[1]. Very large data-sets are split up into smaller data blocks distributed over the cluster for fast data processing. Implemented in most NoSQL solutions, it can deal with unstructured data like images, videos, photos, etc.

With this kind of data processing new ways of analytical capabilities can be used within business solutions. Map-Reduce allowed Google to perform simple computations while hiding much complexity in the background.

Large data-sets are split-up on to multiple machines. Each machine performs a mapping-function filtering the data-set and generating key-value pairs out of it. These key-value pairs are saved as temporary results. The framework aggregates them by grouping values with the same key together. Grouped result sets are passed to the reduce-function, where further calculations are performed.

To give a better overview, Figure 5.1 shows an environment with all involved worker-processes. A worker-process is either performing a Map or Reduce-task, but not both at the same time. Leskovec et al. (2012) argues that a system should not use too many Reduce-tasks and one Map-task per input-chunk. This is because of the huge number

[1] http://hadoop.apache.org/

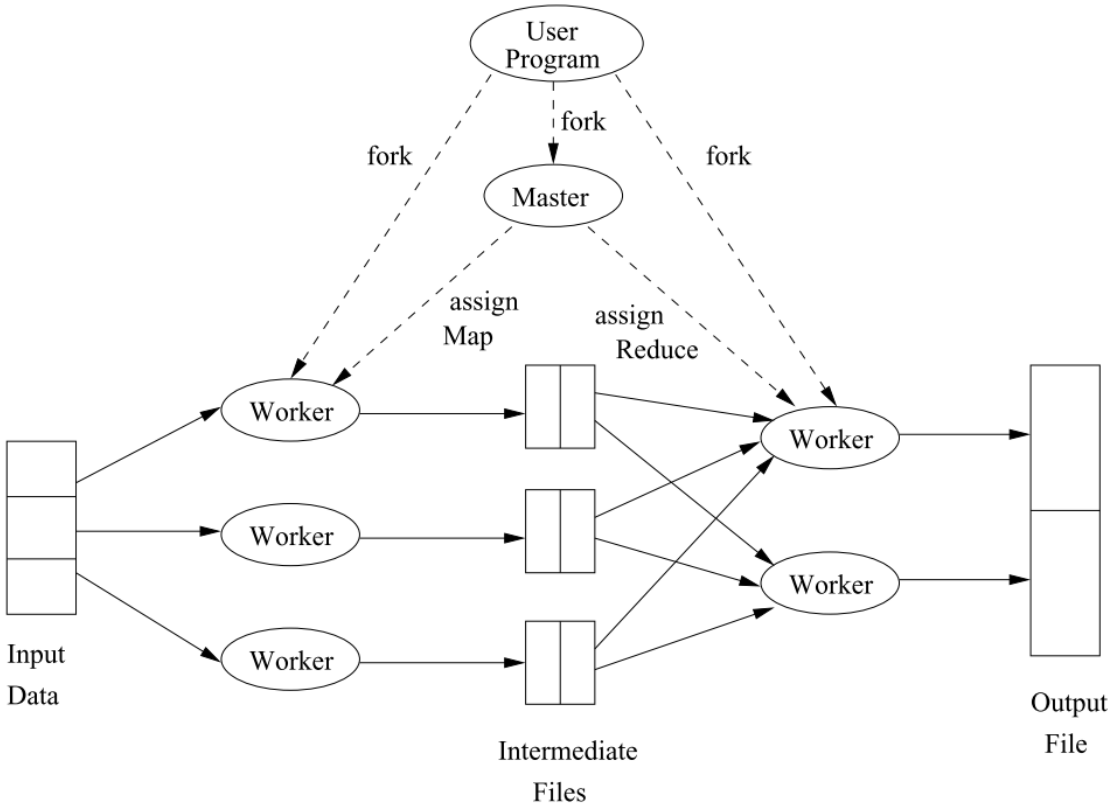


Figure 5.1: Environment for the usage of Map-Reduce explained by Leskovec et al. (2012)

of intermediate files generated by the Map-task. To distribute work on all processes, a predefined master schedules new tasks to the worker threads, when they are finished.

When using this framework, the programmer just needs to write the map and reduce routines. Functional languages make it possible to handle multiple jobs in parallel and ensure that no mapping function is depending on another (Dean & Ghemawat 2008). An example showing how one can count the number of occurrences of all words within a text is shown in Figure 5.2.

5.3 Advantages

- **Simple structure:** The usage of structures like key-value or document-based stores makes development easy for novice programmers. This flattened and portioned structure completely differs from the traditional relational data model

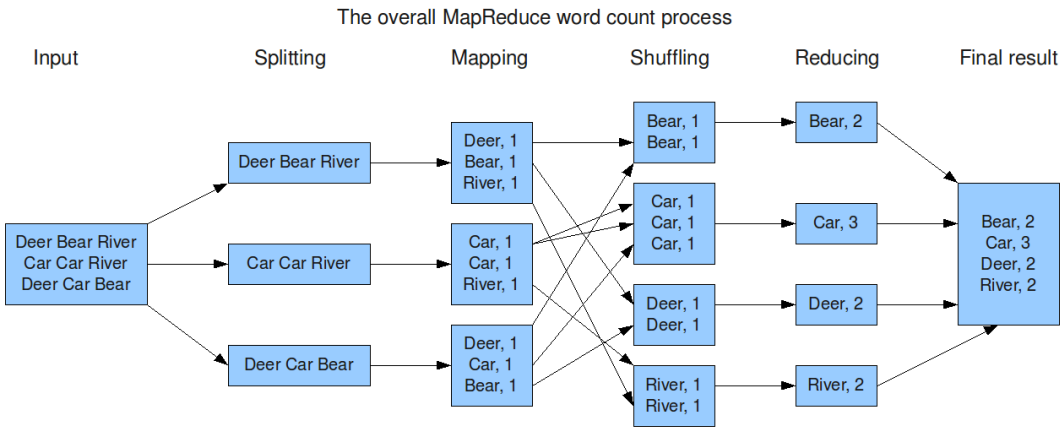


Figure 5.2: Counting the number of occurring words within a larger data-set

and allows more variable structures. Using e.g. JSON within a document-store makes it easy to store multiple emails in the same document.

- **High Throughput:** Built for performance, some NoSQL solutions provide a higher throughput than traditional RDBMS. Lai (2009*b*) mentions that Google is able to process 20 petabytes of data per day with its BigTable solution.

- **Avoidance of Object-Relational Mapping:** Using simple data structures is a great benefit, when relational table structures are not needed. For developing a simple application, RDBMS often give you too much and object-data has to be twisted into a relational schema (Lai 2009*a*).

- **Predictable scalability:** NoSQL cluster can easily and cheaply be expanded without switching from a single database to a more complex relational cluster solution. According to Strauch (2011), scaling can be predicted easier, when using a non-relational databases with a simple data model. Within relational models, it is easy to determine table relations, but hard to tell how often tables interact via foreign-key constrains.

- **Version history:** Using MVCC makes it necessary to save multiple database version. This can also be used to determine data manipulations and restore old database states. With the help of timestamps and vector-clocks, Google-Docs and its BigTable storage allow showing a complete modification history. It also allows users to restore older document versions. Within RDBMS version information might be quite difficult to persist for each table and has a large

performance impact (e.g. composite-key with timestamp and an application layer dealing with that).

- **Schema evolution:** When applications grow and/or requirements change, the database schema has to adapt as well. Non-relational databases have a distinct advantage, because they offer more options for how an update can be performed. Changing the schema in relational models with ALTER statements leads to the problem that all records have to be changed and migrated. This makes schema changes a very expensive task, where gigabytes of data have to be written back to disk atomically and in real-time.

  In non-relational databases a schema is enforced in the application layer making it easy to tell which application version matches the corresponding database version. This leaves each entity updated as it is touched and low priority "Gardener" processes can periodically sweep through the data store and update all remaining nodes. Strauss (2009) argues that this leads to more complex code in the short-term, but prevents downtimes in the long-term.

5.4 Disadvantages

- **No standardised query-language:** The benefits of a standardised interface have already been mentioned in subsection 4.1.1. NoSQL solutions have nothing similar, but use many custom languages like HiveQL or AQL. Manipulating data may seem easy without an underlying schema, but there is no easy-to-use standardised interface to do so. There rarely is ad-hoc access to the data, which makes it hard to perform quick fixes on the database layer.
- **Schemaless structure:** Despite the benefits a schemaless storage has, there are drawbacks as well. Inspecting the database gets harder, when structure and meaning of the data is unknown. The database can become invalid as there is no schema enforcing basic validation constrains.
- **Possible security flaws:** NoSQL solutions are not designed to specify and enforce any security-model. Working with highly sensitive data, these storages are vulnerable and leave much room for attacks.

  Typical use-cases operate under the assumption that the web application and database are working in a trusted environment relying on firewalls to restrict access. Using these protection mechanismns is far not enough, as it is possible to use JavaScript and JSON attacks to get access behind firewalls. Often there exists no granular access-control, roles or responsibilities (Software

Group 2013). Data is commonly stored unencrypted with no possibility to enable encryption.

As security becomes a more popular topic for web companies some NoSQL solutions started implementing a better security-model. MongoDB introduced SSL-encryption and more granular access control. According to Software Group (2013), securing data in NoSQL databases can be challenging and needs huge development effort. Implementing this can diminish all the benefits NoSQL stores provide.

- **Lack of experience:** Many companies are unfamiliar with NoSQL solutions and feel not knowledgeable enough finding the correct solution for their problem. Solutions often provide no customer-support and unreliable management tools. This field is rather young and experimental, which makes finding experienced developers hard (Leavitt 2010).

5.5 Types of solutions

According to the NoSQL-archive[2], there exist more than 120 different NoSQL solutions. Instead of going through every solution, the most popular four distinct categories are introduced.

5.5.1 Key-value stores

The idea behind key-value stores is really old, but has been rediscovered by Amazon a few years ago. Amazons Dynamo storage was the starting point for the emergence of multiple new key-value stores like Redis, Riak or MemcachedDB.

It is a large two-column table, each row containing a single unique key pointing to a value. The value can contain more complex structures like sets, lists or hashes, but may be a single string as well. Horizontal partioning and scaling is easy with this kind of structure.

Querying data is only possible by using get-operations and searching through all key fields of the whole dataset. To represent understandable semantics, it is rather important to define good keys from the start. Normally a key consists of multiple

[2]http://nosql-database.org/

values and IDs which are separated by colons. Relations between data-sets are represented this way. The key of a data-set becomes part of another data-sets value. Data-set relations based on these keys are then established in the application layer. Search queries can either be full-text or with string-matching pattern like "Pr?gress".

Advantages

- Simple structures enable scalability onto multiple machines.
- Usually fastest NoSQL solution by holding much data in-memory.

Disadvantages

- Simple structure makes it hard to map complex object-relations and increases application-logic complexity.
- Limited query-ability makes data manipulation hard. As data values can not be searched, these values have to be saved as keys. This leads to much redundant and unreadable data. According to Strauch (2011), the lack of joins and aggregation functions combined with no rich ad-hoc querying interface make performing analytical operations very difficult.
- Multiple operations are necessary to get all needed data. If errors occur during those operations, the whole dataset is at risk.
- Some data-stores like Redis operate only in-memory and make snapshots which are saved to disk in predefined intervals. If a machine crashes between those intervals, the data is irretrievable lost.

Applications

Functionality is very limited for developers, but Decandia et al. (2007) says it is sufficient for some applications needed at Amazon. Key-value stores represent lightweight databases used in applications that have a clear and predefined purpose. When high-availability with data structures unlikely to change after distribution is needed, they seem like a good choice.

Data is most of the time held in-memory. This makes it perfectly suited to act as temporary database cache. Used as query-cache layer in front of a RDBMS, it prevents expensive database queries. Other use-cases are possible as well, so is e.g. DynamoDB [3] used for Amazons shopping cart. Sadalage & Fowler (2012) mention

[3] https://aws.amazon.com/de/dynamodb/

that they are a good fit for storing session- and user-information as well.

They are a bad choice in large datasets, where only small parts are accessed often or when the whole dataset does not fit in-memory (Tiago Macedo 2011).

Redis is used by many popular projects like Github, Stack-Overflow, etc. and stores all data in-memory. Maximum database size is therefore limited by the RAM. Benefits of Redis include its wide acceptance and the large set of existing clients. In contrast to many other key-value databases, it allows structuring data in meaningful semantical ways. Doing so enables data-type specific operations. An example for the structure of a key-value store taken from Naesholm (2012) is listed in Listing 5.1.

```
(customerIds,                    [12, 67])
(customerId:12:name,             Smith)
(customerId:12:location,         Seattle)
(customerId:67:name,             Andersson)
(customerId:67:location,         Stockholm)
(customerId:67:clubCard,         82029)
```

Listing 5.1: Example of a Redis key-value-store

5.5.2 Document stores

Strauch (2011) explains that document databases can be seen as next step of key-value stores. Documents are a set of fields similar to relational tables or objects. Each field defines a key-value pair. In contrast to key-value stores, fields allow nested structures within their values. Popular solutions include MongoDB, CouchDB, and Lotus Notes.

Within each document there is nothing like a schema or set of rules defining what kind of content is allowed. Embracing feedom of choice as one core paradigm, the syntax can be freely chosen and either be JSON, XML or anything else. Data restrictions can be forced on the application layer. A JSON based example including IDs and revisions can be found in Listing 5.2.

There are no relations between documents, which can lead to much redundant data if fast queries are needed. If that is not the case, objects can be referenced with keys as well. This is often better, especially when many-to-many relations should be established. Queries are performed by either using Map-Reduce procedures or provided REST-APIs. Jansen (2010*a*) says that huge joins on multiple tables are not necessary as one document is sufficient to store all the data.

CouchDB, as one of the most often used document stores, does not consider writing conflicts as an exception. Instead they are seen as normal where each application should know how to deal with. They do not use locks during updating as versioning is performed at each write-operation (copy-on-modify). To enforce MVVC explained in section 5.2 a MD5 hash based revision key-value pair is stored in each document. It helps to identify competing data access and prevents wrong writing operations. A new data-set and revision hash is generated each time data changes.

Advantages

- They permit querying data structures more efficiently, as they do not necessarily reply the whole BLOB when a key is requested.
- Easy to understand and use by humans. As the storage format can be freely chosen in many cases, solutions provide great web-compatibility by using JSON.
- Performance is not as good as column-based or key-value based stores, but still very efficient.
- High flexibility is provided as most attributes can be extended and shrunk anytime necessary. This is done without any schema migration effort. The atomic nature of stored data makes document-stores lean and easy to distribute via auto-sharding.

Disadvantages

- No self-defining constraints or triggers are included in the database. Applications have to implement security features like data validation on their own. MongoDB and CouchDB started implementing a simple access-control protocol and user authentication. This promises more security features in the future.
- Implementing large document operations needs more effort than data manipulations with standardised languages like SQL. During updating data, possible redundancies have to be considered and updated as well.

Applications

They are well suited for semi-structured data, where it is hard to tell which data has to be saved beforehand. By just saving necessary data, this can lead to different data entities.

Due to this flexibility and the common usage of JSON documents, they work well in content-management-systems or similar web-facing applications. Often used application parts include e.g. user comments and profiles. Another benefit is the ease in evolving data models without performing expensive database migrations and refactorings (Sadalage & Fowler 2012).

MongoDB, unlike many other NoSQL database solutions, supports ad-hoc queries. To answer them efficiently, a build-in query optimiser is used. Strauch (2011) states that Map-Reduce procedures can be used for easy batch manipulations on the server. This makes document-stores very popular for the usage within real-time analytical applications. It is easy to store page-views and document-parts can be updated without changing the database schema.

```
{
        "_id": "124",
        "_rev": "1-13a34ff65a7ed9b8",
        "first": "John",
        "last": "Doe",
        "age": 39,
        "sex": "M",
        "salary": 70000,
        "registered": true,
        "interests": [ "Reading", "Mountain Biking", "Hacking" ]
}
```

Listing 5.2: Example of a JSON-based data object

5.5.3 Wide-column stores

At first sight column-oriented databases look a lot like relational databases. In practice they are rather different from each other and it is not possible to apply the same principles used in relational databases (Rèmy Frenoy 2013). Figure 5.3 shows that within column oriented databases a record is a set of values corresponding to the same attribute, but different entities.

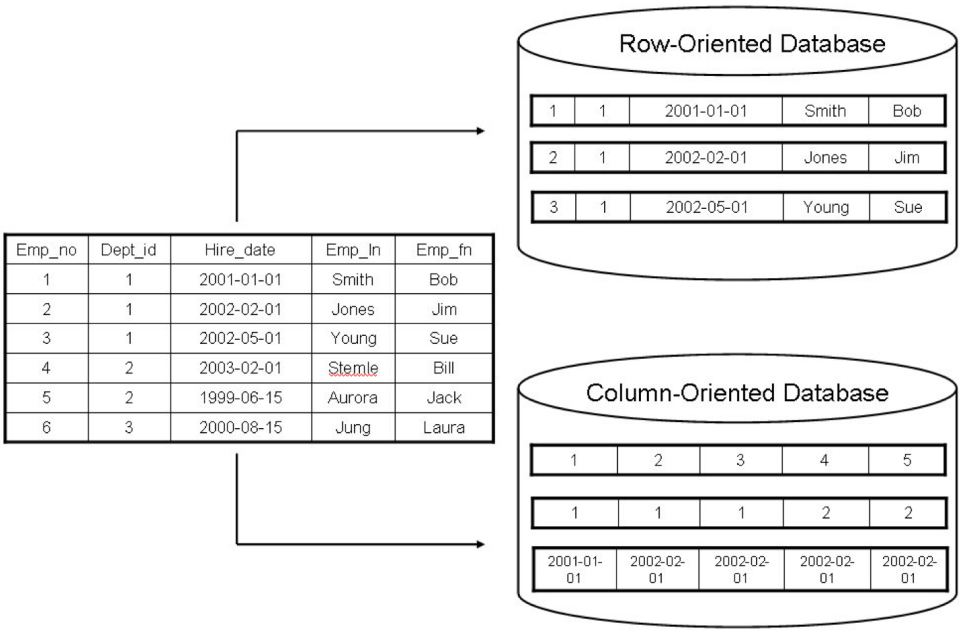


Figure 5.3: Different representations of column- and row-oriented databases (Rèmy Frenoy 2013)

All data is stored in a single entity. This entity can be seen as multidimensional key-value store. Each dataset has its own key, normally having two or three dimensions (row-key, column-key, timestamp). There exist no data-types and everything is simply saved as byte-array. Nevertheless, structures can be created by using the following concepts:

- **Column:** Representing the smallest unit of all existing data containers, it is simply a JSON notioned data-field and timestamp. By specifying a timestamp, each cell is automatically versioned, old copies can be identified and competitive access can be recognised.
- **Super Column:** They can be seen as catalogue or collection of regular columns. The value of a column is a string, where the value of a super column is a map of columns. It must not include other super columns within its structure. An example for nested structures is the super column *Timeline* within the column family *UserTweet* in Figure 5.5.
- **Column families:** Related columns and super columns can be part of column families. It is a collection of rows containing any number of columns. There is no logical structure or restriction for this data. Column families can hold thousands of columns and are identified by keys.

Key	@ayende	
Columns	Location	Vienna
	Name	Ayende Hola
	Profession	Student

Figure 5.4: Column family *User* for multiple records, e.g. Ayende

Key	@ayende		
Data	Timeline	Timeline/00000-000-000-0005	Tweets/00000-000-000-0001
		Timeline/00000-000-000-0006	Tweets/00000-000-000-0002

Figure 5.5: Super column *Timeline* containing a list of columns

With their help it is possible to predefine how data is stored on disk or in-memory. It is the closest thing to tables within all mentioned non-relational data-stores and has to be defined upfront. In contrast to relational databases there just have to be defined two attributes, the name and the sort-key options. A simple example showing the column family *User* is illustrated in Figure 5.4.

Column families heavily influence data distribution. Data within the same column family is saved on the same server and grouped together in-memory and on-disk. By exposing the actual physical model to the users, more efficient usage is possible. Schema design is rather important here, because getting it wrong means that one might have problems retrieving the data in a later stage.

There exist two ways column-stores can be queried. One is by key and the second is by key-range. Data is stored based on the column families sort-order, which is hard to change afterwards.

When two related columns should be combined without key-constraints there is normally used a column family. In the example shown in Figure 5.5 this is done by defining a user as key and the value as all tweets he is associated with. In the super column defining the tweets there are no tweets embedded, but links to the tweet-columns. Retrieving this data means performing multiple queries in the application layer, as one can only query data by key.

Advantages

- Columnar stores can work with petabytes of data and scale very well.
- Reading processes are faster than any other relational or non-relational database. This is because only needed data is read from the database.
- Column-stores are very flexible, because there exist no data-types.
- Queries which include aggregation functions like sum, average, etc. are tremendously faster and outperform relational databases (Abadi & Madden 2008).
- With the query language HiveQL[4] even ad-hoc queries are possible. HiveQL is a query-language similar to SQL, which transforms queries into Map-Reduce procedures. These procedures operate well on wide-column-stores. They are extraordinary fast, when columns are loaded in parallel.
- Since columns are stored separately, entire table columns can be dropped without downing the system.
- Vast improvements are possible via data compression, eliminating the need to store multiple indexes and views.

Disadvantages

- Writing processes affecting multiple columns are slow as multiple tables have to be accessed.
- Restoring objects which have been saved into multiple tables is rather complicated. Relations have to be generated from either nested structures or references within the application layer.
- Typification and relations over multiple tables have to be ensured by the application.
- Converting data sources into the columnar format can be very slow when much data is involved (Rudra 2012).
- It is not well suited for early prototypes, because it is not clear how the query-pattern and column-design may change. Changes are very expensive and slow down development productivity (Sadalage & Fowler 2012).

[4] http://hive.apache.org/

Applications

Most solutions like HBase or Hypertable are inspired by Googles BigTable. They are good for analytical information-systems, because direct data processing is needed. Analytical operations like data-mining and business-reporting make extensive use of well supported aggregation functions. Column-based approaches are good when just portions of the whole data-set are needed. They are slow if the complete dataset has to be retrieved.

In business applications this kind of storage is very useful, because fast calculations can be performed on huge amounts of data. Map-Reduce is a perfect fit when data has to be aggregated efficiently in a distributed environment. This can be very useful when counters for further analytics have to be computed. It is used in read-mostly, read-intensive applications that have large data repositories. Facebook uses Cassandra for its inbox-search within the user-profile. A similar application is event-logging, where reading and aggregating data has to be fast.

5.5.4 Graph databases

Storing graphs within relational databases almost ever ends in large queries containing complex nested joins. Graph databases store this data in a traversable graph representation. Especially designed for use-cases where relations have an important role and navigation between nodes should be easy.

Popular solutions include Neo4j, Pregel or FlockDB, which mostly differ only in their support of different graphs. Different graphs and properties are not always supported to the same extend. Most notable differences are graphs supporting self-loops, directed edges, or multi-edged nodes. Which database to choose largely depends on the usage scenario.

The database model consists of nodes and edges, where nodes represent tuples from RDBMS and edges represent relations between those tuples. Each edge has its own type telling how two nodes are related. In Figure 5.6 each undirected edge is associated with its meaning. Nodes can store inherent information as well. This is done in a key-value based structure without any predefined schema.

Querying data is done with declarative languages like Cypher[5]. Graph paths are traversed with a predefined starting node. Depth-/Breath- first search or heuristic algorithms are used afterwards to walk across the network until the result is found.

[5]http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

Additionally to that, it is possible to use indexes associated with certain edges and/or nodes. This helps finding nodes based on specific attributes without the need to traverse through the whole graph. An index always has a name, type (node, edge) and provider. When e.g. location is an index element, this can help retrieving all nodes associated with that location.

Advantages

- Good performance through natural path traversal instead of join-intensive queries. Using traversal also allows more complex query-operations on the underlying data.
- It is easy to understand and offers a good visual data representation. People can easily see which nodes have been passed during traversal.
- Neo4j has full ACID support, providing the same functionality as RDBMS on the database level (Jansen 2010*b*).
- When the number of edges and nodes gets larger, a graph can be partioned into multiple sub-graphs. Each sub-graph can then be hosted on a different server within a cluster.

Disadvantages

- Searchable attributes have to be indexed, which makes searching for them slow.
- Many different query languages make portability hard.
- It is hard to find the right spot to partion a graph into multiple sub-graphs. There exist heuristic clustering algorithmn to find optimal partitioning spots, but this is still an unsolved mathematical problem.
- Many solutions lack documentation and are not well supported yet (Hughes 2010).
- Not well suited in situations where all entities or a subset of them has to be updated (e.g. analytics). This is because global graph-operations are non trivial (Sadalage & Fowler 2012).

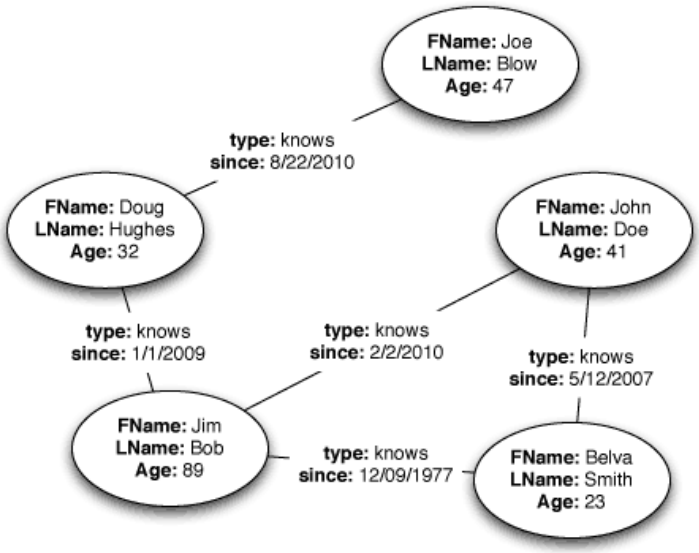


Figure 5.6: Simple graph database containing multiple relations taken from Hughes (2010)

Applications

The most popular graph-database Neo4j is used by Google and Twitter. The core of most projects are underlying networking- or routing-problems. This includes projects with many entity relations and a need for complex queries over those relations. This can be geographic data in traffic networks, where the shortest distance has to be determined. In social networks a possible task can be determining the path between two people.

6 Survey and implementation

After going through various NoSQL data-store types and explaining their individual advantages and disadvantages a real-life scenario with a concrete implementation is evaluated in this chapter. The evaluation criteria include performance, scalability and complexity. The concrete use-case and its requirements are described in section 6.1. An appropriate solution is chosen in section 6.2 and evaluated against the existing approach in chapter 7.

The implemented solution combines the advantages of two different technologies (NoSQL and RDBMS). By building a storage architecture that includes multiple data-stores, different problems can be solved in a single environment. This was first mentioned by Sadalage & Fowler (2012) and is called *Polyglot persistence*. An example how different types of data can be stored within a single e-commerce platform is shown in Figure 6.1.

Inspired by early adopters like Facebook, which use a combined approach of MySQL and NoSQL (Diehl 2011), it is investigated if this makes sense for smaller data-sets.

6.1 Analysis

6.1.1 Structural properties

The evaluated application is a mid-sized web application. Users can write about their physical activities and share training data with their friends. Each user has a personal activity-stream, holding new information in sequential order. The backend data-store is based on a MySQL cluster database holding two data-nodes. The cluster uses built-in replication to ensure that each node always holds the same data. This guarantees all time availability at the cost of complicated replication procedures. Data manipulations are performed with the help of the Doctrine ORM[1] included in the Symfony 2 framework[2].

[1] http://www.doctrine-project.org/
[2] http://symfony.com/

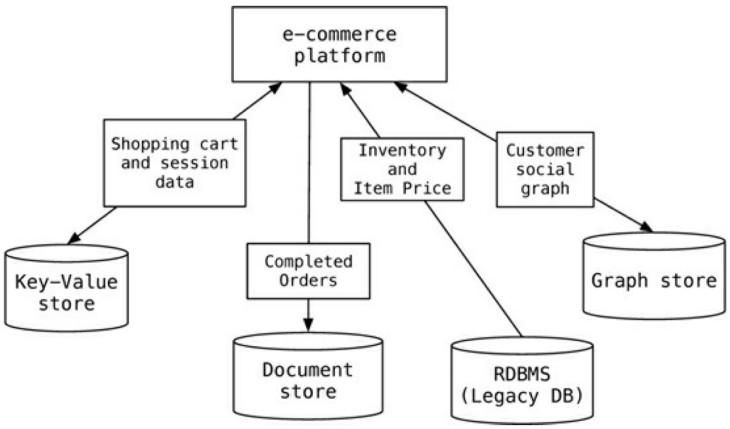


Figure 6.1: An example for polyglot persistent data-stores taken from Sadalage & Fowler (2012)

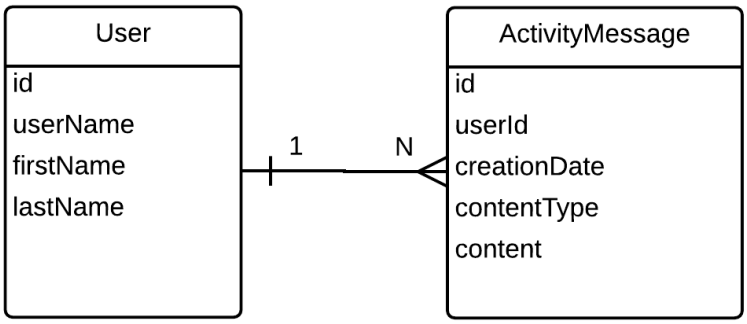


Figure 6.2: MySQL data setup

In this evaluation only data used in the activity-stream is considered. Experience showed that this component gets slower with increasing amounts of data.

Currently there is one *User* table holding about 35000 records and an *ActivityMessage* table holding 8 million records. These two tables are linked together with a 1:N relationship as shown in Figure 6.2. The number of activity messages is growing with each user using the platform. Due to this matter, loading the results gets slower leading to unacceptable response times.

In order to achieve good and reliable results, the properties of our data are explained in more detail:

- **User**
  - Expected average growth is 2000 users/month. This is important to estimate the future dataset-size and is based on existing data using a linear growth model.
  - Data is expected to change slightly, when additional fields are added.
  - This entity is linked to other components within the system and should stay within the relational database. Moving it into a non-relational datastore would mean to rewrite large parts of the current implementation.
- **activityMessage**
  - One user has between 10 and 500 activities stored within the database. This is determined by the users engagement level with the platform and approximately matches a gaussian-distribution.
  - The content-field stores serialised JSON data. MySQL is not able to query this data in a meaningful and easy way. An example how this data looks within a database-table is presented in Table 6.1
  - The data schema is not expected to change in the near future. The data within the content-field is already very diverse. Future activity-types and their content are expected to be different from existing ones as well.
  - Existing messages almost never change and new ones get appended in the database table. This means that update procedures are rare and less important in this usage scenario.
  - Most of the time, the last 50 activity messages of a given user are needed. When accessing this entity, there are around 80% read-operations and 20% write-operations.
  - When new records are saved into the database, they can be written in batches up to 5 records per database flush.
  - Present activity-messages are never searched in full-text, but this may become relevant in the future.

id	uid	creationDate	type	content
1	1	2014-04-28 22:41:43	1	{ "Author": { "AvatarUrl": "http://site.com/avatar.gif", "Name": "Bob", "ProfileUrl": "http://site.com/profile.php", "Username": "admin", "Id": 2100 }, "Title": "Hello", "Body": " Hans posted something." }

Table 6.1: JSON content within the activityMessage table

6.1.2 Requirements

The proposed solution has to fulfill certain criteria to provide greater benefit over the existing solution. These requirements are split into functional and non-functional ones. Functional requirements include functionalities the system must provide, e.g. technical details or data manipulations. Non-functional requirements are often called qualities of a system and include attributes like stability and portability.

Functional requirements

- To provide redundancy, all data has to be stored on at least two machines.
- It should be possible to migrate existing relational data into the non-relational data-store.
- Querying capabilities for the *activityMessages* should be equally good or better than currently offered by MySQL.
- Performance should be significantly better, at best twice as good as the current implementation using MySQL.

Non-functional requirements

- Horizontal scaling should be easy and natively supported by the chosen solution. Distributing large amounts of data should guarantee good performance with high read-loads.

- The database solution should be easy to integrate into the existing environment:
  - Integrability with the PHP-framework Symfony2.
  - Integrability with the current server infrastructure providing two dedicated servers for all operations.
  - Integrability with current deployment procedures done with the help of console commands.
- The system should be easy to maintain and it should take an experienced developer not more than one week to understand all structural changes and how to deal with them.
- Writing application code for the NoSQL solution should be about as much effort as it is now, when using MySQL in the backend.
- Object mappings currently used in the application layer must be retained and it should be possible to integrate them in a meaningful way with the proposed NoSQL data-store. This is especially important to keep maintenance simple when multiple data-stores are used.

6.2 Technology selection

Looking at all non-relational types explained in chapter 5, it is obvious that graph-databases are too application specific for the stated use-case. They also lack documentation and support, which makes them hard to use in a production environment.

Wide-column stores promise good read-scalability and there already exist community based attempts to integrate existing databases like Cassandra into the Symfony2 framework[3]. When using them, JSON data would still reside within a single content-field. As well would later database changes be hard to perform. The most important features wide-column stores offer are fast aggregation functions and their good fit for huge datasets (Kovacs 2014). Considering this use-case, these features are currently not needed and no larger benefit is expected by integrating wide-column-stores.

Key-value stores offer great performance by holding their data in-memory. They are used for frequently accessed data. Activity messages almost never change and are only shown on a single spot throughout the whole application. Keeping all these messages in-memory is not necessary, as only few of them are shown to the user.

[3]https://github.com/MmdBundles/CassandraBundle

Document-based data-stores are often used for web-content where data structures do not fit into a predefined schema. Often mentioned use-cases include form-data or wizard-style metaphors. The content of activity messages looks different with each type and this diversity will continue in the future. Document-based stores offer a great solution for unstructured JSON data like such we are dealing with. They also offer better querying capabilities for unrelational data than MySQL currently does.

The documentation for their most popular successors MongoDB and CouchDB is very good and Doctrine even offers MongoDB[4] and CouchDB[5] ODMs within the Symfony2 framework. Even though these components are not fully tested yet and still in an early development state, they look very promising and well supported.

Kahwe (2010) explains that using an ODM for document-based data stores makes sense because of the following points:

- Model classes provide some sort of schema. This abstraction layer provides data managment capabilities on application level.
- Model classes provide structural validation constraints, which are missing on database level.
- For data migration within non-relational databases, old data does not have to be migrated. Doctrine introduced a concept called *Eventual migration*. It is based on the idea that rules within the model define how old data is read from the application and written back to the database. Data is stored in the new format after it has been accessed for the very first time. This means that data migration occurs just-in-time and not immediately after all structural changes have been deployed.
- ODMs provide high-level access to bulk-operations and explicit support for document-conflict-resolution.

When looking back at Figure 3.1, it becomes clear that the core difference between MongoDB and CouchDB in the CAP-theorem space is that MongoDB drops some of its performance to provide better consistency. As both solutions look promising, a closer look is taken on their individual advantages and disadvantages. A general comparison can be found in Figure 6.3.

[4]https://github.com/doctrine/mongodb-odm
[5]https://github.com/doctrine/couchdb-odm

	Couchdb	Mongodb
Data Model	Document-Oriented (JSON)	Document-Oriented (BSON)
Interface	HTTP/REST	Custom protocol over TCP/IP
Object Storage	Database contains Documents	Database contains Collections Collections contains Documents
Query Method	Map/Reduce (javascript + others) creating Views + Range queries	Map/Reduce (javascript) creating Collections + Object-Based query language
Replication	Master-Master with custom conflict resolution functions	Master-Slave
Concurrency	MVCC (Multi Version Concurrency Control)	Update in-place
Written In	Erlang	C++

Figure 6.3: Comparision of MongoDB and CouchDB

6.2.1 MongoDB

A major goal of MongoDB is the achievement of great performance. By adding dynamic padding to its documents and preallocating data files, it gives up disk space to achieve good and consistent performance. To achieve good performance, it is important to use the right configuration for the application. Then it can be even faster than CouchDB.

When the system has plenty of RAM, often used queries and indexes are cached to provide fast data-access. Generic secondary indexes allow fast queries and provide full-text indexing capabilities (Kristina Chodorow 2010). 10gen[6] recommends to keep the working-set always in RAM. The working-set is the most frequently accessed data. This can either be the whole data-set or a part of it. When data beside the working-set is accessed within MongoDB, page-faults will occur and the system will be tremendously slower.

By offering an easy-to-use SQL-like query language, people coming from relational databases adopt quickly to the new environment (Kovacs 2014). MongoDB supports an aggregation pipeline that allows building complex aggregations from simple database optimised pieces.

Summarising all this functionality shows that it functions more like a RDBMS than most other NoSQL solutions do. This makes adoption easy and guarantees a short learning curve for developers. As Market (2014) puts it, MongoDB is easy to use and fast to adapt due to its wide support. He also says that the community is very active, good documentation is existing and commercial support is available as well.

MongoDB stores its data in BSON[7], which is a binary-encoded serialisation of JSON documents. En- and Decoding is fast and by adding extra information to its documents it provides easy document traversal. By making use of this format, MongoDB can build index-keys or simply match serialised objects against a query (Brown 2012).

Market (2014) mentions some problems and trade-offs they had to take when shifting from MySQL to MongoDB:

- **Lock Contention:** Database locking within MongoDB is done globally and will cause problems when large amounts of requests have to be served. Possible solutions make the system more complex, but better working:

[6] The company behind MongoDB
[7] http://bsonspec.org/

  - One machine using multiple MongoDB instances and multiple databases provides significantly higher throughput.
  - Tonytam (2010) notes that querying records before issuing an update gets the object into RAM and increases update speed.

- **Data consistency/durability and performance:** Read and write concerns have to be lowered to provide high performance. This leads to lower consistency and data loss in the long run. To react quickly, comprehensive monitoring support has to be implemented.
- **More normalized data and fewer network round trips:** When data within the documents is highly normalised, the client application has to issue more queries to fetch data from different collections. When scattering the same data into multiple collections, additional disk space is wasted and data inconsistencies occur more frequently.
- **Giving up multi-document or multi-collection transactions:** While using write-concerns and read preferences can mitigate some of the data consistency and durability problems, MongoDB is still not using atomic updates. This makes it inappropriate for critical data which needs transactions.

Consistency

Sirer (2013) claims that MongoDB does not provide any reliable consistency as data-writes are not even written to one of MongoDBs replications. They are just written into an outgoing socket buffer on the client host until MongoDB runs an asynchronous write. When any of the clients crashes, which is considered normal in todays environments, no written data will reach any of MongoDBs replications.

Kristina Chodorow (2010) describes this setting as unacknowledged write, which can be set within each MongoDB configuration. There can be used different write-concerns, defining how safe a write should be stored before continuation.

Unacknowledged writes do not tell if a write succeded, but acknowledged writes do. In general, applications should use acknowledged writes. Only if you are dealing with low-value data (e.g. logs), you may not want to wait for a response until you proceed with your application code.

To ensure that a write has been propagated to n-replicates, MongoDB provides the getLastError() API call. Using this call for ensuring successful writes doubles the cost for each write operation. Additional problems with this call include the lack of multi-threading support (Sirer 2013).

MongoDB documentation says, instead of using getLastError() manually, one should set write-concerns. As a first step, it must be defined how consistent application-data needs to be. This is, because MongoDB supports a huge variety of consistency levels.

To ensure that writes will always be persisted, the majority setting should be used. It guarantees that each write propagates to a majority of the replica-set. If such a strong consistency setting is not needed, it is also possible to specify the number of replicas to which the data should be propagated. The problem in using lower consistency settings is that data might not be available on the next read-request.

Sometimes replica-sets have more complex requirements, as you may want to make sure that a write makes it to at least one server in each data-center. Replica-sets allow to create such custom rules in various combinations (Kristina Chodorow 2010).

When unacknowledged writes are used, it is possible that writes are fired faster than MongoDB it is able to process them. These writes will pile up in the operating systems socket-buffer are written to disk on the next batch-write. By default, MongoDB writes to the disk-journal every 100 ms. This means that not each write-operation is flushed to disk immediately. When using this default setting, it is possible to lose up to 100 ms of writes in the event of a crash.

It generally does not take this long to flush writes to disk, which makes using batch-operations very useful. Committing writes to disk should therefore only be used for very important writes.

Replication

MongoDB is working with a single primary node and multiple secondary nodes to provide failure resistence. The same data is hold on multiple servers where each keeps an oplog of all write-operations the primary performs. This log is stored within a collection on the primary and queried by the secondaries for replication.

Each secondary maintains its own oplog recording each operation it replicates from the primary. This allows any member to be used as a sync source for any other member. If a secondary goes down, it will start syncing from the last operation in its oplog after a restart. As operations are applied on the data and written to the oplog afterwards, the secondary may replay operations that it has already performed.

To ensure that each node within a replica-set knows about the other nodes, members send out heartbeat requests to all other nodes every two seconds. If a primary node

can no longer reach a majority of the replica-set, it will demote itself and become a secondary node. If one node can not reach the primary node, it will seek an election with all other reachable nodes (Kristina Chodorow 2010).

Sharding

MongoDB supports auto-sharding to simplify administrative overhead and decouple the application architecture from the code.

Deciding when to start sharding is a hard decision to make. Sharding too early adds operational complexity to the deployment and needs design decisions that are difficult to change later. On the other side, waiting too long makes it hard to shard an overloaded system without downtime. To find the perfect time to start sharding, various metrics like available memory and disk space have to be monitored.

When shards are added, performance is increasing roughly linearly per shard. Usually there is a performance drop if you move from a non-sharded system to just a few shards. Due to the overhead of moving data, maintaining metadata and additional routing, a small numbers of shards will generally result in higher latency and lower throughput than a non-sharded system. Sharding therefore only makes sense with at least three or more shards. According to 10gen (2014*a*) one should start with an non-sharded deployment and extend MongoDB when the dataset grows.

When using a sharded cluster, it is impossible to perfectly back it up while it is active. This limitation is generally negligible when the cluster is growing and a complete backup is almost never needed. Instead smaller pieces of the system are backed-up individually (Kristina Chodorow 2010).

Kingsbury (2013) notes that backup-problems, different write-concern settings and its architecture make MongoDB complicated to administer and maintain correctly. Nevertheless, MongoDB is developing and evolving in astonishing speed to fulfill current and future customer demands (Edlich 2012).

6.2.2 CouchDB

CouchDB has a flexible and easy-to-learn document-based structure where data is received from a REST API and stored as JSON. The REST API provides an unified interface and allows people to use CouchDB without any platform specific drivers.

A core philosophy in this project is that problems should be rare and they should be easy to find (Brown 2012). CouchDB is therefore build to be fault-tolerant and includes a very structured versioning system. Each time an attribute is updated, a new document gets uploaded to the data-store and a new revision is created for this particular document. This makes sense for CouchDB, but creates additional overhead for all database operations. Network traffic also increases as untouched attributes are sent to the server as well.

The used data-store uses a B-tree storage engine for all internal data, documents and views. It is quick for retrieving single keys and key ranges. To improve single and range-based key-lookups, a view model allows to write key-value pairs directly into the B-tree storage engine (Brown 2012).

Performing queries in CouchDB is slightly more complicated than in MongoDB. Instead of using a query language like SQL, the user needs to write more complicated Map-Reduce jobs. In the explained use-case, Doctrine encapsulates easy operations with its own querying capabilities[8]. If more complicated data operations are needed, it is still possible to create Map-Reduce scripts or views[9]. CouchDB does not include full-text search capabilities. To take advantage of such functionality, one has to use a text-search engine on top of it. CouchDB is often used in combination with Lucene to provide this kind of functionality[10].

Using MVCC for database updates prevents database locking. Revisions make it possible to use an old record while the new one is written. When using relational databases under high load, it can take longer figuring out who is allowed to do what than doing the actual work. With the help of revisions, throughput can be higher than with traditional RDBMS. A visual comparison of both mechanisms can be found in Figure 6.4.

When working with CouchDB, Hazel (2012) encountered some problems which lead to additional administrative overhead. This includes reindexing problems for views or even broken views which prevent all other views from working. He also needed additional monitoring-systems and had to periodically run scripts to keep availability and performance high.

[8] http://doctrine-couchdb.readthedocs.org/en/latest/reference/working-with-objects.html
[9] http://doctrine-couchdb.readthedocs.org/en/latest/reference/map-reduce-queries.html
[10] https://github.com/rnewson/couchdb-lucene

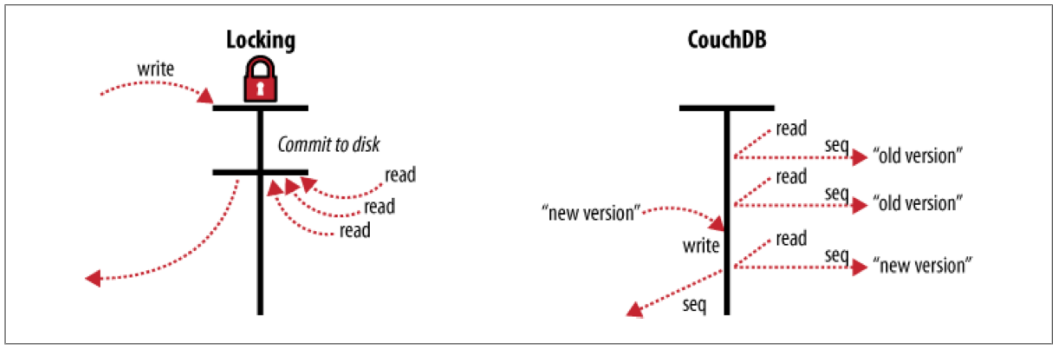


Figure 6.4: CouchDB MVCC compared to traditional locking mechanisms (Brown 2012)

Performance

CouchDB is designed for varying traffic without failing over (J. Chris Anderson, Jan Lehnardt 2010). To provide great performance, enough RAM is important as CouchDB makes heavy use of filesystem caches. It is possible to install CouchDB on basically any device (e.g. mobile phones) and sync databases with a centralised CouchDB data-store. This creates fast data retrieval and good user-feedback.

Similar to MongoDB, it is possible to avoid indexing and disk-syncing overhead associated with individual document writes. With a certain option enabled, CouchDB can build up document batches in memory and flush them to disk, when a certain threshold is reached. By default, CouchDB flushes the in-memory updates once per second to keep data loss minimal. To reflect the reduced integrity guarantees when batch-updates are used the HTTP response code is 202 - Accepted instead of 201 - Created.

The ideal use for batch-mode are logging-type applications. Many distributed writers store discrete events to CouchDB, where rarely losing a few updates is worth the increased storage throughput.

The probably most important configuration setting for increased performance is *delayed commits*. It allows to run operations against the disk without an explicit fsync after each operation. This setting is disabled by default and should stay like that, unless you can effort losing some durability to raise performance (J. Chris Anderson, Jan Lehnardt 2010).

Riyad Kalla, John Lynch (2014) claim that with high durability constraints enabled, CouchDB is more resilient than MongoDB can every be. According to them, it is very efficient when it comes to fsync'ing and constantly recopying portions of the data.

Concurrency plays an important role in CouchDB. It behaves differently when there are 100 or 1000 clients accessing the database at the same time. Erlang, the language CouchDB is written in, tends to perform better under high-load.

Replication

CouchDB supports fast peer-to-peer replication and sync between remote CouchDB databases. If you have an application with various synced databases, where each can go off- and online at any time, CouchDB can handle a lot of low level conflict resolution for you (Riyad Kalla, John Lynch 2014). In contrast, MongoDB has no comparable peer-to-peer replication and uses more conventional master-slave redundancy mechanisms with automatic failover.

CouchDB operations take place within the context of a single document. This avoids that two database nodes have to be in constant communication. Eventual consistency is achieved by incremental replication where document changes are periodically copied between servers. This shared-nothing architecture creates independent, self-sufficient nodes within the system (J. Chris Anderson, Jan Lehnardt 2010).

When the same document has been changed in two different databases, the build-in replication system comes with automatic conflict detection and resolution. The conflicting document is flagged as in-conflict, the losing version is saved in the document history and the winning version is saved as the most recent version in the database. Both database run a deterministic algorithm leading to the same winner. Later manual change is always possible by reverting or merging with the older version of the document (J. Chris Anderson, Jan Lehnardt 2010).

Sharding

As CouchDB on his own does not support sharding, applications often end up using BigCouch[11]. It is an auto-sharded and replicated version of CouchDB. According to one announcement of Apache, it should get incorporated into the CouchDB source-code by January 2013. Even though the repositories have been merged, its functionality is still not available in the newest CouchDB version (Slater 2013). This shows that CouchDB support is currently not as reliable as MongoDBs.

[11] http://bigcouch.cloudant.com/

Riyad Kalla, John Lynch (2014) say that a sharded CouchDB database is simpler to scale than MongoDB. In terms of scalability Foundation (2014) notes that with up to tens of thousands of documents you will generally find CouchDB to perform well no matter how you write your code. Once you get into the millions of documents, you need to be a lot more careful.

6.2.3 Architectural integration

Shifting the data schema from a relational model to a document-based model normally starts with the schema design process. Data should be modelled in a way that takes advantage of the document models flexibility.

A general pattern is that parent-child relationships are embedded into a single document. An example for embedded activity messages within an user document is shown in Listing 6.1. If relations are more complicated, it makes sense to split-up data into multiple documents. They are then linked as sub-documents. This can be a user document as in Listing 6.3 and a separate activity-messages document like in Listing 6.2.

```
{
   "_id": ObjectId("5423be4448177eed658b494e"),
   "userName": "username1",
   "firstName": "firstname1",
   "lastName": "lastname1",
   "activityMessages": [
     {
         "contentType": "0",
         "content": "[\"widget\": [\"title\": \"Sample activity message\"]",
         "creationDate": ISODate("2014-09-25T07:03:54.0Z")
    },
    {
         "contentType": "1",
         "content": "[\"widget\": [\"title\": \"Sample activity message 2\"]",
         "creationDate": ISODate("2014-09-25T07:04:54.0Z")
    }
  ]
}
```

Listing 6.1: User document with embedded activity messages in MongoDB

```
{
   "_id": ObjectId("5423be4548177eed658b4978"),
   "contentType": "0",
   "content": "[\"widget\": [\"title\": \"Sample activity message\"]",
   "creationDate": ISODate("2014-09-25T07:03:54.0Z")
}
```

Listing 6.2: ActivityMessage document in MongoDB

```
{
   "_id": ObjectId("5423be4448177eed658b494e"),
   "userName": "username1",
   "firstName": "firstname1",
   "lastName": "lastname1",
   "activityMessages": [
     {
       "$ref": "MongoActivityMessage",
       "$id": ObjectId("5423be4548177eed658b4977"),
       "$db": "project"
    },
    {
       "$ref": "MongoActivityMessage",
       "$id": ObjectId("5423be4548177eed658b4978"),
       "$db": "project"
    }
  ]
}
```

Listing 6.3: User document with references to ActivityMessage documents in MongoDB

Hadjigeorgiou (2013) claims that MongoDB allows faster transactions when dealing with embedded data compared to finding data in other collections. This is because the binary format used for internal representation is optimised for queries in a single collection. Additional advantages of using embedding over linking are (10gen 2014*a*):

- An aggregated document can be accessed with a single database call rather than joining multiple documents together.
- As documents are self-contained, splitting them across multiple locations becomes easier. This enables good horizontal scalability while sharding.

When deciding if documents should be embedded, the following criteria should be considered:

- Embedding documents makes sense when there is a 1:1 or 1:N relationship. The core criteria is that one document is always viewed in the context of its parent.

- Fields that are modified together atomically should be embedded.

According to 10gen (2014*a*) linking should be preferred over embedding, when the following is the case:

- The parent document is accessed often, but its child is rarely needed. Embedding would lead to a higher unneeded memory footprint.
- The document size exceeds MongoDBs document limit.
- One part of the document is frequently updated and constantly growing, while the rest is relatively static.
- References make sense when one document would otherwise be embedded in multiple other documents. Embedding takes up more space and syncing data is harder.

All this conclusions are based on the assumption that there is used a document-based store only. The decision between linking and embedding becomes more complicated when multiple data-stores are used at the same time.

Figure 6.5 summarises previous considerations regarding the decision for a good schema design. Activity messages for a certain user are expected to change often, which makes references look like the better solution. Even through embedding brings benefits while sharding, this is not yet used in the evaluated mid-sized dataset. From an object-oriented view-point, activity messages clearly belong to the user object, but they are rarely needed when the user object is needed. They are just shown in one single view and the user-document has to be transferred to the client much more often.

If activity messages are embedded in a user-document, the user-entity residing in the relational database has to be migrated into the document-based store as well. This is unfeasible in this use-case, as the user entity has too many further dependencies within the system.

The unnecessary transaction overhead and complications while integration lead to the decision of referencing activity messages even though they are in a 1:N relationship with the user.

Combining a relational database with a document based one brings the challenge of syncronization. Doctrine (2014) provides an example how this can be achieved on the application layer. In this example, the ID of the ODM document is referenced within the ORM entity object. Based on this example, we created convenient access to all activity messages a user has.

Embedding is better for...	References are better for...
Small subdocuments	Large subdocuments
Data that does not change regularly	Volatile data
When eventual consistency is acceptable	When immediate consistency is necessary
Documents that grow by a small amount	Documents that grow a large amount
Data that you'll often need to perform a second query to fetch	Data that you'll often exclude from the results
Fast reads	Fast writes

Figure 6.5: Comparision of embedded and referenced documents taken from Kristina Chodorow (2010)

To achieve a convenient data-store mapping, Table 6.2 shows a *MongoActivityMessagesId* field added to the *User* entity. This value is linking to the ObjectId within the *activityMessages* document of the non-relational data-store. As shown in Listing 6.4, all activity messages are contained within there.

Id	UserName	FirstName	LastName	MongoActivityMessagesId
1	JacksonL	John	Hanger	5459cd8548177ec8538b462d
2	SingStar11	Arianne	Schmidt	6559c93648177ec74c8b4570

Table 6.2: User entity within MySQL

```
{
   "_id": ObjectId("5459cd8548177ec8538b462d"),
   "activityMessages": [
     {
       "$ref": "MongoActivityMessage",
       "$id": ObjectId("5459cd8548177ec8538b462e"),
       "$db": "project"
    },
     {
       "$ref": "MongoActivityMessage",
       "$id": ObjectId("5459cd8548177ec8538b462f"),
       "$db": "project"
    }
  ]
}
```

Listing 6.4: ActivityMessages document in MongoDB

Data migration

Migrating existing data from a relational database to a document-based store is a feasible task, because both data-stores build upon similar structures. Table 6.3 represents a simple mapping, showing data-structures both systems are using. These structures act as an orientation when the migration process is planned. Depending on the complexity of the database, there are multiple ways of data migration:

- Using auto-migration tools like Mongify[12] or Techgene[13].
- Using an ORM and ODM within the migration scripts and perform migrations on the application layer. For this process, the development framework needs to provide data connections to both databases and conversion logic within the migration script.
- Some document based databases provide JSON import functionality that can be used for data migration. This allows data exports independent from the new platforms programming language. The only requirement is that the old platform supports some automatic or manual JSON export.

When performing database migrations on a running system, one should perform them in a step-wise incremental migration process. This describes a paradigm of moving little data-chunks sequentially from one data-store to another. Incremental migration eliminates problems with service availability and allows easy fall-back to the relational database if problems occur. The disadvantage of this incremental data transfer is the higher administrational effort and the fact that both databases need to run in parallel (10gen 2014*a*).

RDBMS	**MongoDB**
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Table JOIN	Embedded document or linking

Table 6.3: RDBMS and MongoDB concept comparision (10gen 2014*b*)

[12]http://mongify.com/
[13]http://www.techgene.com/services/data-migrator/

7 Empirical analysis

To determine advantages and disadvantages beside those revealed in the theoretical evaluation of the previous chapters, this chapter is all about a practical evaluation of different datastore characteristics.

The use-case and its architectural integration have already been discussed in section 6.1 and subsection 6.2.3. The test environment and environmental conditions that need to be taken into account for result interpretation are explained in greater detail now.

The evaluations performed in this chapter use the following software components:

- Debian GNU/Linux 7.6 (Wheezy)
- Kernel Linux 3.2.0-4-amd64
- Apache 2.2.22
- PHP 5.4.4-14
- MySQL Cluster NDB 7.0.36
  - query_cache_limit = 1024MB (default)
  - query_cache_type = ON (default)
  - query_cache_size = 0 (default)
- MongoDB 2.6.4
  - storage.journal.enabled = true (default)
  - storage.journal.commitIntervalMs = 100 (default)
  - storage.syncPeriodSecs = 60 (default)
  - slaveOk = false (default)
- CouchDB 1.6.1

The underlying hardware consists of:

- 64bit OS
- 1x Intel(R) Core(TM) i7 CPU, 2.7GHz
- 2.3 GB RAM
- Intel 520 SSD hard drive, Read 550MB/s, Write 520MB/s

On the application layer following software components are used:

- Symfony 2.5.6
- Doctrine ORM 2.2
- Doctrine MongoDB ODM 1.0-Beta11
- Doctrine CouchDB ODM 1.0-Alpha2

Due to this setting, network latency can be ruled out as an influential factor in all performed test-cases. This is because all operations are performed on the same machine. Instead, additional latency from conflicting file-system operations can influence the results negatively. This is expected to be about the same for all used technologies and is therefore neglected.

In this evaluation, MongoDB is used with the most secure write-concern settings available. This allows better comparison between performance and scalability measurements on approximately the same consistency level MySQL cluster and CouchDB offer.

7.1 Performance

According to J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber (2007), performance testing is done to address risks related to expenses, opportunity costs, continuity and/or corporate reputation. It enables to predict performance characteristics of an application in production. Based on those predictions, prospective performance concerns can be addressed.

Performance testing makes it possible to either evaluate the current capacity and stability or to perform benchmarks on different application configurations. Predicting the likelihood of user dissatisfaction or revenue losses are typical use-cases covered by such evaluations.

NoSQL performance evaluations often lack standardised tools and good testing protocols that explain what is actually tested. As applications and technical setups

strongly diverge, it is hard to find a common evaluation framework. The closest attempt in evaluating different data-stores against each other has been made by Yahoo and its YCSB[1] benchmark. They are using different workload scenarios representing common business cases. Different workloads include e.g. Balanced-Read/Write or Heavy-Read load.

YCSB is very general and does not provide a full-stack test-framework that includes our application/server-setup (e.g. combining multiple data-stores). Within this section, database operation performance is evaluated. A test-configuration more similar to YCSB can be found in subsection 7.1.2.

7.1.1 Database performance testing

Configuration

The Debian machine used in this evaluation has all databases installed, but only those activated which are needed for the selected test-scenario. All disks were over provisioned, leaving at least 21% of disk space empty. This is consistent with recommendations made by 10gen.

The MySQL cluster architecture is depicted in Figure 7.1. Separate illustrations for MongoDB and CouchDB can be found in Figure 7.2 and Figure 7.3. In test-setups where multiple data-stores are used, mentioned architectures are combined on application level.

For measuring datastore operation times in the evaluated test-scenarios, each database offers its own measurement methods. MySQL can log queries that take longer than a predefined amount of time in the slow-query-log or system database. Using this approach affects database performance and distorts the received performance results.

MongoDB records query times in a similar manner, but results are stored in a special system profile database. According to Hadjigeorgiou (2013), this method is inadequate for measuring performance due to its impact on performance.

To achieve comparable test results, time is measured on the application layer with the help of the PHP microtime-function[2]. The measured operation times are written to disk after successful operation to ensure that no performance impact influences the results.

[1] http://labs.yahoo.com/news/yahoo-cloud-serving-benchmark/
[2] http://php.net/manual/en/function.microtime.php

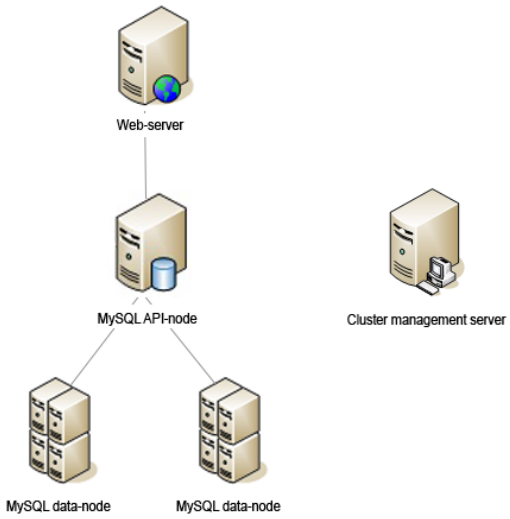


Figure 7.1: MySQL cluster architecture

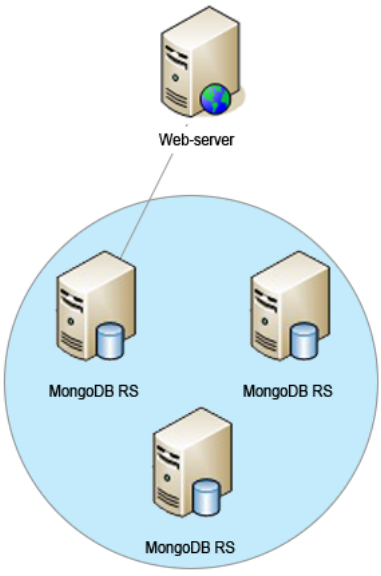


Figure 7.2: MongoDB architecture using 3 replica-sets

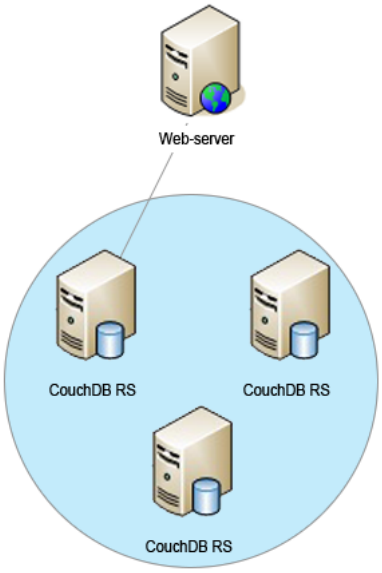


Figure 7.3: CouchDB architecture using 3 replica-sets

Results

All upcoming figures in this section are split into multiple sub-graphs. Each sub-graph represents a different number of users for which statistics have been generated. The throughput is defined by the number of activities created for a certain user. All performed operations use the relational *activityMessage* table defined in Table 6.1 or the document defined in Listing 6.2.

When combined data-stores like MySQL and MongoDB are used, the *User* object is always saved within MySQL and *activityMessages* are stored in the document-based database. The implemented data-store combination is equivalent to the one explained in subsection 6.2.3.

Different MongoDB write-concern settings evaluating the same characteristics have been included as well. Unacknowledged writes define the lowest possible setting in MongoDB and return a successful operation message right after data has been sent to the server. Using the majority write-concern setting guarantees that the written data is present on the majority of nodes.

Insertion performance For the performance evaluation of insert-operations, two measures have been taken. Figure 7.4 shows the performance of batch operations on

different data-stores. Additionally can performance statistics for single-record creation be found in Figure 7.5.

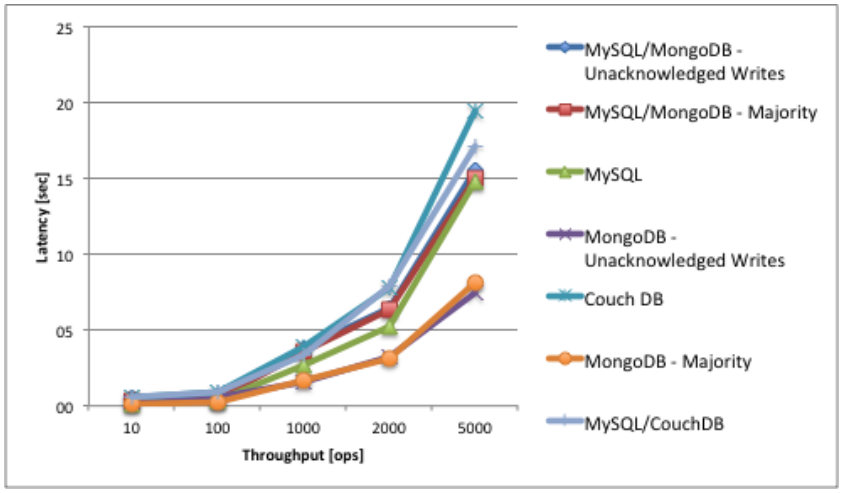
Having a closer look at Figure 7.4, makes clear that record-insertion performance for the combination of MySQL and MongoDB get worse with increasing throughput. Even the lowered write-concern setting in MongoDB does not largely improve this behaviour.

This can be the case if the infrastructure is not providing enough resources for running both, MySQL cluster and MongoDB. It can also mean that the linking of the *activityMessages* document to the *User* table creates too much additional overhead in MongoDB. This is possible as *activityMessages* are hold in a separate document which has to be fetched and updated each time an *activityMessage* is added.

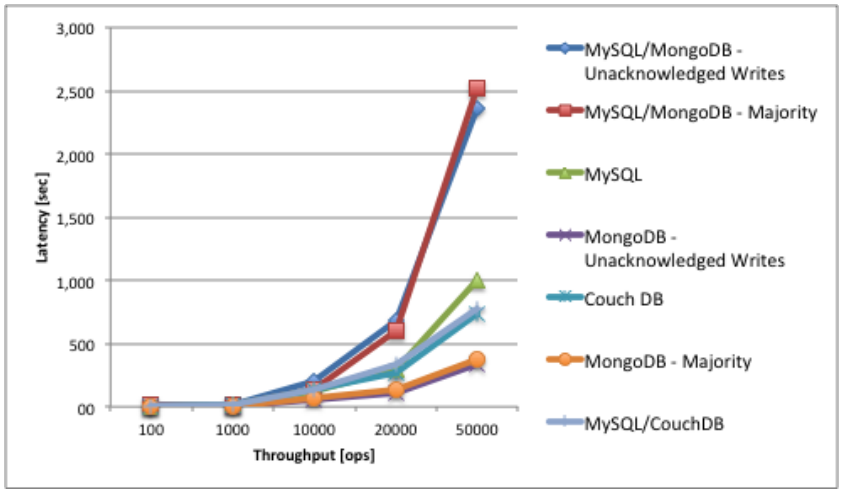
The results show that MongoDB on his own performs far better than in combination with MySQL. As the same structures are used and the amount of memory is limited on the testing machine, it is more likely that the bad performance of the data-store combination is due to a lack of computation power.

With increasing amounts of data, it becomes clear that document-based stores are offering slightly better insertion performance. This difference is not significant and probably diminishes when a MySQL expert configures the used database. The performance between MongoDB and CouchDB is about equal, but it seems that CouchDB becomes more performant with increasing amounts of data.

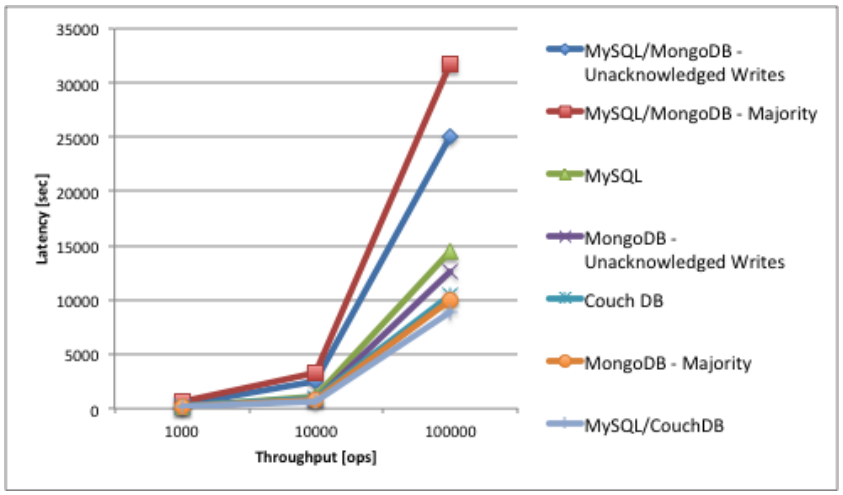
Single insertion operations in Figure 7.5 show similar performance characteristics as the batch operations described before. The only notable difference is that CouchDB does not seem to become more performant with increasing amounts of data.



(a)10 users

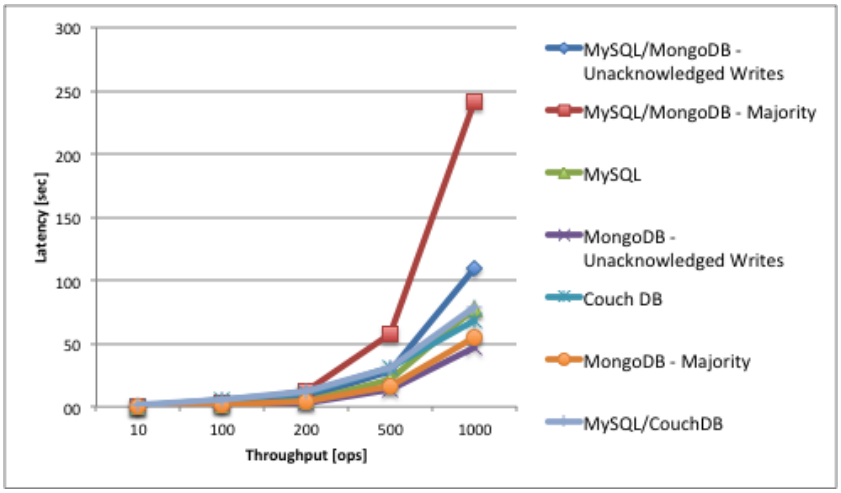


(b)100 users

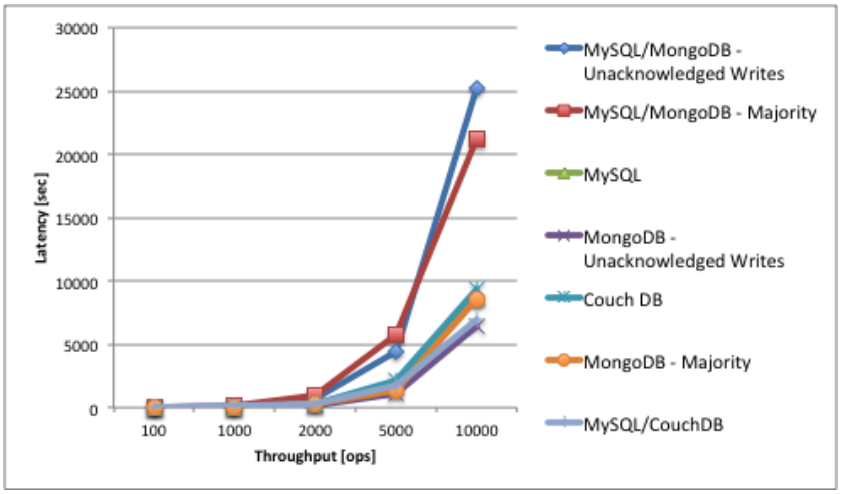


(c)1000 users

Figure 7.4: Insert batch operation performance



(a)10 users

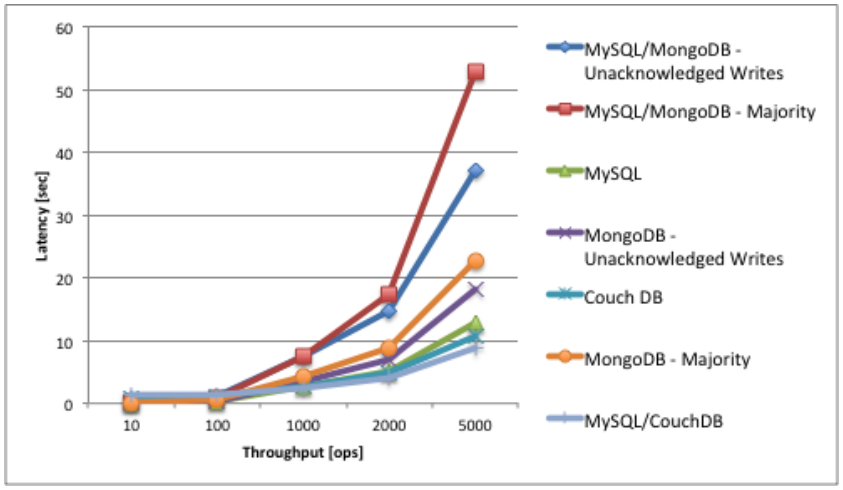


(b)100 users

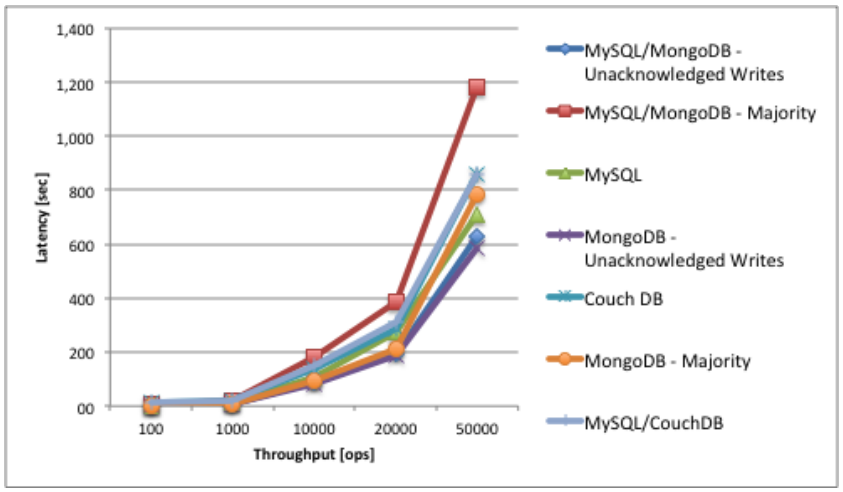
Figure 7.5: Insert operation performance

Update performance Batch update procedures in Figure 7.6 show similar characteristics as those already discovered in the evaluation of batch insertion operations. MySQL in combination with MongoDB is the slowest, but this time it is not as far behind the others when the amount of written data increases. The difference between the unacknowledged and majority write-concern setting is also much higher when the number of operations increases.

When single data operations are used, Figure 7.7 shows that MongoDB performs very well and even the combination with MySQL achieves better result than MySQL alone. CouchDB seems to become slower with increasing load in both updating procedures.

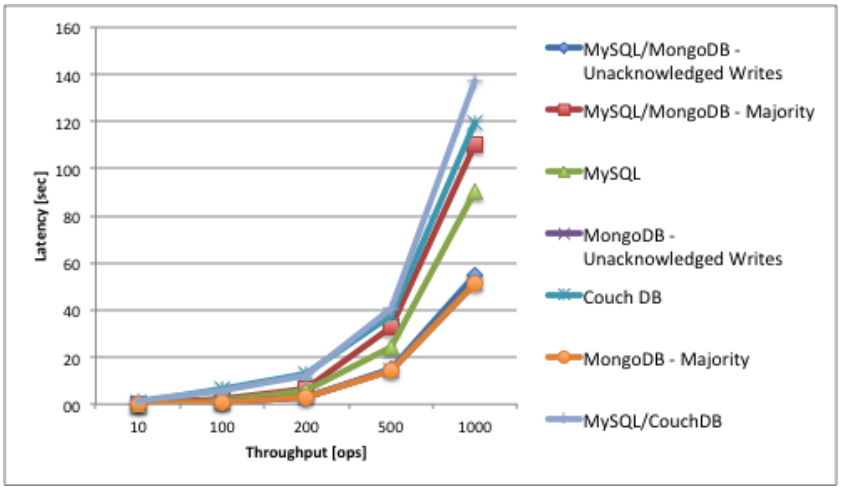


(a)10 users

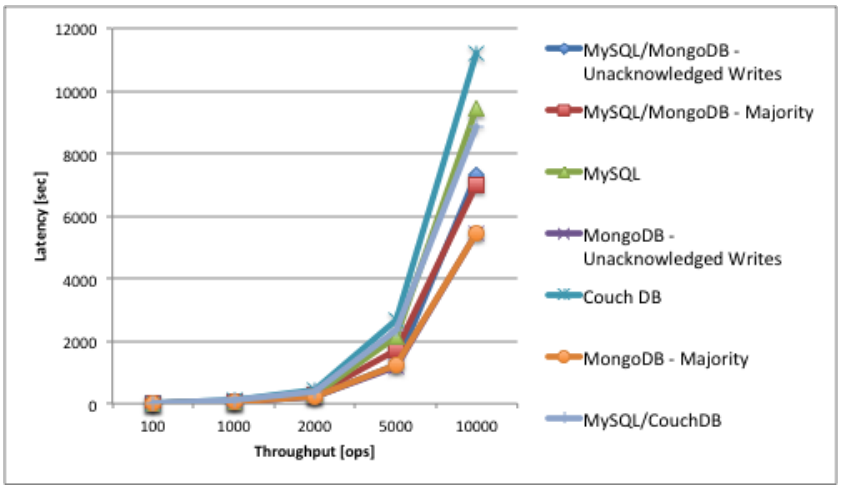


(b)100 users

Figure 7.6: Update batch operation performance



(a)10 users

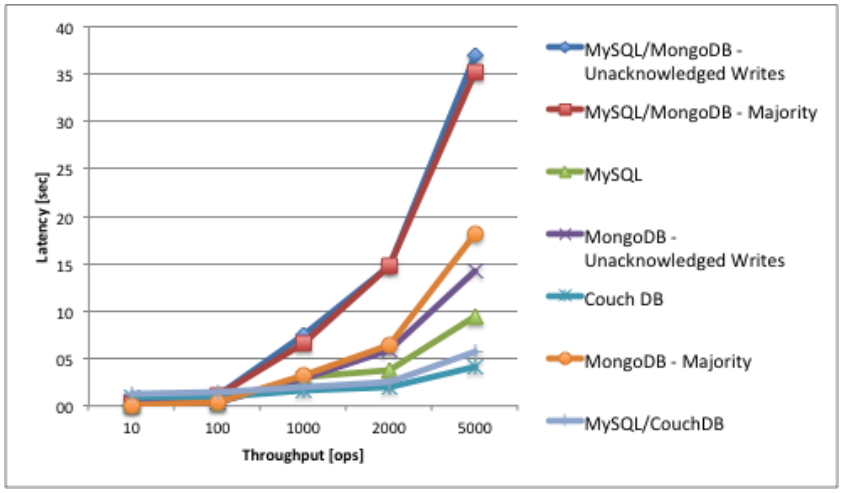


(b)100 users

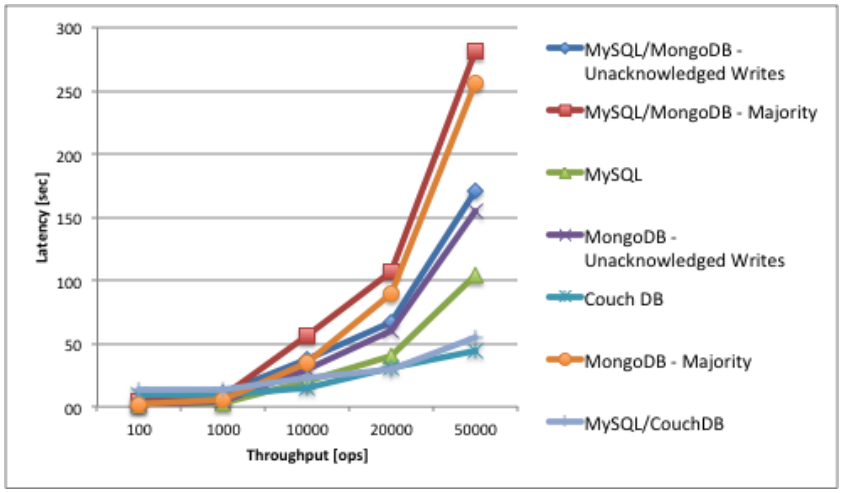
Figure 7.7: Update operation performance

Deletion performance When data-records are removed from the data-store Figure 7.8 shows that MongoDB has a really bad performance if batch operations are used. It is far behind MySQL, which performs best with increasing load. Interestingly MongoDB combined with MySQL also achieved better results than only MongoDB.

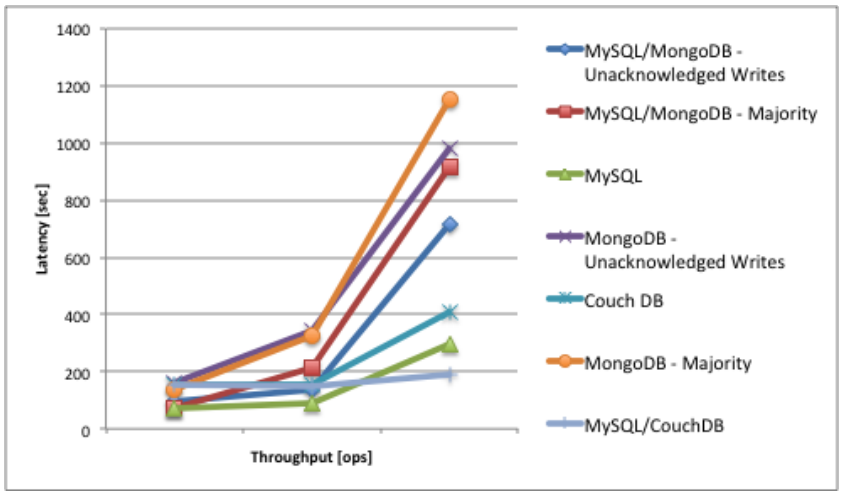
Single removals seem to work better with MongoDB, especially with a lower write-concern setting. With this setting its performance is equal to MySQL and far better than CouchDB.



(a)10 users

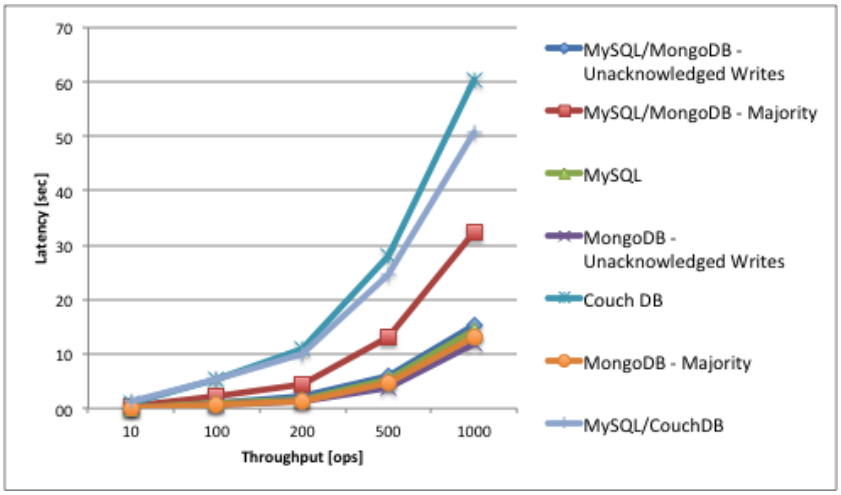


(b)100 users

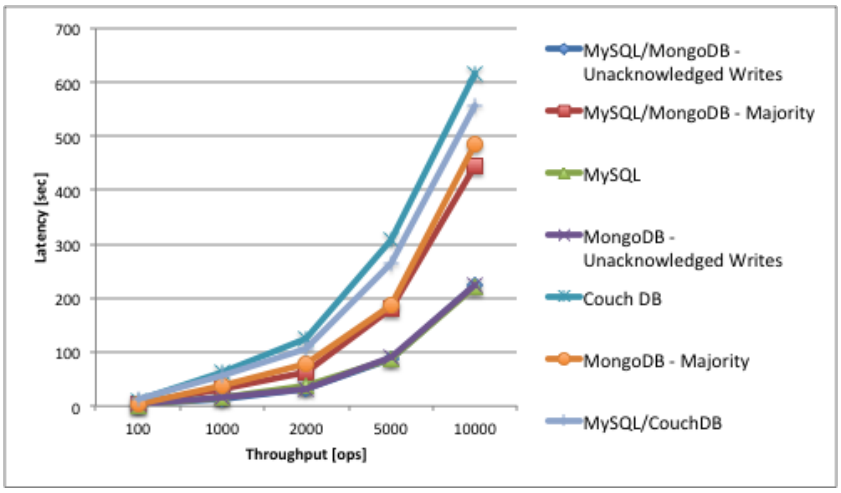


(c)1000 users

Figure 7.8: Delete batch operation performance



(a)10 users



(b)100 users

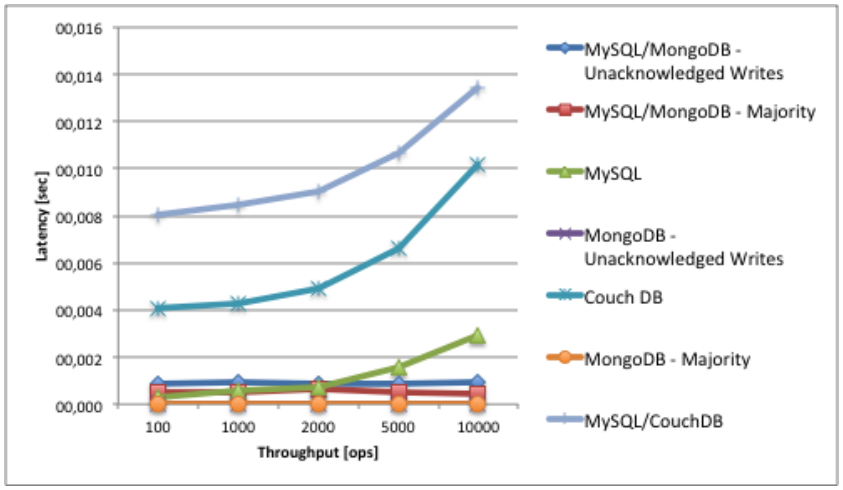
Figure 7.9: Delete operation performance

Query performance In this use-case, the amount of fetched activity messages equals exactly the amount of messages a user has associated in the database. The results shown in Figure 7.10 do not really emphasise it, but they show that querying MongoDB performs really good and is a magnitude faster than MySQL with increasing amounts of data.

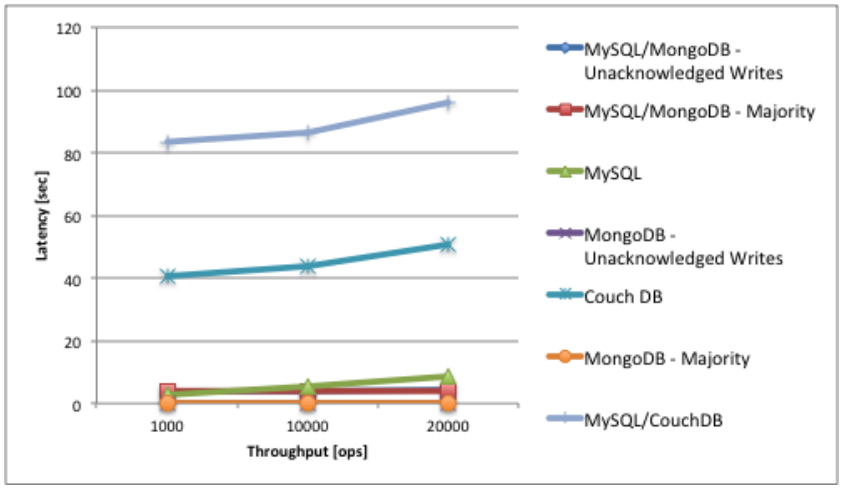
The query-statistics show that CouchDB and especially the combination with MySQL perform really bad. The reason why CouchDB performs so bad is not because the database is inefficient. After debugging and monitoring the application it became clear that CouchDB objects handled in PHP are enormous in their size. Iterating and

counting these large objects in PHP leads to miserable performance characteristics. In contrast to MongoDB where objects handled by the ODM are about 35000 bytes, CouchDB object are around 830000 bytes. This can be either caused by the framework or through the datastore itself.

In this scenario the query-time increases not because of the query duration. It is because of the array conversion after retrieval. As this is a common task in todays PHP development and has been done for MongoDB as well, the results remained unchanged and are compared to the others.



(a)100 users

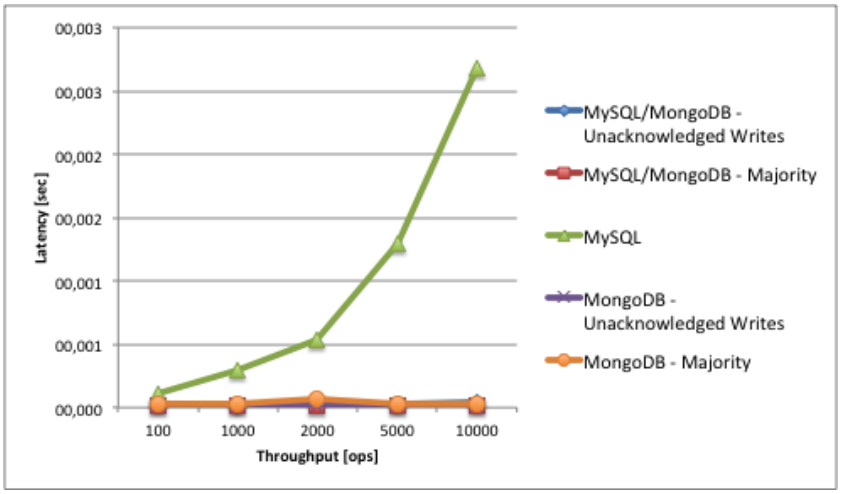


(b)1000 users

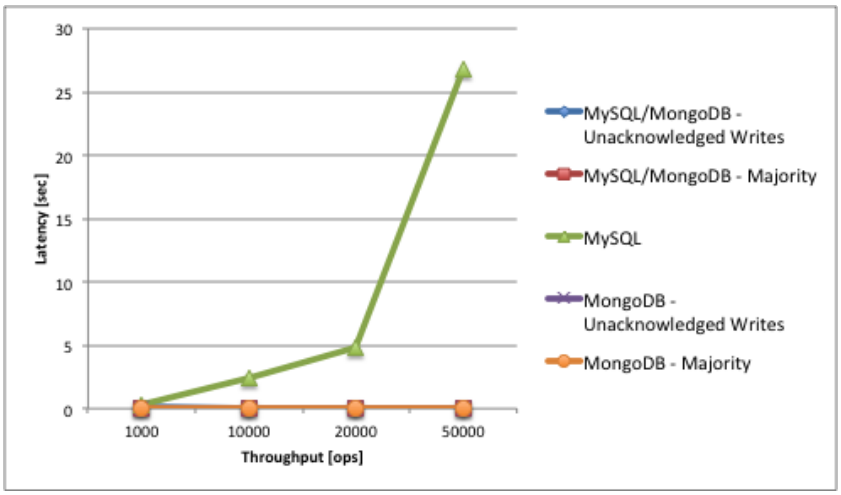
Figure 7.10: Query operation performance

Search performance Figure 7.11 shows the throughput/latency comparison of MySQL and MongoDB regarding search-operations. CouchDB is not included, because full-text search is not natively supported and a separate package needs to be installed.

Search-operations represent text-searches within the content-field of the *activityMessage* table. In MySQL this has been done with the help of LIKE statements. In MongoDB, find-operations provided by the ODM have been used. In all cases, the search-query string is included in each activity message. It has already been mentioned that document-based stores offer great support for unstructured data like the JSON data stored within the content-field of the *activityMessage* table.



(a)100 users



(b)1000 users

Figure 7.11: Search operation performance

Evaluation

The given results show that MySQL reached an average performance in all test-scenarios. This corresponds with the results revealed in the theoretical evaluation and explains the wide range of applications MySQL is used for. The only use-case where MySQL performed significantly worse than document-based solutions was searching JSON based data structures.

These results can definitely be improved with database indexes and views. As well can a caching layer like Memcached[3] or a non-relational search-engine like ElasticSearch[4] improve the situation for often used database operations. Floratou & Dewitt (2012) proved in their paper that an optimised RDBMS can achieve better performance that NoSQL solutions. In their evaluation, a Microsoft SQL Server PDW was about nine times faster than a MongoDB based solution using HiveQL as data manipulation language.

The bad results in MySQL search-performance do not affect this evaluation negatively because good search-performance is no mandatory requirement for the scenario described in chapter 6. The performance result of combined solutions like MongoDB and MySQL showed that this approach needs large amounts of computation power and memory.

With regard to performance, CouchDB and its combination with MySQL led to very good results that can be compared to those achieved by MySQL. The only exception are query-operations where results are far behind other technologies. The underlying problem has already been discussed and seems to be caused by the used PHP module. This problem can probably be solved by using self-written PHP components or implementing memory optimisations.

As expected, bulk operations should be used on as many database operations as possible. Comparing all evaluated data-stores, Figure 7.6 and Figure 7.8 show that bulk-operations in CouchDB have a larger impact on performance than they have on any other data-store.

7.1.2 Load performance testing

Load testing focuses on simulating and often exceeding expected production load by throwing many concurrent requests at a site for an extended time-period. The key to

[3] http://memcached.org/
[4] http://www.elasticsearch.org/

load testing is realism. You need to know your application and your audience well, so that you can balance load across your application in a way that closely mimics the actual or expected production traffic.

Application speed can be measured under various conditions, load levels and scenario mixes. Predefined validation measures are compared with the same test scenario, but different amounts of users (J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber 2007). The functionality of an application under load must not be disturbed and well working. This can be evaluated by validating created database entries or returned response data.

An actual use-case of the an user is called a scenario. This particular use-case is carried out by a single user within the load-test. In technical terms this would be a thread or thread-group firing requests against the applications API.

The way a given scenario is executed reflects a test-plan. This plan defines how many users are accessing the platform, what ramp-up time each of them has and how often a particular user is interacting with the application.

When users are accessing the platform, think-times between two user actions must be considered. Think-time represents the time where the user decides what to do next or where time elapses and he is reacting to other interruptions like a phone call.

Load-testing can measure different evaluation metrics. This ranges from the consistency of response times and their variance on different data and user loads. Additionally when data- or user-load increases it can be distinguished between gradually and stair-step load raise.

Configuration

The tests carried out in this section use the same server configuration already described in subsection 7.1.1. Network traffic is neglected as an influence factor, but the small amount of available RAM can be seen as strong performance limitation.

Sharding is not evaluated in this section, because this use-case does not require it and additional overhead for the used maintenance procedures should be prevented. Another measure not taken into account is the systems behaviour when multiple machines are automatically added or removed from the cluster.

JMeter It provides a GUI for test building that allows to create complex load tests with many different scenarios. Load tests are written in a simple XML format and are run against a configurable web-server. Within the configuration file it is possible to specify the number of concurrent virtual users sending requests to the web-server.

The number of users is not the number of concurrent users executing tasks on the system. The number of concurrent users depends on the duration of the defined scenario and the ramp-up time configured for the thread group (Vahlas 2010).

The ramp-up time within a thread group is the actual time taken by JMeter to spawn all threads. Vahlas (2010) notes that if the ramp-up time is small compared to the number of threads and the mean duration of a scenario, then the number of concurrent threads accessing the system will be high and vice versa. For example if 10 threads are used with a ramp-up period of 100 seconds, each thread will start 10 seconds after the previous thread. Ramp-up time needs to be high enough to avoid a too large workload at the start of a test and short enough to create enough throughput to stimulate otherwise unnoticed problems.

With JMeter it is possible to define the number of loops in a thread group. This defines the number of times a scenario will be executed by each thread. Think-times can be transformed into gaussian waiting functions, so that they can be variable among multiple users.

Scenario The usual access-pattern of the *activityMessages* table is defined here. Using this scenario does not reflect platform user behaviour. It is used to determine performance characteristics of different data-stores and theit replication mechanism.

Each user performs actions leading to data-store operations that are executed. This operations and their execution-distribution are shown in Table 7.1. User actions are performed with the help of sending HTTP-request to the Apache web-server. In comparison to the YCSB benchmark this scenario relates to a read-mostly workload.

Operation	Percentage	Description
Query	80%	Query the last 50 activity-messages of a user.
Bulk-Write	7%	Write 3 user-related activity-messages into the data-store.
Write	13%	Write 1 user-related activity-message into the data-store.

Table 7.1: User operation distribution

At the beginning of the scenario, all used data-stores are empty and there exists a predefined amount of threads running HTTP-requests against the platform. To simulate different engagement-levels on the platform, some users perform the defined operations more often than others. This user engagement is represented as gaussian-distribution.

The operating user is therefore chosen when a new thread is spawned. The needed userID is selected by a gaussian distribution. This makes userIDs in the middle of the userID range more likely to be chosen.

When, for example, users with an userID between 0 and 100 are able to perform operations on the platform, the user with userID=50 is most likely to be chosen for the given operation. In gaussian terms he represents the median μ. In contrast users with userID 0 or 100 are unlikely to be chosen, as they are on the borders of this distribution.

The gaussian distribution states that 99.73% of all occurrences are within the range $\mu \pm 3 * \sigma$. This range and a randomly generated gaussian value z are used for the calculation of the userID. All calculated numbers are rounded and checked for their validity to generate correct userIDs. Using this approach leads to an approximated gaussian distribution of user-engagement.

The JMeter configuration leading to the results presented in section 7.1.2 can be found in Table 7.2

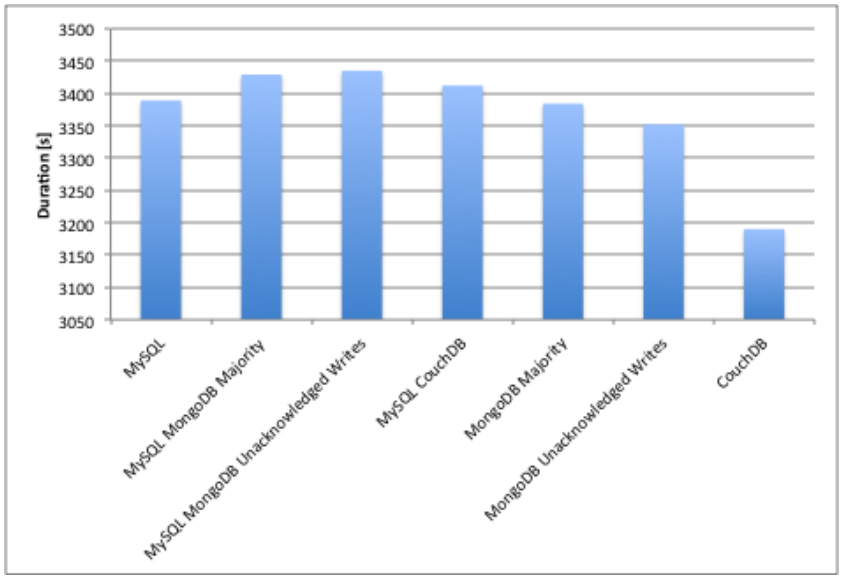
Variable	Value
Number of users	100
Ramp-up time	200/2000 ms
Number of loops	100

Table 7.2: JMeter configuration

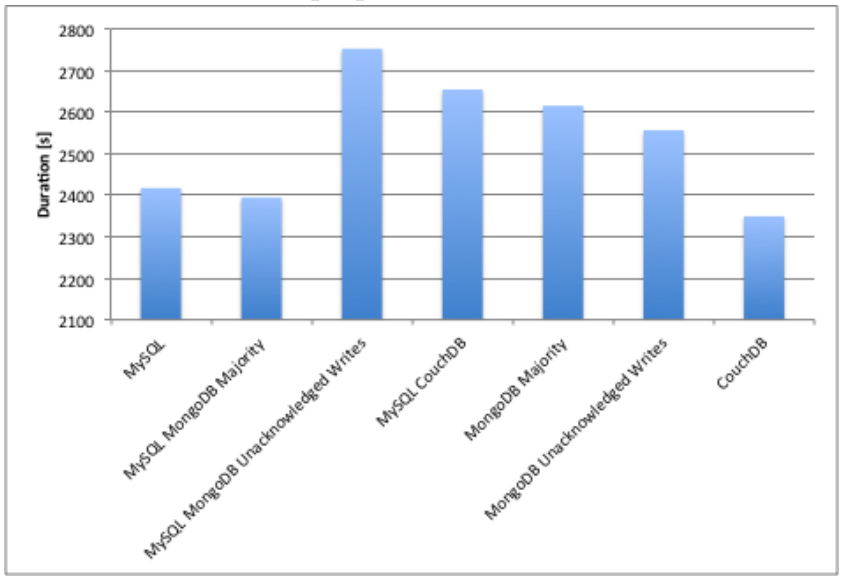
Results

All upcoming figures are split into multiple sub-graphs. Each graph consists of two subgraphs using different ramp-up times. The first subgraph represents the database behaviour under average load with occasional database errors. In the second graph, the number of concurrent users on the platform has been increased leading to high database load on the platform.

Test duration Figure 7.12 shows how long a complete test-run took with each database solution. CouchDB operations are very fast with all tested ramp-up times. Figure 7.12b shows that MySQL and its combination with the MongoDB majority store are also very fast. With high-load, all data-stores beside CouchDB are equally fast.



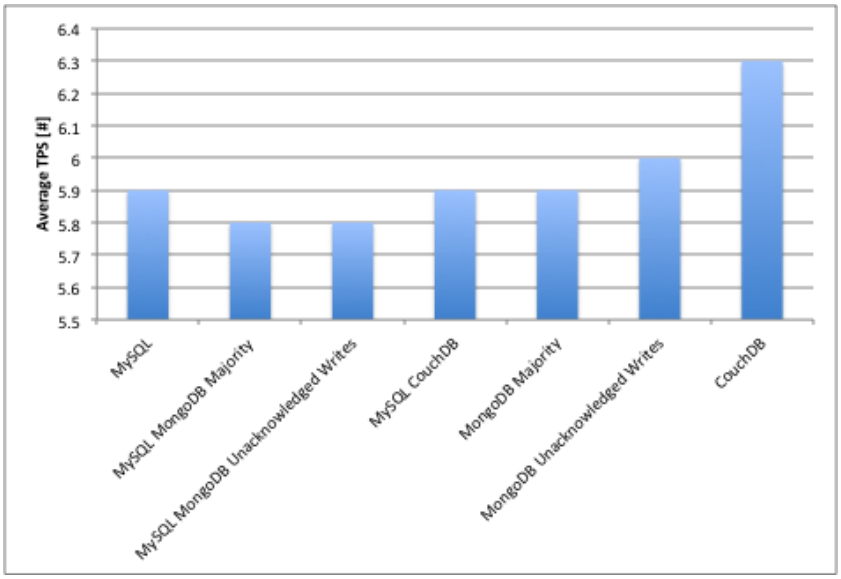
(a)Ramp-up time: 2000 seconds



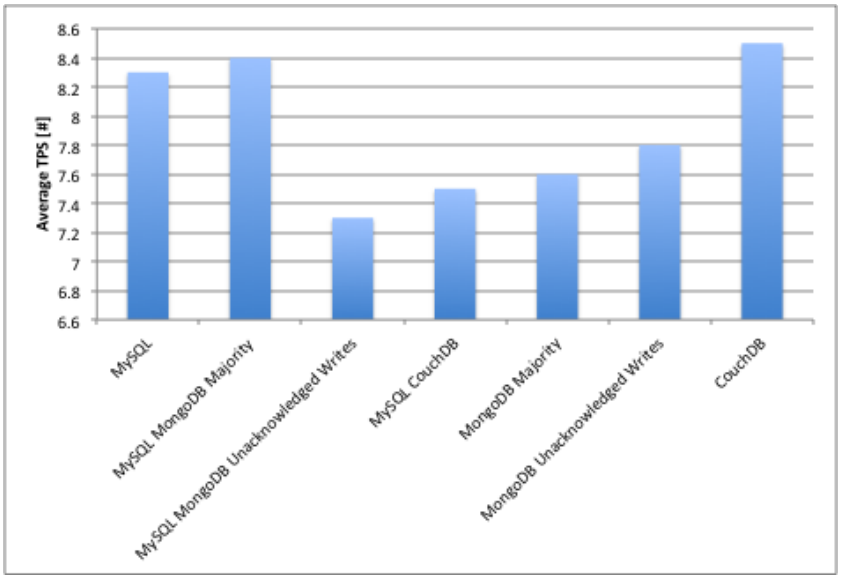
(b)Ramp-up time: 200 seconds

Figure 7.12: Test duration comparison

Average TPS The average amount of transactions per second is shown in Figure 7.13. As in the previous evaluation, the strength of MySQL and CouchDB can be confirmed.



(a)Ramp-up time: 2000 seconds

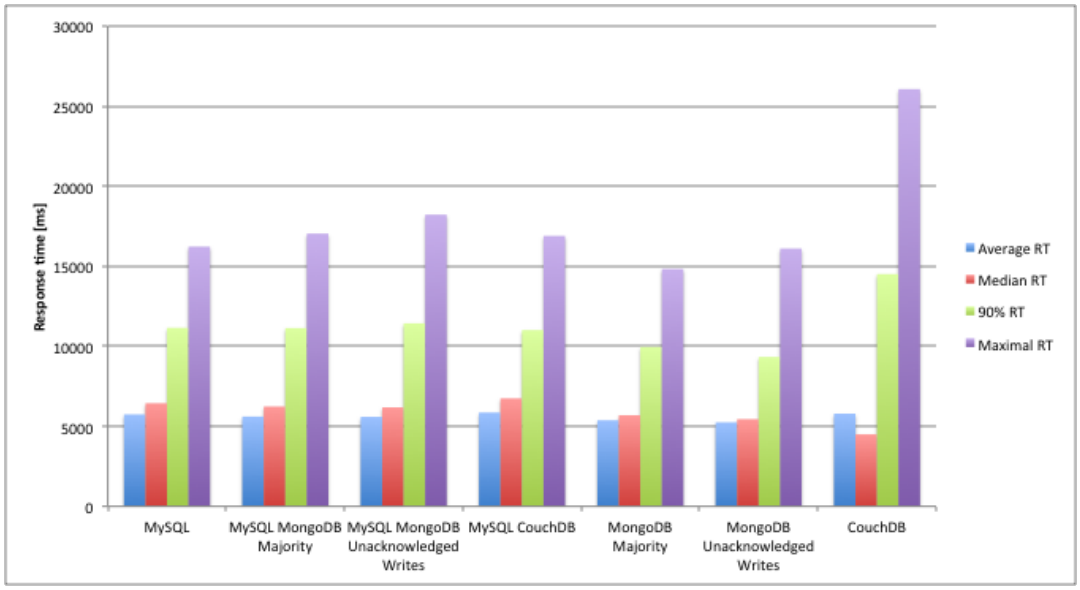


(b)Ramp-up time: 200 seconds

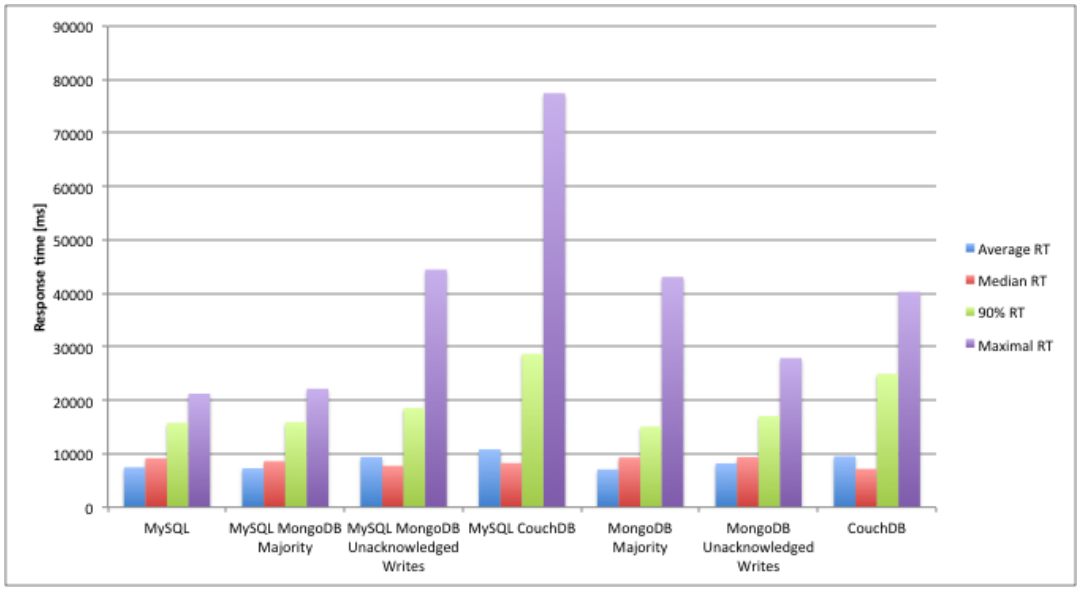
Figure 7.13: Average TPS comparison

Response time CouchDB has the highest throughput, but Figure 7.14 shows that it also has very unstable response-times. In contrast to MySQL, where the maximum and 90% response time is just twice as high as the average, times measured with CouchDB are much higher. This gets even worse with increased ramp-up times, where response times of CouchDB are significantly worse. The statistic shows that CouchDB generally responds very fast, but when the database is busy and errors occur, this can lead to long waiting times and failing requests.

Figure 7.14a shows that MongoDB takes about as long as MySQL with a high number of concurrent users accessing the platform. With average load, MongoDB shows worse response times than MySQL and only performs well in combination with MySQL.



(a)Ramp-up time: 2000 seconds

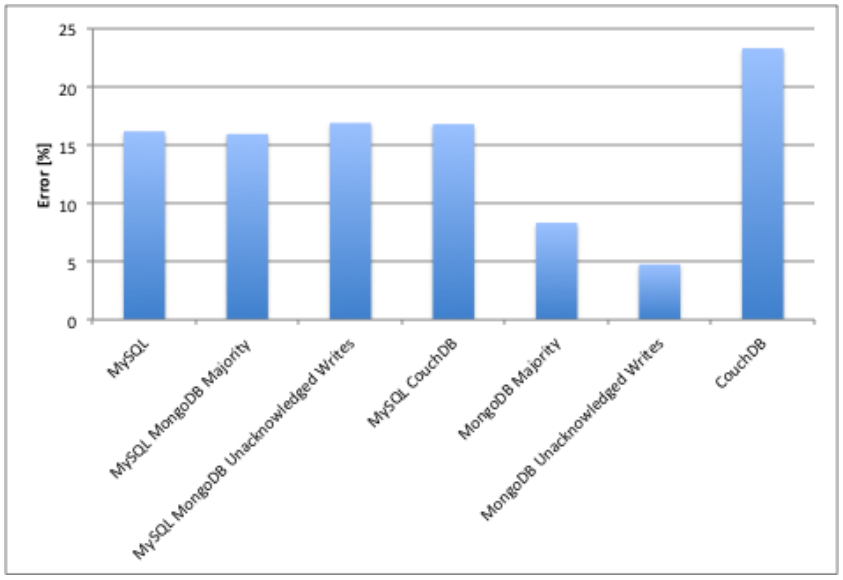


(b)Ramp-up time: 200 seconds

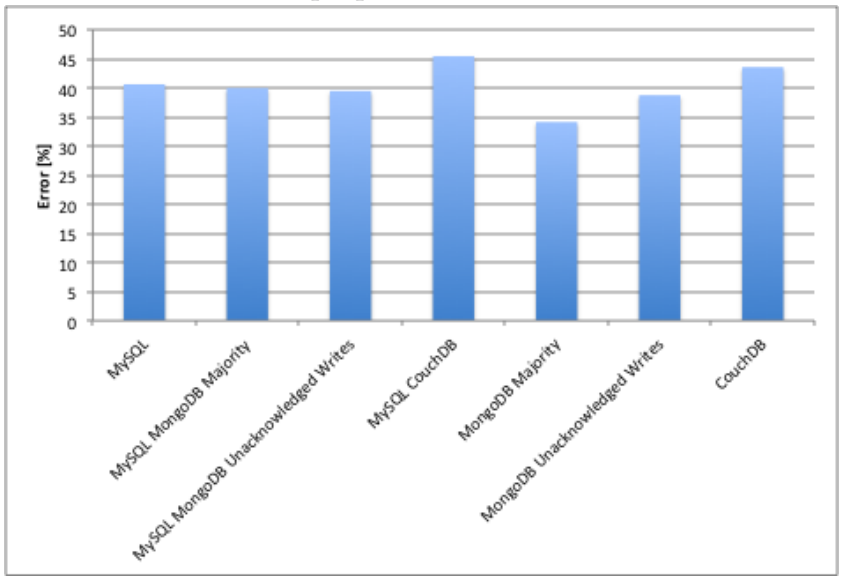
Figure 7.14: Response time comparison

Response error The number of errors stated in Figure 7.15 shows that even though the throughput of CouchDB was very good, this does not mean that all sent requests were successful. So had CouchDB the highest error-rate in the average-load test-scenario. It had a moderate performance with the high-load setting shown in Figure 7.15b.

The datastore performing best with concurrent requests was MongoDB, which had the lowest error-rate in both test-cases. Compared to the other solutions, it had an average throughput and response time. Figure 7.15b demonstrates that almost all data-stores have an error-rate of about 40%. When a database problem occurred, many subsequent requests failed until load went down again.



(a)Ramp-up time: 2000 seconds



(b)Ramp-up time: 200 seconds

Figure 7.15: Response error comparison

Evaluation

In this test-scenario CouchDB behaves like it was described in Figure 3.1. Throughput and performance are traded against consistency and fault tolerance. Test duration and TPS were great, but the error-rate was far behind the other solutions. MongoDB is closer to MySQL, showed less errors, but only average throughput.

Interestingly, the combination of MySQL and MongoDB (Majority) performed more like MySQL than MongoDB. The statistics show that if a document-based store interacts with MySQL, the results are closer to those of MySQL than they are to the non-relational structures. MySQL did not show any extraordinary results and performed moderately throughout all evaluations.

7.2 Complexity

7.2.1 Code complexity

Including data manipulation functionality into a Symfony2 application shows that writing code with an ODM takes about the same amount of expertise and time as if code is written for any other ORM. All performed operations are relatively simple and it can not be guaranteed that complexity stays low when more sophisticated operations are needed. This is probably the case when Map-Reduce operations are needed by CouchDB or custom query operations have to be written in MongoDB.

Researching different solutions revealed that the high level of database support provided by the Symfony2 framework is very rare. Including NoSQL solutions into other frameworks is probably more effort. Like already mentioned in the theoretical evaluation, a transition between relational databases and document-based datastores is feasible.

7.2.2 Maintainability

A very important part while maintaining multiple data-stores, is their easy understanding and widespread knowledge among the development team. When using an ODM, additional code-complexity is low and easy to maintain.

Many organisations implemented monitoring systems that send out e-mail alerts when problems occur. This makes it possible to react quickly on unexpected errors.

Already mentioned in the theoretical part, setting up the non-relational database is easy, but as soon as MongoDB becomes large and sharding is used, it is hard to maintain. Even though a small setup was used in this evaluation, there were also occurring write-concern exceptions during the evaluation. The reasons for those problems remain unclear, but it is suspected that it is because of exhausted heap space.

If systems are used for extended time-periods, with many different users administrating and maintaining the database, the limited privilege and user settings offered by non-relational databases can become a problem. Many systems require specific security mechanisms to protect it from intruders or because of organizational structures.

An important and useful feature for identifying structural database changes, is offered by CouchDB. The underlying versioning system is good to keep track of changes and helps in conflict resolution. Old data can always be restored if application errors occur. In terms of availability, using a MVCC system brings many benefits when structural database changes are needed. No downtimes or schema migrations are necessary to introduce new functionality on a running system.

Already mentioned in the theoretical evaluation, code-quality within the application layer becomes more important when using document-based data-stores like CouchDB and MongoDB. This is because integration checks are largely missing in the database layer.

Knowing a lot about the inner works of the used data-store is very important and understanding the challenges is necessary to get NoSQL solutions to work at a decent speed. This is true for relational and non-relational data-stores and employee training is therefore increasingly important.

Companies developing small, flexible databases are often more interested in their customers needs and may act faster in implementing desired features.

8 Conclusions

8.1 Discussion

The desired benefits expected by combining a MySQL cluster with a document-based store have mostly not been fulfilled. Given the provided infrastructure, which was limited in terms of memory and computation power, multiple storage solutions never showed a significant improvement over the MySQL solution. Even though the combinations sometimes performed better than their non-relational counterpart, they also achieved worse results in many more test-scenarios.

The combination of document-based and relational data was easy to accomplish as both data-stores use similar structures that can easily be combined in the application layer. Migrating data from a relational data-store to a document-based store is feasible if the relational schema is not too complex. When data is used by the ORM and ODM, the developer always has to keep in mind where the data is stored. Combining data from multiple sources always increases complexity and retrieval times.

For the implemented solution, data of the document-based store was linked between multiple collections. It is expected that using just a single MongoDB instance and making use of embedded structures leads to better query-performance, because internal BSON structures are optimised for that.

The heap-space was limited and MongoDB was sometimes not able to hold the whole working-set in-memory. This negative performance impact can be solved with a larger cluster and more physical resources. With larger amounts of data the number of database-nodes within the cluster increases and the benefits of non-relational models should become more obvious. The same should be true for CouchDB, which claims that Erlang works more efficient than other programming languages when many parallel connections are used.

In terms of performance, a mid-sized database using a wide range of different operations still seems to work well with MySQL. This is of course only true for the given setup. Different frameworks and tools have a large impact on the database performance, as we have seen in section 7.1.1. If the use-case involves many complicated

queries using nested join-operations in MySQL, non-relational stores can be the better choice. So was the full-text search provided by MongoDB extraordinary fast and outperformed MySQL even for simple queries.

Non-relational models definitely make sense for temporary data that does not really belong in the main data-store. This can be shopping carts or site personalisations. They can also be an option for web applications having a simple and flexible data-schema expecting sudden growth. The JSON structure used in document-based stores is very appealing and better suited for web applications. When using non-relational databases one should always consider the security risks and maturity issues they have and provide appropriate counter-measures. MongoDB can be a very good solution for dealing with geo-spatial data and provides a large number of indexes and query mechanisms for exactly that purpose.

Before implementing a non-relational schema, extensive tests are necessary. Its performance largely depends on the infrastructure. A web application has many influence factors which lead to given performance measures. So can write-concerns of MongoDB play a more important role in a distributed system dealing with network latency.

When using a new system hard questions should be asked and detailed answers should be excepted. Rigurious testing and vendor cooperation is needed to ensure that the used product is right for the use-case. Corresponding with the results of this evaluation, Stonebraker et al. (2007) argue that there does not exist any specialised area of application for RDBMS and their performance can be beaten by specialised solutions in all scenarios.

Existing relational database vendors have taken notice of NoSQL's success, so did Oracle join the NoSQL camp and new products fulfilling new customer demands are rolling out. Their small competitors are numerous and mature quickly to provide better solutions in the future. Technical leaders have the important role of understanding and evaluating all available options. Having a logical strategy for choosing the right product is going to be the difference between success or failure in technology adoption.

8.2 Future work

This thesis is a starting point for future evaluations of polyglot persistent architectures and gives an impression how different data-stores can be combined and used within a larger infrastructure. To represent an evaluation which is closer to a real-world environment the different data-stores should be run on multiple machines and network latency should be considered. Machines running MongoDB or CouchDB should be evaluated with more computation power available to ensure that they can hold the full working-set in-memory. These performance measurements can then be evaluated against the existing ones to get a more complete picture of their capabilities.

When testing data-store combinations in a large setup, sharding over multiple nodes has to be considered as well. Built-in sharding capabilities of MongoDB and BigCouch can be used and evaluated.

It is also interesting how the database size on each shard influences the performance result. This gets important when large amounts of data are embedded within documents and duplicated data has to be updated as well. The amount of time taken for these operations should always be lower than the performance benefits gained by preferring embedding over linking.

To test how such an architecture operates under changing conditions, it is also interesting to evaluate automatic node reconfigurations. When using a larger cluster with sharding, it should be evaluated at which number of nodes this makes sense. While testing this configurations under high-load, it is expected that some data is lost. Especially with different write-concern settings it should be evaluated how much data is lost for additional performance gains.

When reduced consistency should be combined with a document-based solution, ACID based document-stores like HyperDex[1] can be evaluated in terms of their integrability into larger infrastructures.

In the performance testing more advanced features like caching and indexing capabilities have not been used. It would be interesting if document-based search advantages and performance gains are still that high when an optimised MySQL datastore is used. A caching layer or optimised query-index can be sufficient for the described use-case and lead to better results than those achieved by the data-store combination.

[1] http://hyperdex.org/

Bibliography

10gen (2014*a*), RDBMS to MongoDB Migration Guide, Technical Report October, 10gen.

10gen (2014*b*), 'SQL to MongoDB Mapping Chart'.
URL: *http://docs.mongodb.org/manual/reference/sql-comparison/*

Abadi, D. J. (2007), Column-Stores For Wide and Sparse Data, *in* 'Proceedings of the Conference on Innovative Data Systems'.

Abadi, D. J. & Madden, S. R. (2008), Column-Stores vs. Row-Stores: How Different Are They Really?, *in* 'Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data', ACM, New York, USA, p. 14.

Brewer, E. (2012), 'CAP Twelve Years Later: How the "Rules" Have Changed', *Computer* .
URL: *http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed*

Brown, M. (2012), *Getting Started with CouchDB*, O'Reilly Media.

Cal Henderson (2006), *Building Scalable Web Sites*, 1 edn, O'Reilly Media.

Cass, S. (2009), 'Designing for the Cloud'.
URL: *http://www.technologyreview.com/video/414090/designing-for-the-cloud/*

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. & Gruber, R. E. (2008), 'Bigtable : A Distributed Storage System for Structured Data', *ACM Transactions on Computer Systems (TOCS)* **26**(2), 205–218.
URL: *http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf*

Codd, E. F. (1970), 'A relational model of data for large shared data banks', *Commun. ACM* **13**(6), 377–387.
URL: *http://www.ncbi.nlm.nih.gov/pubmed/9617087*

Dan, P. (2008), 'BASE: An Acid Alternative', *Queue* **6**(3), 48–55.

Dean, J. & Ghemawat, S. (2008), 'MapReduce : Simplified Data Processing on Large Clusters', *Commun. ACM* **51**(1), 107–113.
URL: *http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf*

Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. (2007), 'Dynamo : Amazon's Highly Available Key-value Store', *SIGOPS Oper. Syst. Rev.* **41**(6), 205–220.

Diehl, M. (2011), 'Facebook application development', *Linux Journal* **20011**(208).

Doctrine (2014), 'Blending the ORM and MongoDB ODM'.
URL: *http://doctrine-mongodb-odm.readthedocs.org/en/latest/cookbook/blending-orm-and-mongodb-odm.html*

Edlich, S. (2012), 'The State of NoSQL'.
URL: *http://www.infoq.com/articles/State-of-NoSQL*

Edlich, S., Friedland, A., Hampe, J., Brauer, B. & Brückner, M. (2011), *NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*, 2 edn, Carl Hanser Verlag, Munich, Germany.

Elmasri, R. & B., N. S. (2009), *Grundlagen von Datenbanksystemen*, 3 edn, Pearson Education Inc., Munich, Germany.

Floratou, A. & Dewitt, D. J. (2012), 'Can the Elephants Handle the NoSQL Onslaught?', *Proc. VLDB Endow.* **5**(12), 1712–1723.

Foundation, A. S. (2014), CouchDB 1.7.0 Documentation, Technical report, Apache Software Foundation.
URL: *http://docs.couchdb.org/en/latest/maintenance/performance.html*

Geisler, F. (2011), *Datenbanken - Grundlagen und Design*, 4 edn, mitp Professional, Heidelberg, Munich.

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

Göbel, A. (2011), Verbindung relationaler Datenbanksysteme und NoSQL-Produkte, *in* '23. GI-Workshop Grundlagen von Datenbanken', pp. 31–36.

Hadjigeorgiou, C. (2013), RDBMS vs NoSQL : Performance and Scaling Comparison, Master thesis, The University of Edinburgh.

Hazel, S. (2012), 'Goodbye, CouchDB'.
URL: *http://sauceio.com/index.php/2012/05/goodbye-couchdb/*

Hoff, T. (2010), 'MySQL And Memcached: End Of An Era?'.
URL: *http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html*

Hughes, D. (2010), 'A Quick Overview of Graphing Databases'.
URL: *http://alagad.com/2010/07/13/a-quick-overview-of-graphing-databases/*

J. Chris Anderson, Jan Lehnardt, N. S. (2010), *CouchDB: The Definitive Guide*, O'Reilly Media.

Jansen, R. (2010*a*), 'CouchDB - angesagter Vertreter der "NoSQL"-Datenbanken'.
URL: *http://www.heise.de/developer/artikel/CouchDB-angesagter-Vertreter-der-NoSQL-Datenbanken-929070.html*

Jansen, R. (2010*b*), 'Neo4j – NoSQL-Datenbank mit graphentheoretischen Grundlagen'.
URL: *http://www.heise.de/developer/artikel/Neo4j-NoSQL-Datenbank-mit-graphentheoretischen-Grundlagen-1152559.html*

J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, D. R. (2007), Performance Testing Guidance for Web Applications Performance Testing Guidance for Web Applications, Technical report, Microsoft Corporation.

Kahwe, L. (2010), 'Doctrine2 CouchDB support! But Why?'.
URL: *http://pooteeweet.org/blog/1832*

Keep, M. (2011), 'Scaling Web Databases: Auto-Sharding with MySQL Cluster'.
URL: *https://blogs.oracle.com/MySQL/entry/scaling_web_databases_auto_sharding*

Kingsbury, K. (2013), 'Call me maybe: MongoDB'.
URL: *http://aphyr.com/posts/284-call-me-maybe-mongodb*

Kovacs, K. (2014), 'Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs OrientDB vs Aerospike vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris comparison'.
URL: *http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis*

Kristina Chodorow, M. D. (2010), *MongoDB: The Definitive Guide*, 2nd edn, O'Reilly Media, Inc.

Lai, E. (2009*a*), 'No to SQL? Anti-database movement gains steam'.
URL: *http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_ga*

Lai, E. (2009*b*), 'Researchers: Databases still beat Google's MapReduce'.
URL: *http://www.computerworld.com/s/article/9131526/Researchers_Databases_still_beat_Google_s*

Leavitt, N. (2010), 'Will NoSQL Databases Live Up to Their Promise?', *Computer* **43**(2), 12–14.

Leskovec, J., Rajaraman, A. & Ullman, J. D. (2012), MapReduce and the New Software Stack, *in* 'Mining of Massive Datasets', 1 edn, Cambridge University Press, p. 315.
URL: *http://infolab.stanford.edu/ ullman/mmds/ch2.pdf*

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C. & Byers, A. H. (2011), *Big data : The next frontier for innovation, competition and productivity*, 1 edn, McKinsey Global Institute.
URL: *http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation*

Market, P. (2014), '4 Reasons Perfect Market Bets on MongoDB'.
URL: *http://perfectmarket.com/four-reasons-perfect-market-bets-on-mongodb/*

MySQL (2014), 'Guide to Scaling Web Databases with MySQL Cluster'.
URL: *http://www.mysql.com/why-mysql/white-papers/guide-to-scaling-web-databases-with-mysql-cluster/*

Naesholm, P. (2012), Extracting Data from NoSQL Databases, Master thesis, University of Gothenburg.

Parker, Z., Poe, S. & Vrbsky, S. V. (2013), Comparing NoSQL MongoDB to an SQL DB, *in* 'Proceedings of the 51st ACM Southeast Conference', ACM Press, p. 6.

Rèmy Frenoy (2013), Design and implementation of a NoSQL solution on a Software As a Service platform, Master thesis, Linköpings universitet.

Riyad Kalla, John Lynch, R. H. (2014), 'How does MongoDB compare to CouchDB? What are the advantages and disadvantages of each?'.
URL: *http://www.quora.com/How-does-MongoDB-compare-to-CouchDB-What-are-the-advantages-and-disadvantages-of-each*

Rotem-Gal-Oz, A. (2006), 'Fallacies of Distributed Computing Explained'.
URL: *http://www.rgoarchitects.com/Files/fallacies.pdf*

Rudra, S. (2012), 'Rise of Column Oriented Database'.
URL: *http://www.slideshare.net/srudra25/rise-of-column-oriented-database*

Sadalage, P. J. & Fowler, M. (2012), *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1 edn, Addison-Wesley Professional.

Service, S. S. & Cloud, E. C. (2009), 'Eventually Consistent', *Communications of the ACM - Rural engineering development* **52**(1), 40–44.

Sirer, E. G. (2013), 'Broken by Design: MongoDB Fault Tolerance'.
URL: *http://hackingdistributed.com/2013/01/29/mongo-ft/*

Slater, N. (2013), 'Welcome BigCouch'.
URL: *https://blogs.apache.org/couchdb/entry/welcome_bigcouch*

Software Group, I. C. (2013), 'NoSQL does not have to mean no security'.
URL: *http://public.dhe.ibm.com/common/ssi/ecm/en/nib03019usen/NIB03019USEN.PDF*

Stonebraker, M., Abadi, D. J., Harizopoulos, S. & Helland, P. (2007), 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', VLDB Endowment, Vienna, Austria, pp. 1150–1160.

Strauch, C. (2011), 'NoSQL Databases'.
URL: *http://www.christof-strauch.de/nosqldbs.pdf*

Strauss, D. (2009), 'Giving schema back its good name'.
URL: *http://fourkitchens.com/tags/couchdb*

Strozzi, C. (2010), 'NoSQL – A relational database management system'.
URL: *http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home Page*

Tiago Macedo, F. O. (2011), *Redis Cookbook - Practical Techniques for Fast Data Manipulation*, O'Reilly Media.

Tonytam (2010), '12 Months with MongoDB'.
URL: *http://blog.wordnik.com/12-months-with-mongodb*

Vahlas, N. (2010), 'Some thoughts on stress testing web applications with JMeter (Part 1)'.
URL: *http://nico.vahlas.eu/2010/03/17/some-thoughts-on-stress-testing-web-applications-with-jmeter-part-1/*

Wennmark, T. (2012), 'MySQL & NoSQL: The Best of Both Worlds'.
URL: *http://nosqlroadshow.com/dl/NoSQL-Cph-2013/Presentations/MySQLandNoSQL_Ted Wennmark.pdf*

Wilson, C. (2013), 'Introduction to Sharding'.
URL: *http://www.slideshare.net/mongodb/introduction-to-sharding-28536018*