Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

MICHAŁ NIEĆ

BAZY DANYCH NOSQL W APLIKACJACH NAUKOWYCH

PROMOTOR:
dr inż. Maciej Malawski

Kraków 2014

OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Computer Science, Electronics and Telecommunication

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

MICHAŁ NIEĆ

NOSQL DATABASES IN SCIENTIFIC APPLICATIONS

SUPERVISOR:
Maciej Malawski Ph.D

Krakow 2014

Abstract

Today, scientific computations undeniably play very important role in research, engineering and business applications. Their increasing adoption and capabilities are also driven by exponential growth of available computing power and disk storage. Researchers are able to execute very complex simulations and experiments which involve processing petabytes of data using clusters consisting of thousands of machines. Already established solutions for storing and accessing data like relation databases become bottleneck in such situations and create demand for more scalable solutions.

In the scope of this thesis aim to answer whether using new, modern databases such as MongoDB and Riak could help in designing and executing large scale scientific computations. As a result we analyzed real biological experiment to create new benchmark tool set that could simulate it and generate comparable scores. Multiple tests were executed using AWS EC2 to try different configurations of load and databases. We carefully examined obtained figures trying to show strengths and weaknesses of both databases.

The results show that new, different types of database engines can take advantage of modern computing environments and accelerate creating scientific computations. However most of their performance and scalability features brings certain limitations in other areas which makes choosing right data store a hard decision that needs to be backed up by proper study of a given use case and a database.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor, Maciej Malawski, for his invaluable suggestions, time and help.

I would like also to thank Tomasz Gubała for sharing his research materials with me.

Contents

1. Introduction

This chapter introduces purpose and background of this thesis. In section 1.1 we explain requirements of scientific computations. In section 1.2 we explain problems regarding accessing computational data. In section 1.3 we present our reference application. Section 1.4 formulates goals of this thesis.

1.1. Scientific computations

Computing undeniable plays big role in today's science and engineering. The advanced mathematical models analyzed by computers help scientists to understand, verify, simulate and overcome many difficult problems. Often they are so complex that require vast amount of computational power and data storage. The demand for faster, cheaper computations seems be limitless and for years it has driven development of supercomputers in research centers around the world. In the Figure **??** we can see that in 19 years amount of computational power was increasing exponentially leaving us with capacity that is more then 10000 times bigger then what we had in 1993 when TOP500 ranking was established.

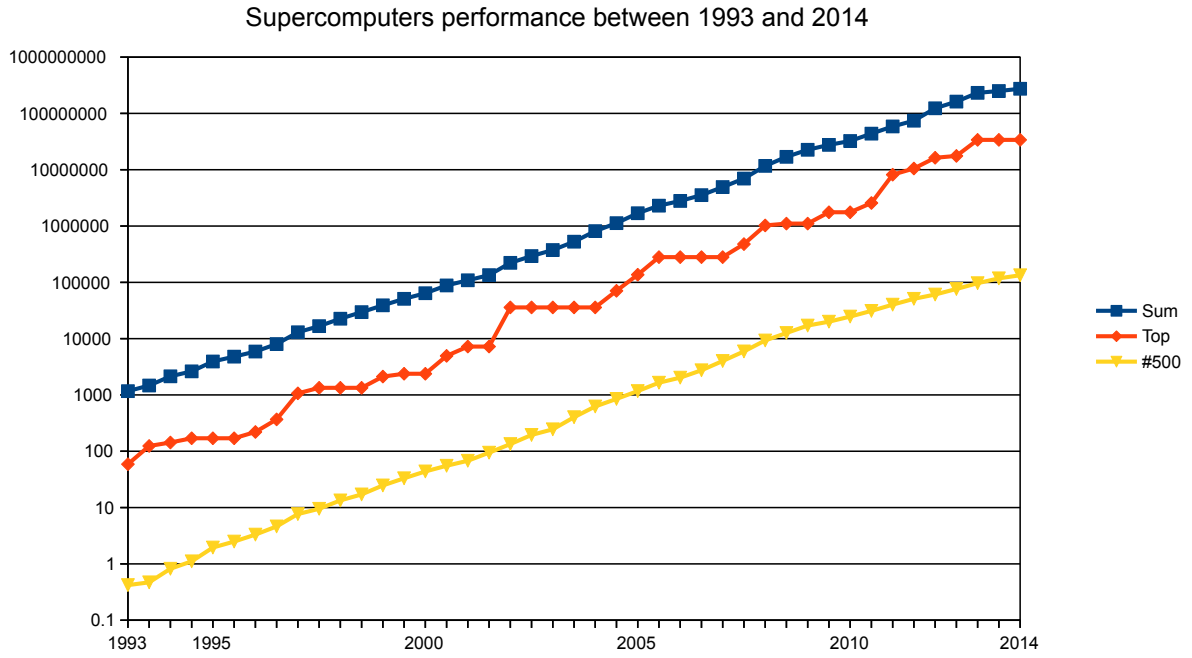

Figure 1.1: Expotential growth of supercomputers performance, based on data from top500.org site.

The logarithmic y-axis shows performance in GFLOPS. The "Sum" series combines performance of 500 largest supercomputers. The "Top" shows the fastest supercomputer. The "#500" shows power of a last computer on the list (source: [1]).

The disk storage size has also been growing exponentially in recent years. In 1993 a disk with 1500 MBytes costed 1459 USD but today we can buy similar device with 3000000 MBytes for about 110 USD [28].

Such fast development enables us processing much bigger volumes of data in more complex ways. Currently researchers try to collect and process petabytes of data in their experiments. Such ideas could not be even considered couple years ago.

However hardware capabilities are not the only requirement for such such computations. The software that that runs experiments also needs to be able to take full advantage of all additional capacity. Even after years of developing applications, libraries and other tools it is very difficult to create such computations. It requires researchers to write and test many complicated custom programs and procedures. One of their biggest problem is accessing large volumes of data in distributed environment where many nodes need to get random pieces of input data very fast and being able to write results to one place so they don't wait wasting time and power.

1.2. Processing large volumes of data

The most common approach for accessing large data sets was to persist data in form of binary or text files which were directly consumed by computation programs directly from file system.

The file system approach has significant advantages like simplicity and scalability. It is usually built-in in operating systems and offers raw performance for writing and reading data.

However it lacks any advanced data management, validation, partitioning, querying forcing user to choose between writing custom programs and procedures or using database management systems (DBMS).

The choice isn't that obvious as most popular, mature and widely adopted DBMS in both scientific and commercial applications are relational databases (RDBMSs). They are designed to offer consistency and availability sacrificing ability to work in highly distributed, heterogeneous environments. It usually means that they require architecture in which multiple concurrent workers must be connected to one data store which quickly becomes a bottleneck. This makes them very difficult to use in grid or cloud environment where large scale computation is often performed on thousands of independent nodes.

Nowadays alternative databases were introduced by NoSQL movement [41] engineered to deal with problem of scaling. They seem to be able to take full advantage of clouds and clusters. However they achieve it by limiting some of the features like consistency or availability. They are not standardized and each one has different strength and weaknesses. Right now there there is not enough information available on how they behave when used for large scale computations.

1.3. Motivating application

There are many types of scientific applications that involve processing large volumes of data. Some disciplines like bioinformatics completely rely on such experiments. In this thesis we focused on single biological experiment that would serve us as a reference of such computation and provide us requirements and realistic sample data.

In this experiment then 40 000 RNA fragments were compared against genes CDS from 23 human chromosomes. The computation used SSEARCH program from FASTA [35] suite which implements Smith-Waterman algorithm and as a result it produced more then 1 000 000 files containing thousands of positions. Such large number of results is useless without further processing which filters and returns most interesting scores.

Researchers were facing problem how to extract useful information fast without having to traverse all these files repeatedly. Inserting such data into database which could index it and preprocess allowing fast searching and

aggregations seemed like a good idea but question was raised what type of database engine could be used and if NoSQL solutions that could take advantage of distributed environment would be suitable.

1.4. Goal

The goals of this thesis is to review possibilities given by selected NoSQL data stores available today in terms of scientific research. It aims to provide answers to questions like: when to use NoSQL databases? What are benefits and what are the drawbacks? The goal will be accomplished by:

- studying existing data stores to understand their purpose, advantages and drawbacks,
- examining benchmarks, workload generators and test results available today,
- designing test procedure, implementing it and preparing execution environment,
- evaluating results,
- formulating conclusions.

1.5. Summary

In this chapter we described motivation and goals for this thesis. We discussed briefly scale related challenges to scientific computations. We introduced our reference experiment and explained potential benefits of using NoSQL databases in it.

2. Comparison of database management systems

In this chapter we describe systems for managing data as this thesis focuses mostly on them. In section 2.1 we present history of databases and try to explain their purpose and origin. Later in section 2.2 we discuss the NoSQL movement. Next section 2.3 presents in detail two databases chosen for the experiment.

2.1. Evolution of database engines

2.1.1. Early database management systems

The need for systems specialized in managing data have been present since early days of computing. The constantly growing data that needs to be persisted and accessed through various programs requires more and more complex algorithms and approaches.

History of computer databases begun in 1960' with availability of direct-access storage devices which allowed shared, interactive use in opposite to daily batch processing offered by tape systems.

First database engines used navigational approach in which records or objects are searched by following references from other object. Through many systems using this pattern two solutions had biggest influence - the IBM *Information Management System* and General Electric *Integrated Data Store*. The IMS used hierarchical model and was designed at first for the Apollo program and then adopted for civil usage. It is still used around the world and available to buy. The second was designed by Charles Bachman who received A.M. Turing in 1973 for his work [2] on navigational databases. The biggest advantage of these databases was raw performance for transactions on large-scale data sets. It was superior to relational DBMSs invented later. To date these systems are used to help DBMSs in most throughput demanding scenarios.

2.1.2. Relational databases

Second generation of database engines are relational DBMSs. They are based on relational model invented by PhD Edgar Codd - IBM employee who wanted to mostly improve search capabilities of DBMSs used then [11]. He proposed to organize data in many different tables consisting of fixed-length records. In Figure 2.1 relation (table) is shown where a single column represents set of attributes and a row defining the dependency between values in columns. Every record which is always identified by key can relate to others in one-to-many relations (like in hierarchical model) and many-to-many relation (navigational model). Development of relational databases in 1970' also led to invention of SQL language. This special-purpose language was designed to provide a convenient interface for all data manipulation (inserting, updating) and browsing. As a result of these innovations IBM and Oracle released first commercial RDBMSs in the late 1970s which started to replace old navigational systems.

From 1980s to today RDBMSs dominated data stores market being an automatic choice for people developing new systems. What is more before started to gain popularity databases often acted as a integration layer between diffrent systems in for example company. The monolithic system design do not assume any communication with

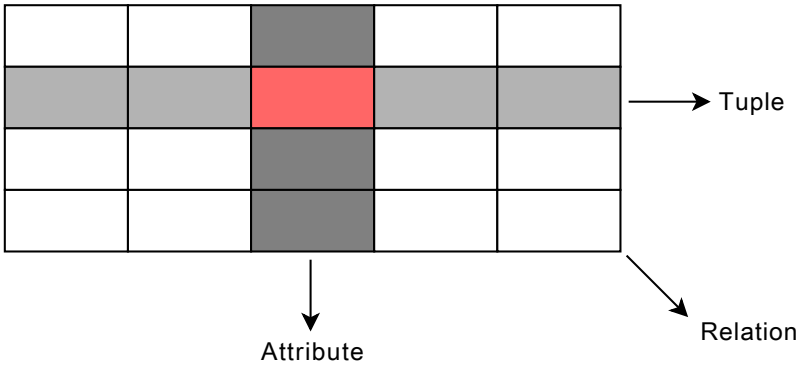

Figure 2.1: Codd's Relational Model

external services and systems. All components and layers are tightly coupled so that introducing such interface is never easy. For years many system architect were using databases for that. The RDBMS has a few advantages in such configuration:

Strong consistency

The RDBMSs come with strict schema and support for different validation techniques of records being inserted or modified. This guarantees that many different subsystems will coexists using the same data without interrupting and breaking each other.

Availability of connectors/drivers

Most RDBMSs have grow with many libraries and connectors allowing wide range of programming languages and middlewares (linke ODBC) to work with them. Therefore systems really different in architecture and technologies used were able to communicate with the same database.

Business logic

The databases allowed to implement Stored Procedure which could be used to implement operations on data that many DB users can access. This allowed to implement business logic that written once could be available for every application that is using DB.

Qualified staff

As the SQL language and relational database model became a common skill for an average software engineer. Almost every programmer knew concepts behind it and had knowledge how to build systems on top of it. Therefore specialists building different systems were able to integrate their systems through database easily.

Today still relational databases are most popular DBMSs (figure **??**). Both commercial (Oracle, MS SQL Server, IBM DB2) and open-source (MySQL, PostgreSQL) solutions gained wide adoption. Being developed through out many years they become stable, reliable and grew in community and tools.

2.1.3. Object databases

In 1980s object oriented programming gained wider adoption so that software engineers started to treat data stored in databases as objects. The problems of object-relational mismatch were mostly seen in data types differences and supporting polymorphism and inheritance. This lead to research and development of object databases, object-relational databases object query language (OQL). The object databases have never gained wider adoption while object-relational features were introduced to almost all major RDBMSs by becoming part of the

Rank	Database	Score
1	Oracle Relational DBMS	1544.44
2	MySQL Relational DBMS	1324.83
3	Microsoft SQL Server Relational DBMS	1304.96
4	PostgreSQL Relational DBMS	182.22
5	DB2 Relational DBMS	162.94
6	MongoDB Document store	155.99
7	Microsoft Access Relational DBMS	150.88
8	Sybase Relational DBMS	81.59
9	SQLite Relational DBMS	79.44
10	Teradata Relational DBMS	53.83

Table 2.1: Top 10 most popular databases according to DB-engines.org (2013-09-01).

SQL:1999 [19]. On the programming side ORM (Object Relational Mapper) libraries were developed to bridge gap between objects and RDBMSs.

2.1.4. NoSQL movement

Next generation of DBMSs started to be known as NoSQL databases [41]. This name refers to databases that do not use relational model and do not use SQL query language to operate on them. The NoSQL label does not stand for any specific model of storing and operation on data. However databases which claim to belong to this category are usually key-value stores and document-oriented databases where there is no schema, joins are avoided and data is stored in denormalized state. Most of them were designed to tackle problem of massive scale which is a problem of many web oriented companies. Therefore they offer advanced horizontal scaling and replication mechanisms.

Popular NoSQL databases nowadays are MongoDB, Riak, CouchDB, Cassandra, HBase, Redis, Memcache. They are becoming more and more popular (see figure 2.4 but they are still shadowed by most popular RDBMSs engines (figure 2.3). Nevertheless the whole NoSQL movement constantly gain more and more attention (figure 2.2).

2.1.5. NewSQL relational databases

Another interesting breed of new DBMSs is called NewSQL. These databases use relational model but claim to provide almost the same scalability as NoSQL DBs. This movement is quite new and it seems like it is driven mostly by commercial DB producers for now. Browsing technical blogs and journals on web we might found attempts to describe and categorize such databases.

Michael Stonebraker (MIT, co-creator of VoltDB) published blog post [40] in which he points out that leading NoSQL databases do not offer strong consistency of data which is crucial for many applications. Without them NoSQL are not able to replace relational databases so new SQL databases are required guaranteeing consistency but also scalability and one-node performance.

On other technical blog [44] there is article trying to justify NewSQL as beineg a solution to constantly increasing volume of data handled by classical OLTP which cannot be easily replaced by NoSQL and cannot be scaled by today RDBMSs. Authors also try to divide NewSQL databases into 3 categories.

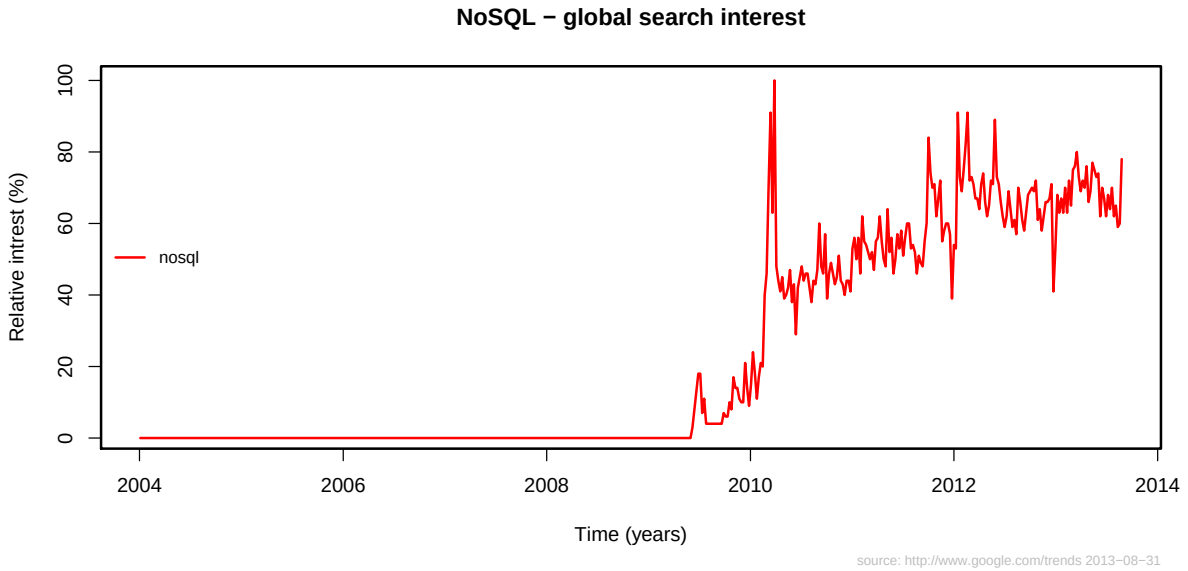

Figure 2.2: Chart showing how often term ‘nosql’ was searched in Google search engine over last few years

New databases

Designed from scratch to provide scalability and performance. Delivered as software or appliances. Examples: VoltDB[1], NuoDB[2], Drizzle[3].

New MySQL storage engines

They replace the original MySQL engines (InnoDB, MyISAM and others) to overcome performance and scalability issues maintaining same interface as MySQL. Examples: TokuDB[4], MySQL NDB[5].

Transparent clustering

Systems which retaining OLTP in original shape adding pluggable features for transparent clustering and scalability. Examples: dbSchards[6], ScaleBase[7] MemSQL[8].

To sum up, the NewSQL databases aim to solve performance and scalability issues of today’s OLTP. They are RDBMSs with completely redesigned internals maintaining the same interface (SQL language) and ACID transactions. They are most suitable for:

– extending current applications that need to operate on bigger volumes of data,

– developing new systems that require OLTP features,

– taking advantage of developers skilled in SQL and OLTP.

[1] http://www.voltdb.com
[2] http://www.nuodb.com
[3] http://www.drizzle.org
[4] http://www.tokutek.com/products/tokudb-for-mysql/
[5] http://dev.mysql.com/doc/ndbapi/en/index.html
[6] http://www.codefutures.com/dbshards/
[7] http://www.scalebase.com/
[8] http://www.memsql.com

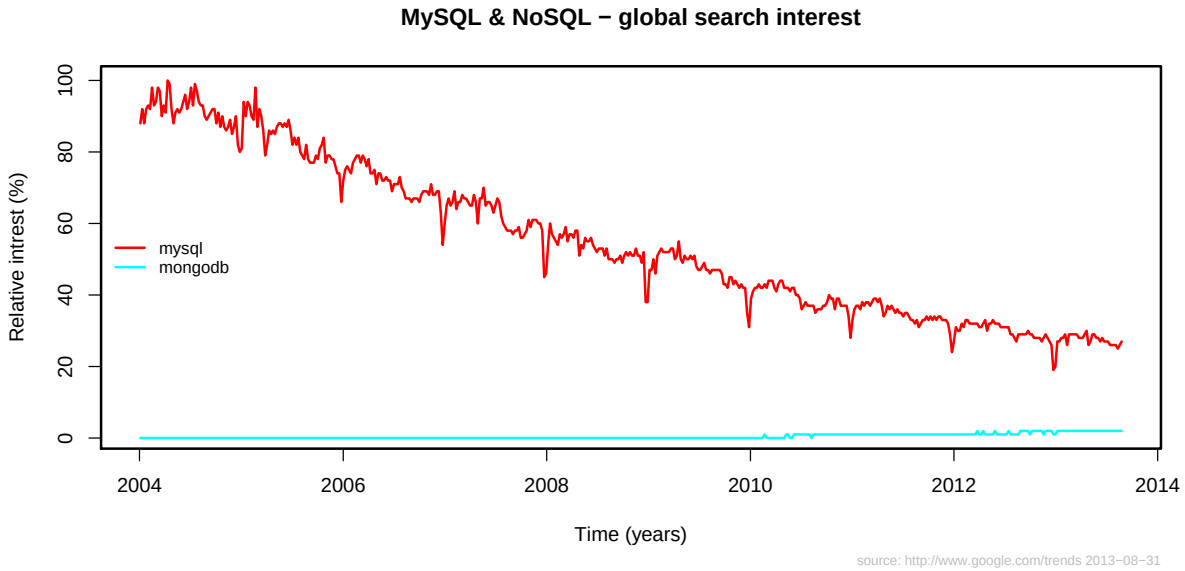

Figure 2.3: Chart showing how interest in one of the leading RBMS engine is dropping compared to leading NoSQL database engine

2.2. NoSQL Databases

2.2.1. Genesis of NoSQL term and it's meaning

The origins of NoSQL data stores are hard to define. Rick Cattel translates term "NoSQL" to "Not Relational" or "Not Only SQL" saying also that there is no generally agreed definition [8]. J. Sadalage and M. Fowler in their book about NoSQL [36] claim that NoSQL term began its career by accident in 2009. Johan Oskarsson was trying to find short, memorable name for a meeting regarding alternative to RDBMSs data stores. The meeting was held to discuss Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB. Each of databases share some concepts and ideas but they are far from compatible between each other and they serve different purpose. What they all had in common was the time they were created and the fact that they were very different from SQL solutions. Therefore term once used in Twitter[9] started becoming more and more popular in IT world and now refers not only to them but also to other similar solutions. It is imporatant to know that event if it is quite wide set it does not include other not relational databases which were created in before 2000.

2.2.2. Purpose

Figure 2.5 shows the data collected by *Cisco* company. The charts shows estimation of average monthly traffic that was generated in global internet. There is no doubt that after year 2000 there was major increase in internet adoption. The constantly increasing number of people connected to Internet alongside with new concept of WWW called Web 2.0 created completely new challenges for IT industry.

As co-author of Web 2.0 term Tim O'reilly said in one of this presentations "The key to competitive advantage in internet applications is the extent to which users add their own data to that which you provide." [33]. That meant that servers and systems delivering the data to user via Internet had to change dramatically. They needed not only to sent published content to the user but also to receive, validate and store data from the user. Allowing the same data to be accessed by other users. This task required usage of databases capable of processing many concurrent

[9] http://www.twitter.com

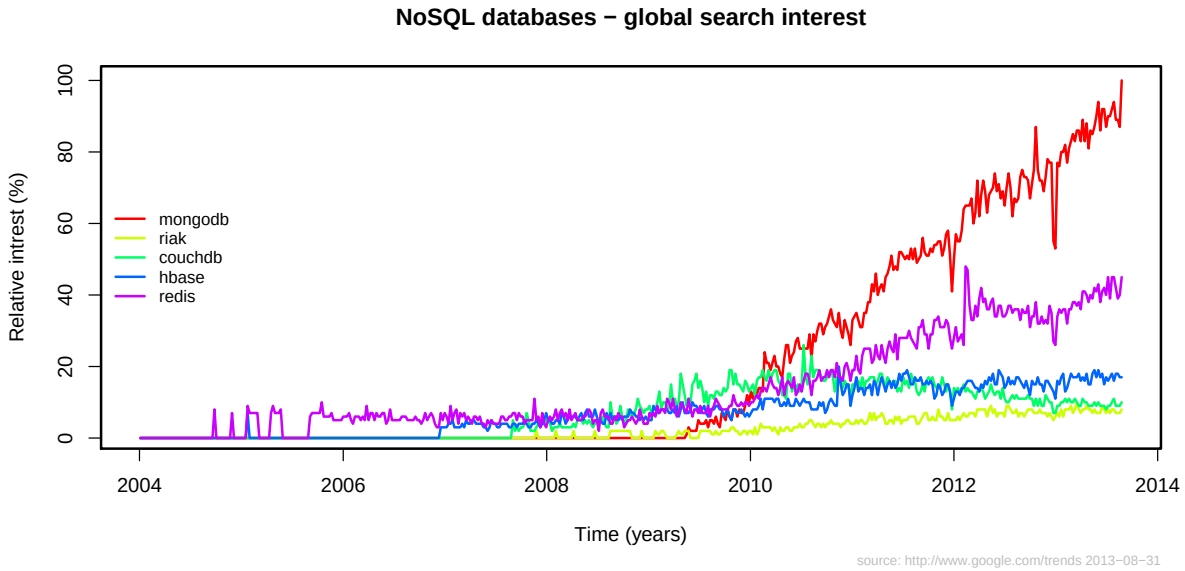

Figure 2.4: Chart showing interest some of the most popular NoSQL database engines

requests each second. At first RDBMS as leading industry standard were used rather succeeding and maintaining strong position in web development to date. Nevertheless they showed a few drawbacks which were quite relevant for some systems facing constantly growing number of users and requests.

The RDBMSs were designed to run of single computer. Scaling database could have been done only in so called *vertical* way - meaning buying better hardware (CPU, Memory) to database node. Increasing throughput or availability *horizontally* - by adding more nodes to cluster was not available. The *vertical* scaling worked well for many solutions. Also problem of scaling could have been solved by decomposing systems to smaller subsystems having separate databases and working with each other using SOA architecture.

Nevertheless web companies which gained enormous popularity among the globe (such as Google, Amazon) could not satisfy their need for persisting and accessing data with RDBMS. Therefore they started to research solutions possessing following features:

Distibuting load horizontally

Multiple nodes automaticly split the incoming request so that each one server only part of the requests.

Replication to many nodes

Data is partitioned and replicated to multiple cooperating nodes.

Simple interface

Removing SQL language layer in RDBMS.

Weaker concurrency model

Ability to execute operations more parallel using modern multicore CPU.

Efficient use of hardware

Taking advantage of modern hardware rich in large volumes of fast memory and many fast CPU cores.

No strict schema

Ability to add new attributes and record types without changing database schema.

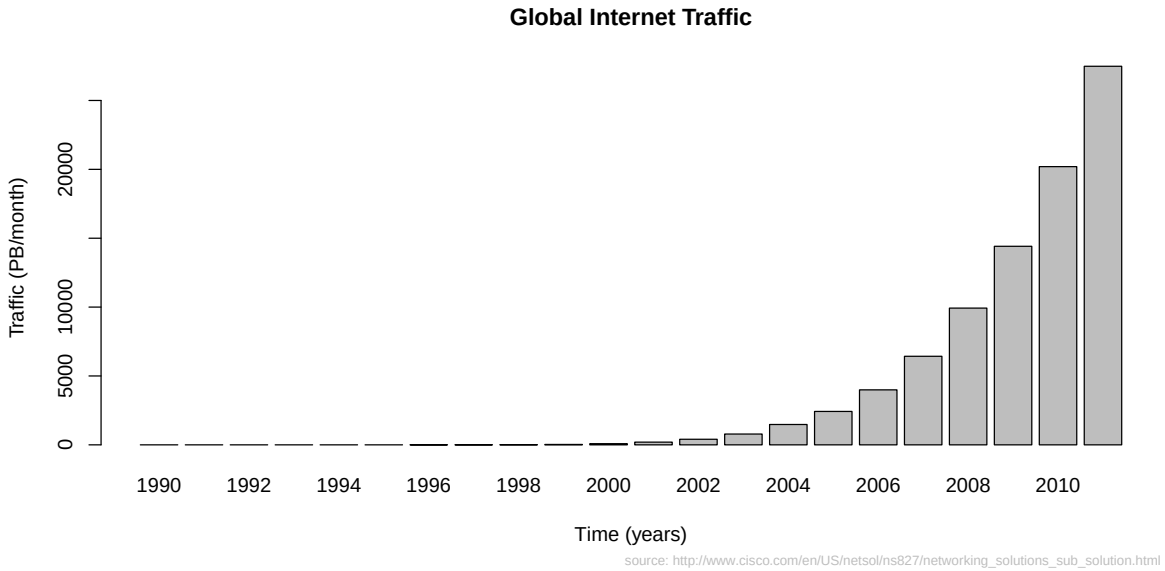

Figure 2.5: Rapid growth of the Internet

As a results many products were created. The most significant ones that inspired many others were Google BigTable [9], Amazon Dynamo[17] and open source project - memcached[10].

BigTable showed ways of storing petabytes of data spread accross thousands of nodes.

Dynamo introduced concept of *eventual consistency* which is way of achieving high availability sacrificing strong consistency. Data fetch might not be up to date but the changes are guaranteed to be propagated to all nodes when it is possible.

memcached showed how scalable simple key-value stores can be.

These concepts and products became a foundations for many later popular data stores such as Riak, Cassandra, Voldemort - implementations of Dynamo white paper [17]. Hypertable[11], Accumulo[12], HBase[13] - modeled after BigTable. Couchbase[14], Redis[15] - inspired by concepts of memcached.

Alongside with these data stores graph databases emerged (such as Neo4j[16]) being included as NoSQL because of solving data releated problems that current SQL database engines cannot. Most

2.2.3. CAP theorem

In 2000 Eric A. Brewer in his presentation "Towards Robust Distributed Systems" [5] at *Symposium on Principles of Distributed Computing* formulated conjecture based on his experiences with large scale systems. He stated that if distributed system is meant run in a environment that allow situations like:

– disk and computer hardware failure,

– software upgrade/maintenance time off,

[10] http://www.memcached.org/
[11] http://hypertable.org/
[12] http://accumulo.apache.org/
[13] http://hbase.apache.org
[14] http://www.couchbase.com
[15] http://redis.io
[16] http://neo4j.org

– operating system upgrades,

– power failures,

– network outages,

– need to relocate hardware.

It can only posses maximum two features from:

Consistency - all nodes in system access the same data.

Availability - system always can report back if operation was successful or not.

Partition tolerance - system is able to operate when there is failure in communication between parts of the system.

In 2002 Seth Gilbert and Nancy Lynch proposed formal model and prof of the Brewer's theorem [21]. Therefore we can use CAP theorem to divide distributed systems into 3 categories for each different combinations o 2 qualities mentioned earlier. Next sections try to describe systems belonging to each category using databases as an specific example of DS. However note must be taken that most of the NoSQL databases given in examples are highly configurable in terms of qualities described here. By putting them to one of 3 categories we take into the consideration their default parameters and use case.

Forfeit Partitions

This is category which includes all "classic" RDBMS, cluster databases, some distributed file systems. In order to achieve consistency in distributed environment the write or read operation must be done exclusively. This means that all nodes must take part and in such operation. Such property excludes the database being able to operate in face of network split when hosts cannot communicate between each other. Node or network split in cluster mean that database is unavailable.

Forfeit Availability

Systems like scalable NewSQL databases, Google's BigTable, HBase or specific distributed databases (for example Mnesia[17]) are designed to run on many nodes while always accessing the same data. Nevertheless they cannot guarantee the consistency while running when one of hosts is down. In such conditions some operations on database might be successful while some which need disconnected node to participate might fail.

Forfeit Consistency

To this category belong systems like DNS, World Wide Web (WWW) caching and highly available data stores like Dynamo. These systems cannot promise consistency because they allow clients to read and write data even when communication between nodes is down. This leads to conflicts which are not rejected during write but are promised to be resolved as soon as it's possible after communications is back again.

In Figure 2.6 popular databases were linked to the CAP features they posses. All SQL data stores are placed in consistency-availability zone while NoSQL databases are placed in every combination of three features.

2.2.4. NoSQL database types

There is no officially agreed taxonomy for NoSQL database however in vast amount of sources following categories could be spotted.

[17] http://www.erlang.org/doc/man/mnesia.html

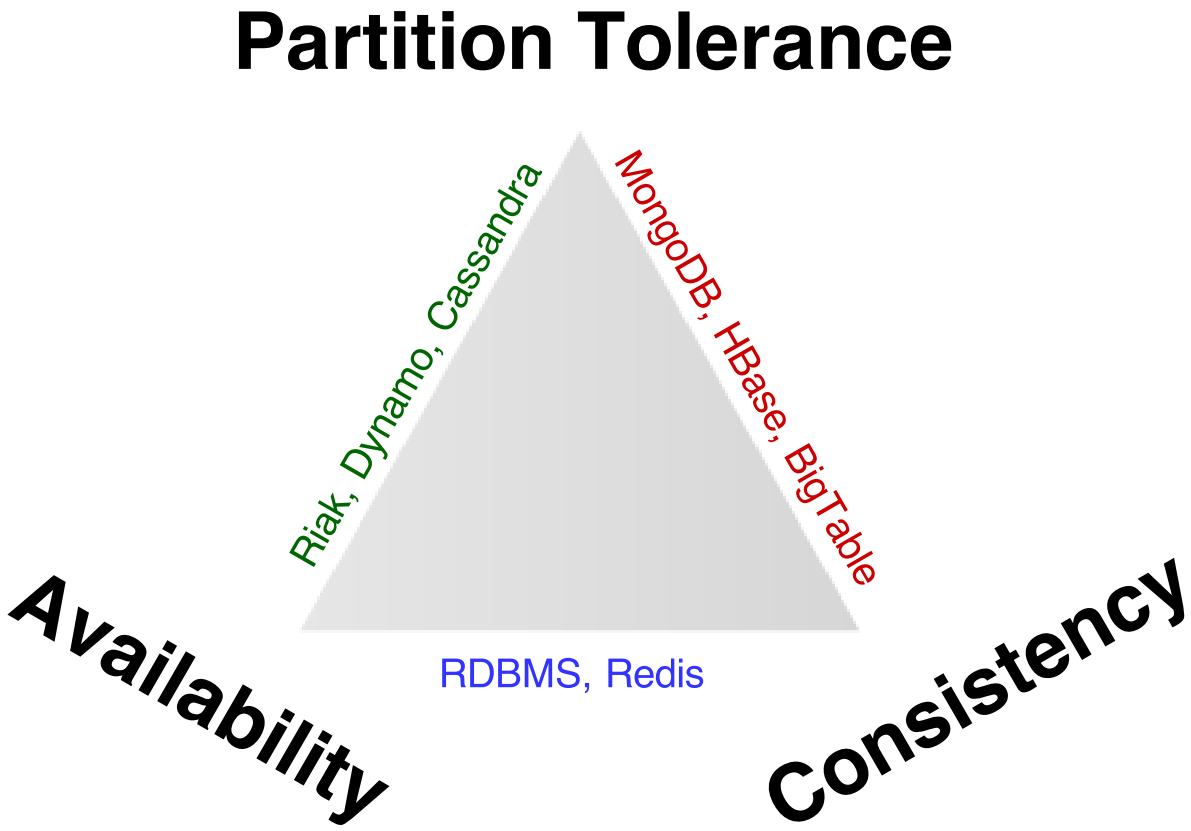

Figure 2.6: Popular databases in context of CAP theorem

Key-Value

Allow the application to store any data in any format under specified key.

Document oriented

Operates on data encapsulated and encoded to document. ‘

Wide-column

Stores data into multidimensional sorted map.

Graph

Focuses on storing rich relations between data units.

2.3. NoSQL databases selected for this thesis

There is vast number of NoSQL data stores available nowadays. In this thesis two of them were analyzed and tested: MongoDB and Riak, for the following reasons. These databases are quite different in concept and features. Their tests and deep comparison should point out advantages and shortcomings in terms of scientific applications. Also having tested these NoSQL data stores alongside popular, open-source member of RDBMS family – MySQL should help answer the question: under what circumstances NoSQL are better option then traditional database engines?

2.3.1. MongoDB

The creators of MongoDB present it [32] as world's leading NoSQL database. They claim [29] that MongoDB posses features such as "full index support", "replication & high availablity", "automatic sharding", "rich document-based queries", "fast in-place updates", "map/reduce".

All these things together promise scalability and flexibility of an NoSQL database while maintaining consistency and advanced querying mechanisms known from RDBMS engines.

Advanced indexing

Despite the fact that MongoDB is schema-less database it supports creating various types of indexes that should dramatically fasten searching large number of documents. There are different types of indexes:

Main – the main unique index on `_id` field which is mandatory to every document. If new document does not posses this field the data store generates it.

Secondary – an index creted by user that can be based on any field in document. It can be created anytime however adding new index to a large collection could start long lasting indexing operation.

Unique – an index that causes MongoDB to reject a document that posses duplicated value for the indexed field.

Sub-document – an index holding sub-documents. Allows queries that select documents comparing whole embedded documents.

Embedded fields – an index based on a field of a sub-document.

Compound – an single index but based on two or more fields of a document.

Multikey – an index on a field that contains an array. Supports querying for documents that posses specified keys in these arrays.

Sparse – an index containing entries only for documents possessing the indexed field.

Hashed – an index containing entries which are hashed values of the indexed field. Used for equality comparison and for partitioning.

Replication and Partitioning

MongoDB defines Replica Set and Shard. Both solutions assumes running mongodb in distributed environment where each node is a *mongod* instance running exclusively on one machine.

Replica Set aims to provide redundancy and increased availability [30] by managing a group of *mongod* instances and replicating data between them. The replica set consists of one *primary* instance, one or more *secondary* instances and optional *arbiter*.

The *primary* node is responsible for handling all write operations to *replica set*. It writes data to disk and to oplog.

Secondaries are nodes which mirror data that *primary* is writing. The repliation is done in asynchronus way. The operations from oplog are streamed from *primary* to *secondaries*.

Arbiter nodes are optional nodes that help in election of the *primary*. Whenever *primary* is not reponding to heartbeat requests the *Replica Set* elects new *primary* from *secondaries*.

All read operations goes to *primary* node by default. Therefore without changes on client side introducing replica set helps only to make service more fault tolerant and prevent data loss in case of primary node crashes,

power failures etc. The client can configure driver when it comes write/read operations. By manipulating with *write/read concerns* levels it can trade performance for durability and fault tolerance.

Write concern levels are:

Errors Ignored – the write operations are not acknowledged and all connectivity errors are ignored.

Unacknowledged – the write operations are not acknowledged but the driver reports all connectivity related errors.

Acknowledged – driver wait for the write operation to be successfully written to database. This allows to detect errors like *duplicated key*.

Journaled – the write operation returns after write completes and the operation is written to oplog.

Replica Acknowledged – the write is acknowledged by specified number of *secondary* nodes.

Read concerns levels are:

Primary – all reads are performed using *primary* node.

Primary preferred – if *primary* is unavailable, operations red from *secondaries*.

Secondary – all reads are performed using *secondaries* nodes.

Secondary Preferred – only if *secondaries* are occupied *primary* is used.

Nearest – all reads are performed using the nearest node.

If MongoDB clients turn on reading from secondaries nodes the throughput of read operations is greatly improved at cost of data no longer guarantees to be up to date. If this is unacceptable the client can configure the write operation to succeed only if all replicas are acknowledged. This however affects the write performance since each operation need to wait for all replicas to acknowledge write. This configuration might be find itself really useful in application when number of reads out-spaces number of writes.

Setting *Primarry preferred* on the other hand causes the MongoDB to always use *primary* guarrainting consistency while also ensuring that if *primary* crashes the reads will be served by *secondaries*.

Another example of read/write configuration tighten to application need is *nearest* level which works really well in database distributed across distant geographical locations. The *mongos* instances ran usually alongside client applications measure the latency to particular nodes and choose the nearest. This ensures optimal performance while retaining fault tolerance and automatic syncing between data centers.

Journaled level is very desired if highly valuable data is stored. Writing to the journal ensures that even if nodes crashes and corrupts database state the data can be recovered from oplog. Nevertheless the *journaled* level waits only for *primary* node to complete oplog writing. It cannot be configured to wait for *secondaries* to write to oplog as well.

Despite many configuration options for read and write operations, MongoDB was criticized [39] for not being sufficient in some failure scenarios. For example user cannot set write concern that results in data being persisted on disk on more then one node. It's because journaled concern affects only primary node no matter now many replicas we have.

Shard in terms of MongoDB is a partition taking care of only part of data set. MongoDB claims that use of shards lets achive high throughput of read/write operations and allow to handle very large data sets [31].

In theory each document in sharded collection is stored only on one shard. Therefore client who write it or access it operates on only one instance of *mongod* without even knowing. The decision to which shard particular

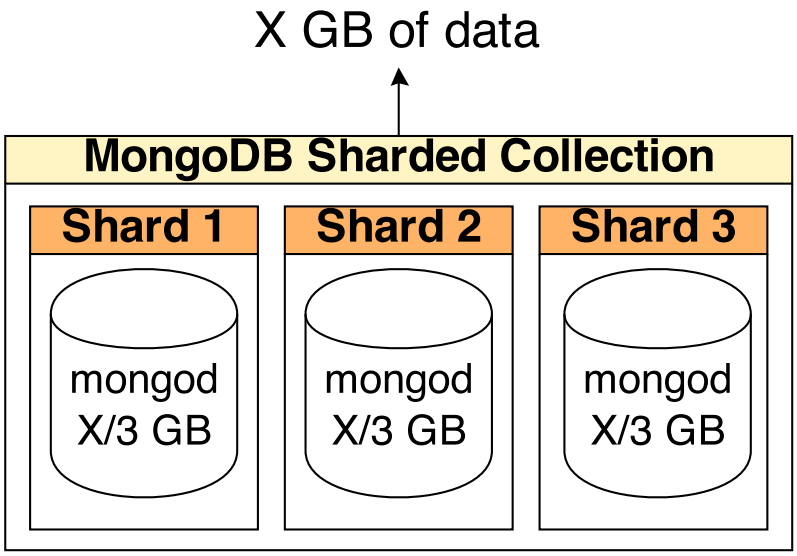

Figure 2.7: Single collection evenly divided into 3 shards running on separate machines.

document belongs is based on *sharding key* which is a MongoDB index. It is performed in mongos processes that are also responsible for configuring shard in config server and maintaining it (balancing).

Figure 2.7 shows a typical setup in which 3 nodes run *mongod* instances which together host single dataset. The data is distributed evenly meaning that MongoDB always tries to achive state in which each shard holds equal portion of data. Internals try to dynamically split items into equal partitions and distribute them between shards. The database allows users to choose the field which will be used to make decision to didvede the data. This field is called *shard key* and it needs to be a indexed field.

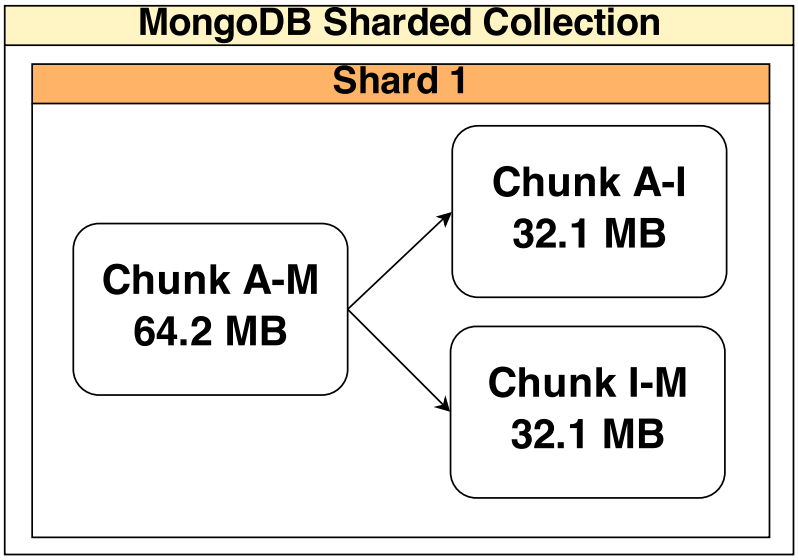

Figure 2.8: Diagram of a shard with a chunk that exceeds the default chunk size of 64 MB and triggers a split of the chunk into two chunks.

[18]

[18] http://docs.mongodb.org/manual/core/sharding-introduction

At first all items go to single chunk on a single shard. As soon as chunk exceeds globally set limit 64 MB[19] the MongoDB attempts to split the chunk into two. Each one contains items that are within calculated key range which guarentees same size for both newly created chunks. Figure 2.8 shows example split.

According to documentation[20] whole mechanisms works best when three conditions are fulfilled:

1. The collection already consists of chunks located on different hosts.
2. Consecutive inserts contain keys from different partitions.
3. Key values are distributed uniformly.

Assuming that DB MongoDB is using default settings, if there is only one chunk in collection (less then 64 MB of documents) all inserts will go to only one shard until chunk splits and data can migrate to remaining shards.

When subsequent writes are very similar they are can be usually handled only by single node which also limits ability to fully utilize power of every host in cluster.

If keys are not distributed uniformly (for example there is big probability of items having the same id) some chunks are overloaded. Therefore on single host occur lots of of expensive splits and migrations.

The problems mentioned above are important when designing and tuning the performance of a particular application that uses MongoDB, and during the work on this thesis we had to deal with them and find solutions.

2.3.2. Riak

Riak is open-source, distributed database that was designed to provide maximum data availability. It was very inspired by concepts described in Amazon's Dynamo white-paper [17].

It is designed to run in distributed environment on a number of hosts. The documentation [25] highlights its four main goals: availability, operational simplicity, scalability, masterless.

Masterless, distributed architecture

Riak database is designed to run as a distributed system, using more then one host. Their creators advise running at least 5 nodes to achieved desired performance under default, safe settings[21].

The cluster consists of nodes which are instances of riak program – usually one per host. Cluster is very uniform, meaning that each node is equal and perform same tasks as rest of the nodes. This architecture in threory has following advantages.

No single points of failure

Each node in cluster is equal and performs same tasks, cooperating with the others. Although cluster members do not depend on each other which means that failure of one node or a network split does not stop database from functioning.

Simplified maintenance

All Riak hosts are configured in very similar way. Also whole database does not require complicated startup procedure. Therefore configuration and maintenance can be simplified and streamlined lowering probability of operator's error.

Extendibility

Due to usage of gossip protocol all nodes in the cluster communicate with each other exchanging information

[19] http://docs.mongodb.org/manual/core/sharding-chunk-splitting/
[20] http://docs.mongodb.org/manual/core/sharding-introduction/
[21] http://docs.basho.com/riak/latest/theory/why-riak/

about database state and actual members of the cluster. This allows new hosts to be join or leave cluster automatically without any manual configurations.

Keys and consistent hashing

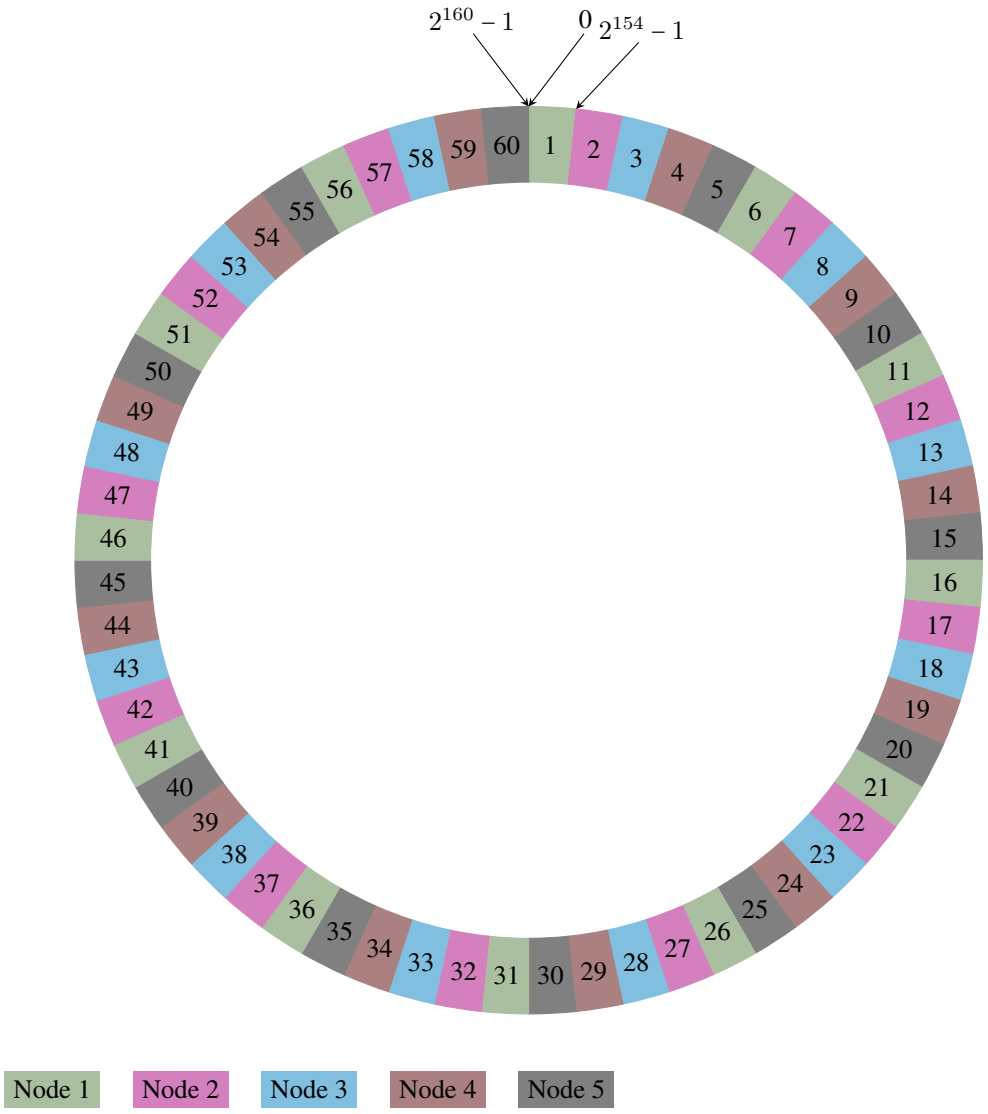

Figure 2.9: Diagram of Riak's key space divided into 64 partitions assigned to 5 distinct nodes.

Riak is key-value store therefore unlike traditional RDBMSs or MongoDB it doesn't put any constraint on data being stored nor examine it's content. It requires only each data item to be associated with a unique key.

The unique key is a string used to access the object and also to determine it's actual location inside the database. It is transformed to 160 bit long value out of it using SHA-1 algorithm. This value indicates object's position in the *ring* which is a continuous space divided into 64 equal partitions. As presented in Figure 2.9 every partition in the rings managed by one of the Riak node's. These partitions are called *virtual nodes* or *vnode*. Information about vnode mapping is stored on every node and kept in sync.

Such architecture enables efficient routing because every node knows where exactly documents are physically stored so it can route directly a request from client to target node.

It also simplifies any migration of data between Riak nodes as partitions are fairly even and have fixed boundaries.

Replication and Partitioning

One of the main features of Riak is strong, configurable fault tolerance which can be configured by changing replication and querying parameters known as N, R, W.

N – defines how many hosts should store single value. Upon write algorithm always stores object in node that is assigned to item's partition. Additional nodes are the ones that are responsible for consecutive partitions. Because adjacent partitions are always assigned to distinct nodes there is no risk that 2 or more replicas are stored on one physical host that could go down and irreversibly lose data.

R – defines how many replicas should be successfully fetched before read operation is considered successful and returns to client with result.

W – defines how many replicas should acknowledge writing item before write successfully returns to client.

Different combinations of parameters above allow to do trade between latency, availability, partition tolerance and level of consistency.

Storage backends

One of Riak's unique features are pluggable storage backends which allow to choose different method of persisting data on a node. The database comes with 3 backends built-in.

Bitcask

This is default option for storing data on Riak node. This engine is integrated with Riak as it is Erlang application which coexists with Riak inside one Erlang virtual machine.

Information about position of specified document in disk file is stored in fast hashtable. Each read operation requires constant lookup in hashtable, one *seek* call to operating system and reading bytes. Write operations consist of hashtable operations and appending a file. These are all operations that are considered optimal in such scenario.

Therefore it's possible to achive low latency and predictable performance. Also thanks to append only method of writing database file backups are greatly simplified.

Weakness of descried solution lies in hashtable that stores positions associated with keys. It must fit in the host memory otherwise it terminates abnormally.

LevelDB

The backend that uses an open source key-value store LevelDB[22] written by Google Inc. It performs fast compression using Snappy[23] algorithm so it can save significant amount of space when storing large documents containing lots of redundant information. It is not limited by host's memory as Bitcask. It is also proven to be very fast when it comes to both read and write operations [26].

Memory

Memory backend is very simple engine that stores all data in memory. Disk is not used at all therefore performance is greatly improved but available space is very limited and node failure always results in data being lost.

Table 2.2: Comparison of key characteristics for MySQL, MongoDB and Riak

Database	Type	Consistency	Availability	Clustering	Indexing
MySQL	RDBMS	Strong consistency achieved by transactions.	High availability, no dependencies on remote hosts.	Master-slave replication only, cannot divide data into partitions handled by separate hosts.	B-tree indexes on columns, R-tree spatial indexes
MongoDB	Document store	Strong consistency achieved by readers-writer locking.	Low availability, only primary nodes can alter data set	Supports partitioning, "Replica sets" mechanism for increasing fault tolerance and availability.	B-tree indexes on fields of document
Riak	Key–value store	Eventually consistent, multiple conflicting version of data might occur. Uses vector clocks.	Always available, no primary hosts. Failure of single host doesn't affect delivery.	Distributes and replicates data uniformly between all available host.	Secondary indexes appended to key-value pair

2.4. Summary

In this chapter we presented history of databases and explained how they evolved during last 50 years into very essential component of current systems. Later we described and compared in detail different types of databases explaining their purpose and principles. Next we choose two NoSQL products to be take part in our test and we described them in detail. Finally in table 2.2 we tried to summarize their key features and compare them against SQL database.

[22] https://code.google.com/leveldb/
[23] https://code.google.com/snappy/

3. Related Work

Section 3.1 presents transactional benchmarks maintained for years by TPC organization. Section 3.2 describes newer approach taken in Yahoo Labs that was tailored for NoSQL solutions. Last section 3.3 tries to discuss results described in different research papers.

3.1. TPC

The *Transaction Processing Performance Council* is a non-profit organization that delivers independent, reliable, transparent benchmarks for transaction processing and databases. It was founded in 1988 and today it gathers many IT companies alongside researchers and independent experts. Together they work on defining test procedures and conducting tests of different data stores.

All maintained test procedures require database that posses Atomicity, Consistency, Isolation, Durability (ACID) properties and their specifications often operate on terms specific to relational databases.

Currently the organization has number of benchmarks types and specifications. They are defined in detail and try to recreate conditions known from typical business applications. They aim to provide reliable info not only about the performance but also about cost and energy efficiency.

The results are available online[1] and they contain hundreds of test runs conducted on different databases, hardware and operating systems.

3.1.1. TCP-C

The TCP-C [15] benchmark is simulating computer environment used for managing warehouses or stocks where large population on independent clients executes transactions against several databases simulating placing orders, entering products etc. Such conditions are rather common for a system that is involved into managing, selling or distributing products and services.

The primary result of this benchmark is number of transactions per minute (tpmC) and associated price per transaction ($/tpmC).

3.1.2. TCP-DS

Another benchmark *TCP-DS* [16] which models load that is generated by typical decision support systems used in business. In scenario like this user operates on large data set executing reporting, ad hoc, iterative OLAP and data mining queries.

The result of benchmark is *QphDS@SF* which is a metric that is calculated from scale factor, number of queries divided by total time of tests.

[1] http:/www.tpc.org/information/results.asp

3.1.3. TCP-E

The *TCP-E* suite simulates the workload typical for brokerage firm. The test driver simulates customers of the firm and stock exchange that performs transactions on the same data. The goal here is to measure the performance of a single, central database that holds customer accounts information.

The performance is measured in transactions per second (tpsE).

3.1.4. Others

Remaining specifications define standards for measuring performance in virtual environments (*TPC-VMS*), cost effectives (*TPC-Pricing*) and energy efficiency (*TPC-Energy*).

3.2. YCSB

One of the most notable tool used to measure the performance of NoSQL system is the *Yahoo! Cloud Serving Benchmark* written by Brian F. Cooper. It was described for the first time in [14] providing also performance figures for a few databases that were available for use at that time. The tool tried to establish a standard routine for checking most important characteristics of distributed databases used in general web development. It consists of a two tiers: Performance and Scalability.

The Performance tier focuses on mean response time for each request because it assumes that typically in web applications database drives the responsiveness of whole application seen from user perspective. If requests to database take very long user might be unsatisfied or it will not receive any response due to timeout in higher layers. It answers what is maximum throughput under which database is still reasonably responsive.

The Scalability tier is oriented around measuring performance and elasticity in regards of number of nodes in a cluster. First, the same test runs are executed against database cluster with different number of hosts to establish the benefit of adding new nodes to cluster. Second, the database is added a node during the test to measure how fast it will take advantage of additional host.

The benchmark presented in [14] was testing Cassandra 0.5.0, HBase 0.20.3, 2 MySQL based solutions. Cluster consisted of six server-class machines and it revealed differences in performance under different work loads and more interestingly showed what are limits for throughput they can handle.

The scalability results showed good horizontal scalability when it comes to read performance. It also highlighted *Cassandra's* issue with adding new database host to cluster under load. This operation worsen the performance mostly due to inefficient replication mechanism which was moving data to new host.

The YCSB project is still actively developed and available open source at [13]. It has became a popular tool used to perform benchmark. For example [7] is a benchmark published in 2012 which used newer version of YCSB and tested more recent versions of databases mentioned above and also Riak 1.1.1 and MongoDB 2.0.5. The test was conducted using 4 Amazon AWS[2] instances of type *m1.xlarge* and highlighted impact of different databases setting and features such as deferred log flushing or mapping files in memory.

3.3. Studies regarding NoSQL databases

In [27] researchers use YCSB and their custom framework to measure impact of adding new hosts to database cluster in cloud environment. Authors were using from 8 to 24 virtual machines from private OpenStack instance

[2] http://aws.amazon.com/ec2

to run HBase, Cassandra and Riak cluster. They measured time that it takes for the cluster to rebalance and the impact of this rebalance to latency and other metrics. Interestingly they showed differences in amount of data that was moved during rebalance highlighting the the strengths and weaknesses of different implementations.

Another interesting research [18] studied MongoDB as a backend for Hadoop, comparing it to it's default storage HDFS. Elif Dede at al. established a number of different tests for single and 2 node configuration of MongoDB. First they tested it as storage for Hadoop workers checkpoints measuring how many concurrent workers can write every 10s large data (1MB or 64KB) without causing any overhead on Hadoop job. Next test compared HDFS and MongoDB performance in writing and reading small documents. The results showed that HDFS is faster in both read and writes by 3:1 and 24:1 ratio. Third test reveled that MongoDB built in MapReduce functionality is on average five times slower then a setup consisted of 2 Hadoop workers that use MongoDB as data store. Following tests tried to compare this setup against Hadoop using HDFS. Again the results showed that MongoDB is much slower compared to competitor. Interestingly study showed that there was almost no difference in the performance when using one or two MongoDB host in sharding setup.

3.4. Summary

In the past there were successful attempts to obtain objective, comparable performance figures for SQL database. The TPC organization still maintains benchmarks and rankings of transactional systems helping IT experts in making important design decisions.

Unfortunately those tools cannot be used to measure NoSQL databases. Subject of comparing NoSQL performance and capabilities gains more attention as the adoption of these databases constantly grows. Number of studies is still low and they show big differences in results. Because there is no standard for these type of databases they often show big difference in the way they try to deliver the same functionality to client. Often implementation and configurations details have big effect on the results leading to confusion and wrong conclusions.

It seems that almost every decision whether to use NoSQL solution or not should be made after careful and detailed studies. These can be very difficult to conduct without proper literature and well documented tests. Therefore until industry accepts well defined and accepted standards for implementing and benchmarking NoSQL databases it's worth to conduct experiments similar to this thesis.

4. Design of the experiment

In section 4.1 we analyze requirements of scientific computation that we used as reference and design benchmarking procedure based on them. Section 4.2 describes test environment that was used. Last section 4.3 describes how we gathered various metrics of systems under test.

4.1. Benchmark application and workload

To find if and how NoSQL databases can be used in scientific applications benchmarking involves using actual scientific data that was acquired due to courtesy of Mr Tomasz Gubała from ACC Cyfronet. The data is part of a research that included finding matches between 46655 RNA fragments and of all human genes.

In this study each RNA fragment was compared to every sequence from each of 24 human chromosomes by running *SSEARCH* program from FASTA [35] framework. The result of *SSEARCH* program is stored in single file (PASTA format, described later) which has size between 9-12MB. Whole available data set consists of 1,119,820 files and weights almost 13TB.

Such large data set of results requires a lot of computations in order to extract informations that could be used to formulate some conclusions about the whole experiment. This could be achieved by running many parallel tasks, executed against these files on a cluster using PBS queuing system. Such procedure returns single results after all documents are read. The biggest drawback of this solution is that a single change in map reduce procedure requires running it against all input files.

4.1.1. Requirements for experiment

The goal of these tests was to simulate a job that would insert all the documents to database first and then access it directly from there. This should be much more efficient then accessing data by directly reading it from files. The job should a single, simple program that could be executed in any grid or cloud environments.

To get the requirements for such job a number of tests and manual checks were done using real grid *Zeus* which is a supercomputer ranked in TOP500 list [43] and operated by Academic Computer Center *CYFRONET*.

In environments like this user usually don't allow direct access to shell and underlying operating system. User has to submit a shell script that is then executed by a scheduler on random nodes. The API and environment variables are very limited especially when it comes to running many concurrent jobs that need to be coordinated.

The test procedures should address the following problems we faced when running the experiments in a distributed environment such as the Zeus cluster:

Synchronization - The database nodes requires to be initialized first, before workers are started. Also reading test shouldn't begin when there are some workers that still hasn't finish writing the data.

Communication - Workers responsible for reading and writing data need to be provisioned with database endpoints which are hostname or ip addresses. In grid or cloud environment they are assigned randomly.

Different hardware requirements - Workers and database nodes require different nodes. Databases utilize CPU, RAM and disk differently but in general the performance can be greatly improved by using increasing nodes specification. Also discrepancy in nodes specification can lead to bottlenecks and benchmark unreliability. On the other hand, worker's script usually doesn't have such requirements and it's meant to be ran on any available hardware which reduces costs or speed up time spent in the job queue.

The testing procedure described in next section was designed to tackle these problems. It is very modular, consisting of stand alone components that are highly configurable, written in Python and delivered as a single package. It includes benchmark program that reads input and writes and then reads, monitoring which gathers information about system under test, test executor which schedules execution of right steps on different hosts and provider which creates and configures hosts in the cloud.

4.1.2. FASTA files

The test procedure involves operations on real scientific data. In this case these are results of genome related computation conducted in order to find which RNA fragments.

For the sake of experiment random 500 of these were chosen and placed on disk. So that each machine taking part in experiment had access to them no matter which role it played in experiment.

The FASTA output is in PASTA format and posses well defined structure containing results of a computation that need to be loaded, processed, filtered and then put to database by benchmarking programs.

Listing 4.1 shows the beginning of such file. It contains information regarding target RNA sequence and parameters received by *SSEARCH* program – values prefixed with pg_ and *mp_*.

Listing 4.1: Begening of FASTA results file. Contains information about parameters and examined fragment.

```
1  >>>gi|100413100|ref|NR_003084.1|, 1641 nt vs cds_chr1.fa library
2     ; pg_name: /fasta-36/bin/ssearch36
3     ; pg_ver: 36.06
4     ; pg_argv: /ARGUMENTS/
5     ; pg_name_alg: Smith-Waterman (SSE2, Michael Farrar 2006)
6     ; pg_ver_rel: 7.1 Aug 2009
7     ; pg_matrix: +5/-4 (5:-4)
8     ; pg_open-ext: -12 -4
9     ; mp_extrap: 60000 7949
10    ; mp_stats: Expectation_n fit: rho(ln(x))= 9.4720+/-0.00156; mu= 26.9838+/- 0.103 ↙
          ↳ mean_var=198.4934+/-36.994, 0's: 0 Z-trim(115.5): 3 B-trim: 0 in 0/51 Lambda= ↙
          ↳ 0.091034
11    ; mp_KS: -0.0000 (N=0) at 20
12    ; mp_Algorithm: Smith-Waterman (SSE2, Michael Farrar 2006) (7.1 Aug 2009)
13    ; mp_Parameters: +5/-4 matrix (5:-4), open/ext: -12/-4
```

For each compared Coding DNA Sequence there is an entry (see Listing 4.2) describing similarity between Coding DNA Sequence and target RNA. An entry contains score (line 7) which informs how good matched fragments are. After that their exact location (line 11) is given alongside with fragments themselves.

Listing 4.2: Fragment of a file containing FASTA results

```
1    >>gs_sequence|5412|gs_gene|5412|chromosome|1
2     ; sw_frame: f
3     ; sw_s-w opt: 219
4     ; sw_z-score: 145.2
5     ; sw_bits: 37.9
6     ; sw_expect: 0.053
```

```
7    ; sw_score: 219
8    ; sw_ident: 0.673
9    ; sw_sim: 0.673
10   ; sw_overlap: 147
11   >gi|100413100 ..
12   ; sq_len: 1641
13   ; sq_offset: 1
14   ; sq_type: D
15   ; al_start: 432
16   ; al_stop: 574
17   ; al_display_start: 432
18   AAAGTATATGTGTGTGT--GTGTGGAGCTGAGACAGGCTCGGCAGCGGCA
19   CAGAATGAGGGAAGACGAGAAAGAGAGTGGGAGAGAGAGAGGCAGAGAGG
20   GAGAGAGGGAGAGTGACAGCAGCGCTCGAGAC-GGACGGCA-AGCGG
21   >gs_sequence|5412|gs_gene|5412|chromosome|1 ..
22   ; sq_len: 453
23   ; sq_offset: 1
24   ; sq_type: D
25   ; al_start: 161
26   ; al_stop: 303
27   ; al_display_start: 161
28   AAAGTAAGAGAGAGAGAAAGAGAGGAAGAGAGAGAGGAGAGGAAGGGGGG
29   GTGGA-GAGG-AAGAC-AGAGAGGAAGAGGGAGAGGGAGAGGAAGAGAGA
30   GAGAGAGGGAGAGGAAGAG-AGAGAGGGAGACAGGAAGAGAGAGCGG
```

4.1.3. Benchmark program

The benchmark application is a program which loads pasta file, parses its content from FASTA format and converts it to JSON (see Listing 4.3) that is accepted by both databases, and writes it to desired database. It also saves document ids that will be randomly picked during read test that follows write test. The program counts how many separate documents were written or read in each second.

Figure 4.1: Write and read characteristics of benchmark program running without any database connected.

The application was designed with speed in mind so that it has minimal impact on database performance measurement while still being realistic example of scientific computation. Results of benchmark without any database connected presented in Figure 4.1 shows that single program was able to process 100 input files in 233s creating 5017837 documents. This gives average speed of 21444 writes/s that is maintained during whole run. In read phase program showed that it's capable of doing more then 60000 requests/s. This proves that these program run concurrently on many machines is able to generate load that will be sufficient to overload desired databases. The test was conducted in the same environment as real tests described in chapter 5.

Listing 4.3: Example documents created by benchmark program from pasta files

```
1  //tasks
2  {
3      "_id" : ObjectId("536426579182bf0b79da7954"),
4      "t" : "601",
5      "mp_Parameters" : "+5/-4 matrix (5:-4), open/ext: -12/-4\n",
6      "pg_name" : "/people/ymgubala/fasta/fasta -36.2.7/bin/ssearch36\n",
7      "pg_open-ext" : "-12 -4\n",
8      "pg_matrix" : "+5/-4 (5:-4)\n",
9      "pg_ver" : "36.06\n",
10     "mp_Algorithm" : "Smith-Waterman (SSE2, Michael Farrar 2006) (7.1 Aug 2009)\n",
11     "mp_KS" : "-0.0000 (N=0) at  20\n",
12     "mp_stats" : " Expectation_n fit: rho(ln(x))= 11.3408+/-0.00165; mu= 21.4283+/- 0.108  ↙
           ↳ mean_var=227.0713+/-43.292, 0's: 0 Z-trim(115.5): 81  B-trim: 0 in 0/51  Lambda= ↙
           ↳ 0.085112\n",
13     "mp_extrap" : "60000 7878\n",
14     "pg_argv" : "/people/ymgubala/fasta/fasta -36.2.7/bin/ssearch36 -Q -H -m 10 -W 0 -E ↙
           ↳ 1000000000 1.0 /scratch-lustre/ymgubala/pasta/seqs/input-34801.fasta ↙
           ↳ /scratch-lustre/ymgubala/pasta/cds_chr1.fa\n"
15 }
16
17 //rna
18 {
19     "_id" : ObjectId("536426579182bf0b79da7955"),
20     "t" : "602",
21     "b" : 3369,
22     "e" : 3797,
23     "l" : 428,
24     "s" : ↙
           ↳ "GCTCGCCAGCCCCAAGGT-CTGCAGCATCTCTCTCTGGTCTTTGT--CC-CCAGGGCC--GATGTGCCTCCGAGCGAAGTCGTCGTGTCTGGGCAG
25     "ti" : ObjectId("536426579182bf0b79da7954")
26 }
27
28 //gs
29 {
30     "_id" : ObjectId("536426579182bf0b79da7956"),
31     "t" : "603",
32     "b" : 1922,
33     "e" : 2351,
34     "l" : 429,
35     "s" : ↙
           ↳ "GCAGGCCATCGAGCAGGTGCTGAACCACCACCGTGGGGGCCTGGCGGCCGCCTGGGCCCTGCGGCGCCCCCAGATAAGGCCGCCGTGGGTG--CTG
36     "seqid" : ">>gs_sequence|1569|gs_gene|1569|chromosome|1",
37     "ti" : ObjectId("536426579182bf0b79da7954")
38 }
39
40 //align object
```

Document	Average size
Header doc	750B
RNA doc	201B
DNA doc	258B
Align doc	122B

Table 4.1: Average size of all stored documents

```
{
    "_id" : ObjectId("536426579182bf0b79da7957"),
    "t" : "604",
    "b" : 43.4,
    "e" : "0.0028",
    "s_w" : 268,
    "f" : "r",
    "i" : 0.5649999999999999,
    "rna" : ObjectId("536426579182bf0b79da7955"),
    "o" : 446,
    "p" : "",
    "si" : 0.5649999999999999,
    "ti" : ObjectId("536426579182bf0b79da7954"),
    "gs" : ObjectId("536426579182bf0b79da7956"),
    "z" : 154.7
}
```

Formats of documents are almost identical for all databases with exception to some additional metadata required by specific DBs. It also tires to be as similar to original document as possible. Sample documents are shown in listing 4.3. The fields were shorten to save space for databases that doesn't use any compression (such as MongoDB). In table 4.1 all documents have size less then 1kB.

The program has two modes - one for writing and one for reading. First phase in each run is writing data to database in following steps:

1. Get database endpoint from command line arguments.
2. Get input files locations from command line arguments.
3. Get output file location from command line.
4. Connect to database.
5. Open files that need to be read.
6. Parse them and write data to database, saving ids of all entities to disk by using pickle[1] protocol.

Second run is executed after all instances writing document finish working. During this phase programs load files containing ids written in previous phase and start to read random ids.

4.1.4. Test scenario

A single test run in terms of this thesis means completing tasks shown in Figure 4.2. Each test run involves creating from scratch every virtual machine using predefined VM image that contains all needed libraries and tools. After each run all VMs are destroyed completely leaving only logs and results that were uploaded to S3.

[1] https:/docs.python.org/2library/pickle.html

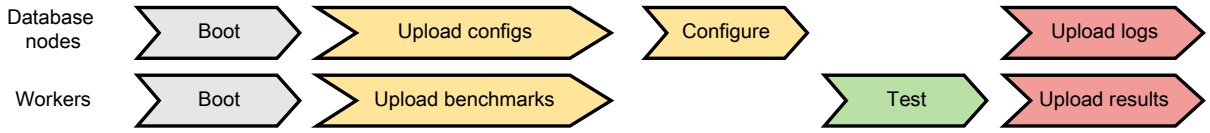

Figure 4.2: Tasks executed during single test run.

All machines in cluster were split into 2 basic groups

Database – database nodes.

Workers – responsible for all parsing and operations on database.

Thanks to that, database and workers processes were not competing for the same resources. It also introduced normal network related latency in communication between server and client.

Boot

At first all needed machines were spawned in cloud. The script waited until all of them were booted properly and every one of them was accessible via SSH.

Provision

After all the machines were started properly, the script started to upload the benchmark programs and configuration files needed for the experiment. It was also starting the databases connecting each other. At the end it was ensuring that whole cluster works together in a desired way.

Write test

During this phase worker machines start parsing and writing data from files to databases. Each instance is writing log containing information of how many documents were written during specified period of time.

Read test

After all data is inserted to data store a constant number of random documents is fetched from database to record access speed.

Tear down

After all benchmarks are done each machine uploads data to *S3* storage tagging it appropriately.

4.1.5. Test variants

Each database was tested several times using different setup and environment to see how different variables affect the performance. The main focus was on 3 scenarios: horizontal scaling, vertical scaling and how increasing data volume affects performance.

Horizontal scaling

In these test scenario number of workers and volume of data is kept constant. However in each test run additional database nodes are added. Intention of this test scenario is to measure how horizontal scaling is affecting performance.

Vertical scaling

This scenario assumed constant data volume and number of nodes. The goal here was to see the speedup generated by more powerful database nodes.

4.2. Test environment

As this thesis focuses on experimenting on large, horizontally scalable databases it required grid or cloud environment to conduct experiments. Thanks to courtesy of Amazon the experiments could be executed using Amazon Web Services one of the biggest cloud providers. Nevertheless it is worth mentioning here that experiments use open source middleware which can execute them using other than Amazon's cloud.

4.2.1. Amazon AWS

Amazon AWS is a collection of commercial web services designed to offer computational power for all sort commercial applications and also for scientific computations.

Officially launched in 2006 gained instantly popularity among people searching for on-demand cloud computing and accompanying services. Thanks to AWS in Education Grant from Amazon we were able to utilize following products in this thesis:

– EC2 supplying hosts

– EBS providing persistent storage for data

– S3 storing the results of each run

Therefore the experiments were not tightly coupled to any custom, unique services like SWF, Elastic Load Balancing.

Elastic Compute Cloud

The Amazon Elastic Compute Cloud is one of the core services. It can deliver many virtualized hosts as available almost instantly from the Internet. They can be chosen from wide range of instance types that differ in CPU power, amount of memory or IO performance. From the experiment and thesis point of view the EC2 had following advantages:

Popularity
: As previous chapters describe most NoSQL databases were developed with cloud computing in mind. EC2 is one of the most popular clouds in the world used for deploying all sorts of distributed systems. Testing there ensured us that we face the same conditions as many other cloud related experiments.

Availability
: The instances are available almost instantly as requested. This is very convenient compared to classical grid environment where experiments described in this thesis would need to be scheduled to a queue as batch jobs and wait for execution for unspecified amount of time.

Scalability
: The Amazon AWS is one of the largest clouds in the world and it's capable of delivering up to 100 or more instances at once.

Flexibility
: The virtual machines provided by Amazon are highly configurable and come in many different flavors (as shown in table 4.2). They can run many various operations systems allowing the experiments to configure the environment according to it's needs.

From various configurations available (as shown in table 4.2) tests were conducted using *m1.xlarge* instance type as it offers moderate performance typical for common commodity hardware used in large scale deployments.

Family	Instance Type	Architecture	vCPU	ECU	Memory (GB)	Network Performance
General	m1.small	32-bit or 64-bit	1	1	1.7	Low
General	m1.medium	32-bit or 64-bit	1	2	3.75	Moderate
General	m1.large	64-bit	2	4	7.5	Moderate
General	m1.xlarge	64-bit	4	8	15	High
General	m3.xlarge	64-bit	4	13	15	Moderate
General	m3.2xlarge	64-bit	8	26	30	High
Compute	c1.medium	32-bit or 64-bit	2	5	1.7	Moderate
Compute	c1.xlarge	64-bit	8	20	7	High
Compute	cc2.8xlarge	64-bit	32	88	60.5	10 Gigabit
Memory	m2.xlarge	64-bit	2	6.5	17.1	Moderate
Memory	m2.2xlarge	64-bit	4	13	34.2	Moderate
Memory	m2.4xlarge	64-bit	8	26	68.4	High
Memory	cr1.8xlarge	64-bit	32	88	244	10 Gigabit
Storage	hi1.4xlarge	64-bit	16	35	60.5	10 Gigabit
Storage	hs1.8xlarge	64-bit	16	35	117	10 Gigabit
Micro	t1.micro	32-bit or 64-bit	1	Variable	5	Very Low
GPU	cg1.4xlarge	64-bit	16	33.5	22.5	10 Gigabit

Table 4.2: Amazon EC2 instance types available 12th September 2013.

Elastic Block Store

The Elastic Block Store is another core service of Amazon's Web Services. It's tightly coupled with EC2 aiming to provide very flexible storage system for virtual machines spawned in Amazon's cloud. It persist and manages virtual block disk devices which can be dynamically attached and detached to hosts appearing as normal hard drives from operation system perspective.

The biggest advantage of using EBS were

Replication

All data and configration that was needed by every host taking part in experiment could be stored in one EBS snapshot which could be replicated to many EBS volumes attached to every dynamically spawned machine.

Snapshots

Ability to capture storage content as a snapshot and then use it as a new EBS volume replicated to every host speed the development of experiments a lot. Also allowed to easily manage diffrent versions and variants of experiments.

Figure 4.3 shows how EBS volumes were used in the experiment. In almost every single run the same EBS volume image was used. This volume stored

– configuration files for each experiment variant,

– specially prepared directory structure for database files and configurations,

– experiment input data which were meant to be stored in databases in the experiment.

The volume image had fixed size of 50 GiB of which only 8GiB was used. Therefore a lot of space was left for any computation result.

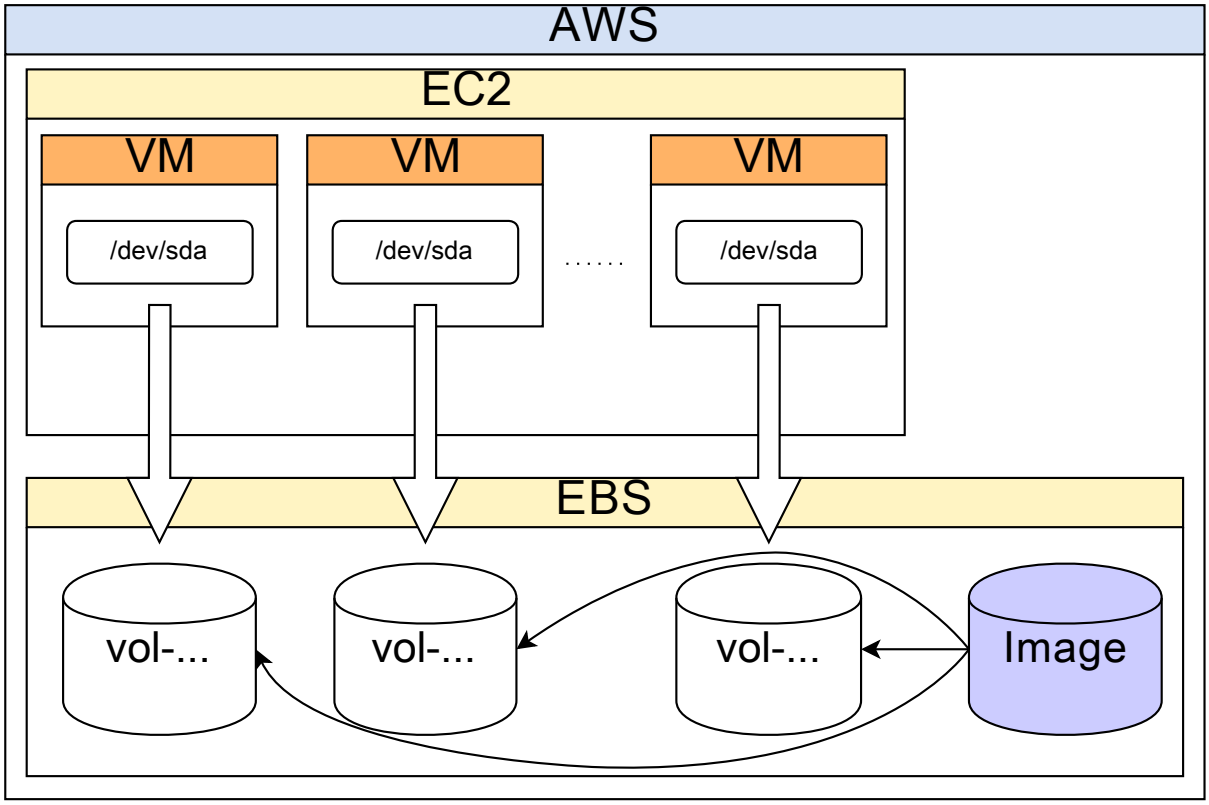

Figure 4.3: EBS Storage image used to provision all hosts in experiment.

Simple Storage Service

Amazon Simple Storage Service (S3) is yet another core service of AWS. It excels in storing and serving files via simple HTTP calls.

In the experiment S3 is used to store all experiment results such as Coma Separated Values (CSV) files with timings. Compared to EBS that also is used for storing files the S3 is

– easy accessible due to simple HTTP API available from Internet,

– allows to store huge amount of data without any concerns about running out of disk space.

Each experiment is stored as directory with name based on current time, configuration and custom name.

4.2.2. PRECIP cloud middleware

This thesis assumed using multiple nodes to run database and simulate workers using it. Therefore there was need for a tool that allows to start and prepare whole cluster for each test given just a configuration. As a result a set of programs in python was developed to execute such experiments in very controlled manner ensuring that each results can be repeatable and comparable to others.

Almost each cloud provider offers some kind of Software Development Kit containing Application Programming Interface for programmable resource management. Ussually these are mature tools that enable users taking advantage of every feature of particular platform. However the drawback of such tools is that they can only manage vendor's platform and as a result force users to stay with particular cloud provider. In this thesis experiments were expected to be vendor independent as possible so that they could be also run on other clouds in the future.

PRECIP[2] - python library were used as a foundation for experiment scripts. As it's documentation says "Precip is a flexible exeperiment management API for running experiments on clouds. Precip was developed for use on FutureGrid infrastructures such as OpenStack, Eucalyptus (>=3.2), Nimbus, and at the same time commercial clouds such as Amazon EC2.". Alongside from being cloud provider independent *PRECIP* offers many features regarding provisioning such as: syncing machines, executing shell commands, transferring files, grouping machines, easy authenticating and others.

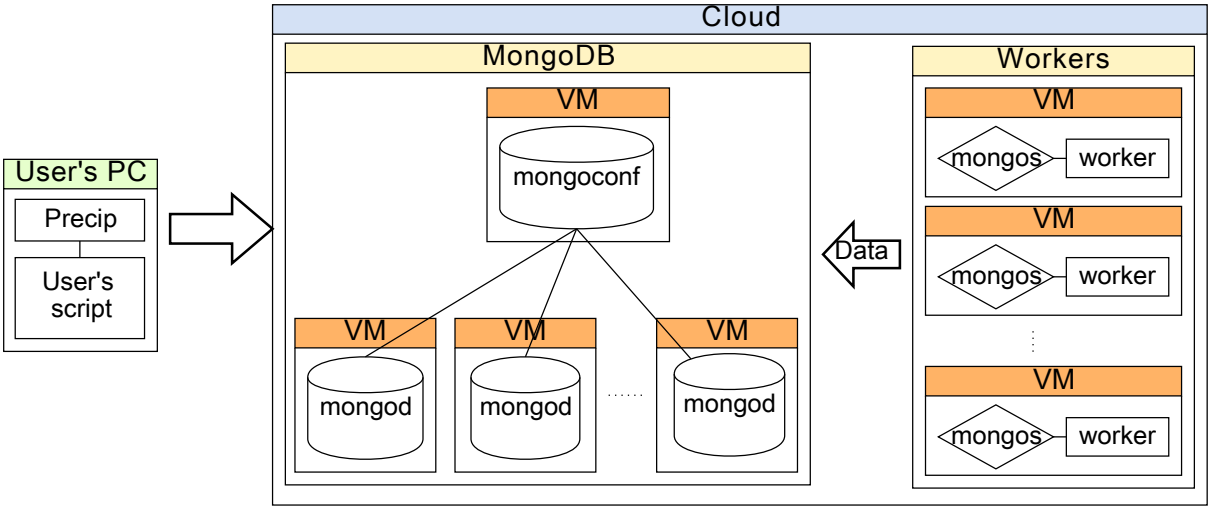

Figure 4.4: Experiment running in cloud, supervised by PRECIP script.

Figure 4.4 shows how sample experiment which uses all components mentioned previously. The whole experiment code is encapsulated in one python module which depends on *PRECIP* library. An user on it's computer can runs it. PRECIP communicates with Amazon AWS API to dynamically spawn machines, attach EBS volumes and then waits for them until they are booted. After that it configures each machine in parallel according to it's role. Next it runs shell commands which start monitoring and the experiment itself. In the end it uploads the result to S3 storage and removes instances from the cloud.

4.3. Measuring performance

To answer which database offers better performance and how it achieves this it is required to analyze many different metrics and variables. The sole numbers showing number of read or written objects are not sufficient as they cannot show many vital informations.

The focus was to capture as many informations as possible about the running database and operating system. Such data can be used to describe:

CPU utilization – showing how well database uses multiple cores and computation power in general.

Memory consumption – which tells how well database manages available memory and how much does it need to operate.

Network utilization – exposing amount of traffic directed to particular hosts.

[2] http:/futuregrid.org

```
01:06:17 AM   CPU    %user    %nice   %system   %iowait    %steal     %idle
01:06:18 AM   all     0.00     0.00      0.00      0.00      0.00    100.00
01:06:19 AM   all     0.00     0.00      0.00      0.00      0.25     99.75
01:06:20 AM   all     0.00     0.00      0.00      0.00      0.00    100.00
01:06:21 AM   all     0.00     0.00      0.00      0.00      0.00    100.00
01:06:22 AM   all     0.00     0.00      0.00      0.00      0.00    100.00
```

Figure 4.5: Listing showing output of sar command displaying mean CPU utilization for every second

In order to capture information listed above and many more useful statistics the SYSSTAT[3] project was used. It is a collection of performance monitoring tools for Linux operating system originally created by Sebastien Godard in 2002 [22]. It was designed as counterpart to Solaris's sar utility and it's mostly compatible with its it.

The package consists of many utilities. Amongst them the most notable from thesis points of view were:

sar – main executable collecting all data from the system and displaying it to user as shown in Figure 4.5.

sadc – program that runs as daemon, collects statistics and store them in binary format in desired location.

sadf – program that loads binary data from sadc, filter it and output in desired format.

During every test run *sadc* was configured to capture all possible metrics every second and write it to file. When test run was examined *sadf* converted binary data to CSV files which were matched against other results.

4.4. Summary

This chapter provided detailed information regarding test design and procedure. First we looked at an example of real scientific computation and tried to design simulation based on it. Next we explained all details regarding implementation. Later we described the AWS cloud that we used as test environment. We explained it's key features and presented PRECIP middleware which enabled us to dynamically recreate testing infrastructure. In the end we showed how we gathered useful metrics using SYSSTAT project.

[3] http://sebastien.godard.pagesperso-orange.fr

5. Results

This chapter presents the results of running benchmark procedure against target databases (MongoDB and Riak) with different configurations and number of hosts. In section 5.1 we explain in detail how and why we configured databases for these tests. Next sections describe each test series focused on different aspects of scaling. In section 5.5 we recap the results to show advantages and drawback of chosen NoSQL databases.

5.1. Test configuration

Both databases were configured to provide possibly similar capabilities. In all MongoDB tests data was written to single database called *bio_test* that had 4 collections. Riak setup used 4 buckets that are more or less equivalents of MongoDB's collections. Each of these containers was dedicated for storing one document type.

- *tasks* – storing information about *SSEARCH64* input parameters that are RNA and DNA ids
- *rna* – maintaining matched RNA fragments
- *gs* – maintaining matched DNA fragments
- *align* – saving information about position and quality of match between RNA and DNA

5.1.1. MongoDB sharding preferences

During the tests sharding was configured to optimize write operations therefore it used *hashed* sharding key described in section 2.3.1 that guarantees that data is distributed uniformly between the all hosts. Without this, MongoDB tried to divide sharding key space dynamically which sometimes crippled the performance depending on data values, especially when tests start with empty database as in our case.

5.1.2. Riak NRW settings

Riak tests were ran in configuration which tried to offer the same functionality as MongoDB configuration. Each bucket was configured to write documents to one node only (no replicas). Therefore the same level of redundancy was achieved as in MongoDB. For the configurable backend the LevelDB was chosen as it provides caching in memory that is most similar to how MongoDB caches it's data.

5.2. Number of concurrent workers

First series of tests was designed to establish the limits and the performance impact of running many parallel clients. All concurrent programs were started on a single machine. In case of the MongoDB they all used single, local instance of *mongos* program that was routing requests to actual cluster.

During each test run different number of workers spawned on a single host (called “Worker 0”) attempted to process content of 80 FASTA files which resulted in writing 4014329 to a database running on a single host (called “DB 0”). After writing workers tried to read 80000 random documents.

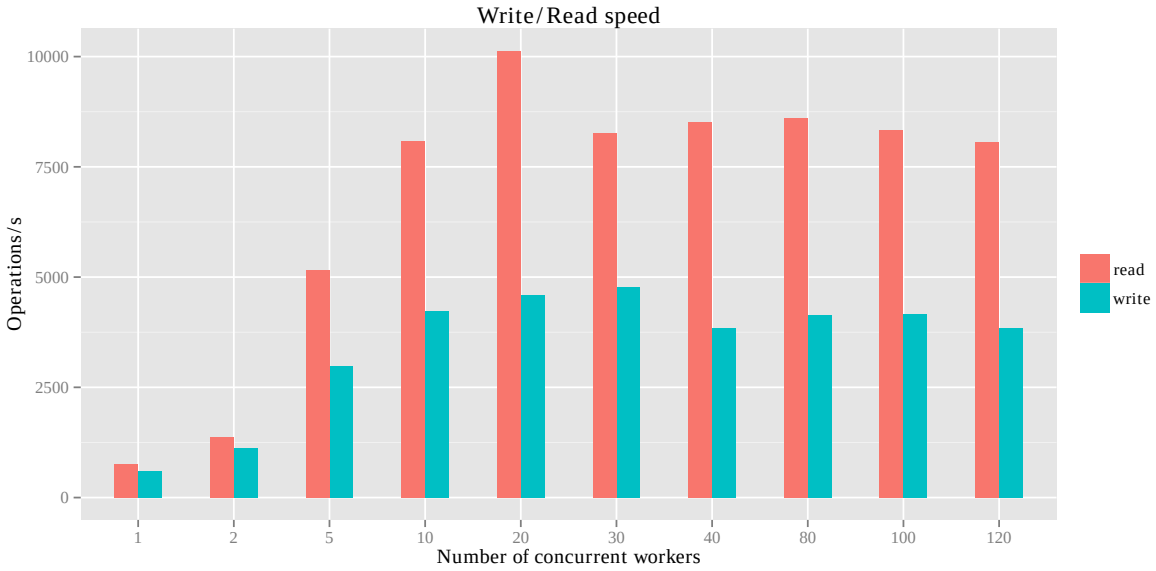

Figure 5.1: Mean throughput when running different number of concurrent workers against single MongoDB instance.

The results presented in the Figure 5.1 indicate that best average number of operations per second: 4593 for write and 10126 for read was achieved when using 20 concurrent workers. Increasing this number to 40 resulted in decrease of throughput by about 16% for both read and write. Further adding of concurrent connections didn’t have any significant impact.

Riak database in this test was more predictable but slower for every number of workers as shown in Figure 5.2. With each new worker added to database write and read throughput was improved until 10 workers were running in parallel when average number of read and write ops/s was accordingly 4705 and 3141.

Further analysis of CPU usage shows that in case of MongoDB using 40 workers caused database to spend much more time on waiting for disk operation to finish (higher iowait) which leads to conclusion that too many concurrent writes lead to less optimal way of accessing hard drives. Also CPU graphs for *Worker 0* shows lots of pauses which are probably result of workers waiting for DB to finish long lasting IO.

What is more in all tests the CPU could not reach 50% remaining around 25% for best runs. The most likely reason for this is the fact that MongoDB has global write lock which acts as single point of serialization allowing only one document to be written at the same time in one database. As 100% equals full load on all 4 cores 25% indicates that only single core was utilized.

Riak on the other hand can utilize all cores regardless of numbers of buckets and clients. It reached maximum usage when 10 workers were reading and writing as presented in Figure 5.4. The usage pattern and scores looks almost exactly the same for 10 and 80 workers suggesting that number of workers has minimal impact on performance.

Overall in this test MongoDB used far less resources achieving better performance for each number of concurrent workers. Also both databases didn’t collapsed or showed big drop in performance due to large number of simultaneous writes and read.

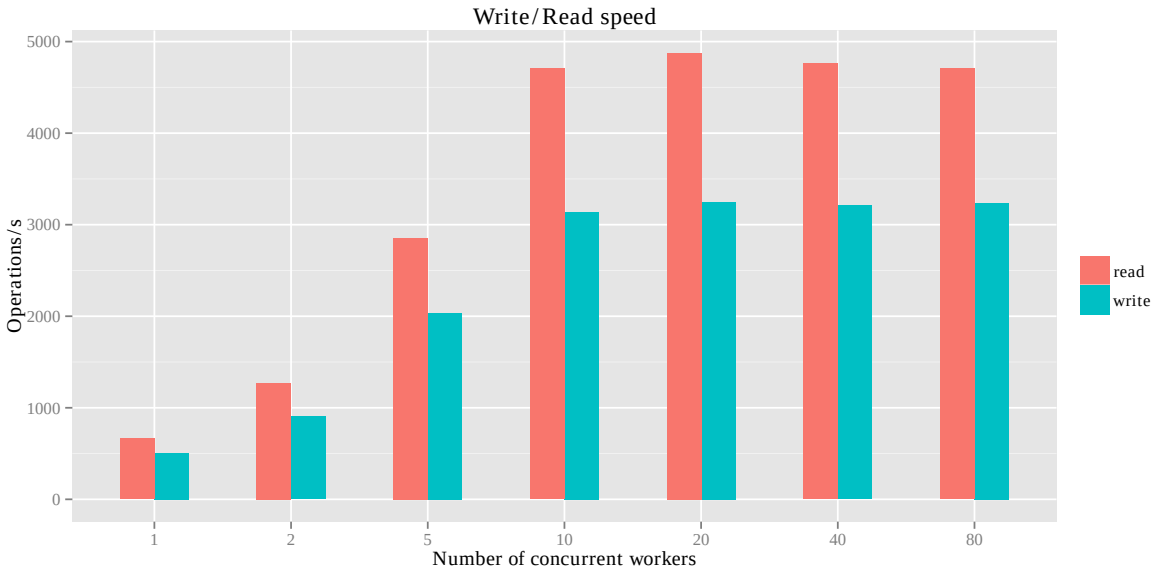

Figure 5.2: Mean throughput when running different number of concurrent workers against single instance Riak cluster.

5.3. Number of independent mongos

Second round of tests were executed only for MongoDB as it tried to answer what is the impact of using multiple mongos instances in comparison to using only one. In Riak all clients communicate with all the nodes directly and routing happen on all Riak hosts. Therefore there is no difference if using many distinct hosts or not.

The results presented in Figure 5.5 show that using multiple hosts decrease write performance linearly by about 5% for each additional mongos. The best write speed achieved for single host 4450 ops/s dropped to 4071 for 2 hosts and to 2731 for 8 hosts.

On the contrary, read speed was greatly improved by each additional mongos. Tested MongoDB database was able to deliver up to 38000 documents per second when using 8 independent mongos compared to 9090 for single worker setup. It seems that database can really take advantage of caching it's content in RAM and the bottleneck for read operation is number of mongos with their workers.

5.4. Horizontal scaling

Next series tested horizontal scalability of both databases. In each run 4 virtual machines with 30 workers each writing and then reading from MongoDB and Riak databases consisting of different number of nodes. Previous tests indicated that this number of workers and nodes seems to be optimal for achieving best results.

We expect that ideal horizontal scalability is when each host added to cluster increases performance by 100% of it's capabilities measured in single node mode. However in the real world it is almost impossible to achieve because of overhead that is added by things like synchronization and other communication occurring between database hosts. exchanged between database hosts.

The write throughput of MongoDB shown in Figure 5.6 scaled up when additional nodes were added. Second node improved scores from 4142 ops/s to 7032 which is 70,74% better. Adding 3rd and 4th node did not have as good impact but the performance divided by number of nodes showed that each node delivered 62% and 56% of

single node performance. When 13 nodes were cooperating in one MongoDB cluster the write throughput went up to 11642 ops/s which is 22% of a ideal throughput.

While write performance was greatly improved the read throughput was dropping with each shard added to database. The optimal number of ops/s was the same as best result in previous series of tests 32429. Adding second node decreased this number by 12% to 28568 ops/s. 3rd and 4th also decreased overall read performance by similar factor. During test with 13 hosts the read performance dropped to 14999 ops/s which is 4,5% decrease per additional node.

The reason for this behavior is that main id used to fetched the object didn't have any information about which shard is storing this document. In this case all shards are requested to fetch the document. Each shard added to database increases number of operations needed for retrieving each document.

To verify this assumption second series of tests was conducted using primary key as sharding key. In Figure 5.7 the read performance is still best for a single node setup 32429 reads/s but drops very little about 28000 reads/s when additional shards are added.

As presented in Figure 5.8 Riak database showed very stable, linear growth of throughput with each node added to the cluster. The 3291 ops/s achieved for single node was improved by 67% when second node was added. When 13 nodes configuration was tested the performance went up to 16312 ops/s which means that each node gave the 38% of single node throughput.

When it comes to read performance Riak databases also showed that each additional node can improve the throughput. The single node performance was 4938 ops/s and it was improved by 87% with second node. 3rd node added 44% of single node performance and 4th gave next 32%. For 13 nodes the read performance was best and it reached 25000 reads/s. Adding one more node lowered the write speed which might suggest that for this configuration 13 nodes is the optimal number of nodes.

5.5. Outcome

Tests presented above showed that both databases can be used for storying and accessing big data sets when using multiple hosts to increase their capacity. During tests we were able to run each database in configuration consisting of 12 hosts achieving up to 14 000 inserts/s, 30 000 reads/s for MongoDB and 16 000 inserts/s, 25 000 reads/s when using Riak.

We saw that adding number additional host to database can improve. In case of the Riak speed up is 495% for both reads and writes when using 14 nodes. When it comes to MongoDB the write speedup is 306% when using 12 nodes and read performance remains the same but it still much higher then Riak best score.

This leads to question if Riak and MongoDB are using multiple nodes efficiently enough to justify their usage. When we used Riak throughput per node was 35% of values achieved when running with single instance. When it comes to MongoDB this figure was even lower for write operations (26% with 12 nodes) and read performance couldn't be improved without using additional mongos which increases further complexity and resources needed to use MongoDB.

The Riak database proved to be very predictable that seemed to utilize 100% of given hardware and thanks to very uniform architecture which implies that all nodes play the same role in cluster and simple configuration which didn't rely on predefining almost any settings.

The worse speed of Riak is probably due to number of reasons. First, routing occurs on database nodes that clients are connected to. In MongoDB architecture there is separate program *mongos* for routing requests to specific database nodes. Thanks to that if we run *mongos* on the same machine with client we always communicate with host responsible for data. Also Riak is written in Erlang and in Erlang virtual machine which can add overhead compared to native implementation like MongoDB.

The biggest advantages of Riak database had no use here. This database specializes in high availability and fault tolerance which achieves by replicating data between hosts and being able to survive transparently to user many nodes failures.

The MongoDB showed overall better performance for simple read and writes but getting correct configuration to achieve satisfying performance was problematic. Also its performance during test was not quite uniform with periods of high and slow speed which are probably caused by operating system that tried to cache MongoDB files into memory or was preallocating next disk file. Also in these tests we didn't encounter problem of global database lock that lowers greatly write performance but would be also visible when doing queries and interleaving read and write operations.

5.6. Summary

In this chapter, we explained in detail the MongoDB and Riak configurations that were used to run the tests. Later we analyzed results of each single test run commenting most interesting figures and scores. In the end, we compared databases based on performance figures and studies described in previous chapters.

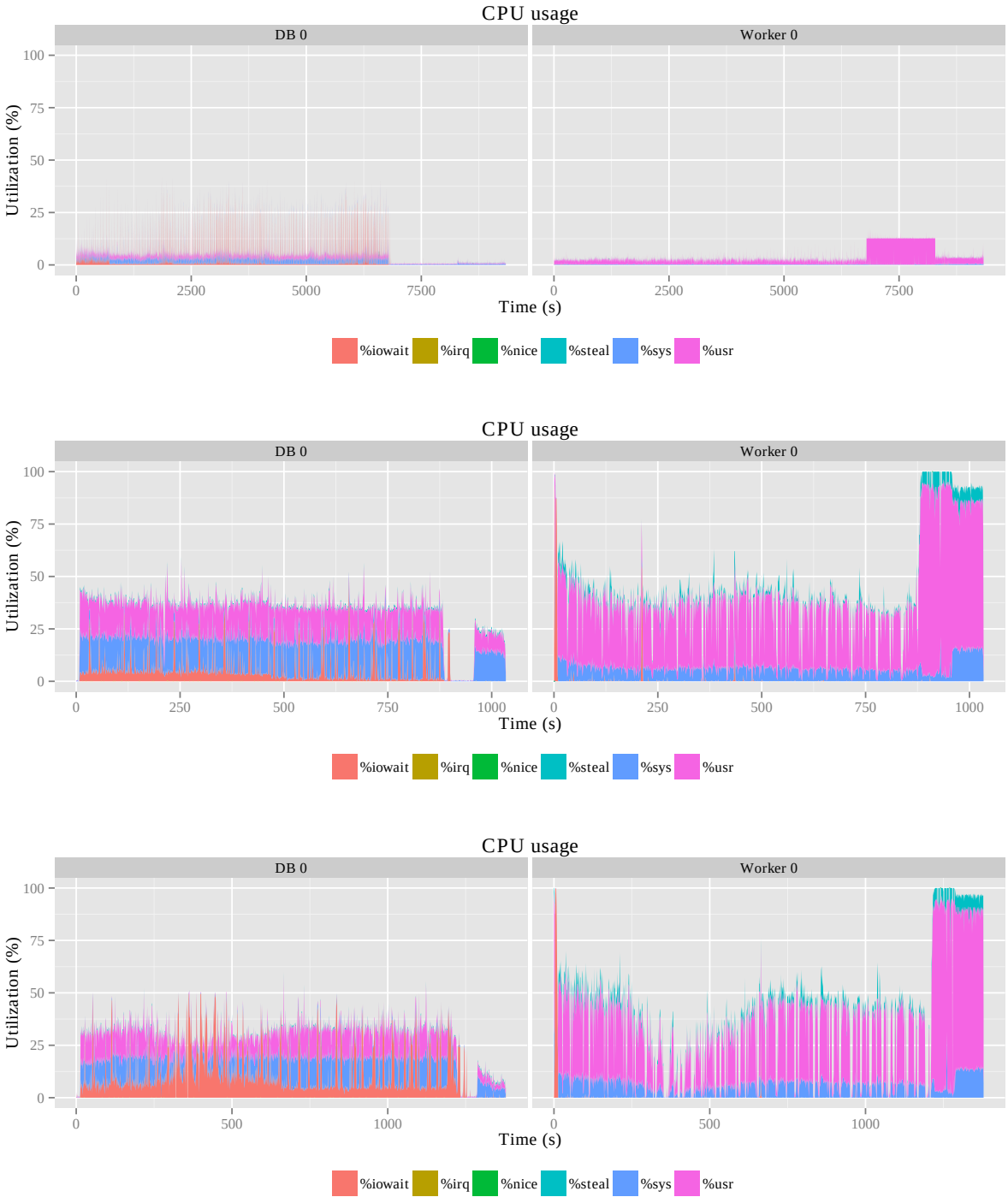

Figure 5.3: Mongo CPU utilization for 1 (above), 20 (middle) and 40 workers (below).

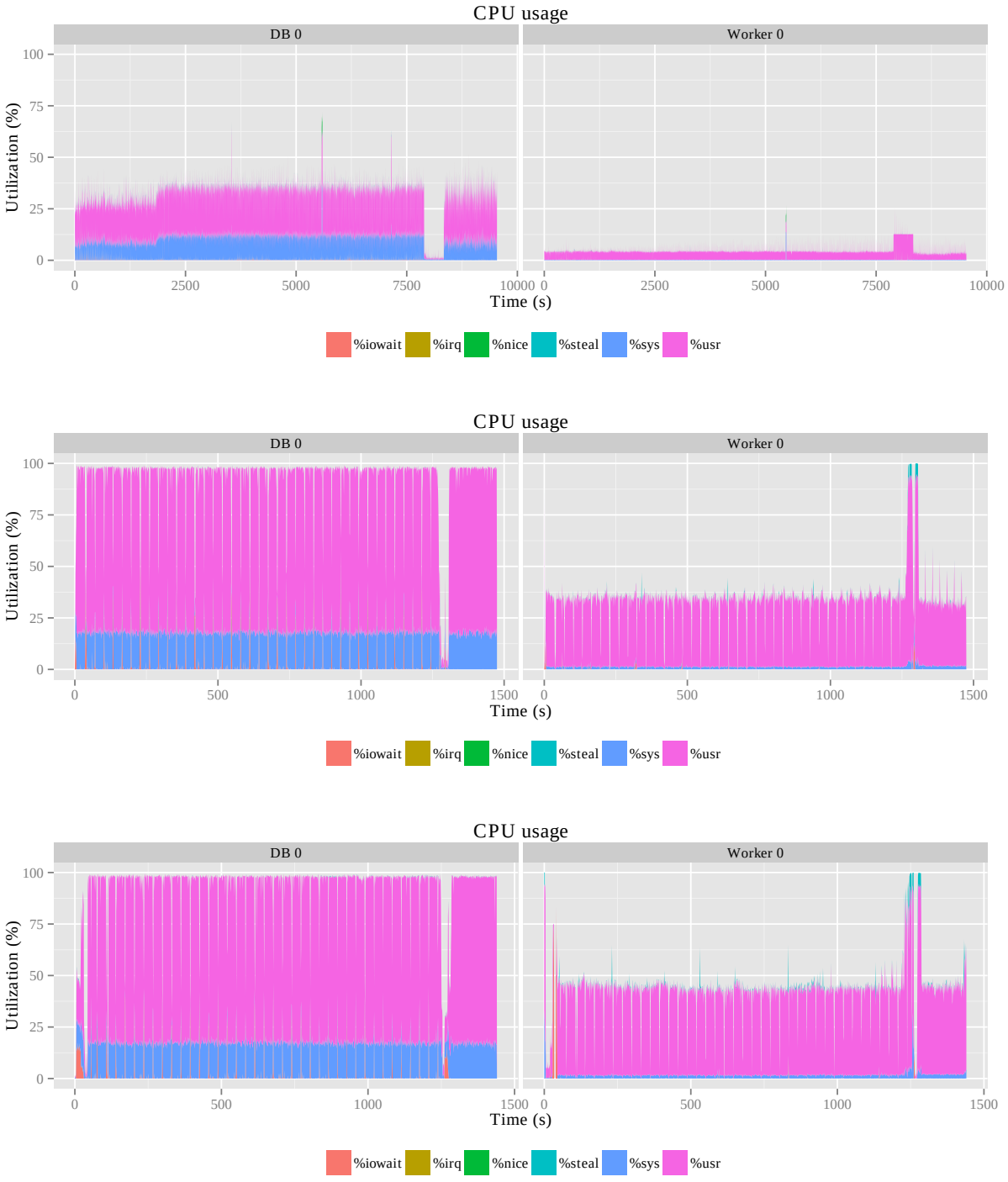

Figure 5.4: Riak CPU utilization for 1, 10 and 80 workers.

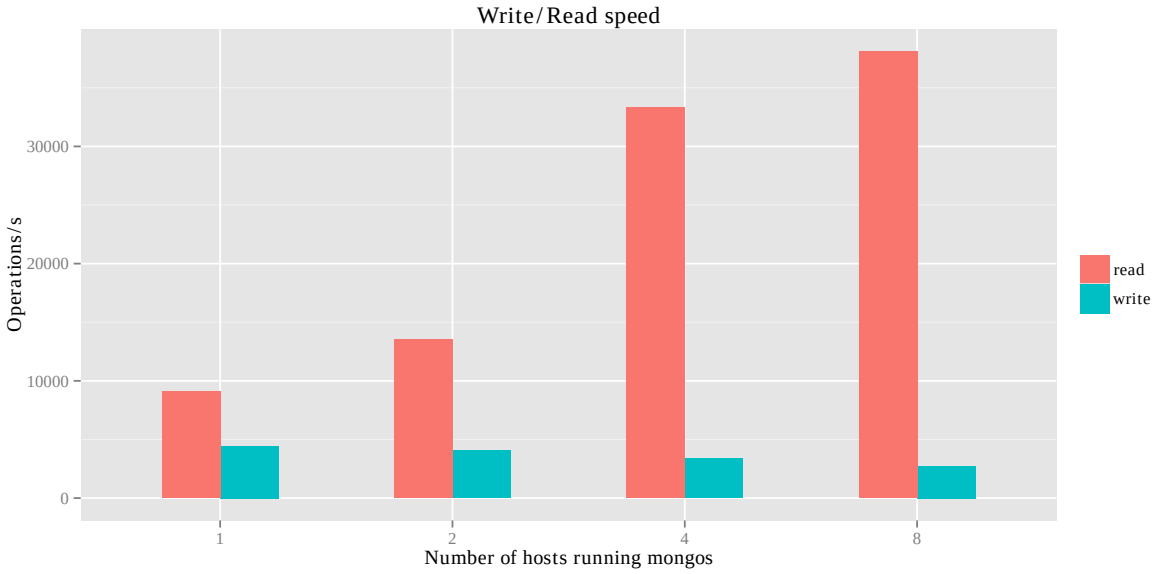

Figure 5.5: Mean throughput using different number of hosts with mongos and 20 workers.

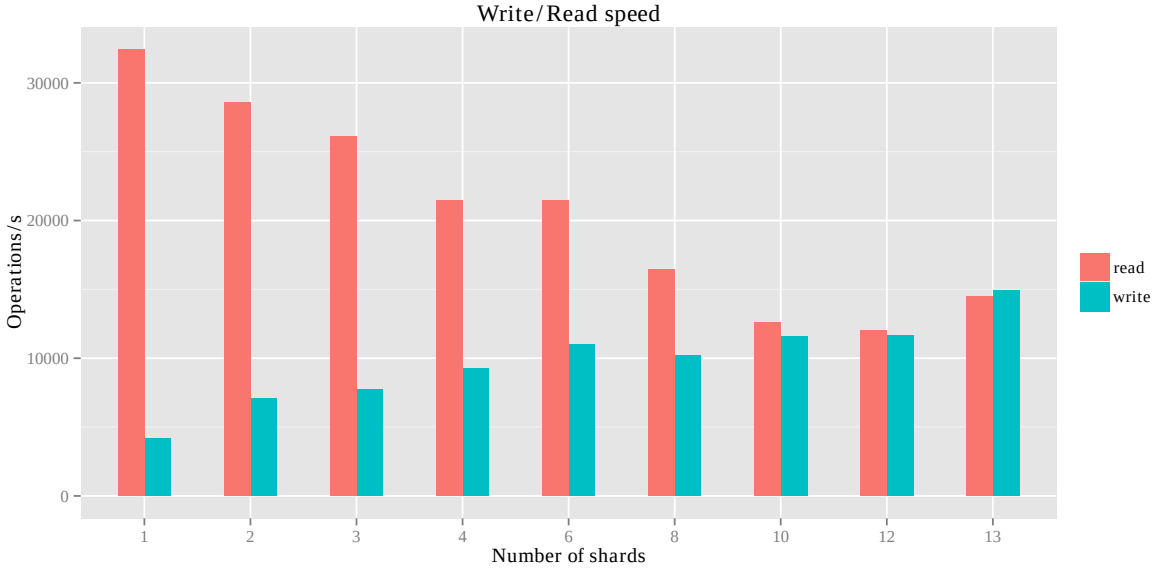

Figure 5.6: Mean throughput using different number of MongoDB nodes with sharding key different then primary key.

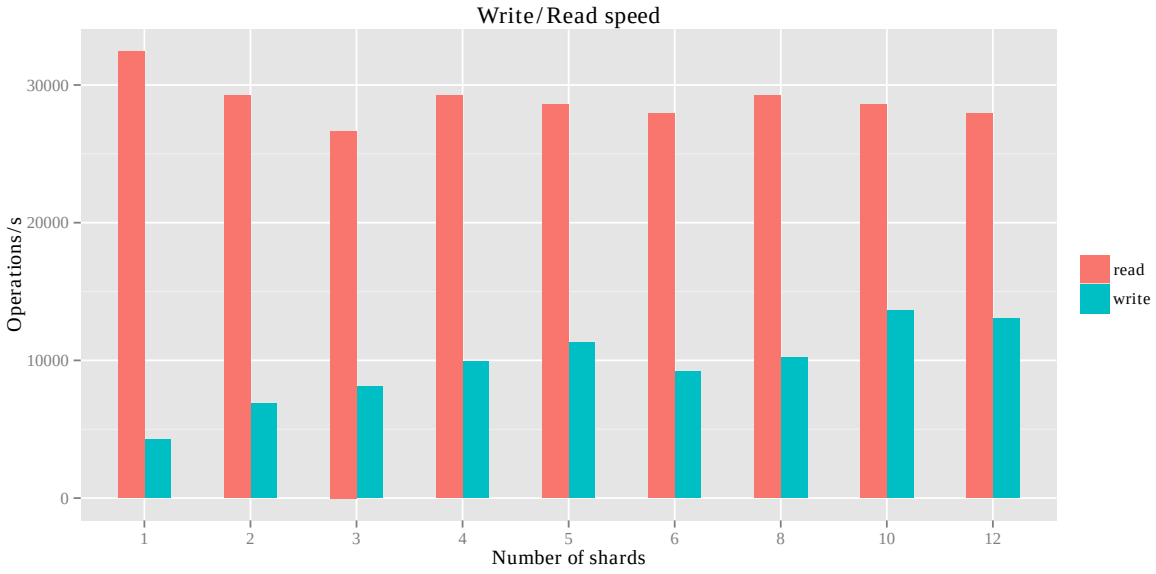

Figure 5.7: Mean throughput using different number of MongoDB nodes using the primary key also as sharding key.

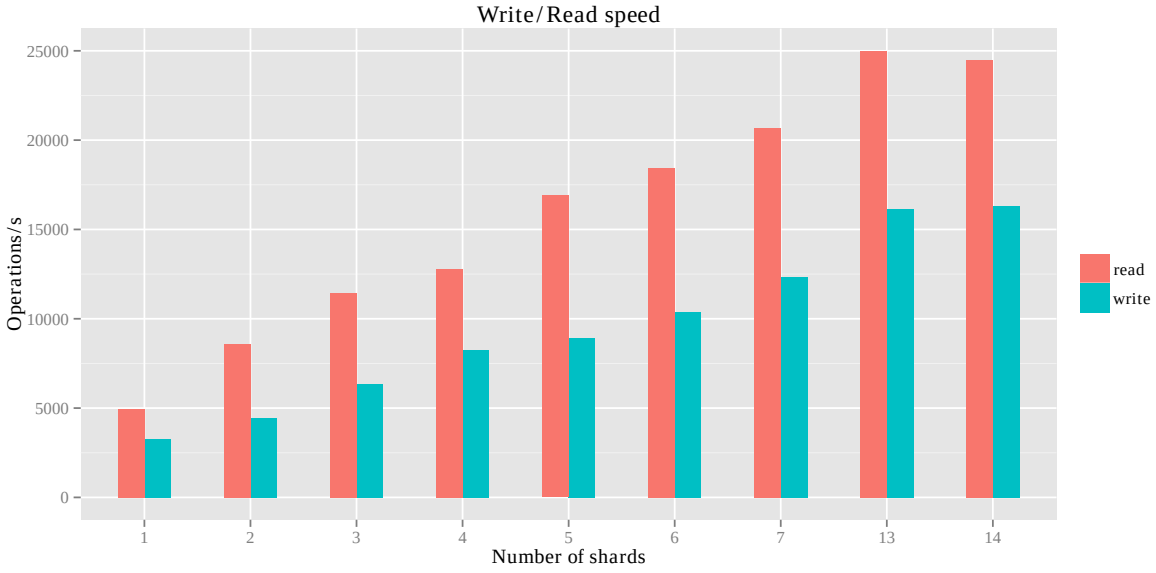

Figure 5.8: Mean throughput using different number of Riak nodes.

6. Conclusions and future work

6.1. Conclusions

Presented work analyzed many aspects of design and implementation challenges regarding storing and processing large volumes of data in scientific computations using NoSQL databases. The goal of this thesis was to gain theoretical knowledge about their capabilities and put them under test to obtain useful performance figures which could be used by researchers to implement their next experiments. To achieve it we needed to deeply analyze and compare currently available solutions and come up with adequate method of testing.

First we looked at evolution of data stores trying to capture the most important aspects and features of all types of databases. We mostly focused on SQL solutions which became the industry standard and compared them against data store from new families NewSQL and NoSQL. Thanks to that we were able to evaluate results of different benchmarks, workloads and test scenarios that were created by others. We designed our own method of testing that was based on real scientific experiment and allowed us to conduct fully automated, configurable and repeatable tests which use real scientific data to generate load on databases.

We conducted series of tests which helped us to understand and compare read and write performance of two NoSQL databases MongoDB and Riak under the same workloads and in identical environment (Amazon EC2). The tests were executed in various different configurations which allowed to measure how number of clients and number of database hosts affect the performance and scalability.

We conclude that both databases can be useful in scientific applications. They offer good write and read performance and also horizontal scalability which is unique feature compared to standard SQL solutions. The MongoDB proven to be generally faster then Riak, especially when it comes to read speed. However it is also much more complicated and problematic to setup and use. Riak, on the other hand, proven to be very predictable, easy to configure and when using large number of nodes it even showed better write performance then MongoDB.

6.2. Future work

The implemented testing framework proven to be a good solution for testing databases for scientific computations. Extending its test methods and adding support for new databases could produce very interesting results that would definitely help researches in making good decisions when designing software for computational experiments.

In addition to measuring write and read performance, testing framework could also check performance and capabilities of built-in MapReduce, queries and indexes, interleaving writes and reads.

When it comes to databases we could consider benchmarking other NoSQL databases such as: Apache Cassandra, Apache CouchDB, Apache HBase alongside NewSQL solutions.

Last, but not least, it would be interesting to conduct the same tests on different set of hardware and even using other cloud provider than AWS to compare their quality and cost effectiveness. quality and cost effectiveness.

Bibliography

[1] AI.Graphic. Aug. 2014. URL: http://commons.wikimedia.org/wiki/File:Supercomputers-history.svg#mediaviewer/Plik:Supercomputers-history.svg.

[2] Charles W. Bachman. "Data structure diagrams". In: *SIGMIS Database* 1.2 (July 1969), pp. 4–10. ISSN: 0095-0033. DOI: 10.1145/1017466.1017467. URL: http://doi.acm.org/10.1145/1017466.1017467.

[3] Inc Basho Technologies. *Why Riak*. May 2014. URL: http://docs.basho.com/riak/1.4.2/theory/why-riak/.

[4] Eric A Brewer. "Lessons from giant-scale services". In: *Internet Computing, IEEE* 5.4 (2001), pp. 46–55.

[5] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. 2000, p. 7.

[6] M. Bubak et al. "Evaluation of Cloud Providers for VPH Applications". In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. 2013, pp. 200–201. DOI: 10.1109/CCGrid.2013.54.

[7] Sergey Bushik. *A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak*. Oct. 2012. URL: http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html.

[8] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919. URL: http://doi.acm.org/10.1145/1978915.1978919.

[9] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[10] Inc. Cisco Systems. *Visual Networking Index (VNI)*. Sept. 2013. URL: http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html.

[11] E. F. Codd. "A relational model of data for large shared data banks". In: *Commun. ACM* 26.1 (Jan. 1983), pp. 64–69. ISSN: 0001-0782. DOI: 10.1145/357980.358007. URL: http://doi.acm.org/10.1145/357980.358007.

[12] "Collins English Dictionary – Complete & Unabridged 10th Edition". In: (July 4, 2013). URL: http://dictionary.reference.com/browse/emotional%20intelligence.

[13] Brian F Cooper. *brianfrankcooper/YCSB*. May 2014. URL: https://github.com/brianfrankcooper/YCSB/.

[14] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[15] Transaction Processing Performance Council. “TPC Benchmark C Standard Specification, Revision 5.11 (2010)”. In: *URL: http://www.tpc.org/tpcc* ().

[16] Transaction Processing Performance Council. “TPC Benchmark DS Standard Specification, Version 1.1.0 (2012)”. In: *URL: http://www.tpc.org/tpcc* ().

[17] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *SOSP*. Vol. 7. 2007, pp. 205–220.

[18] Elif Dede et al. “Performance evaluation of a mongodb and hadoop platform for scientific data analysis”. In: *Proceedings of the 4th ACM workshop on Scientific cloud computing*. ACM. 2013, pp. 13–20.

[19] Andrew Eisenberg and Jim Melton. “SQL: 1999, formerly known as SQL3”. In: *SIGMOD Rec.* 28.1 (Mar. 1999), pp. 131–138. ISSN: 0163-5808. DOI: 10.1145/309844.310075. URL: http://doi.acm.org/10.1145/309844.310075.

[20] Python Software Foundation. *9.6. random — Generate pseudo-random numbers*. May 2014. URL: https://docs.python.org/2/library/random.html.

[21] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59.

[22] Sebastien Godard. *SYSSTAT changelog*. May 2014. URL: http://sebastien.godard.pagesperso-orange.fr/old_changelog.html.

[23] The Open Group. *Service Oriented Architecture : What Is SOA?* Sept. 2013. URL: http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition.

[24] Nils Homer, Barry Merriman, and Stanley F Nelson. “BFAST: an alignment tool for large scale genome resequencing”. In: *PLoS One* 4.11 (2009), e7767.

[25] Basho Inc. *Why Riak*. Jan. 2013. URL: http://docs.basho.com/riak/latest/theory/why-riak/.

[26] Google Inc. *LevelDB*. May 2014. URL: https://code.google.com/p/leveldb/.

[27] Ioannis Konstantinou et al. “On the elasticity of nosql databases over cloud management platforms”. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. 2011, pp. 2385–2388.

[28] John C. McCallum. *Disk Drive Prices (1955-2014)*. Aug. 2014. URL: http://www.jcmit.com/diskprice.htm.

[29] Inc MongoDB. *Agile and Scalable*. Sept. 2013. URL: http://mongodb.org.

[30] Inc MongoDB. *MongoDB Replication Guide*. Sept. 2013. URL: http://docs.mongodb.org/master/MongoDB-replication-guide.pdf.

[31] Inc MongoDB. *MongoDB Sharding Guide*. Sept. 2013. URL: http://docs.mongodb.org/master/MongoDB-sharding-guide.pdf.

[32] Inc MongoDB. *MongoDB – The Leading NoSQL Database*. Sept. 2013. URL: http://www.mongodb.com/leading-nosql-database.

[33] Tim O’reilly. *What is web 2.0*. 2005.

[34] Tim O’reilly. “What is Web 2.0: Design patterns and business models for the next generation of software”. In: *Communications & strategies* 1 (2007), p. 17.

[35] William R Pearson and David J Lipman. “Improved tools for biological sequence comparison”. In: *Proceedings of the National Academy of Sciences* 85.8 (1988), pp. 2444–2448.

[36] Pramod J Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.

[37] Susan M Sanchez et al. "Designs for large-scale simulation experiments, with application to defense and homeland security". In: *The Design and Analysis of Computer Experiments* 3 (2012), pp. 413–441.

[38] Eric E Schadt et al. "Computational solutions to large-scale data management and analysis". In: *Nature Reviews Genetics* 11.9 (2010), pp. 647–657.

[39] Emin Gün Sirer. *Broken by Design: MongoDB Fault Tolerance*. Jan. 2013. URL: http://hackingdistributed.com/2013/01/29/mongo-ft/.

[40] Michalel Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps*. http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps. Blog. 2011.

[41] Christof Strauch. "NoSQL databases". In: *URL: http://www.christof-strauch.de/nosqldbs.pdf* (2011).

[42] Top500.org. *Top500*. June 2014. URL: http://www.top500.org/.

[43] Top500.org. *Top500 List - November 2013*. June 2014. URL: http://www.top500.org/system/177388.

[44] Prasanna Venkatesh. *NewSQL — The New Way to Handle Big Data*. http://www.linuxforu.com/2012/01/newsql-handle-big-data/. Blog. 2012.

List of Figures

List of Tables

A. Publications

The tests conducted in scope of this thesis were presented during Cracow Grid Workshop 2013 and in a paper which describes PRECIP framework. The author of this thesis is also a co-author of these publications.

List of publications

- Azarnoosh, S., Rynge, M., Juve, G., Deelman, E., Niec, M., Malawski, M., Silva, R. F. D. (2013, December). *Introducing PRECIP: an api for managing repeatable experiments in the cloud*. In Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on (Vol. 2, pp. 19-26). IEEE.
- M. Niec, T. Gubala, K. Sarapata, M. Malawski *Evaluation of NoSQL Databases for Bioinformatics Applications*, Cracow Grid Workshop 2013.