RDBMS vs NoSQL: Performance and Scaling Comparison

Christoforos Hadjigeorgiou

August 23, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

The massive amounts of data collected today by software in fields varying from academia to business and many other fields, is increasingly becoming a huge problem due to storage technologies not advancing fast enough to provide the performance scalability needed. This is even more true for data which are highly organised and require analysis while being stored in databases and being accessed by various applications simultaneously. Databases also have the added advantage of providing failover mechanism in case of disruptions or one node failing.

As database vendors struggle to gain more market share new technologies emerge attempting to overcome the disadvantages of previous designs while providing more features. Two popular database types, the Relational Database Management Systems and NoSQL databases are examined. The aim of this project was to examine and compare two databases from these two database models and answer the question of whether one performs and scales better than the other.

From the comparison of the results it was found that MongoDB can perform much better for complicated queries at the cost of data duplication which in turn results to a larger database. Also the database size did not appear to be a deciding factor as performance was not crippled significantly for larger database sizes. Writing benchmarks were also run which showed that MySQL performs best at deletion whereas MongoDB excels at inserting documents. The last comparison showed that using sharding to split up the database in MongoDB did not provide a performance advantage which may be related to the routing done by the MongoDB system.

Contents

List of Figures

Acknowledgements

I would like to sincerely thank Dr Adam Carter. Without his advice, guidance and encouragement this dissertation would not have been possible.

I am also very grateful to Mr Gareth Francis for helping in setting up the cluster and troubleshooting.

I would like to thank my family as without their support this Master's degree would not have been possible for me.

Chapter 1

Introduction

Efficient storage and retrieval of data has always been an issue due to the growing needs in industry, business and academia. Larger amounts of transactions and experimentation result in massive amounts of data which require organised storage solutions. Databases were created in order to satisfy this need of storing and retrieving data in an organised manner. Since their inception in the 1960's different types have emerged, each using its own representation of data and technology for handling transactions. They began with navigational databasess which were based on linked-lists, moved on to relational databases, afterwards object-oriented and in the late 2000s NoSQL emerged and has become a popular trend [1].

Two of the most widely used database types are relational databases and NoSQL databases. Although NoSQL databases are relatively new compared to other types they have become popular due to their ability to handle very fast unrelated and unstructured data, mainly because they do not require a fixed schema and they use metadata heavily in order to achieve fast performance. Relational databases came to the forefront in the mid 1970's and were proven successful by the end of the decade. The essential concepts involved in relational databases were laid out by Edgar Codd in order to overcome the disadvantages of the previous linked lists implementations in databases.

Although the two types differ in many aspects depending on the implementation they could be used for similar applications although it is not recommended as one is not meant as an alternative to the other [2]. One of the main reasons for this recommendation is the problem of NoSQL databases being less reliable compared to relational databases due to less data-integrity and reliability. Through this dissertation a comparison of the two types is performed in terms of performance and scalability. Scalability in databases is their ability to handle an increasing amount of transactions and stored data at the same amount of time. The NoSQL database MongoDB and the relational database MySQL Cluster, which will be the databases tested, support separating the data stored to different nodes and have support for a number of programming languages. Other databases were considered as well but due to previous experience with MySQL and the simple programming layout of MongoDB as well as its document structure these two were preferred. The databases considered were PostgreSQL, CouchDB and Apache Cassandra. The reasons why these databases were not chosen include tuning of the configuration in order to achieve good performance for PostgreSQL and the limitation of interface to HTTP/REST API and Java for CouchDB and Apache Cassandra respectively. [3] [4] [5]

Comparing these two databases is important as it allows to draw conclusions regarding their ability to process data and how they handle large amounts of transactions and data. It also provides insight to how well suited they are to today's issue of massive amounts of data collected otherwise known as Big Data. Databases play an important role in applications and the wrong choice at the beginning may have disastrous effects as it is difficult to migrate to another database system, more so a completely different type. The performance and scalability of the databases are the most important factors besides reliability when weighing the various options and comparing them for different databases can be difficult due to different designs, configurations and data access methods.

This dissertation takes the two database systems and compares them by trying to find a middle ground where their implementations are as close as possible and the benchmarks performed do not favour one database system over the other. Amongst the most important issues are finding a data set and representing it in both databases effectively. In addition the correct choice of benchmarks is vital as stressing a database can be a tedious task.

Chapter 2 is an analysis of the technologies used followed by an analysis of relational and NoSQL databases. Also the terminology and concepts used in developing databases is discussed. An overview of the machine used in the experiments, EDIM1, is provided at the end of the chapter together with a description of the operating system and the software used to run the benchmark in parallel.

Chapter 3 details the design of the software used for the experiments. This includes an examination of the database deployed for each database system. The methodology used for measuring the performance and scalability of the databases is described together with what options were available in doing so.

Chapter 4 analyses the results of the experiments and builds a hypothesis regarding the performance of the databases. This hypothesis is then examined through further experimentation.

Chapter 2

Background

2.1 Databases

Databases are defined as organised collections of data [6, p. 6]. Although when using the term database we refer to the entire database system, the term actually refers only to the collection and the data. The system which handles the data, transactions, problems or any other aspect of the database is the Database Management System (DBMS). What follows is a description of the two database types which will be compared in this dissertation.

Early designs and implementations were based on the use of linked lists to create relations between data and to find specific data. These models were not standardised and required extensive training in order to make efficient use of them. These models and other important types are explained briefly as well.

2.2 Relational Databases

Relational databases use the notion of databases separated into tables where each column represents a field and each row represents a record. Tables can be related or linked with each other with the use of foreign keys or common columns. On an abstract level tables represent entities, such as users, customers or suppliers. This abstraction is helpful when designing the database schema as real world objects need to be mapped to the database in addition with the relations between them. The design of a database schema can be visualised using diagrams such as the one in Figure 2.1.

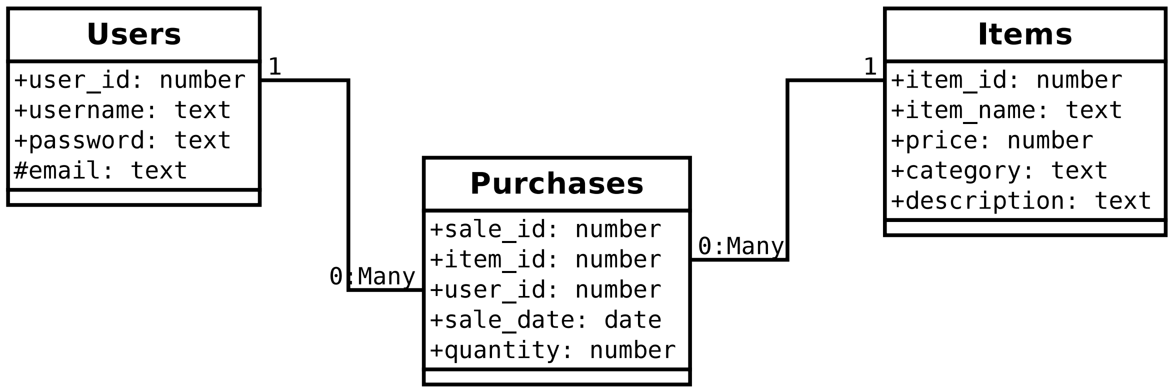

Figure 2.1: Database schema example

An important design aspect of relational databases is the normalisation of the schema. This involves 3 steps which are described below:

1. **First Normal Form (1NF)**: Eliminate groups of repeating data by creating a new table for each group of related data which is identified by a primary key.
2. **Second Normal Form (2NF)**: If a set of values are the same for multiple records move them to a new table and link the two tables with a foreign key.
3. **Third Normal Form (3NF)**: Fields which do not depend on the primary key of a table must be removed and if necessary be put into another table.

It is also necessary for a database schema to satisfy the 2NF to first satisfy 1NF and the same applies for 3NF correspondingly. While there are other forms a schema is considered normalised if it satisfies the above 3 conditions.

2.3 ACID properties

An important aspect of relational databases which guarantees the reliability of transactions is their adherence to the ACID properties: **A**tomicity, **C**onsistency, **I**solation, **D**urability [6, p. 405]. Each property is explained below in the context of databases.

- **Atomicity**: Either all parts of a transaction must be completed or none.
- **Consistency**: The integrity of the database is preserved by all transactions. The database is not left in an invalid state after a transaction.
- **Isolation**: A transaction must be run isolated in order to guarantee that any inconsistency in the data involved does not affect other transactions.
- **Durability**: The changes made by a completed transaction must be preserved or in other words be durable.

2.4 NoSQL Databases

NoSQL databases started gaining popularity in the 2000's when companies began investing and researching more into distributed databases [7]. For this reason the category of NoSQL databases grew and included many subtypes each better suited to specific datasets than others. The most common NoSQL database categories are the following:

- **Document stores**: The notion of "documents" is the central concept here with documents being the equivalent of records in relational databases and collections being similar to tables. [8]
- **Key-value stores**: Data is stored as values with a key assigned to each value similarly to hash-tables. Also depending on the database a key can have a collection of values. [8]
- **Graph databases**: Like graph theory the notion of nodes and edges is the primary concept in graph databases. Nodes correspond to entities such as a user or a music record and edges represent the relations between the nodes. An important aspect which differentiates graph from relational databases is the use of index-free adjacency, this means each element contains a pointer to its adjacent element and does not require indexing of every element. [9]

An important aspect of NoSQL databases is that they have no predefined schema, records can have different fields as necessary, this may be referred to as a dynamic schema. Many NoSQL databases also support replication which is the option of having replicas of a server, this provides reliability as in the case one goes offline the replica would become the primary server. All servers execute the same transactions and synchronise their data in order to eliminate any errors. Also an important difference between relational databases and NoSQL databases is they do not fully guarantee ACID properties. Their lack of ACID guarantees is attributed to their deployment architeture which typically involves having multiple nodes in order to achieve horizontal scalability and recovery in case of failover. This deployment, which is also referred to as replication, creates issues with synchronisation which can result in a secondary node becoming primary but not have an up-to-date copy of the data. NoSQL databases, apart from using an Application Programming Interface(API) or query language to access and modify data, may also use the MapReduce method which is used for performing a specific function on an entire dataset and retrieving only the result. [2]

2.5 Other databases

Beyond relational and NoSQL databases other types have emerged in the past as well.These types have not become as successful as the types examined in this project but have been an important part of the evolution of databases nonetheless.

2.5.1 Navigational Databases

Navigational Databases were the first generation of databases developed which worked by using pointers from one record to another. The major downside of navigational databases was the fact that the user would have to know and understand the underlying physical structure of the database in order to query for records. In order for an extra field to be added to a database the entire underlying storage scheme had to be rebuilt. In addition the lack of standardisation among vendors was a disadvantage as it became difficult to choose a suitable implementation. [1]

2.5.2 Object-oriented databases

An important part of the database evolution are object-oriented databases which emerged in the 1980s and their main usage was in the object-oriented field such as in conjunction with object-oriented programming languages. In this model information was stored as objects which could accommodate for a large number of data types and could also offer advanced features such as inheritance and polymorphism which were characteristic of object-oriented programming languages as well. The disadvantages which prevented object-oriented databases from gaining more popularity was the need to rebuild an entire database in order to migrate from another database management system and also the issue that most of these databases were bound to a specific programming language. [1]

Object Relational Database Management System & Object Relational Mapping

In order to overcome the disadvantages of object-oriented database and use the advantages of relational databases the Object Relational Database Management System emerged. This system attempted to bring together the features of RDBMSs and object-oriented modelling techniques which were used in popular programming languages. This however was not successful with a more popular approach being the Object Relational Model whereby a relational model database was used with an object relational mapping software which allowed developers to integrate their own types and their methods into the DBMS. [1]

2.6 Database Replication

Database Replication is the practice of deploying multiple servers which are clones of each other. This practice is used in NoSQL databases often in order to provide higher reliability and performance. MongoDB recommends deploying a replica set, which is a set of replica servers, in all production deployments. [10]

In MongoDB replication is deployed using a primary-secondary server configuration whereby one server is the primary and all others are secondary. The primary server, or primary replica, handles all write operations and logs them in a special collection where the secondaries read and apply them. Secondary replica servers can also read the operations from another secondary thus limiting the amount of load on the primary server. This processes are shown in Figure 2.2. By having a copy of the data replica servers not only provide more reliability but can also be used for read operations from client applications, this can have a massive performance effect but also carries with it the chance that the data provided will not be the most up-to-date. The reason that data is not always up to date is due to the asynchronous replication model which MongoDB uses. This asynchronous replications provides better reliability as secondary members can continue to function when another member of the replica set is unavailable but prohibits the guarantee of the ACID properties as a secondary replica which becomes primary may not be up to date thus violating the Consistency and Durability properties.

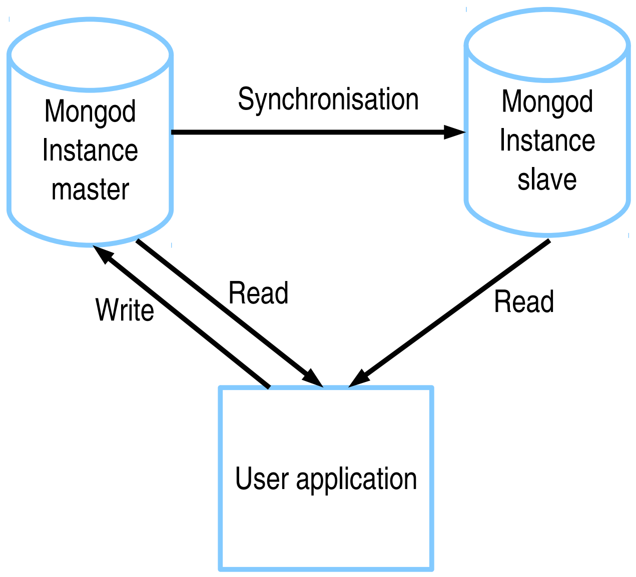

Figure 2.2: MongoDB Replica deployment and usage

In the case when the primary replica fails for any reason and goes offline voting takes place to promote a secondary replica to primary, this is when the concept of an arbiter server becomes relevant. An arbiter server does not handle any load or store data but only exists to provide a vote in these elections, this is necessary in cases where there is an even number of secondary replicas and the voting could result in a draw.

MySQL Cluster also supports replication but is currently limited to a main replica server and a slave replica server at maximum. This limitation can be alleviated with the use of node clusters which work as replicas of other node clusters. In addition, the default configuration does not allow direct connections to the replica in order to distribute the load and also the use of replication in clustering instead of thte simple Master-Slave model requires the use of the NDB engine to cluster successfuly. The MySQL implementation of the Master-Slave model is similar to MongoDB as it is used to send write operations to the

Master node and read queries can be routed accordingly either to the Master or the Slave. This enables to distribute some of the load from the Master node and improve performance.

2.7 Database Sharding

Sharding is the term used to describe practice of using multiple servers of the same database and configuring them in order for the data stored in the database to be split or separated to different machines. This allows increased performance as each server handles different sets of data thus if a single database becomes too large its performance may diminish due to the increased time a query takes. Despite the obvious advantage of added performance database, replication is recommended over sharding as it provides both performance and reliability. Specifically MongoDB developers suggest to deploy databases without sharding and only shard when the data set increases in size. In addition, MongoDB shards can be accessed directly but the transactions which can be performed are limited to the collections which are sharded on the specific shard.

MySQL Cluster handles sharding automatically but is limited to using the NDB storage engine. The NDB storage engine is limited mainly due to its lack of foreign keys support and a storage limit of 3TB compared to 64TB of the InnoDB storage engine [11] making it unsuitable for the benchmarking harness which uses foreign keys.

2.8 EPCC Data Intensive Machine 1 (EDIM1)

EDIM1 is an experimental platform designed for data-intensive research at EPCC. It consists of 120 back-end nodes, a login node and data staging server. Each back-end node consists of an Intel Atom CPU, a programmable GPU, 4GB of memory, a 6TB hard disk and 256GB SSD. Nodes are also interconnected with a gigabit ethernet network and use ROCKS as the operating system which is a GNU/Linux distribution designed for running clusters [12]. EDIM1 was designed as an Amdahl-balanced cluster in order to eliminate the I/O bottleneck which exists in most systems due to the inability of the I/O system to provide data as fast as the CPU can process it. This is an important feature of the machine which affects the project as the databases will be able to make use of all the processing power instead of delaying while the I/O is sending data. The significance of this effect is that observations can be made on how the database works according to the number of connections it handles. For example, having two simultaneous connections requesting the same data could cause a slow down to the transations.

2.8.1 ROCKS Linux

The operating system running on EDIM1 is the ROCKS Linux is a GNU/Linux distribution which is aimed at setting up clusters easily. It is a 64 bit operating system which allows to build, manage and monitor clusters through a frontend which controls the rest of the nodes. In case of a node failing the frontend automatically reinstalls the base system on that node any preselected packages which in ROCKS are called rolls. [13]

2.8.2 GNU Parallel

As the default installation of ROCKS Linux on EDIM1 lacked a parallel programming library and using one was beyond the aim of the project an alternative method was used. The software package of the open source GNU Parallel program was installed which allows to execute commands and scripts in parallel by specifying the number of threads. This was used in conjunction with the **rocks** command which is part of the operating system to execute the benchmarking package multiple times on multiple nodes. [14]

Chapter 3

Benchmarks

3.1 Benchmarking harness

The benchmarking harness was built using the C programming language and used the latest stable drivers available for each database. Database drivers are the libraries which provide a method for connecting with the database and performing transactions.

The harness allows to change different options such as the number of benchmarks and queries to be performed, the use of random strings for populating the database and whether to populate the database or use the data already there. Specifically the procedure followed by the harness is as follows:

1. Depending on the database selected create a connection using either the default values or user-supplied parameters.
2. If database populating was requested the current tables are dropped and repopulated either with random strings or a string differentiated by adding an index integer.
3. Otherwise if no populating was requested the database is queried and the data is stored to be used during the benchmarking.
4. Each benchmark is performed which include a simple query, a complicated join operation and a similar complicated operation which uses nested queries instead.
5. The time spent for each benchmark is recorded and at the end of all benchmarks an average is calculated and all the results are output.

Since all the actions performed are crucial to measuring the performance correctly the benchmarking harness will halt if at any point an operation fails. This is also important for finding errors related to the configuration of the database.

3.2 Measurements

Three C methods were considered for taking time measurements . The first was to use the **clock()** function provided by C but was quickly ruled out as it only provides the time a process spent using the CPU. Although this could be used in conjunction with other functions to estimate the time which was spent for a benchmark other better and more direct methods were investigated. The second method was the **gettimeofday()** function which provided timings with millisecond accuracy. Due to a misconception on how the MySQL C driver works, specifically the MySQL database does not return the result of a query until it is requested to do so, it resulted in getting a different time record compared to what was expected. This had the consequence of investigating the use of the **MPI_Time** function but it was quickly ruled out as MPI and its libraries were not installed, a process which could take a large amount of time relative to the time available for the entire dissertation. Instead after investigating the reason for the unexpected results reported by **gettimeofday()** and the explanation was found subsequent timings were as expected.

Each database also offers its own method for measuring time. MySQL can record queries which take longer than a predefined amount of time to the **Slow Query Log** or a system database. By default the time threshold for logging a slow query is 10 seconds and can be configured for microsecond accuracy. Despite the convenience of the automatic logging it holds some disadvantages to using the timing functions of the harness. First, the log takes times for individual queries complicating the analysis of the overall result. This is due to the large size of the log file and the more complicated logging format used as each query is also recorded in the log together with other information such as number of rows returned. MySQL offers a tool for analysing the log file but its purpose is to summarise the contents, either by sorting the queries or select queries which match a specific pattern, all of which do not assist in measuring the overall performance of the database. Secondly, the log itself is a method for measuring time from the server itself and not a client application. This is not an ideal measurement as time will also be spent for applications to retrieve the result. [15]

MongoDB records query times in a similar manner but the results are stored in a system specific database and collection, **system.profile**. This method is inadequate for measuring performance due mainly to the lack of overall measuring of performance but also as it has a minor effect on performance [2].

3.3 Measurement metrics

To measure the performance of the databases a common metric is required. The most important factor for an application is the time required for completing a task, and in the case of databases the time required to complete a transaction. The benchmarking harness measures the time required to complete a set number of transactions as each transaction on its own is negligible. Also the average of all the benchmarks during a single run of the benchmarking harness is calculated in order to have a better overall idea of how the database behaves as a single benchmark may deviate due to outside factors such as an operating system process temporarily using the CPU or performing I/O. Furthermore, to measure the performance of the databases and their scalability the metric of total queries per second is used as it allows to extrapolate how well the databases cope with different loads and different numbers of connections. To calculate the queries per second the formula in Figure 3.1 is used

$$Queries\ per\ second = \frac{Total\ number\ of\ queries\ *\ Total\ number\ of\ threads}{Average\ query\ time} \tag{3.1}$$

Another metric used is the total queries per second per in order to quantify how the databases respond to each thread.

$$Queries\ per\ second\ per\ thread = \frac{Total\ number\ of\ queries}{Average\ query\ time} \tag{3.2}$$

3.4 Database schema

The database schema used was designed from the ground up and was modelled around a music application which would use different algorithms to suggest songs to users according to their tastes and other users who have similar tastes. The database schema shown in Figure was designed for the MySQL database implementation. It has been normalised and eliminates any data duplication between tables. The database schema is shown in Figure 3.1.

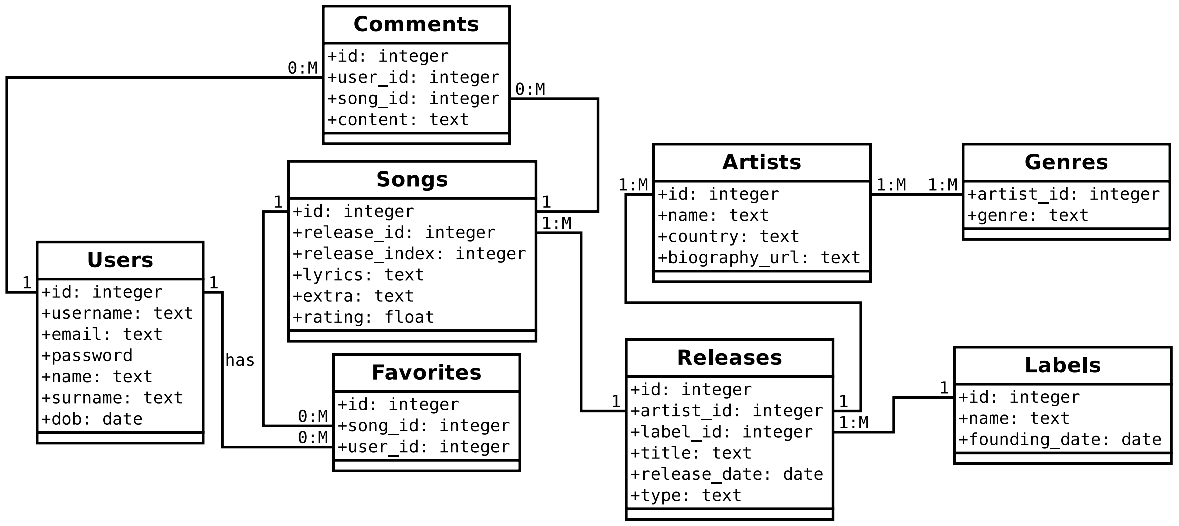

Figure 3.1: Relational database schema

Although the same schema can be informally used in MongoDB some compromises have been made to make up for the fact that MongoDB does not support complicated operations such as JOINS which are typical for RDBMSs. Instead the MongoDB database schema combines a number of tables to take advantage of sub-documents. The process of combining tables is denormalisation although in this case it is only partial, complete denormalisation would mean the combination of all the tables into a single table. In addition, the design of the MongoDB schema allows to more directly compare with MySQL for similar tasks such as JOINs or nested SELECTs. In Figure 3.2 the schema for MongoDB is shown including the subdocuments which were adapted from the relational schema. The Release collection now includes the attributes from the **Labels** table as each record only has one label and also includes a **songs** subdocument which replaces the **Songs** table. The justification for using a subdocument for the songs in each release is that typically the number of songs is relatively small, 10-20, and also they do not change after a release is made. In addition, MongoDB allows faster transactions when dealing with subdocuments compared to finding data in other collections due to the binary format it uses for internal representation.

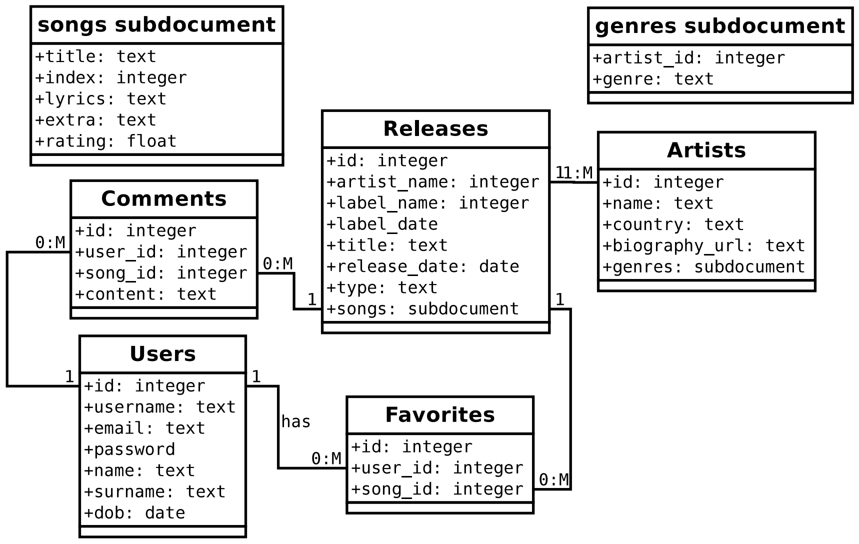

Figure 3.2: MongoDB Schema

MongoDB stores documents using its own binary format called BSON. This format is a binary version of the widely used JSON (**J**ava**S**cript **O**bject **N**otation) format and stands for Binary JSON. Although BSON is used internally by MongoDB the manipulation of documents in the MongoDB shell interface and client software is done using JSON due to its readability and open standard. A document from the Releases collection is shown in Listing 3.1.

Listing 3.1: Document from Releases collection

```
{
  "_id": ObjectId(""),
  "artist_name": "Deftones",
  "label_name" : "Maverick",
  "label_date" : 01041992,
  "title"            : "Change(In the House of Flies)",
  "release_date" : 27052000,
  "type"             : "single",
  "songs": [
      {"title": "Change(In the House of Flies)",
       "index": 1, "lyrics" : "", "extra": "", "rating" : 5
          },
      {"title": "Crenshaw",
       "index": 2, "lyrics" : "", "extra": "", "rating" : 3
          },
      {"title": "No Ordinary Love",
       "index": 3, "lyrics" : "", "extra": "", "rating" : 4
          },
    ]
}
```

3.5 Database Queries

The benchmarks involved using three queries to stress the databases. The first query which was a simple query containing only a SELECT statement, a second more complex one with an INNER JOIN and a third with a SELECT statement containing an INNER JOIN to a second table and inside the WHERE clause a nested SELECT containing an INNER JOIN as well. In addition a second version of the complex query which contained a single INNER JOIN was created using nested SELECTs as to benchmark the performance of MySQL when a query similar the actions performed in MongoDB is used. These queries are described in SQL terms and are shown in Listing 3.2.

Listing 3.2: Database queries

```
// simple

SELECT * FROM Users WHERE username = 'username'

// complex 1.1

SELECT  'Favourites.song_id' AS fSID,
        'Favourites.user_id' AS fUID
FROM Favourites AS b INNER JOIN Favourites AS a
ON b.user_id = a.user_id
WHERE a.song_id = 123456 AND a.user_id != 987654

// complex 1.2
SELECT  'Favourites.song_id' AS fSID,
        'Favourites.user_id' AS fUID
      FROM Favourites
      WHERE user_id IN
            (SELECT 'Favourites.user_id' AS fid
            FROM Favourites
            WHERE song_id = 123456 AND user_id != 987654

// complex 2
SELECT  'Songs.release_id' AS sId,
        'Releases.id' AS rId
FROM Songs INNER JOIN Releases
ON Songs.release_id = Releases.id
WHERE artist_id IN
```

```
 (SELECT 'Genres.artist_id' AS gAID
FROM Genres AS c
INNER JOIN Artists AS d
ON c.artist_id = d.id WHERE d.name = 'artist_name'
```

3.6 Benchmarking configurations

The benchmarking configurations varied from one to four benchmarking harnesses on each node in parallel. The reason for limiting the maximum number of harnesses to four on each node is due to the CPU having two hyperthreaded cores which virtually appear as four cores, running more harnesses than the number of available cores could affect the performance and in consequence the results.

Other configurations were used as well to examine the behaviour of the database. These configurations included running a single harness on two nodes instead of two harnesses on the same node. To run the harnesses in parallel on a single node the GNU Parallel program was used and to run harnesses on multiple nodes GNU parallel was used together with the **rocks** command by specifying which nodes and which command to run on each node.

An important aspect of the dissertation is to compare the performance of the database depending on the database size. For this benchmark the harness was first run to populate the database with a specific amount of data and run some initial tests. Then the harness was run again but only to perform benchmarks on the existing data and take time measurements. The reason for running a number of initial benchmarks is due to indexing and as MySQL in particular performed badly.

3.7 Database configurations

The database systems can be configured in various ways to take advantage of features such as replication and sharding. For this dissertation the configurations for each database were changed to examine different situations and set-ups.

3.7.1 Single node

The simplest configuration for both databases was to use a single data node without any replicas or shards. MySQL is slightly more complicated to configure compared to MongoDB which only requires a single mongod instance to be executed. For MySQL a management node, a data node and a MySQL daemon process are required. The management node handles how the cluster is configured in terms of node groups, replicas, shards, shut-down/restart situations and arbitrating when a master fails whereas the daemon process handles connections and transactions.

3.7.2 Multiple Nodes

Deploying a database on multiple nodes can be complicated due to the various configurations available. Here two configurations are used in order to examine the performance of the databases due to replication and sharding. In the case of MySQL though sharding will not be examined as it requires the use of the NDB storage engine which is not suitable for the application used and as it is currently not representative of the overall RDBMS category.

For both databases the replication node implementation involved two nodes each running a data node and one being a primary or master node and the other a secondary or slave node. The limitation imposed by MySQL Cluster for having one or two replication servers in total prohibits from comparing the two databases by adding further nodes.

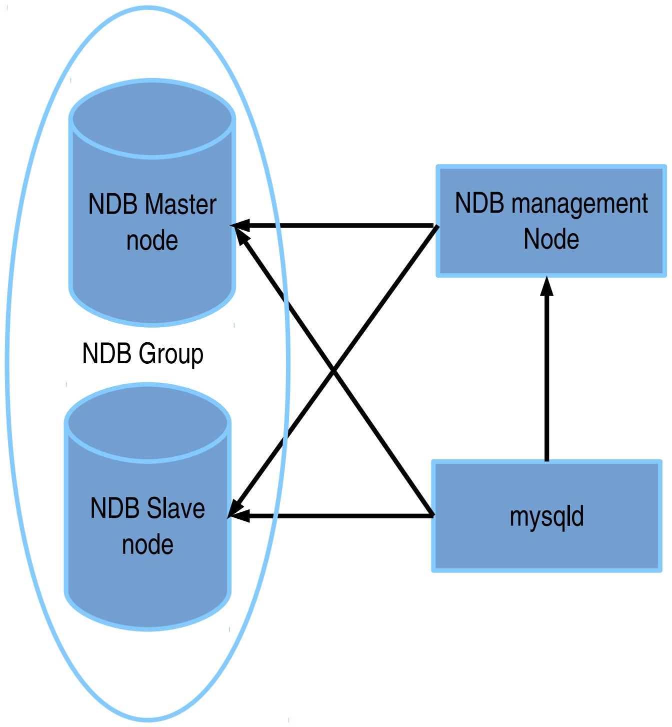

Figure 3.3: MySQL Cluster Architecture

3.7.3 Sharding

Another experiment involved sharding the MongoDB database. Sharding in MongoDB can be done with 1,3 or more shard nodes and it is the process of distributing data across different nodes. In the experiment the configuration of the MongoDB database was as follows:

- Three replica servers each running on a different node.

- Three configuration servers (mongod –configsvr) each running on the same node as a replica server.
- A query router instance (mongos) which chooses which node will execute a query and manages loads.

To deploy the shards a number of settings are configured. The first setting is modifying the chunksize which is the size that a chunk will have when being distributed to a shard. Initially it was set to 1MB but after performing a number of tests and the performance being worse than the single node deployment the values of 128MB and 64 MB were tested as well. Both performed worse than the initial deployment but with 64MB performing slightly better it was chosen as the chunksize for performing benchmarks. According to the MongoDB manual a small chunksize can incur higher CPU costs due to more frequent splitting of collections and migrations to shards and also a large chunksize can cause high I/O due to the size of the chunks being transferred being larger [2].

Another important part of sharding is setting the shard keys. When considering what shard keys to use the following are the main strategies for making a good choice:

- The shard key must be easily "divisible" in order to be easy to split the collection, and by divisible meaning that if the number of possible values is too small then it will be more difficult to create enough chunks.
- It should be highly random. As the chunks are are made according to the shard key and also migrated to shards according to the shard key a bottleneck could be created by having a key not being random enough. This would be the result of a number of shard keys having similar values and their chunks migrating to a single shard whereas another shard could contain a very small number of chunks for the same reason.
- Another approach which conflicts with high randomness is to use a shard key which targets a single shard. This method aims to allow a mongos instance to direct specific queries to a specific shard and is also named **Query Isolation**.
- If a single field in a collection is not optimal for using as a shard key a compound key can be used.

The cardinality of MongoDB which in the database's context refers to its ability to partition data into chunks describes the overall performance of a shard key. Shard keys such as phone numbers have high cardinality as the database can create as many chunks as necessary. Conversely, geographically bound keys, such as post codes or county will depend on the data set [2]. The shard keys chosen for the benchmarks were the **username** for the Users collection, the **artist_name** for the Artists collection, a compound key with the fields **artist_name** and **title** for the Releases collection and the compound key with **user_id** and **release_id** for the Favourites and Comments collections. The shard keys where chosen with the aim of high randomness as multiple queries were used which did not request specific parts of the data set.

Sharding in MongoDB may not be beneficial for read queries as having a query router creates extra overhead and also it is mostly suggested to use sharding when the database size grows extensively [2].

Chapter 4

Results and Analysis

The results which are analysed here firstly show how the databases respond to different query types, both reads and writes, and total number of queries. As it was explained the main method of analysing the results is through 2 metrics, the average time taken by the harness to complete specific numbers of queries and the total queries per second. Both metrics are plotted against the different nodes and thread configurations. Afterwards the results from the sharding implementation of MongoDB are explained and compared with the MongoDB replica implementation and MySQL. Finally the database size is examined regarding whether it affects performance of the two databases.

4.1 Query types

The first set of benchmarks performed measured the average amount of time taken by each database to return specific numbers of queries. For this benchmark the configurations used for the benchmark harness vary from 1 to 3 nodes and from 1 to 4 threads and the number of queries performed are 500-2500. Each harness was ran with 5 tests and each test's results were the average time taken to perform one of the queries. Some configurations result in the same total amount of threads when the number of nodes is multiplied by the number of threads. This helps understand whether the database performance varies when connections are made from different nodes instead of the same node.

The first comparison between the databases is done when performing a simple query which includes only a SELECT SQL statement and the equivalent in MongoDB both of which are displayed in Figure 4.1. MongoDB performs slightly worse for the majority of the benchmarks with the difference being more noticeable as the number of queries increases. This comparison also makes clear that the time doesn't change whether using 2 or 1 nodes when the total number of threads is the same, ie. 1x2 or 2x1. This changes though when moving to 3 nodes as configurations such as 3 nodes with 1 thread perform worse compared to 1 node with 3 threads in MySQL, especially with higher query counts. MongoDB does not show any significant changes with different configurations which leads to the assumption that it is not affected by where the connections were made from.

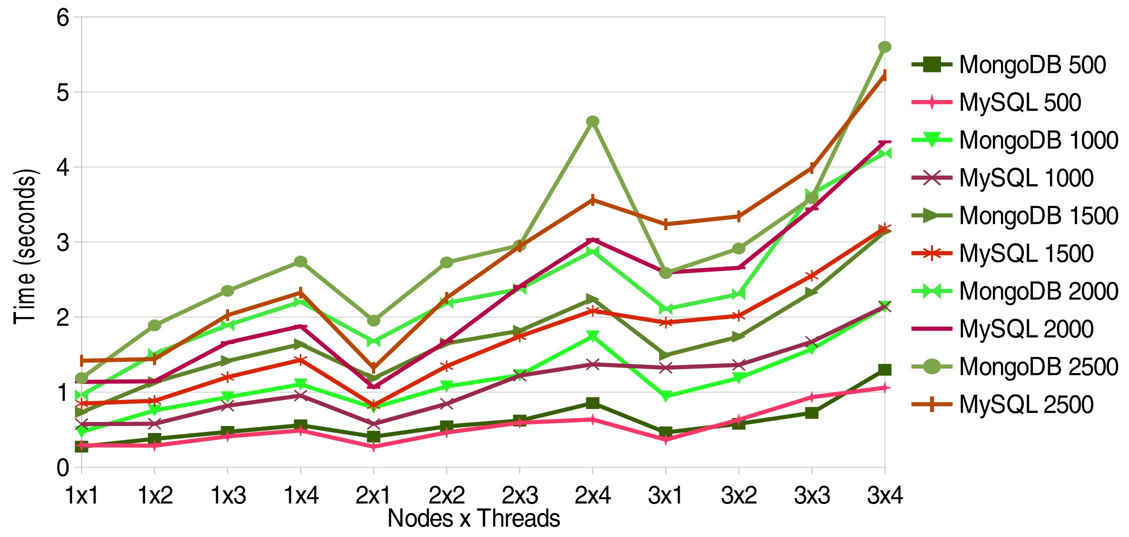

Figure 4.1: Simple query with different configurations

Further conclusions can be made by charting the Queries/second metric instead of the average time. This is displayed in Figure 4.2 against the nodes and threads configuration with only the benchmarks for 500 and 1000 queries shown for more clarity.

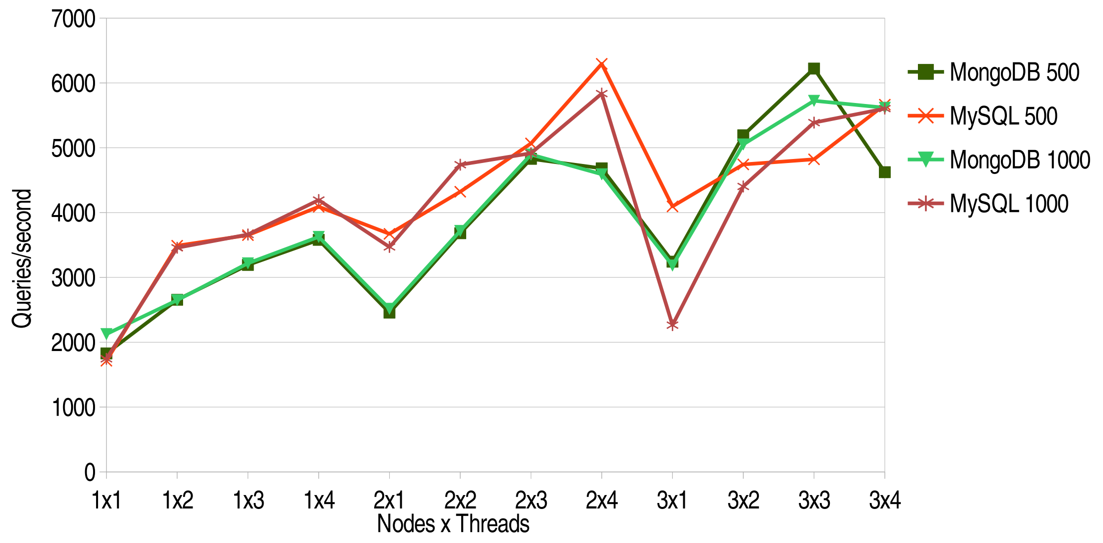

Figure 4.2: Simple queries, total number of queries per second

MySQL has an advantage over MongoDB which appears to diminish as the number of connections increase but not in a definite manner as beyond the configuration of 2 nodes with 3 threads the variation changes and at higher numbers of threads MongoDB shows a small advantage.

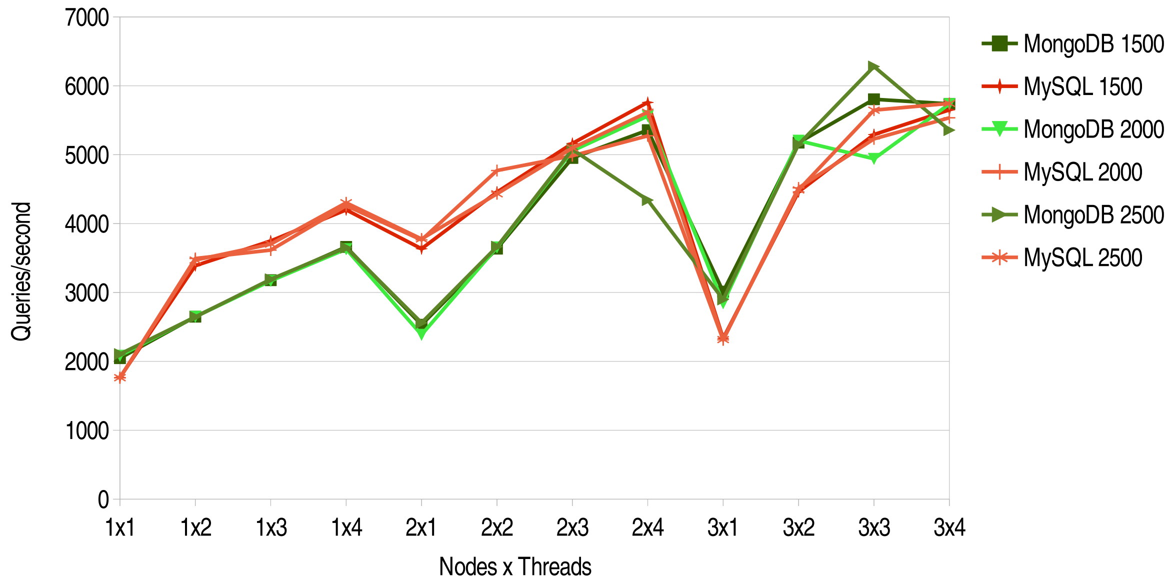

Figure 4.3: Multiple benchmark configurations with simple query measuring queries/second with larger number of queries

To better understand the relation between the number of threads and the queries per second a scatter plot is used as shown in Figure 4.4. The performance is almost identical with slightly better for MySQL up to 4 threads. Beyond 4 threads both databases' performance varies with both declining and no apparent advantage of one over the other.

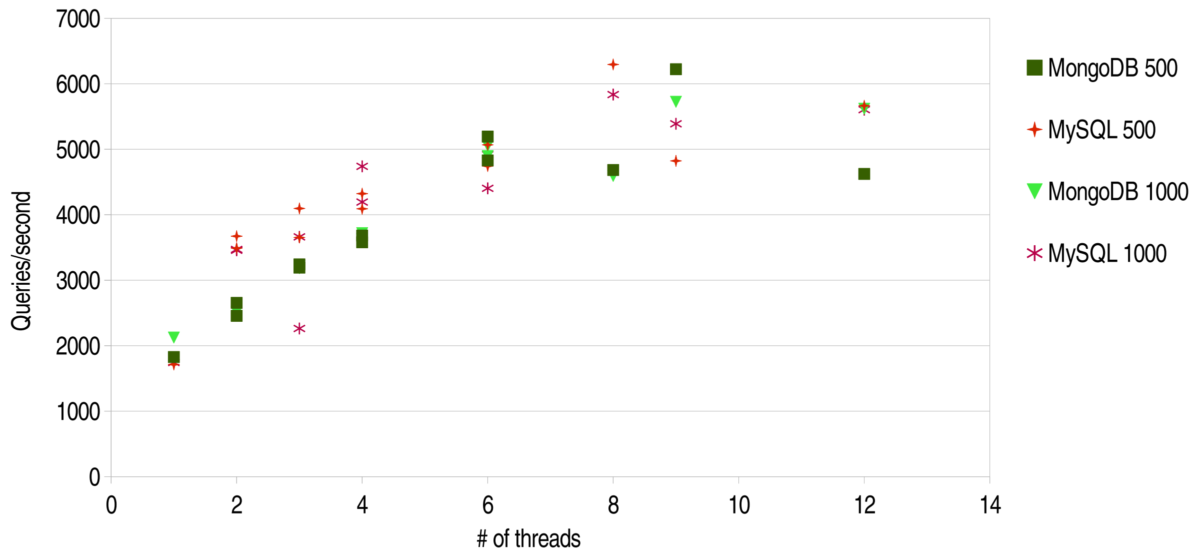

Figure 4.4: Simple query - Queries per second for total number of threads

The second comparison for this configuration when performing a complex query which includes an INNER JOIN in SQL terms is shown in Figure 4.5 for the average query time. MongoDB performs the same no matter where the connections are made from whereas MySQL performs slightly worse when 3 nodes are running each with 1 thread (3x1) instead of 1 node with 3 threads (1x3). Other configurations which indicate the same performance issue are 2x3 with 3x2. This issue is more apparent for runs with higher query counts such as MySQL 2000 or MySQL 2500.

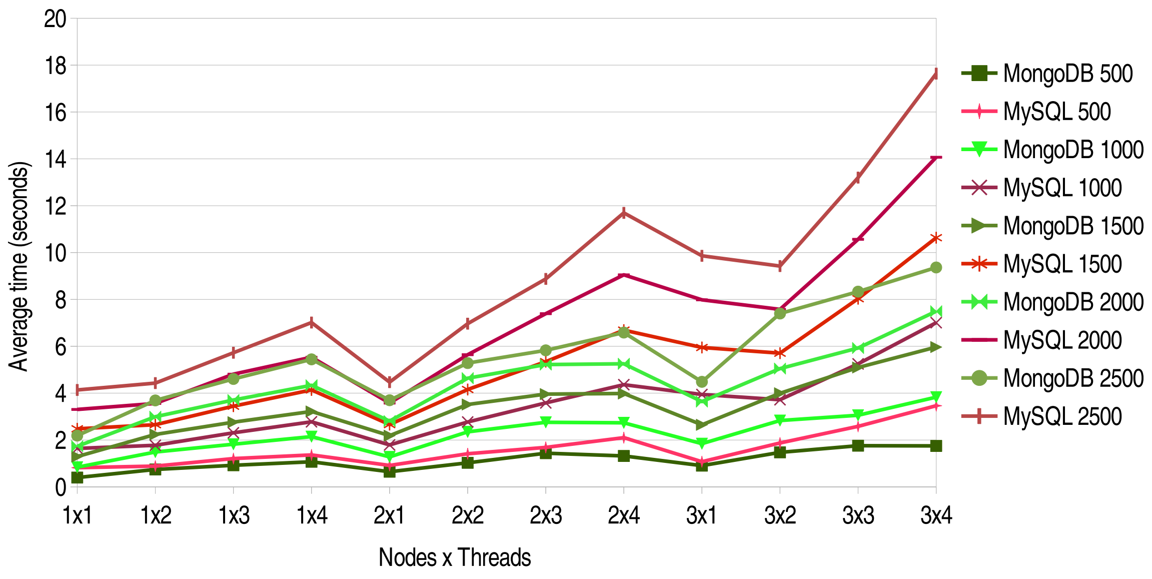

Figure 4.5: INNER JOIN query using multiple configurations

Another conclusion which can be made from the Figure is the overall better performance of MongoDB. Specifically MongoDB is able to execute a larger number of queries in a smaller amount of time in all cases. In measurement the performance is better by about 40% which becomes more apparent as the number of queries grows.

To better understand the performance difference between the two databases Figure 4.6 is used which plots Queries per second against the various nodes and threads configurations. The chart shows the constantly higher rate at which MongoDB handles queries. For all configurations the number of Queries per second is higher compared to the corresponding MySQL rate by almost 20%. The figure also verifies the conclusion made from the previous chart regarding having connections from different nodes being faster although the difference for having 2 nodes with 2 threads each instead of 1 node with 4 threads is significantly smaller but is also more apparent.

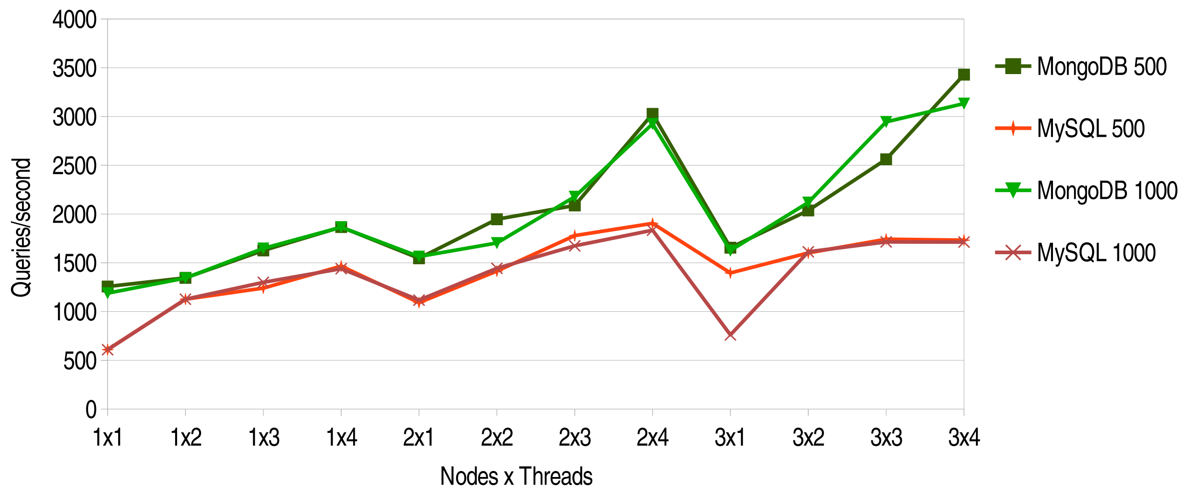

Figure 4.6: INNER JOIN query, queries/second for 500 and 1000 queries

A helpful way to understand whether there is an underlying trend in the results is to scatter plot the total number of threads and the queries/second metric as shown in Figure 4.7 where unlike the trend followed for the simple query the two databases show a different performance. MySQL appears to have a nearly linear trend in terms of performance but with a more gradual decent. MongoDB follows appears to have a linear trend which towards larger numbers of threads becomes more variable.

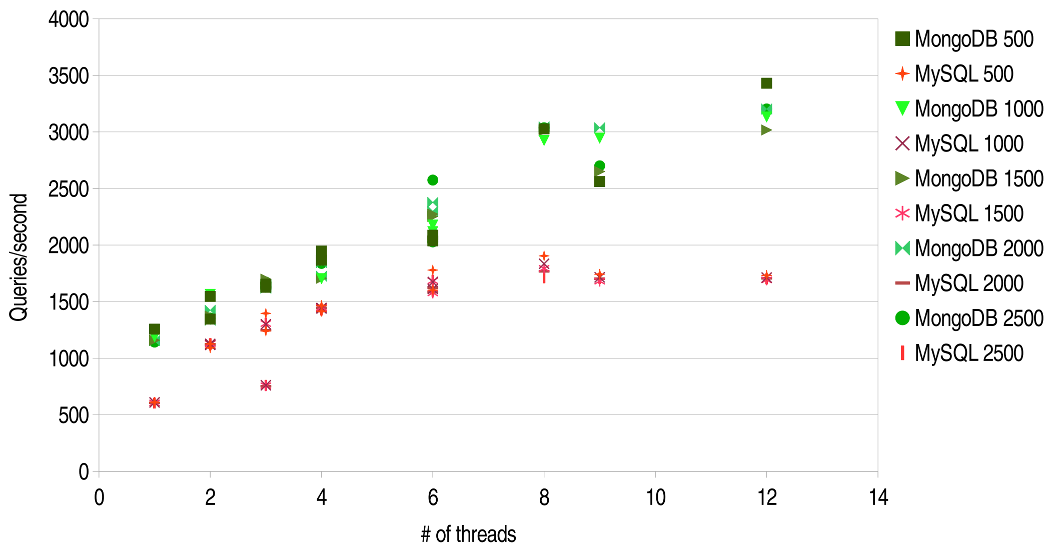

Figure 4.7: INNER JOIN QUERY two runs multiple configurations

The benchmark following is a repetition of the first complex query which was contained an INNER JOIN operation but this uses the alternative nested SELECTs or subqueries. Figure 4.8 shows the resulting average times against the various configurations for both MongoDB and MySQL. MySQL has an advantage over MongoDB for the majority of runs but for the configurations of 3x2, 3x3 and 3x4 the performance is almost identical.

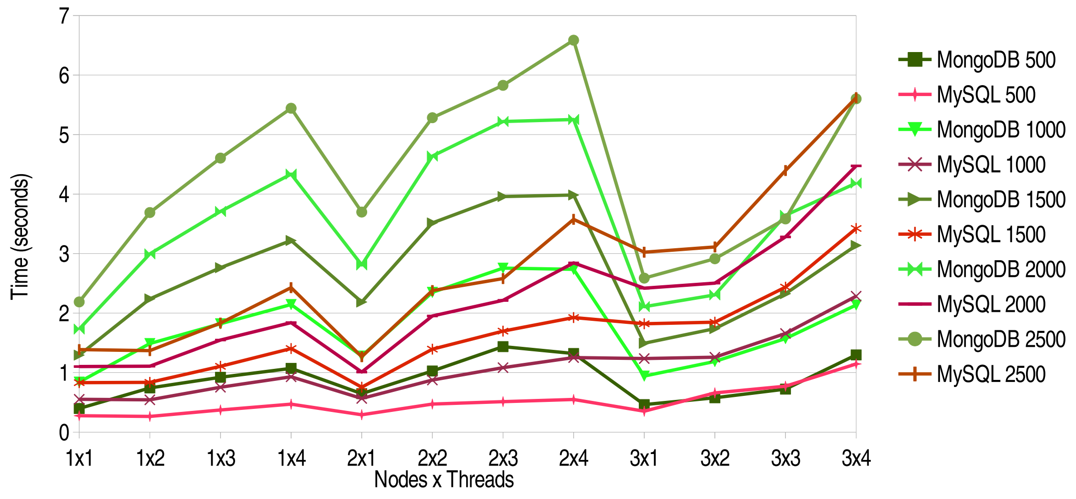

Figure 4.8: Nested SELECT query

To better understand the results the total queries per second chart is shown in Figure 4.9. In this chart the better performance of MySQL is clear up to 2x4 threads. Afterwards the 2 databases' becomes similar. This can be attributed to the higher number of connections from different nodes in combination with the complicated nature of the query executed.

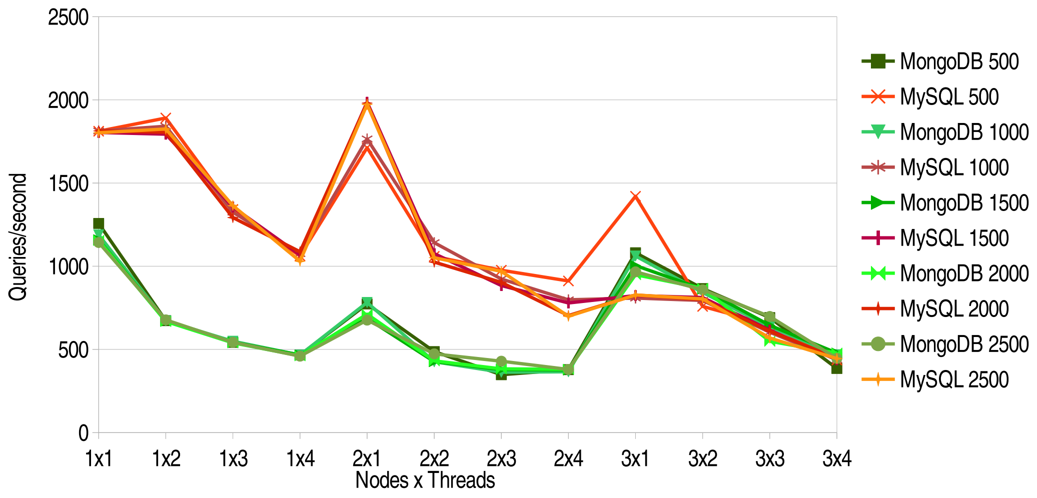

Figure 4.9: Nested SELECT query queries/second

The scatter plot of queries per second against the total number of threads in Figure 4.10 shows the overall declining trend of MySQL whereas MongoDB's performance diminishes up to 6 threads and then remains relatively stable.

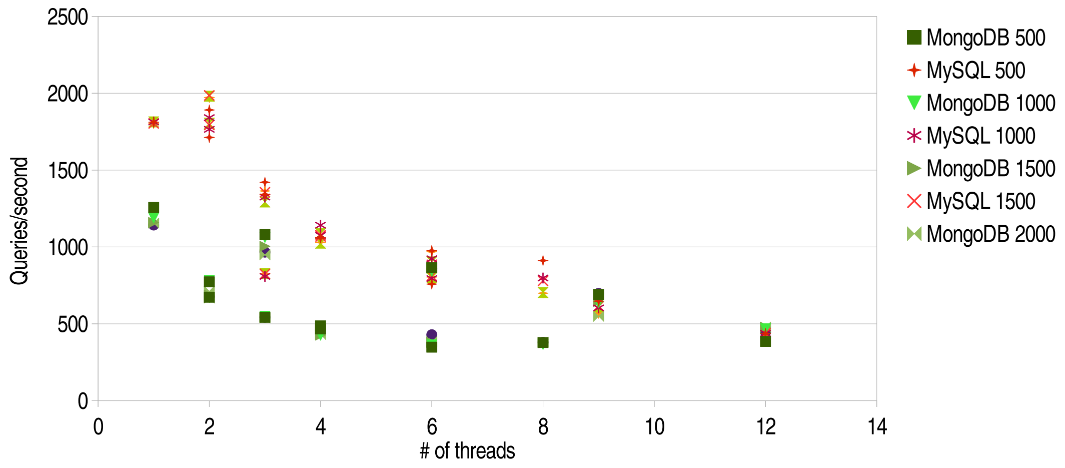

Figure 4.10: Nested SELECT query scatter plot

The next benchmark involving the normal MongoDB deployment was regarding a highly complex query with 2 INNER JOINs and a subquery. The resulting measurements for the average time are shown in Figure 4.11. The chart shows the performance penalty of MySQL having to perform such a complex query whereas the MongoDB equivalent performs much better. This is attributed to the fact that MongoDB uses subdocuments instead of deploying a full database schema and thus saves time by requiring to query a smaller number of collections. This advantage that NoSQL databases have over RDBMSs comes at the cost of data duplication resulting from a denormalised database schema.

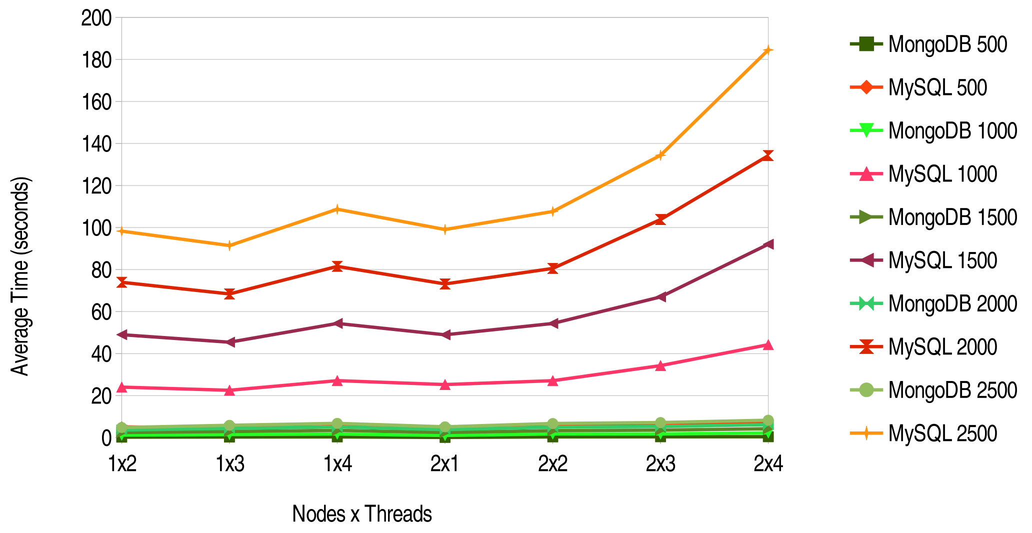

Figure 4.11: 2 INNER JOINS & subquery

In Figure 4.12 the same benchmark is shown by plotting the average queries per second. MySQL appears to have a logarithmic trend which is proved by plotting the same chart using the logarithms of the two axis. This is shown in Figure 4.13 where not only MySQL but also MongoDB appear to have the same trend. This is unclear in the Figures 4.11 and 4.12 due to large difference in the timings of MySQL which enlarge the scale. Despite this though MySQL still has a disadvantage due to its performance.

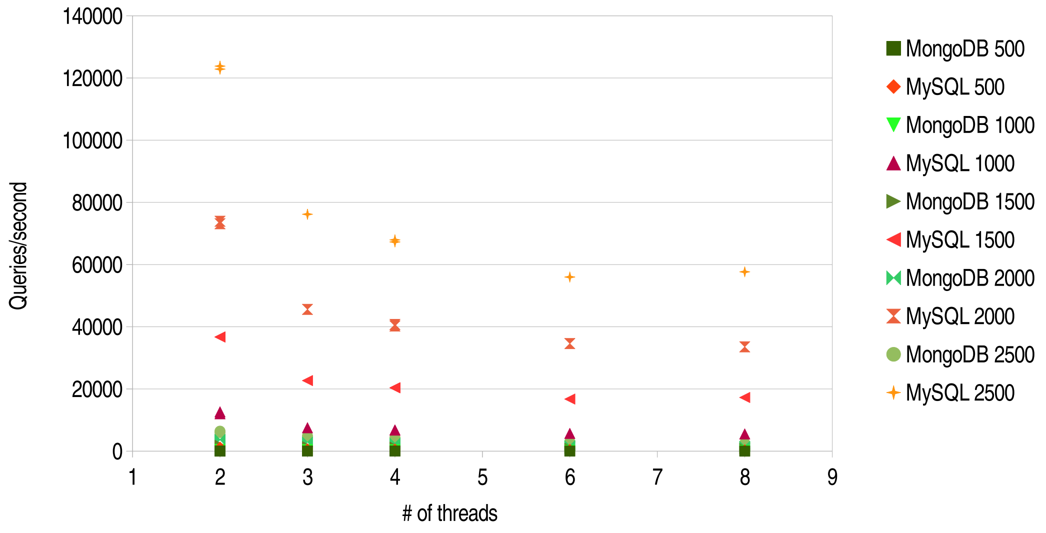

Figure 4.12: 2 INNER JOINS & subquery, queries per second

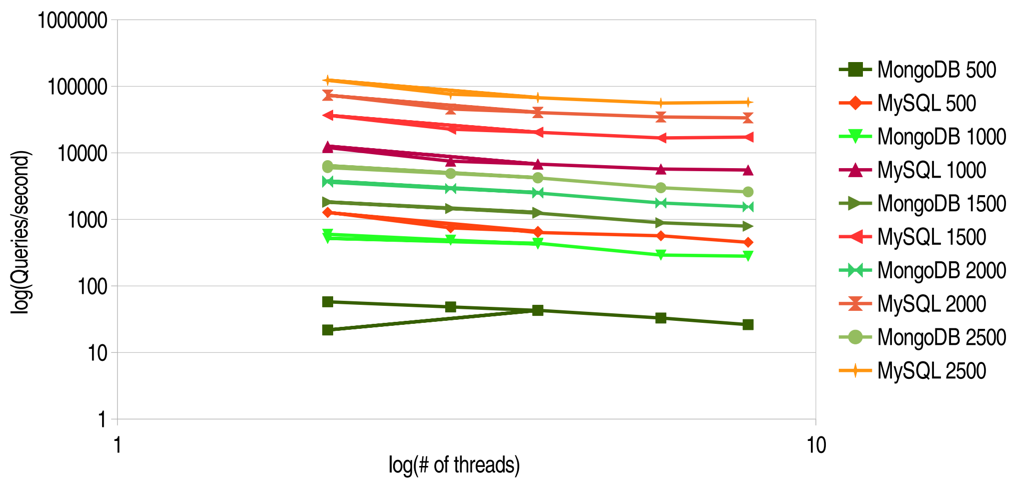

Figure 4.13: 2 INNER JOINS & subquery, number of threads

The last set of benchmarks tested the write operations of the two databases. Although the majority of database commands are expected to be queries for data reading writing is an important part nonetheless. In Figure 4.14 the performance of both databases is shown with both INSERT and DELETE commands. All databases perform well and don't seem to be affected by the number of operations. Also they show a linear trend with deletes being the most costly operations for both. MySQL has a better performance in deletion of data whereas MongoDB excels at insertions. What is also interesting is the difference between MongoDB's operations which when compared with the previous benchmarks which involved a simple query becomes clearer as to why this happens. Specifically, a delete operation requires a search through the database to find the correct documents or record. This means that both databases will execute a search which is similar to a SELECT and MongoDB performed worse than MySQL in the specific case.

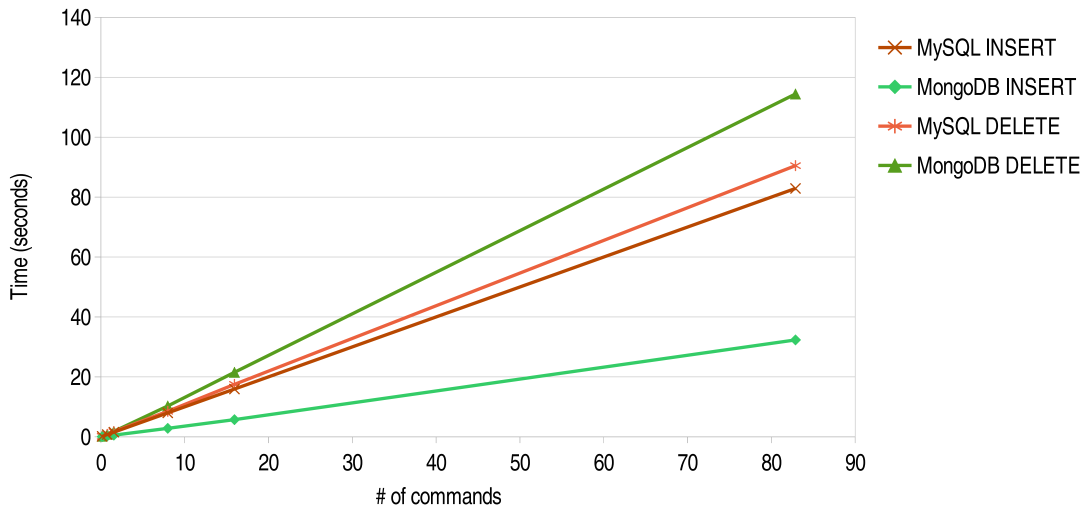

Figure 4.14: INSERT and DELETE operations

4.2 MongoDB Sharding

The following benchmarks were performed on the MongoDB shard deployment which was configured with 3 shard nodes and a query router. These benchmarks which are shown in Figure 4.15 were done using the simple query and the complex query which in MySQL would be the equivalent of having an JOIN operation. The shard configuration does not perform well with the replica configuration being faster in both cases. This can be attributed to the fact that the query router in the MongoDB shard creates an overhead instead of allowing an improvement by separating the load amongst the database nodes.

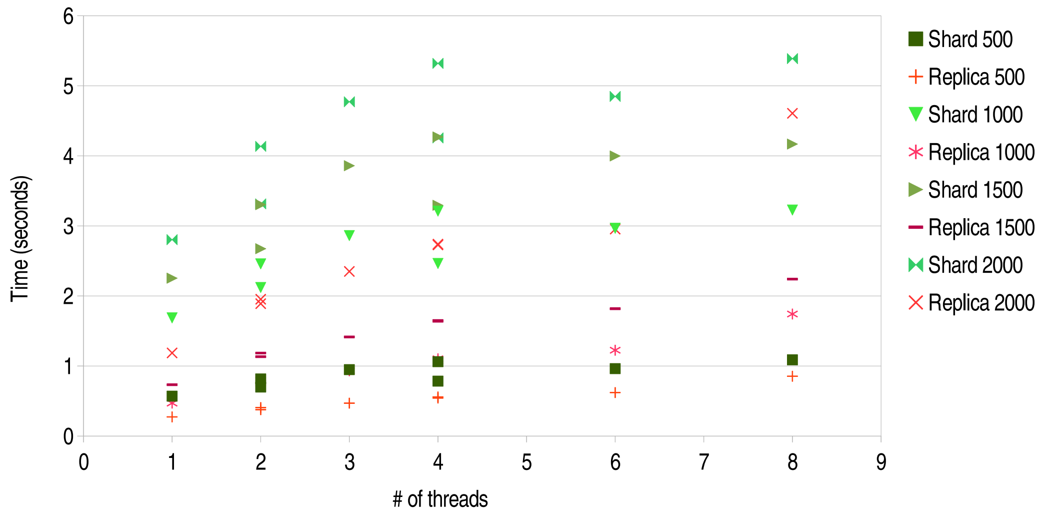

Figure 4.15: MongoDB Replica and Shard configurations, simple query

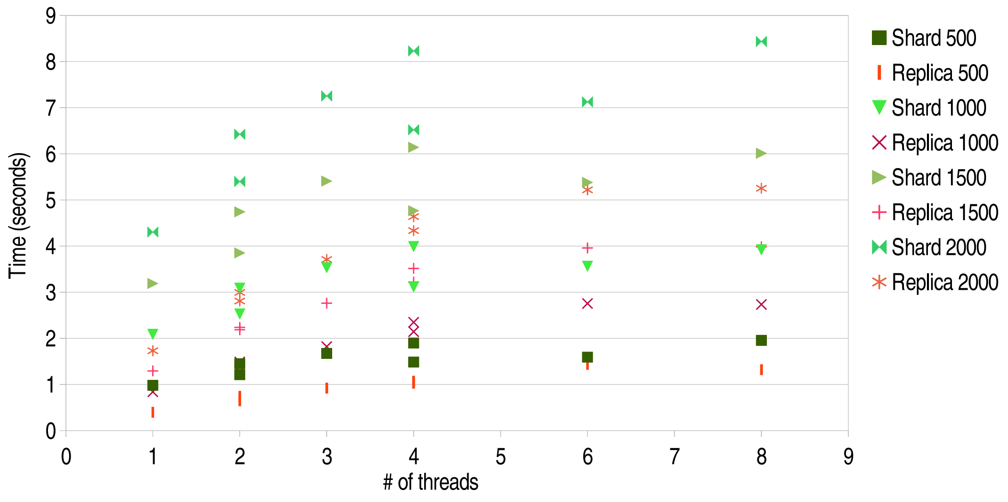

Figure 4.16: MongoDB Replica and Shard configurations, complex query

In Figure 4.17 the comparison the scatter plot of MongoDB shards and MySQL is shown when performing the complex query. The same plot but with the simple query is unnecessary as MySQL outperformed the MongoDB replica configuration which itself was faster than MongoDB shards. The first observation which can be mad is the close performance the two databases systems have. Specifically, MySQL has a linear trend which seems to become more steep as the total number of queries increases. MongoDB shard also follows a linear trend but the most interesting deduction which can be made from the chart, as well as the previous ones is the performance drop caused by the connections made through a single node instead of two, ie. 1x4 versus 2x2. This again shows the effect different connection usage can have on MongoDB.

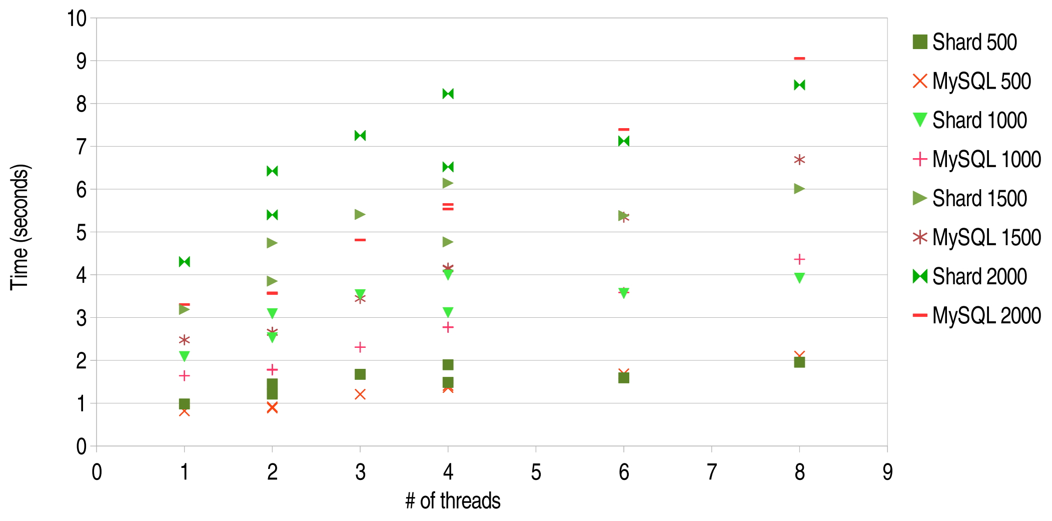

Figure 4.17: MongoDB Shard and MySQL, complex query

4.3 Database size

In order to measure the performance of the database management systems according to the size of the database an iterative approach was considered, whereby a specific process was repeated with only a single variable changing, in this case the variable being the database size. The benchmark initially deleted all existing records from the database and repopulated it with the required data set size. Afterwards a test was performed which was not recorded but it allowed to eliminate any caching issues with newly stored data which would disrupt subsequent measurements. The normal benchmark followed which measured a specific amount of queries. This procedure was repeated for all database sizes and the measurement for the becnhmark with simple queries is displayed in Figure 4.18. The chart which is for a simple query shows a steady trend for both databases with no significant variations as the database size increases.

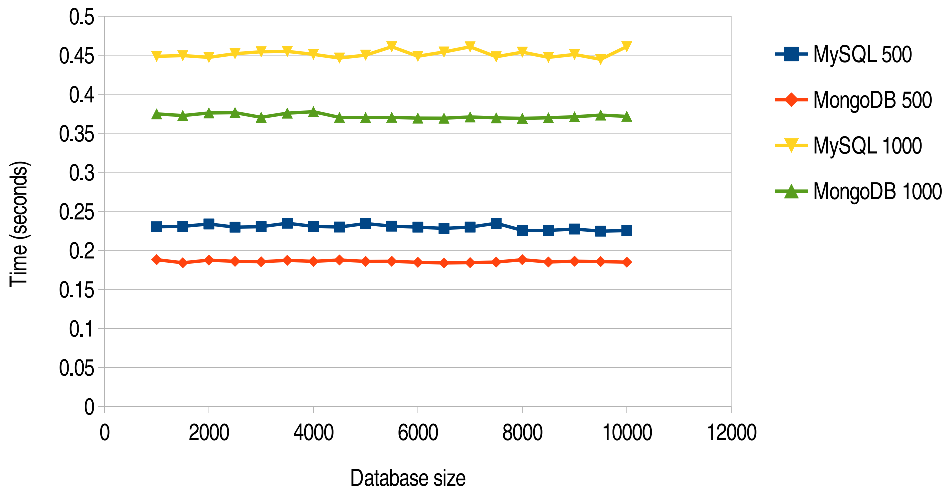

Figure 4.18: Simple query with different database sizes

In Figure 4.19 the variation of execution time of 500 and 1000 complex queries is shown in relation to the database size. Here MongoDB has a similar performance with the previous benchmark with no significant variations. MySQL though has a slightly diminishing performance as time increases when the database size increases. This can be explained by the access of two tables which although is not a linear search due to indexing the fact that the result of each query requires joining the data from two tables is sufficient to explain the small performance penalty as it becomes increasingly more difficult perform searches in two tables at the same time. Such a performance penalty may be observed in much larger database sizes when performing a simple query but the penalty may be negligible. Due to the high memory requirement of the benchmarking harness performing benchmarks which it was unable to handle,subsequent measurements for databases with larger sizes were not performed.

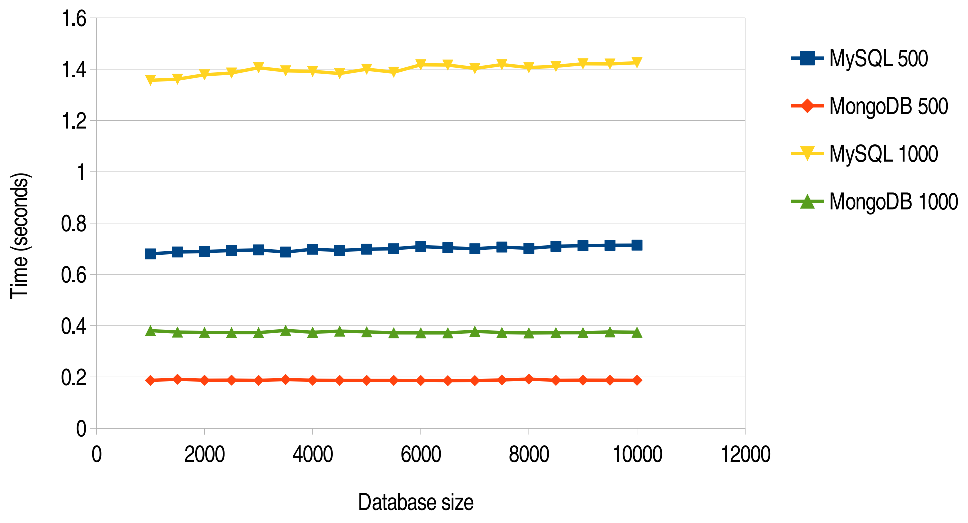

Figure 4.19: Complex query with different database sizes

Chapter 5

Conclusion

The project was an investigation and comparison of the performance and scaling of Relational Database Management Systems and NoSQL databases with the aim of exploring how the different factors affect each database and whether on technology is more suitable than the other and in which situations. The first database type which, relational databases, were developed with structure in mind and built on the premise of tables with rows which followed a pre-specified schema. Their most important feature, the database schema which gave a logical view of the database in combination with relations between tables allowed the building of databases which were fast, easy to develop, and eliminated duplication of data but at the same time guaranteed reliability. NoSQL databases which are relatively new became a popular trend as they provide performance and horizontal scalability which made them suitable for data centers requiring massive amounts of storage and the capability of extending the database easily as no pre-defined schema exists.

The project tested, analysed and compared the performance and scalability of the two database types. The experiments done included running different numbers and types of queries, some more complex than others, in order to analyse how the databases scaled with increased load. The most important factor in this case was the query type used as MongoDB could handle more complex queries faster due mainly to its simpler schema at the sacrifice of data duplication meaning that a NoSQL database may contain large amounts of data duplicates. Although a schema directly migrated from the RDBMS could be used this would eliminate the advantage of MongoDB's underlying data representation of subdocuments which allowed the use of less queries towards the database as tables were combined. Despite the performance gain which MongoDB had over MySQL in these complex queries, when the benchmark modelled the MySQL query similarly to the MongoDB complex query by using nested SELECTs MySQL performed best although at higher numbers of connections the two behaved similarly. The last type of query benchmarked which was the complex query containing two JOINS and and a subquery showed the advantage MongoDB has over MySQL due to its use of subdocuments. This advantage comes at the cost of data duplication which causes an increase in the database size. If such queries are typical in an application then it is important to consider NoSQL databases as alternatives while taking in account the cost in storage and memory size resulting from the larger database size.

The second type of query operations performed in the benchmarks were write operations. The results from these benchmarks showed that MySQL performs better when deleting data which is logical considering that it performs better in simple search queries. This is connected to deletion as deletion requires finding the record to be deleted first. MongoDB performed better during insertions with both databases having a linear trend in these benchmarks.

Another important aspect of the experiments was the use of different configurations for nodes and threads. This part of the benchmarks required running the benchmarking harness on 1, 2 and 3 nodes with multiple numbers of threads in order to test how the databases performed with multiple connections. This benchmark was done in conjunction with the previous benchmark of having different query types. Depending on the query complexity the databases behave differently but at higher numbers of connections the performance in terms of queries/second appeared to converge.

As the two databases behave differently according to the type of queries used the choice of which database to use lies on the type of application the system will be using. In addition it is important to consider the effect that using a database such as MongoDB will have on the hardware storage due to the increased database size. Despite the indication that the performance penalty on both databases is small depending on the database size it is nonetheless an important factor when considering the type of queries which will be performed by applications.

5.1 Deviations from the project plan

The dissertation followed the plan drafted during Project Preparation but deviated at some points during the later stages in order to accommodate unexpected problems with the MongoDB benchmarking part. In order to address the issue the extra time which was set at the end of the plan was distributed to solve the problems. In addition, writing up started in parallel with the development and running of the benchmarking harness due to the limited time available and in case any further deviations were made.

Specifically the first four parts of the project according to the work plan were completed successfully and within the time constraints. Subsequently the stage of implementing the databases was completed but it overlapped with the next stage of building the benchmarking harness as unexpected issues were found while configuring MySQL Cluster. The issues included the difference in hostnames expected by the system when adding nodes and also issues with shutting down and restarting the databases in order to try different configurations. The problems which arose when restarting the servers were mainly due to improper shutdowns or one node being out of sync with the other nodes.

The next stage which was building the harness began as planned but extended late into the project as unexpected problems surfaced mainly because of differences in the documentation of the MongoDB driver and the actual implementation. In more detail a function regarding indexing required different arguments compared to the documentation. This caused a delay as it was considered that indexing may have a serious effect on the performance and as the RDBMS used indexing as well its importance intensified. Other problems with the development of the harness were the fact that MongoDB initialises some MongoDB specific pointers to objects using its own functions and does not require dynamic allocation. This resulted in some confusion as it wasn't clear initially which objects were being initialised thus requiring a MongoDB specific method to deallocate them.

In regards to the dissertation's scope the most important objective which was the comparison of the performance and scalability of RDBMSs against NoSQL databases was completed successfully. Various queries were used to stress both databases and different configurations of the test harness were used in order to understand how the databases behave when using queries of different complexity. In addition the systems were benchmarked in regards to the database size which showed that this factor barely affects performance.

5.2 Future work

As the amount of time available for this project was relatively small the possibilities for further extensions and improvements are numerous. One of the most important aspects which can be explored are the additions of other databases for benchmarking. Especially since many types of NoSQL databases exist the use of a more diverse range would provide insight as to not only how well the different databases perform but also whether they can be used at all instead of relational databases. Also other relational databases are likely to perform differently due to optimisations and different storage engines used. By adding more databases the comparison between relational databases and NoSQL databases would become more objective and provide more measurements to compare the two types.

An important part of the benchmarking harness is the various queries used. Different queries affect the performance of each database differenty and finding how they do is essential in better understanding this. By being able to understand how the queries affect performance someone wanting to use a database for a specific scope, such as queries involving multiple tables with JOINs, can choose the database which performs best in that specific situation.

Although the project gives an indication regarding how well databases cope with large numbers of queries further work can be done by using larger clusters to test the performance. As it was shown different numbers of connections and configurations affect the behaviour of the databases. By extending this to a larger scale a more broad investigation can be performed with more general conclusions as to this factor.

The different sizes of a database may have significant influence in ways other than the ones explored in this project. An example is the complexity of the databases in terms of number of tables or how a database is deployed in NoSQL databases.

The reliability of the two database types may not have been part of the project but in the future this aspect can be examined. The reason for doing so is to understand whether increased performance and scalability by one database type over the other affects the reliability. Specifically, since NoSQL do not guarantee the ACID properties a middle ground can be found whereby performance and scaling reach an adequate level but at the same time the reliability of the database is guaranteed as well.

Appendix A

Benchmark Results

A.1 MongoDB

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.05035	0.074801	0.0942903333	0.11524975
200	0.097766	0.1502885	0.1897333333	0.22523975
300	0.144763	0.22721	0.2821616667	0.33199875
400	0.202356	0.3021275	0.3754526667	0.448554
500	0.273659	0.376707	0.4698933333	0.5587265
1000	0.470587	0.75516	0.9323583333	1.1030975
1500	0.732803	1.1325355	1.4159336667	1.63879775
2000	0.960046	1.5101695	1.8931296667	2.20410125
2100	1.004248	1.5833975	1.978283	2.34381525
2300	1.094791	1.73553	2.1736936667	2.53060125
2500	1.186967	1.889928	2.350015	2.73996525
2700	1.285125	2.0533105	2.548669	2.98266925

Table A.1: MongoDB Simple Query average time, 1 node

A.2 MySQL

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.072884	0.099085	0.130137	0.14807075
200	0.144844	0.2060455	0.262856	0.295738
300	0.225978	0.32249	0.4233533333	0.526319
400	0.29739	0.4310515	0.5097736667	0.69137975
500	0.407062	0.543159	0.6211986667	0.854182
1000	0.794679	1.0757975	1.2257133333	1.74231825
1500	1.185081	1.6500945	1.8171883333	2.240558
2000	1.675761	2.188174	2.3739636667	2.87735925
2100	1.681125	2.2710725	2.4950703333	3.13910225
2300	1.847344	2.50949	2.813234	3.98277125
2500	1.953037	2.728735	2.9550893333	4.60766075
2700	1.989396	2.9323695	3.22776	4.98963175

Table A.2: MongoDB Simple Query average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.080957	0.127828	0.1593146667	0.20512375
200	0.182333	0.240611	0.3160423333	0.4027045
300	0.276998	0.3634	0.441995	0.66105275
400	0.332997	0.4569375	0.6074493333	0.8425005
500	0.462611	0.577711	0.7232206667	1.29780675
1000	0.942273	1.1875285	1.571981	2.1360505
1500	1.493845	1.7390995	2.3258886667	3.1370775
2000	2.108036	2.3059335	3.643536	4.18452625
2100	2.197181	2.4322415	3.9224356667	4.38129575
2300	2.25473	2.6264015	4.04238	4.712586
2500	2.587489	2.9131535	3.5828156667	5.600664
2700	2.861409	3.155052	4.265568	6.4967085

Table A.3: MongoDB Simple Query average time, 3 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.050526	0.147696	0.183089	0.22382425
200	0.129832	0.2953615	0.3669853333	0.438745
300	0.219269	0.4463085	0.5453056667	0.654373
400	0.307257	0.5933595	0.7333143333	0.8804445
500	0.397736	0.7430275	0.922133	1.07154475
1000	0.841227	1.488597	1.8211506667	2.144565
1500	1.294182	2.2384495	2.7625996667	3.21998075
2000	1.731686	2.9950495	3.7085433333	4.335557
2100	1.826487	3.109542	3.866582	4.560364
2300	2.017345	3.3966215	4.2568256667	4.9663935
2500	2.188028	3.6900805	4.60522	5.44234725
2700	2.362815	3.9834745	4.9965503333	5.805082

Table A.4: MongoDB Complex Query 1.1 average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.078912	0.2174985	0.2290246667	0.34936425
200	0.227197	0.4295095	0.5054376667	0.5658265
300	0.37874	0.652467	0.85488	0.81999975
400	0.552387	0.790616	1.0531843333	1.04738
500	0.646774	1.0272245	1.4371743333	1.3214735
1000	1.278552	2.3485735	2.7555153333	2.7352275
1500	2.186346	3.515762	3.9589636667	3.98485975
2000	2.809115	4.6391115	5.219248	5.25347075
2100	3.024854	4.8066975	5.421256	5.47508575
2300	3.603725	4.9771635	5.5982563333	5.98611
2500	3.698275	5.2832795	5.8270666667	6.585386
2700	3.247461	5.4177675	6.4387053333	7.11697425

Table A.5: MongoDB Complex Query 1.1 average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.215255	0.371047	0.499763	0.66309875
200	0.430895	0.7453935	1.0258483333	1.37526975
300	0.656353	1.1197965	1.501206	2.048496
400	0.859615	1.510125	2.0908353333	2.70110475
500	1.074603	1.873236	2.5834286667	3.464211
1000	3.941992	3.720005	5.2496626667	7.0068985
1500	5.948298	5.7065595	8.0305213333	10.6356305
2000	7.983516	7.5759075	10.56764	14.06899475
2100	8.418224	8.02625	11.1193646667	14.82273875
2300	9.298858	8.6862925	12.1055416667	16.15904625
2500	9.863524	9.4254715	13.1997393333	17.63854275
2700	8.742225	10.230226	14.0493186667	18.98359175

Table A.6: MongoDB Complex Query 1.1 average time, 3 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.080393	0.1168715	0.1647213333	0.195534
200	0.150465	0.232192	0.318468	0.41097425
300	0.229106	0.352806	0.4842546667	0.6369915
400	0.316948	0.4775145	0.6612903333	0.8365465
500	0.414129	0.588893	0.8303646667	1.05504875
1000	0.776958	1.1704395	1.663416	2.11225275
1500	1.286811	1.7676875	2.376884	3.16690825
2000	1.723527	2.3641295	3.2713196667	4.257003
2100	1.756042	2.437629	3.5009043333	4.33188225
2300	1.929536	2.7019305	3.6830543333	4.85577425
2500	2.113822	2.9521375	4.0170063333	5.25217325
2700	2.223446	3.152064	4.468722	5.65689275

Table A.7: MongoDB Complex Query 1.2 average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.10911	0.1797775	0.2081716667	0.27008425
200	0.225294	0.411644	0.478181	0.64342325
300	0.337303	0.6055115	0.614586	0.85738225
400	0.460735	0.672302	0.941283	1.19771
500	0.622098	0.8453025	1.0813196667	1.4393685
1000	1.21275	1.707435	2.2559476667	2.947113
1500	1.803349	2.710437	3.5871243333	5.0185145
2000	2.469281	3.658327	4.8133753333	6.72662075
2100	2.677006	3.9482585	5.018753	6.985336
2300	2.838046	4.36349	5.8737736667	7.350817
2500	3.069688	4.578081	6.526868	7.67256425
2700	2.973248	4.6690235	6.7771093333	8.2895225

Table A.8: MongoDB Complex Query 1.2 average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.136032	0.223319	0.312389	0.43331525
200	0.331645	0.4696665	0.7638783333	0.98583925
300	0.511923	0.5745345	0.924682	1.45846425
400	0.609514	0.8365365	1.227705	1.9599605
500	0.755289	1.0066945	1.6918066667	2.467919
1000	1.517494	2.3093185	3.8895786667	5.2573975
1500	2.331923	3.8123925	5.408842	7.82585675
2000	3.245686	5.3170795	7.8764203333	10.8676315
2100	3.362951	5.3938585	7.148951	11.31104425
2300	3.807893	5.712998	8.1956793333	12.0761485
2500	4.017699	6.021579	9.260337	13.33493925
2700	4.277297	6.0331395	9.8787476667	14.0225775

Table A.9: MongoDB Complex Query 1.2 average time, 3 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.061421	0.231165	0.290309	0.34301925
200	0.203652	0.4744265	0.5767286667	0.680468
300	0.337207	0.7198445	0.868374	1.02805675
400	0.47735	0.96294	1.1710863333	1.37168925
500	0.633231	1.196814	1.4644186667	1.70562075
1000	1.325132	2.4163135	2.9124326667	3.406436
1500	2.060693	3.6323355	4.358838	5.07936275
2000	2.755321	4.8103525	5.873472	6.79962275
2100	2.90612	5.053776	6.1517153333	7.164863
2300	3.202521	5.553501	6.760138	7.79666975
2500	3.48786	6.0176825	7.3284903333	8.48593375
2700	3.774195	6.526532	7.9316666667	9.1299555

Table A.10: MongoDB Complex Query 2 average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.087263	0.343942	0.3955633333	0.419485
200	0.347202	0.6618705	0.7332733333	0.808704
300	0.602966	0.999728	1.089874	1.31307575
400	0.848935	1.429643	1.4354313333	1.68168725
500	1.037088	1.7503155	1.7463206667	2.23941025
1000	2.441934	3.3056205	3.5624033333	4.219077
1500	3.786181	4.984063	5.2961383333	6.172839
2000	5.14681	6.7231615	7.1853886667	8.2990265
2100	5.256668	7.1710975	7.6076033333	8.61932925
2300	5.804121	8.140611	8.4075943333	9.4345915
2500	6.595318	8.9809205	9.035602	10.4722445
2700	6.281738	9.6164965	9.486774	11.27073675

Table A.11: MongoDB Complex Query 2 average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.057458	0.061674	0.082752	0.10024275
200	0.113917	0.114558	0.1600176667	0.196732
300	0.171563	0.177839	0.2458713333	0.29511725
400	0.228046	0.2300545	0.3177706667	0.3943075
500	0.291688	0.2865785	0.4106786667	0.48890725
1000	0.573398	0.578496	0.8191866667	0.95386125
1500	0.848657	0.885095	1.2006263333	1.4289275
2000	1.136185	1.14509	1.6596243333	1.87810025
2100	1.185663	1.190022	1.7288023333	1.957549
2300	1.310109	1.3139195	1.8203516667	2.19675025
2500	1.417835	1.440846	2.025522	2.32465
2700	1.524579	1.527794	2.228788	2.55002825

Table A.12: MySQL Simple Query average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.072884	0.099085	0.130137	0.14807075
200	0.144844	0.2060455	0.262856	0.295738
300	0.225978	0.32249	0.4233533333	0.526319
400	0.29739	0.4310515	0.5097736667	0.69137975
500	0.407062	0.543159	0.6211986667	0.854182
1000	0.794679	1.0757975	1.2257133333	1.74231825
1500	1.185081	1.6500945	1.8171883333	2.240558
2000	1.675761	2.188174	2.3739636667	2.87735925
2100	1.681125	2.2710725	2.4950703333	3.13910225
2300	1.847344	2.50949	2.813234	3.98277125
2500	1.953037	2.728735	2.9550893333	4.60766075
2700	1.989396	2.9323695	3.22776	4.98963175

Table A.13: MySQL Simple Query average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.084	0.1393085	0.1759423333	0.238261
200	0.151647	0.2944145	0.3608253333	0.468403
300	0.22147	0.3998765	0.5514316667	0.658146
400	0.292908	0.537042	0.7227463333	0.864767
500	0.366257	0.632505	0.9331723333	1.0591655
1000	1.325605	1.362591	1.6702823333	2.141234
1500	1.927019	2.017488	2.5509003333	3.1857695
2000	2.594995	2.6560925	3.4424663333	4.33593325
2100	2.825633	2.711552	3.3977276667	4.4995645
2300	2.960555	3.163766	3.7804506667	4.87968525
2500	3.236498	3.341857	3.985407	5.223226
2700	3.123149	3.6664755	4.4574293333	5.64654175

Table A.14: MySQL Simple Query average time, 3 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.162953	0.2390285	0.2319773333	0.28192075
200	0.326054	0.3588795	0.4587226667	0.56699375
300	0.489187	0.538394	0.6960813333	0.8192115
400	0.654635	0.6948575	0.917683	1.16042525
500	0.819795	0.887315	1.2090436667	1.364675
1000	1.641447	1.775193	2.30831	2.7781425
1500	2.47847	2.653788	3.4445493333	4.13739025
2000	3.305032	3.5608375	4.8113383333	5.534963
2100	3.470025	3.7006245	4.880723	5.915794
2300	3.804865	4.1072415	5.2554666667	6.3706465
2500	4.136924	4.42951	5.724727	7.013455
2700	4.474203	4.7103945	6.2331216667	7.5139345

Table A.15: MySQL Complex Query 1.1 average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.183805	0.325291	0.314581	0.35671575
200	0.373505	0.614633	0.6772796667	0.83709325
300	0.539237	0.9292885	1.038387	1.22201675
400	0.697565	1.2794935	1.324781	1.74812
500	0.915612	1.413485	1.686512	2.10037875
1000	1.789728	2.769948	3.5846636667	4.36093625
1500	2.658399	4.156596	5.3436446667	6.690046
2000	3.576829	5.6412465	7.391638	9.051557
2100	3.722857	5.849565	7.3606406667	9.430302
2300	4.093636	6.311247	7.9748973333	10.154494
2500	4.470268	6.9644455	8.8771803333	11.69512475
2700	4.842984	7.716159	9.8059606667	12.30326075

Table A.16: MySQL Complex Query 1.1 average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.215255	0.371047	0.499763	0.66309875
200	0.430895	0.7453935	1.0258483333	1.37526975
300	0.656353	1.1197965	1.501206	2.048496
400	0.859615	1.510125	2.0908353333	2.70110475
500	1.074603	1.873236	2.5834286667	3.464211
1000	3.941992	3.720005	5.2496626667	7.0068985
1500	5.948298	5.7065595	8.0305213333	10.6356305
2000	7.983516	7.5759075	10.56764	14.06899475
2100	8.418224	8.02625	11.1193646667	14.82273875
2300	9.298858	8.6862925	12.1055416667	16.15904625
2500	9.863524	9.4254715	13.1997393333	17.63854275
2700	8.742225	10.230226	14.0493186667	18.98359175

Table A.17: MySQL Complex Query 1.1 average time, 3 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.055537	0.0572815	0.0690813333	0.0858525
200	0.11103	0.110718	0.14994	0.17730175
300	0.166791	0.165181	0.211721	0.2777825
400	0.220731	0.203935	0.2776436667	0.35872275
500	0.276008	0.264407	0.372759	0.47084525
1000	0.551619	0.5430735	0.7549166667	0.92998875
1500	0.831501	0.836285	1.106133	1.4046835
2000	1.102743	1.107942	1.5475006667	1.8397505
2100	1.165011	1.1026745	1.555873	1.98983375
2300	1.276238	1.2697725	1.8446083333	2.220829
2500	1.387224	1.369872	1.832457	2.42741525
2700	1.503084	1.435536	2.2262636667	2.52060425

Table A.18: MySQL Complex Query 1.2 average time, 1 node

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.061907	0.0992425	0.101485	0.114072
200	0.117072	0.197654	0.1914056667	0.24000475
300	0.170309	0.277538	0.3270316667	0.35406725
400	0.210309	0.35522	0.3972126667	0.49794175
500	0.291974	0.4722085	0.5123743333	0.548602
1000	0.566497	0.876022	1.083308	1.25298625
1500	0.755442	1.394447	1.697056	1.92558375
2000	1.009866	1.951561	2.215095	2.84342125
2100	1.054684	2.03778	2.1360686667	2.85248025
2300	1.163886	2.1287765	2.304941	2.954352
2500	1.267552	2.378801	2.5799203333	3.57576575
2700	1.367243	2.5910825	3.047513	3.88281075

Table A.19: MySQL Complex Query 1.2 average time, 2 nodes

# of Queries	1 thread	2 threads	3 threads	4 threads
100	0.069579	0.121272	0.172381	0.21339725
200	0.152236	0.2436895	0.3343893333	0.41959675
300	0.216267	0.3746485	0.4850456667	0.669093
400	0.284336	0.5075365	0.635755	0.9839555
500	0.352012	0.6592525	0.7719553333	1.14591575
1000	1.23672	1.259257	1.657809	2.284312
1500	1.821226	1.847769	2.4361673333	3.422406
2000	2.418525	2.502922	3.280702	4.474208
2100	2.662663	2.7220145	3.6390203333	4.700319
2300	3.091954	2.8501815	3.95598	5.300435
2500	3.023243	3.1110105	4.3960646667	5.61990825
2700	2.76028	3.3580175	4.8631396667	6.2929215

Table A.20: MySQL Complex Query 1.2 average time, 3 nodes

Table A.21: INSERT and DELETE

# of commands	MySQL INSERT	MongoDB INSERT	MySQL DELETE	MongoDB DELETE
100	0.152528	0.050925	0.163032	0.192994
500	0.756668	0.277134	0.813595	0.857293
1000	1.547805	0.543094	1.729822	1.794074
5000	7.978989	2.832491	8.52242	10.288447
10000	15.935719	5.731883	17.499579	21.519041
50000	82.883779	32.321015	90.492908	114.414941

Appendix B

Configuration and Scripts

B.1 MongoDB

Create configuration directories for each server. Here the directory /data will contain the configuration directories for each server. Run the mongod replica servers:

```
    /MONGODB_DIRECTORY/mongod --dbpath ~/data/configdb1
    --port 27017 --replSet server1/
./mongod --dbpath ~/data/configdb2 --port 27018
--replSet server1/
```

Run arbiter server:

```
./mongod --dbpath ~/data/arbiter --port 27019 --replSet server1/
```

Run mongo shell and initiate configuration:

```
$ ./mongo
MongoDB shell version: 2.2.3
connecting to: test
> rs.initiate({"_id" : "server1" , "members" : [
... {"_id" : 0, "host" : "compute-0-1:27017"},
... {"_id" : 1, "host" : "compute-0-2:27018"},
... {"_id" : 2, "host" : "compute-0-1:27019",
... "arbiterOnly" : "true"}]})
```

All servers, including arbiter, need to use the same hostname terminology, ie, compute-0-1:27017 compute-0-2:27018 or localhost:27017 localhost:27018 Check the set from the mongo shell

```
server1:STARTUP2> db.isMaster()
{
        "setName" : "server1",
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "compute-0-1:27017",
                "compute-0-2:27018"
        ],
        "arbiters" : [
                "compute-0-1:27019"
        ],
        "primary" : "compute-0-1:27017",
        "me" : "compute-0-1:27017",
        "maxBsonObjectSize" : 16777216,
        "localTime" : ISODate("2013-07-07T05:20:19.308Z"),
        "ok" : 1
}
```

B.2 MySQL Cluster

To setup MySQL Cluster:

- Download the generic version from http://www.mysql.com/downloads/cluster/using wget on one of the compute nodes.
- Follow the quickstart guide provided http://downloads.mysql.com/tutorials/cluster/mysql_wp_cluster_ available on this page as an attachment), but use the following configuration:
  - For the Management Node (ndb_mgmd) - set the hostname to the hostname of the node which will manage the cluster. In this case compute-0-1 was used.
  - For the other 2 nodes (ndbd) the appropriate hostnames must be used which should be different than the Mananagement Node's. In this case compute-0-2 and compute-0-3.
  - The step of initialising the nodes must be executed on the nodes themselves and instead of localhost the hostname of the Management Node must be used, ie. user@compute-0-3]HOME/mysqlc/bin/ndbd -c compute-0-1:1186
  - The status check of the cluster should again be done from the Management Node as well as the initialisation of the MySQL server.

Note: The version of MySQL Cluster used is 7.2.10 64bits.

B.3 Harness

B.3.1 Building

To compile the benchmark first the necessary MySQL and MongoDB drivers need to be installed which can be retrieved from the following websites:

http://www.mysql.com/products/connector/ https://github.com/mongodb/mongo-c-driver

Each driver contains instructions on how to build it. After building the drivers the header files from each driver's include directory need to be copied in the system's include directory, usually /usr/include. The library files which are compiled also need to be copied to the corresponding system library directory, usually /usr/lib. Next as root run:

depmod -a

To compile the application inside the source code directory simply run:

make

B.3.2 RUNNING

To run and see the available options:

./harness -?

Available options are:

```
-o,--output      File to write output to
-i,--iterations  Number of queries to perform
                   for each test, 500-1000000

-t,--testIterations      Number of tests, 1-100
-r,--random 0|1  Use random strings for database
                      populating
-d,--database    Database to perform test on,
                  0=MySQL, 1=MongoDB, 2-Both

-m,--hostsql     MySQL database hostname
-n,--hostno      MongoDB database hostname
-s,--sqlport         MySQL database port
-q,--mongoport   MongoDB database port
```

B.4 Parallel Shell Script

Listing B.1: Parallel execution script

```
#!/bin/bash
cd $HOME/tests
```

```
#Number of threads to run on
JOBS=4
#This command should be run in the command line and not in
#parallel to create the database
#./harness -d 0 -r 1 -t 1 -x 10000 -i 500 -p 1

eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 100 -p 0 -o /data/sql/multnodes/test1-1-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 200 -p 0 -o /data/sql/multnodes/test1-2-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 300 -p 0 -o /data/sql/multnodes/test1-3-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 400 -p 0 -o /data/sql/multnodes/test1-4-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 500 -p 0 -o /data/sql/multnodes/test2-1-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 1000 -p 0 -o /data/sql/multnodes/test2-2-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 1500 -p 0 -o /data/sql/multnodes/test2-3-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 2000 -p 0 -o /data/sql/multnodes/test2-4-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 2100 -p 0 -o /data/sql/multnodes/test3-1-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 2300 -p 0 -o /data/sql/multnodes/test3-2-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 2500 -p 0 -o /data/sql/multnodes/test3-3-{}.out'
eval 'for i in $(eval echo "{1..$JOBS}"); do echo $i; done |
 parallel -q -j "$JOBS" ./harness -d 0 -r 1 -t 5 -x 10000
 -i 2700 -p 0 -o /data/sql/multnodes/test3-4-{}.out'
```

Bibliography

[1] Berg K., Seymour T., and Coel R. History of Databases. *International Journal of Management and Information Services*, 17, 2013.

[2] MongoDB. MongoDB Manual. `http://docs.mongodb.org/manual/`. Accessed: 15-08-2013.

[3] Smith G., Robert T., and Browne C. Tuning Your PostegreSQL Server. `http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`. Accessed: 14-08-2013.

[4] Apache. Cassandra Wiki: Getting Started. `http://wiki.apache.org/cassandra/GettingStarted`. Accessed: 14-08-2013.

[5] Apache. Apache CouchDB Manual. `http://docs.couchdb.org/en/latest/`. Accessed: 14-08-2013.

[6] Beynon-Davies P. *Database Systems*. Palgrave Macmillan, 3rd edition.

[7] Strozzi C. NoSQL: a Non-SQL RDBMS. `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page`.

[8] Strauch C. NoSQL Databases.

[9] Batra S. and Tyagi C. Comparative Analysis of Relational And Graph Databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2), May 2012. ISSN: 2231 2307.

[10] Kemme B., Jimenez-Peris R., and Patino-Martinez M. Database Replication. *Synthesis Lectures on Data Management*, page 17, 2010. doi:10.2200/S00296ED1V01Y201008DTM007.

[11] Oracle Corporation. Differences between the NDB and InnoDB Storage Engines. `http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-ndb-innodb-engines.html`. Accessed: 14-08-2013.

[12] EPCC. EDIM1: Data Intensive Machine. `http://www.epcc.ed.ac.uk/facilities/other-facilities/edim1-data-intensive-machine`. Accessed: 14-08-2013.

[13] Rocks Group. Rocks: Open Source Toolkit Real and Virtual Clusters. `http://www.rocksclusters.org/wordpress/`. Accessed: 13-08-2013.

[14] Free Software Foundation. GNU parallel. `http://www.gnu.org/software/parallel/`. Accessed: 13-08-2013.

[15] Oracle Corporation. MySQL Manual: The Slow Query Log. `http://dev.mysql.com/doc/refman/5.1/en/slow-query-log.html`. Accessed: 14-08-2013.