A COMPARISON BETWEEN MONGODB & COUCHDB ON SEARCH PERFORMANCE

A Comparative Analysis

Bachelor Degree Project in Webutveckling – Webbprogrammering IT606G
Spring Term 2018

Mathias Kinnander

Supervisor: Yacine Atif
Examiner: Henrik Gustavsson
Abstract

When storing and handling Big Data sets a database management system (DBMS) can be implemented to administrate and query databases. The Swedish Internet is a big unstructured data set that contains all published Swedish Websites since the late 1990’s. NoSQL DBMSs such as MongoDB and CouchDB are particularly suited to store the Swedish Internet. Comparing the search performance of MongoDB and CouchDB in this scenario required the insertion of a subset of the Swedish Internet, querying of the data and measuring the search performance. The results show that CouchDB has in general a superior performance but comes with a drawback which is its indexation time. If a query will only be executed a few amount of times MongoDB is generally the better choice. Further studies needs to be conducted in order to assess the performance of NoSQL DBMSs over the whole dataset.

Keywords: NoSQL, MongoDB, CouchDB, Search Performance, Data Archives
## Index

1. **Introduction** .............................................................................................................. 3

2. **Background** ............................................................................................................... 4
   2.1 Big Data .................................................................................................................. 4
   2.2 The Swedish Internet .............................................................................................. 4
   2.3 DBMS & Databases ................................................................................................. 5
      2.3.1 SQL databases................................................................................................. 5
      2.3.2 NoSQL databases ........................................................................................... 7
      2.3.3 NoSQL vs SQL databases ............................................................................. 7
      2.3.4 MongoDB ....................................................................................................... 10
      2.3.5 CouchDB ......................................................................................................... 10
   2.4 Query Performance ................................................................................................. 12
   2.5 Related Work .......................................................................................................... 13

3. **Problem** .................................................................................................................... 15
   3.1 Hypothesis ............................................................................................................... 16

4. **Methodology** ........................................................................................................... 17

5. **Pilot study - Empirical Measurement** .................................................................... 19
   5.1.1 Data insertion ..................................................................................................... 20
   5.1.2 MongoDB query ............................................................................................... 24
   5.1.3 MongoDB indexation ....................................................................................... 27
   5.1.4 CouchDB query ............................................................................................... 30
   5.1.5 CouchDB indexation ....................................................................................... 36
   5.2 Work Progression .................................................................................................... 38

6. **Experiment Setup, Results and Analysis** ................................................................. 39
   6.1 Experiment Description .......................................................................................... 39
      6.1.1 Hardware & Software ..................................................................................... 39
      6.1.2 Pilot study ....................................................................................................... 40
      6.1.3 Experiment 1 – Content type ......................................................................... 41
      6.1.4 Experiment 2 – Query Complexity ............................................................... 42
   6.2 Result analysis ......................................................................................................... 43
      6.2.1 Experiment 1 – Content type ......................................................................... 43
      6.2.2 Experiment 2 – Query Complexity ............................................................... 45
      6.2.3 Evaluation conclusions ................................................................................. 46

7. **Concluding Remarks** ............................................................................................... 48
   7.1 Discussion ............................................................................................................... 48
   7.2 Conclusion .............................................................................................................. 49
   7.3 Future work ............................................................................................................ 50

8. **Research Ethics** ....................................................................................................... 51

References .......................................................................................................................... 52
1 Introduction

The usage of Big Data has seen a rapid growth in popularity since the beginning of the new millennium. Massive amounts of data are consecutively stored into distributed sets across the whole world. The information contained within these data sets hides an immense amount possible value which if used correctly and if unused would be a great loss of valuable information. Extracting data from Big Data efficiently and creating analytics of these data which then can act as a foundation to base decisions upon is required in order to evaluate the performance of each database management system (DBMS).

In order to use data it should be obtained from a database. This would allow the data to be queried via a DBMS. The data must however be stored in such a way that it would make querying possible without losing information in the process. Depending on what type of data needs to be stored DBMSs are suitable. However selecting a DBMS for the handling of Big Data may be a daunting task considering adequate performance implications.

Measuring the search time for each query on the DBMSs yields results that may be indicative of how particular DBMSs perform. In order to increase the precision of the performance measurements the queries should be repeated multiple times in order to establish a reliable search time average. This average would be a relevant indicator of the DBMSs’ performance. Automatization of this querying process would reduce the possibility of human interference in the study which in turn would produce more accurate measurement results. In an effort to further minimize interference and favouritism, different types of queries targeting specific documents within the database should also be implemented. The DBMSs could possibly perform differently depending on the particular query and query type being evaluated. The study concerns itself with the search performance time of the DBMSs and as such focuses on the exact measurements for each of the DBMSs and disregards other possible factors e.g., an implementation with a web application in addition to the DBMSs or the effectiveness of the DBMSs insertion performance.

When the experiment is conducted the average search time, the number of returned documents and the query type should be extracted. The results needs to be averaged out and compiled in a format that allows for relevant analysis and an objective comparison of the DBMSs. Grouping the averaged results by different categories e.g., most amount of returned results, least amount of returned results, longest search time and respectively shortest search time, could arguably make it easier to highlight potential patterns characterizing the DBMSs. Despite the stringent focus on the search performance time other aspects that would be detrimental to an implementation of either DBMS should be taken into consideration. This is in order to convey an accurate depiction of the DBMSs’ performance, strengths and weaknesses.
2 Background

2.1 Big Data

Big data is a term that has been in use since the 1990s and describes data sets that comprises vast collections of revolving data. The data that is contained within data sets is in the majority of cases unstructured, although the data can be semi-structured. The type of data may vary between text, images, video or audio. In general, the amount of data tends to grow exponentially and rapidly as organizations and services tend to generate data using increasingly sophisticated data collection technologies as described by Benjamins, V. R. (2014). Exploiting big data generates analytics which could be used to eventually support some predictions that could aid some future business decisions. Our research focuses on Web data used in the Swedish Internet repository to understand some prevalent trends and evolution of standards used across Swedish web documents.

2.2 The Swedish Internet

The Swedish Internet is a collection of big data sets that encompasses all the Swedish websites that have been published and registered in Swedish domains since the late 1990's. The collection of data has been executed on the request of the Swedish government. An enormous amount of data has been collected throughout the years and stored in an increasing amount of hard drives. In order to utilize the collected data, it needs first to be migrated into a DBMS which in turn can be used to supply information to client side applications and provide the relevant data. Several options are available when it comes to choosing an appropriate DBMS to implement a migration solution.

The Swedish Internet contains a vast amount of data that can be analyzed for research purposes as well as for decision-support purposes. In order to extract the desired information, appropriate queries are employed. Queries such as what the average content length for Swedish websites during 2010 or the most popular image format between the years 2002 – 2012 etc, could reveal pertinent information. A similar study has been made on several Swedish municipality document repositories by Lundell, B. and Lings, B. (2010). Their goal was to analyze document formats that have originated from 1999 and 2008 in order to establish the most frequently used document-format type. This is somewhat similar to the previous query example mentioned as a part of our study. In order to extract information and later use the data set, its structure needs to be stored in such a way that would enable queries to retrieve relevant data.
2.3 DBMS & Databases

A database management system or DBMS is as it implies a system that manages a database. That means the DBMS retrieves information from the database. The database is the where the data itself is stored. The query requests is issued by the user e.g. a human, a program or a machine. Figure 1 shows an example of interactions between DBMS and different actors.

![DBMS interactions diagram]

**Figur 1**  DBMS interactions

2.3.1 SQL databases

Different DBMS for different types of databases exists. The popular type of database is the relational database for which virtually all corresponding DBMSs use the SQL (Structured Query Language) language. These databases employ tables that represent relations represented by rows and columns. A column contains the attributes of the relation shared by all objects e.g. a color, a brand, a name etc. A row contain the *tuple* or *record* which is an instance of the database object as described further by Codd, E. F. (1970). An illustration of an SQL database table is shown in Figure 2.

![SQL table illustration]

**Figur 2**  SQL table
When querying a SQL-based DBMS server specific requirements may be implemented to show only relevant data about one or some particular entities where all the specifications are matched. A demonstration of SQL query can be shown in Figure 3, where all attributes from entities in the table CAR with the color attributed red is selected.

```
SELECT * 
FROM CAR  
WHERE CAR.COLOR = 'RED';
```

**Figure 3**  First SQL select query

The result of this query would list all database object instances for Car entity where the color attribute is equal to red. In Figure 4 there is a demonstration of what the result of the query would look like.

<table>
<thead>
<tr>
<th>REG</th>
<th>BRAND</th>
<th>COLOR</th>
<th>mYEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>15195401</td>
<td>FORD</td>
<td>RED</td>
<td>2003</td>
</tr>
<tr>
<td>231564109</td>
<td>VOLVO</td>
<td>RED</td>
<td>2012</td>
</tr>
<tr>
<td>63532231</td>
<td>FORD</td>
<td>RED</td>
<td>2001</td>
</tr>
<tr>
<td>81253912</td>
<td>FERRARI</td>
<td>RED</td>
<td>2009</td>
</tr>
</tbody>
</table>

**Figure 4**  SQL multiple result

If the brand is relevant for the search query it can be added to selected attributes from the entity CAR where the color is red as demonstrated in Figure 5.

```
SELECT CAR.BRAND 
FROM CAR  
WHERE CAR.COLOR = 'RED';
```

**Figure 5**  Second SQL select query

The possible results of the search query in Figure 5 is illustrated in Figure 6. Only brand instances are listed in Figure 6.

<table>
<thead>
<tr>
<th>BRAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORD</td>
</tr>
<tr>
<td>VOLVO</td>
</tr>
<tr>
<td>FORD</td>
</tr>
<tr>
<td>FERRARI</td>
</tr>
</tbody>
</table>

**Figure 6**  SQL single result

However some types of data is not regular and may not fit easily into a structured table format. Irregular data is where the structure of a particular file or document varies, which is a problem for SQL databases and SQL DBMSs. They are generally designed to employ predetermined templates to structure their entities. If attributes are not correctly inserted into the database it can quickly create a vast amount of NULL values which in turn can corrupt the database and reduce query performance as mentioned by Mirza, G. A. (2015). For such irregular data
NoSQL databases and corresponding DBMS are more suitable to handle the overhead induced by possible NULL values in a SQL database.

2.3.2 NoSQL databases

NoSQL is defined as a blanket term that stands for non-relational database or more recently not only relational database that incorporates several different DBMSs as mentioned by Leavitt, N. (2010). NoSQL databases have been around since the 1960’s (which means even before SQL databases) but have dramatically increased since the beginning of the year 2000’s. One of the reasons of this increase in popularity is the fact that companies such as Facebook, Google, Ebay etc. raised the need to store massive amounts of unstructured data with the introduction of Web 2.0 as argued by Mohan, C. (2013).

When it comes to choosing a NoSQL DBMS there are several options to choose from. Depending on what type of data that is going to be dealt with, different types of NoSQL DBMS are available. Some types of these DBMS include document, column, key-value or graph-oriented DBMS. Unlike SQL databases and DBMS, NoSQL counterparts do not necessarily share a common language or structure. This highlights the fact that some NoSQL DBMS types are objectively better at dealing with some types of unstructured data than other alternatives.

2.3.3 NoSQL vs SQL databases

Why is it that organizations and people use NoSQL DBMS when there exists well-established relational DBMS using SQL? One reason why NoSQL options get chosen over SQL alternatives is that unlike relational database management system (RDBMS), NoSQL primarily does not deal with structured query language. Since we are dealing with webpages structure in our research, these data tend to include different structures and different types of data e.g. texts, pictures, videos, links etc. with diverse instances of stored data content. Another given distinction is:

“[...] NoSQL databases break with tradition in their abandonment of the relational model. To be fair, some data truly does not naturally fit the relational model. This could be because the data changes form or size often, or because the data is completely unstructured”. Bartholomew, D. (2010)

One example of when a NoSQL database would be preferable would be when different objects have different attributes than other objects that are stored within the same database table. For example, let us imagine that the database needs to store data about websites which contains images, audio, text and links. All instances objects have different attributes that may not necessarily share the other entities e.g., one websites is a photo gallery whilst another website only stores text. If the websites were stored within a SQL database there would be several instances of NULL values scattered through the tables for all the websites that do not include all of the available website elements.

In order to use an SQL database efficiently in our research context, the different websites need to be stored in their own table. There are still however some possible problems that can arise if there are further differences between quantities and type of attributes that need to store the related information about an entity. e.g., a website can display several image types and another website display another number of image types. This is illustrated in Figure 7.
The above illustration highlights the problem for SQL database in form of NULL values. In Figure 8 the result of the search query in Figure 7 and the occurrence of NULL value are shown.

<table>
<thead>
<tr>
<th>IDnr</th>
<th>URL</th>
<th>DownloadDate</th>
<th>JPG</th>
<th>PNG</th>
<th>DIVS</th>
<th>SCRIPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://www.example.se">www.example.se</a></td>
<td>2009-10-18</td>
<td>10</td>
<td>NULL</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.anotherexample.se">www.anotherexample.se</a></td>
<td>2016-05-10</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figur 7** SQL website entities

A NoSQL database however can store irregular data that does not require a limited predefined set of possible attributes for an entity but may rather be declared into entity itself depending on requirements as shown in Figure 9 where there is two websites are stored into separate documents that also have a different number of attributes.

```json
{id: ObjectId("5a8838a4dab4b32448c4c60e")
  url: "www.anotherexample.se"
  date: 2016-05-10 12:00:00.000
  JPG: 6
  PNG: 2
  div: 4
  script: 3
  Languages: Array
    0: "HTML"
    1: "CSS"
    2: "JAVASCRIPT"
    3: "PHP"
}
```

**Figur 8** SQL website query result

```json
{id: ObjectId("5a8849b4dab4b32448c60f")
  url: "www.example.se"
  date: 2009-10-18 15:14:10.000
  JPG: 10
  div: 4
  script: 2
  Languages: Array
    0: "HTML"
    1: "CSS"
    2: "JAVASCRIPT"
}
```

**Figur 9** NoSQL document website documents
With the adaptability of NoSQL databases they are often a clear choice over traditional SQL databases since scalability of the NoSQL databases is higher compared to the SQL alternatives. How come then that SQL databases tend to be a more frequent choice for companies and organizations? One of the reasons is that NoSQL in general are not ACID-compliant as motivated by Nayak, A et al (2013). If a DBMS is ACID-compliant it means that it has Atomicity, Consistency, Isolation and Durability. Atomicity means that if an operation must either be fully successful or fail as a cohesive unit. Consistency means that the DBMS enforces rules about how predefined integrity constraints should be structured. For example a telephone number should only contain numbers and no other symbols. Isolation means that one current transaction does not interfere another transaction that is being executed at the same time. Durability means that once a transaction has been executed and saved to the disk the data persist over any potential failure when it comes to hardware of software. When it comes to banking systems and money exchange functions it is vital that is ACID-compliant since any mistake may cause severe problems as outlined by Wright, C. P. et al. (2007) and LIU L. et al. (2009).

Another advantage that SQL DBMS and databases have is that they all share the SQL structured language. This means that if an organization wants to migrate from one SQL database to another there is in general no need for extensive adaptation for the data to suit the new DBMS. This may not be true for all NoSQL DBMS. It will depend on how the data has been structured and if there are some functions that is not supported by the new DBMS. When it comes to migrating between NoSQL DBMS it is more complex since all the majority of NoSQL DBMS do not share the same language and structure because they all are designed to work differently as stated by Bartholomew, D. (2010).

When it comes to deciding between implementing an SQL DBMS solution or a NoSQL DBMS one for our proposed Swedish Internet case study, the NoSQL option would be the most appropriate choice. The reason is that one webpage may differ from the vast majority of other web pages when it comes to content, structure, format and data etc. Implementing an SQL DBMS solution for storing web pages would be a complex and a difficult task to perform if it was required to store all the relevant information about the webpages.
2.3.4 MongoDB
In an article by Butgereit, L. (2016) MongoDB is described as a document-oriented NoSQL database program. The languages that MongoDB written in is C, C++ and JavaScript. As a document oriented DBMS MongoDB stores documents in binary JSON (BSON). However the size limit of BSON are 16MB which is sometimes not sufficient for some large files. For larger files MongoDB provides GridFS which is an alternative tool that can save files exceeding the BSON size limit of 16 MB as described in an article by MongoDB Manual 3.6 – GridFS (2017). Figure 10 demonstrates a theoretical MongoDB document.

Figur 10  A MongoDB doocument

Depending on what type of data that needs to be saved a BSON document can use a specific format that suits its particular needs. That in turn allows developers and programmers the ability to store data sets that contain diverse and complex data in one place rather than having different tables (as it used to be the case with SQL databases).

2.3.5 CouchDB
Figure 11 illustrates how a typical CouchDB document could be structured if a data regarding a website would be stored. The format that CouchDB possesses have a great ability to adapt itself to changes or irregularities in the data structure that would be inserted into the document.
2.4 Query Performance

Since NoSQL databases can store unstructured big data sets why not use them? By implementing the goal template created by Wohlin, C. et al (2012) the scope of the experiment is to analyze the chosen DBMSs for the purpose of measuring their difference with respect to query performance from the point of view of researchers and users in the context of implementing a DBMS for the Swedish Internet.

As previously mentioned there is a multitude of fields, useful applications and decision making scenarios that can benefit from analyzing big data sets. When organizations analyze these data sets, they presumably strive for the most optimal information retrieval performance in order to maximize efficiency and throughput. The response time between the database and the DBMS side differs depending on what type of DBMS, the type of query and the type of data that the database contains. Further aspects that must be considered when it comes to measuring the response time would be the hardware in addition to the amount of traffic between different users and the database as argued by Karniavoura, et al (2017).

One of the most pivotal factors that decide whether or not the required performance goal is achieved relates to the selection of the right DBMS to use. This depends on what type of data needs to be stored and for what purpose it is going to be used. One DBMS type could be detrimental to search efficiency while another type could greatly increase throughput and search performance.

When facing several viable DBMS options of one or several DBMS types the different options may be subjected to testing. The test should presumably cover aspects such as performance, scalability, fault tolerance, security etc. When the tests are complete one may be able to make a well supported decision for one particular DBMS instead of other alternatives. But how would a test measuring search performance look like? How would it be designed and implemented?
2.5 Related Work

We conducted a pilot study based on similar experiments in the Content Management Systems course at the University of Skövde. The selection of DBMS’s was inspired by Nayak, A., Poriya, A. and Poojary, D. (2013), which considers MongoDB and CouchDB performances. Although performance issues have been investigated at large in previous literature, the uniqueness of our work lies in the nature of the dataset used in our empirical study which consists in Swedish Internet content. Up to our knowledge, the investigation of query-processing for this type of large data repositories has not been explored. Our performance-driven work to host Swedish Internet Content into NoSQL databases provides a platform that could support researchers investigating the role of open-source in big data storage and computation.

The initial idea for this study was suggested by a teacher’s at the University of Skövde (Henrik Gustavsson), where the suggestion of incorporating Swedish Internet emerged following a previous study from University of Skövde’s research group called Software Systems Research Group (SSRG). The irregular structure of the dataset and the popularity in NoSQL DBMS promised an interesting study. Other significant influence has been Lundell, B. and Lings, B. (2010) who evaluated the usage of document standards in Swedish municipalities which motivated the end-goal purpose of the candidate queries used in this pilot study.

Similar comparisons between MongoDB and CouchDB have been conducted prior to this study. A study by Henricsson, R. (2011) compared the two DBMS’s performances using a Python interface. The setup of the experiment was also related as they implemented insertion scripts when setting up the databases so that the databases could be queried. Their data set contained documents regarding fictional user information such as age, salary, name etc. In our case, we consider an authentic dataset, with significant irregular structures. In the study they also implement temporary views which are as cited by Henricsson, R. (2011) “Temporary views are supposedly slower than permanent views”. They further go on to stating that “Permanent views would probably have increased CouchDB’s performance, but likely not to the degree of outmatching the performance of MongoDB”, contrasted to this our study only regards itself with permanent views since it is the intended implementation of CouchDB.

In the above-cited, work, one of the queries targeted all persons with a salary of between 18500 and 35000 whilst another query retrieved all persons with a salary of between 18500 and 35000, and with an age of over 25. For each query type they increased the complexity by adding more specifications on what would be returned and which attributes would have to coincide with the query. All queries used in their study were GET request where data is read from the database and disregards requests such as UPDATE, DELETE and POST. Our study also only deals with GET requests since the collection of data into the Swedish Internet only occurs at specific time intervals into file archives by scripts.
Furthermore, the Swedish Internet is supposed to store data and as such would presumably not require the same performance need when it comes to updating and deleting documents. As previously mentioned, this study only deals with the Swedish Internet contents and has as such queries regarding the file formats, dates, size and type of documents etc. The conclusion of their study was that MongoDB performance is better than CouchDB when it came to reading from the database with the employed queries that were mentioned above. The study showed that the divergence in GET performance between the DBMS’s increased exponentially as the number of documents increased. One thing to keep in mind is that they only implemented temporary views in their study and as mentioned earlier this study disregards temporary views and only employs permanent views. Therefore the results from both studies may differ quite significantly.

Another study by Bhardwaj, N. (2016) conducted similar experiments on MongoDB and CouchDB. For their performance measurements, they used Apache JMeter which is designed to load and test functional behavior and the measured DBMS’s performance. The scope of the datasets implemented in their study is tiny in comparison to our study with the number of user documents ranging between 10 – 100, and a sample size of 100. Our study deals with substantially larger data set with the pilot study implementing 14 418 documents and 888 164 documents of varying types and sizes in the main experiments. Unlike the conclusion of the previous study by Henricsson, R. (2011) the study by Bhardwaj, N. (2016) showed that CouchDB performed significantly better than MongoDB in when it comes to READ performance. Arguably this may be in due part to the amount of documents that were stored into the database targeted by the query. It could also be the case that CouchDB performs better when dealing with smaller sets of data. In order to find out if CouchDB performs worse than MongoDB when dealing with larger sets of data additional experiments need to be conducted. An example of a possible experiment would be to implement a large set of data as well as a small set of data then executing the same queries against the data. This would reveal the impact that the quantity of data has on the DBMS’s performance. The results from the testing would have to be measured and evaluated in the same manner which is not the case when comparing the two previously mentioned stories.
3 Problem

In order to support NoSQL DBMS selection for the Swedish Internet data collection, we compare DBMS and assess the results that the test yields. These results will then be analyzed and compiled into statistical measurements which will convey the difference between candidate DBMSs. In order to perform queries upon the Swedish Internet, the data needs to be available in a format that can be stored into a database which then in turn allows the DBMS to execute search queries. The format that the Swedish Internet is stored is a MIME message document format that does not allow the insertion of the data set directly into a DBMS in its current form. Before the insertion of the Swedish Internet into a contemporary database, the data sets have to be converted into a DBMS-compliant format. Thus, the conversion of data stored within the Swedish Internet needs to be accomplished so it can be processed as part of the main problem of project. The dataset has to be structured in such a way that it would contain the same data set for all candidate NoSQL DBMSs. MongoDB and CouchDB are document-oriented NoSQL DBMSs. Since there are two different DBMS types there will be extra specifications and requirements on how a data set has to be structured. It is important that all DBMSs receive the same prerequisites and the study be as standard as possible regarding the testing of the DBMSs as explained by Wohlin, C. et al (2012).

One viable option is to use an application that detects and extracts metadata. The metadata can then later be stored into the DBMS. In order to execute performance tests fairly, the structure of the data allow identical information to be inserted in to the different DBMSs. It may also be the case that depending on which DBMS is tested, the data set adapts to the format that is acceptable for the DBMS e.g., pictures, audio, text etc. Minimizing the differences between the structures and thus ensuring that the exact amount of data exists in all data sets. Then the data needs to be inserted into each of the DBMSs. Since the Swedish Internet has some of big dataset properties it would be inefficient to insert the data manually. Instead the process can be automated so that a greater amount of data can be inserted in a shorter amount of time. The automation of the process may also eliminate the possibility of frequent occurrence of human error. Once the data is stored in the correct format inside the DBMSs they are available for search queries. In order to produce search queries, an application such as a web application that could produce and specify search queries may be considered. There exists specific requirements on the database e.g., specific search queries, measuring search performance, storing measurements, displaying results etc.
Testing of the technical artifacts in the study should cover and reflect the possible real life usage of the implementations i.e., that the results of DBMSs from benchmarking should be representative of web applications as they are used in practice. A quotation from an article by Ratanaworabhan, P. et al (2010) is as follows:

“We conclude that the benchmarks are not representative of real applications in many ways. Focusing on benchmark performance may result in overspecialization for benchmark behavior that does not occur in practice, and in missing optimization opportunities that are present in the real applications but not present in the benchmarks.”

This quote highlights the importance of using benchmarks that are relevant for the experiment in order to avoid biased, irrelevant and erroneous testing that fails to cover all necessary aspects when implementing benchmarking in an experiment. Evaluating how statistical results could be interpreted may also be a potential problem since there exists several correct options on how to evaluate the test results but that in turn might yield slightly different interpretations. For example should eventual spikes in the measurements be ignored? A spike is a statistical abnormality that stands out from the other results e.g., a significant time delay on a particular response from a specific DBMS query. If spikes occur where the line should be drawn for a result to be counted as a spike? When spikes are removed it may ultimately affect the outcome of the tests.

The hardware specifications to use in the tests may also have a significant impact on how well the DBMSs compares against each other. One example could be the amount of available RAM and CPU cores that is available for processing the search queries. During the testing, extra precautions will be considered to minimize any preference towards one or some of the DBMS that exceed well on particular specifications whilst performs poorly on other through normalizing multiple tests across different hardware specifications.

3.1 Hypothesis

The hypothesis is that the search performance of MongoDB will be superior to that of CouchDB when it comes to querying and retrieving results from databases that contain the Swedish Internet and similar data sets.
4 Methodology

The Swedish Internet needs to be stored within a database in order for a DBMS to be able to retrieve information and study the subsequent performance results. The original data set from the Swedish Internet comes in MIME message documents that needs to be converted so that the content of each individual document can be stored within a database. This can be achieved by using the MIME message tool which is further documented at Github (2018) MIME messages for JavaScript. https://github.com/eface2face/mimemessage.js. Each MIME header or element type can then be inserted as an attribute type. Numerical values such as dates, content length etc will be stored as numerical values and attributes such as a URL, subtypes etc will be stored as strings. Large attributes such as the body attribute which stores the entirety of the content from the file will also be stored into one attribute or divided into several sub documents. It is important to use all of the data that is available in the study since the Swedish Internet’s purpose is to store the information regarding governmental computer files such as HTML-pages, png, docx, pdf, mp4 etc.

When the MIME headers has been converted into an acceptable format it can be inserted into the database. Converting all MIME headers from all the files and then inserting them manually would be an arduous process that would be inefficient and also subject to human error. The process needs to be automated to ensure the same standard are applied for all documents. This can be done by implementing a NodeJS script that reads from the files from the Swedish Internet and extracting information from their MIME headers. When all documents have been parsed through the NodeJS script the database is ready to be queried by the DBMS.

The reason why performance testing is conducted is to conclude if there exists query performance differences between the chosen DBMSs. The testing is conducted as a technical experiment. Technical experiments allows exertion of control in order to manipulate the testing in a direct, precise and systematically way as described by Wohlin, C. et al (2012). Since the study is focused on the technical aspects of DBMS performances a technical-oriented experiment is suited for a comparison of the difference in query performance. If the study were conducted using a human-oriented experiment it would entail human bias and unpredictability. Test-subjects opinions and previous experiences regarding a particular DBMS has no bearing on how it actually performs in comparison to the other candidates. A case study might also be a viable alternative method since the study aims to determine which DBMS is the most query efficient candidate in practice of the chosen alternatives. The main reason why a technical-oriented experiment was implemented instead of a case study is the lack of control originates from the unpredictability of the human factor which is present in case studies. An example of this would be that one user might generate more efficient queries than other users and therefore might affect the outcome of the results depending on the knowledge and expertise of the individual user as motivated by Wohlin, C. et al (2012).
Without queries there would not be any generation of performance data. When a query is created, the NodeJS script proceeds to send a search query to the DBMS which then searches through the set of data and sends back the results. The results in turn can later be used to conduct a statistical analysis. Only the response time of the DBMSs will be measured and other aspects such as loading time will be disregarded from the measurements. The response time from a DBMS is the time it took for the DBMS to search through the database and retrieve the correct result. The reason why the experiment will not include any web application or program in conjunction with the DBMSs are that it favors one particular web browser or application types as described by Ratanaworabhan, P. et al (2010). It may also be the case that the web application might bottleneck the performance of a particular DBMS and therefore influencing the end result. Figure 12 demonstrates what is aspect is being measured in the study.

![Diagram](image.png)

**Figure 12** Demonstration of benchmarking

The benchmarking results will be stored into a document on a server other than the server that the search query was made on. The reason why the results should be saved onto another server is because it might affect the performance of the server if more memory is dedicated to store the new test results. Another factor regarding the minimization of interference with our experimental study, is how the setup of the DBMS looks like as the testing might be subject to certain differences in experiment disturbance. If the connection between the client and the DBMS is remote and shares the connection with other entities such as users and applications on a local network, it might create interferences that corrupt the measurements which then in turn influences the testing process, and the final results. Since the goal is to have as little interferences as possible, one option might be to execute the test locally on one or several machines, so that as few influencing-factors are involved as possible.

If testing is not feasible on a private network or a direct link between the client and the DBMS, the testing will have to be conducted over public or uncontrollable connections. The testing should be done over a period of several days and during different time-periods in order to try to minimize the interference effects on the experiments from external sources as motivated by Wohlin, C. et al (2012).
5 Pilot study - Empirical Measurement

This pilot study aims to compare the performance of the chosen DBMS while executing different queries. The pilot study implements one of these queries on the data that were extracted from the Swedish Internet. From the execution of the queries, time measurements of the response time and the returned results of the query itself will be collected. From these results, statistics are analyzed and processed with the intent of establishing differences in search performance between the candidate DBMSs. For each DBMS, there are different ways to execute and implement the queries, ensuring that each DBMS performance is investigated with unbiased considerations is vital. The implementation of the pilot study can be summarized in the following steps:

- Setting-up the databases
- Extracting and inserting the data
- Indexation
- Executing the queries
- Evaluating the results

With the help of the pilot study and the code provided in the appendix the experiments should be able to be reproduced independently and subsequently yielding similar results to the ones obtained in this study. The data subset from the Swedish Internet may be unavailable and as such a similar data set could be used as a substitute in a future study.
5.1.1 Data insertion

In order to measure the performance of the chosen DBMS, a database with data from the Swedish Internet contents should be developed and as such allowing relevant queries to be executed. Inserting the data manually into the databases would be a long and tedious process, therefore automating the insertion is a more efficient option. This can be achieved using a NodeJS script that scans the files and inserts them into each database as a document. Reading MIME-structured messages is achieved by using the mimemessage.js tool found at the guide Github (2018) MIME messages for JavaScript. https://github.com/eface2face/mimemessage.js.

The location and specifics of the databases need to be declared within the script so that it knows where to insert the documents. For each database type, there are slight variations in the code requirements. For MongoDB, node-mongodb-native respectively Github (2018). MongoDB Native NodeJS Driver. https://github.com/mongodb/node-mongodb-native. and for CouchDB couchdb-nano was implemented Github (2018) Apache CouchDB Nano. https://github.com/apache/couchdb-nano. Figure 13 demonstrates the differences of the requirements for each database type.

```javascript
//************** MongoDB initialization **************/
var MongoClient = require('mongodb').MongoClient;
var Mongo_url = "mongodb://localhost:27017/";

//************** CouchDB initialization **************/
var nano = require('nano')('http://localhost:5984');
var dbcouch = nano.db.use('dbo');
```

**Figure 13** DBMS initialization

Storing all unpackaged raw files in one folder enables the script to loop through the folder scanning each document as it goes along. Figure 14 shows how the script loops through the folder that contains the documents from the Swedish Internet.
Each of the documents contains MIME-headers that describe what attributes it has i.e, the type, the size, the URL of the file etc. Depending on how the MIME format was structured in the particular file the MIME header defining which type of file it is might vary in position and subsequently the reading of that particular attribute needs to be dynamic so that the correct file type can be inserted successfully. The solution for the variation of type indication in MIME messages is demonstrated in Figure 15.

```javascript
var filetype;
var filecontent;
var http_headers;
for (var k in msg.body) {
    if (msg.body[k].header('HTTP-part') == 'Content') {
        filetype = msg.body[k].contentType()[]'subtype'];
        //filecontent = msg.body[k].body;
    } else if (msg.body[k].header('HTTP-part') == 'Header') {
        http_headers = msg.body[k].body;
    }
}
```

**Figure 15** Irregular MIME-header function

One of the attributes that are interesting are the dates when the file was extracted from the Swedish Internet. The date attribute in the MIME-header is stored in Epoch time. Storing the date in Epoch time is useful, but in contrast for human-beings it is tricky to implement in practice. For the sake of convenience the date can be converted into a date format using the Gregorian calendar which is internationally the most widely used civil calendar. Figure 16 demonstrates the conversion of the date attribute into the Gregorian calendar.
/* Converting the epoch time into dates */
var utcSeconds = msg.header('HTTP-Archive-Time');
var d = new Date(0); // The 0 there is the key, which sets the date to the epoch
d.setUTCSeconds(utcSeconds);

**Figure 16**  Epoch time conversion function

When the file has been read through the content every attribute is stored within the myobj-variable which in turn is then used to insert the files into the database itself. The myobj variable also contains the newly converted date and the correct content type attribute. Figure 17 illustrates the entirety of the myobj variable.

```javascript
var myobj = {
  //First Header
  MIME_Version: msg.header('MIME_Version'),
  Type: msg.contentType()['type'],
  Subtype: msg.contentType()['subtype'],
  Params: msg.contentType()['params'],
  HTTP_Part: msg.header('HTTP-part'),
  HTTP_Collection: msg.header('HTTP-Collection'),
  HTTP_Harvester: msg.header('HTTP-Harvester'),
  HTTP_Header_Length: msg.header('HTTP-Header-Length'),
  HTTP_Header_Md5: msg.header('HTTP-Header-MD5'),
  HTTP_Content_Length: msg.header('HTTP-Content-Length'),
  HTTP_Content_Md5: msg.header('HTTP-Content-MD5'),
  HTTP_URL: msg.header('HTTP-URL'),
  HTTP_Archive_Time: d,
  HTTP_Headers: http_headers,

  //Second section: file
  Filetype: filetype,
  Filecontent: filecontent,
};

**Figure 17**  The myobj variable

When the myobj variable has been created, the script can proceed to insertion. First the files get inserted into MongoDB. One important thing when it comes to inserting documents into MongoDB is that a connection needs to be established prior to the insertion of a new document. The connection needs to be established every time that a new document is going to be inserted which is in this case is “mongodb://localhost:27017/”. The specifics of the connection were defined earlier in the script so that the insertion of the documents are sent correctly as demonstrated in Figure 18.
If the connection is not closed after the successful insertion, the amount of connections will stack until there is no available memory left on the computer. This will result in a critical failure that disrupts further insertion of new documents. The insertion of documents into CouchDB requires less code than MongoDB with the following line “`await dbcouch.insert(myobj);`”. Await is a newly introduced JavaScript mechanism that helps synchronize function calls. We use it in order to make sure database calls are executed immediately and do not stack up until the program crashes. The document insertion function for CouchDB can be seen in Figure 19.

```
await dbcouch.insert(myobj);
```

**Figure 19** CouchDB data insertion function

In order to execute the script, it can be run in the command prompt by navigating to the location of the script, then using the node command and entering the script’s file name. The script will then proceed to run through the entire folder containing the raw files. When the last file has been processed, the script is finished.
5.1.2 MongoDB query

The MongoDB query is executed in the same way that the insertion of data is conducted i.e., running a NodeJS script using the MongoDB terminal. Prior to executing the query the user must be at correct location for the script to be executed and have the MongoDB server activated. This guide at W3schools.com - Node.js MongoDB Get Started (2018). https://www.w3schools.com/nodejs/nodejs_mongodb.asp. which goes through the steps for the setup of the MongoDB server.

The mongo driver is required in order to connect and use the mongo database within the NodeJS environment. The request library is used to make web requests to the PHP script that stores the query search time results into a text file on the remote server. The time variables define the starting year and the end year implemented in the queries and the loop function. The start year begins at 1997 and will eventually be incremented by one year each time the same query has been executed 100 times. When the start year is equal to the end year which is 2017 the script will stop. By querying each year instead of the entire span between 1997 until 2017 we can divide each year into groups. The dbname variable specifies which database that is going to be used in the script. Since there might be several MongoDB databases running at the same time it is required to specify which database or databases are going to be used by the script. The specific requirements for the script so that it will run correctly is demonstrated in Figure 20.

```javascript
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = "";

// Time variables
var currentYear = 1997;
var endYear = 2017;

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';
```

**Figure 20** MongoDB variables

When the script is executed the DBMS will begin to search through the database looking for documents that are valid for that particular query. The script used within the pilot study contains a query that looks for the number of HTML documents within a span between 1997 until 2017. The query groups the results by querying each time for every New Year. This will automatically group the results by year rather than having a large sample of unordered results. In this study however the execution time of the queries are saved and not the results of the queries themselves. When executing a MongoDB query further specifications may be added to the query so that it returns the output as a specific value. The structure of a MongoDB query is demonstrated in Figure 21.
Since the study aims to compare the difference in search time performance between DBMSs, the attribute that is pivotal is the time that it took for the query to be executed. This can be done by adding “explain.executionStats.executionTimeMillis” to the end of the MongoDB query. The executionTimeMillis specification returns the time it took for the query to be executed in milliseconds. In Figure 22 the difference between an “ordinary” query which would return valid results and a query containing an executionTimeMillis specification that would return the execution time.

```javascript
// Ordinary query
db.mongoarticles.find(
    {
        Filetype: 'html',
        HTTP_Archive_Time: {
            $gte: ISODate("1997-01-01T00:00:00.000Z"),
            $lt: ISODate("1997-12-31T00:00:00.000Z")},
        {_id: 1}
    })

// Query with explain performance specification
db.mongoarticles.find(
    {
        Filetype: 'html',
        HTTP_Archive_Time: {
            $gte: ISODate("1997-01-01T00:00:00.000Z"),
            $lt: ISODate("1997-12-31T00:00:00.000Z")},
        {_id: 1}
    }).explain.executionStats.executionTimeMillis
```

**Figure 22** MongoDB queries
Each variation of the query is executed 100 times in a row so that a larger set of data is generated. This is in order to calculate the average result time of each particular query so that the results is more consistent to the actual performance of the DBMS and not distorted by any abnormality. The execution time results is also be stored into an array that will be divided by 100 when the same query has been executed a total amount of 100 times. The result of the ordinary query that do not contain the execution time in milliseconds will also be stored in to the document once the query has been executed 100 times and this will help to correlate the performance to the actual results. Figure 23 demonstrates how the results are generated and stored within the text file.

![MongoDB event chain](image)

**Figure 23** MongoDB event chain

Saving the results is done by using the request tool which can be found at Github (2018). Simplified HTTP request client https://github.com/request/request, which makes it possible to execute HTTP requests. The location of the text file is also specified in the URL attribute which in example lists the absolute path. The content of that which are going to be stored within the text file is defined within the body attribute. The body attribute stores the averaged response time for a particular query as well as the individual measurements of each instance of returned time result from the executed queries. The result of the queries are also stored within the body attribute. Figure 24 demonstrates the code that stores the result into the text file.
When the data has been stored correctly the currentYear variable gets incremented by one in order generate a new query variation with a new startdate and a new end date. The script will repeat this process a total amount of 2000 times including the first instance and then finally stop. The text files will then contain 20 averaged time performance results, 2000 performance results and 20 results of the returned values.

5.1.3 MongoDB indexation

In order to increase the performance of MongoDB indexation can be used. An implementation of indexation means that MongoDB do not need to scan every document in the database in order to retrieve the result but can rather use the index to find the relevant data the query is looking for as described in Github (2018) Indexes – MongoDB Manual. https://docs.mongodb.com/manual/indexes. There are several types of MongoDB indexes. The index types implemented in this study is single field indexes and compound key indexes.

A single key index sorts the data in an ascending or a descending order based on the index key. In MongoDB there is no impact on performance if the data is stored in a ascending or a descending order and will still have the same level of performance. A demonstration of how a single key indexation works can be seen in Figure 25 were documents are sorted in an ascending order based on the value of the attribute “Score”.

Creating a single index requires the choosing of an attribute which the index will be sorted after. The correct selection of a correct attribute is important since if an unimportant attribute is chosen the index will not be used and therefor redundant. In order to create the index as shown previously in Figure 25 the following code needs to be entered into the mongo shell as demonstrated in the following Figure 26.

```javascript
db.collection.createIndex({
    Score: 1
})
```

Figur 24  MongoDB post data to text file

Figur 25  Single key index example

Figur 26  Single key index creation

A compound key index is similar to single key index since it includes two attributes rather than just one attribute. The two attributes are combined and sorted together. A demonstration of a compound index which is sorted ascendingly primarily by the name attribute and secondarily sorted ascendingly by the score attribute can be seen in Figure 27.

![Compound key index example](image)

**Figure 27**  Compound key index example

Similar to a single key index the selection of the attributes that would be included in the index is important. If the one or both of attributes are irrelevant for the querying the performance gain from the indexation will affect the performance. The code for creating a compound index can be seen in Figure 28 that builds upon the example in Figure 27. Further documentation regarding compound indexation can be found at Mongodb.com (2018) Compound Indexes – MongoDB Manual. https://docs.mongodb.com/manual/core/index-compound.

```javascript
db.collection.createIndex({
    Score: 1,
    Name: 1
})
```

**Figure 28**  Compound key index creation

The MongoDB single field indexes implemented in this study are the shown in Figure 29 and compound key indexes respectively are shown in Figure 30.
Implemented single key indexations

```
// Filetype: 1,
```

```
// HTTP_Archive_Time: 1,
```

```
// HTTP_Content_Length: 1,
```

Figur 29  Implemented single key indexations

```
// Filetype: 1,
// HTTP_Archive_Time: 1,
```

```
// Filetype: 1,
// HTTP_Content_Length: 1,
```

```
// HTTP_Archive_Time: 1,
// HTTP_Content_Length: 1,
```

Figur 30  Implemented compound key indexations

Implementation of an indexation based on the HTTP URL attributes were not possible since some of the URLs were too massive in order for them to be implemented as an index. Creating an index based on the concept of the URLs would however most likely increase MongoDB’s search performance.
5.1.4 CouchDB query


Unlike MongoDB, CouchDB queries are executed using a web browser and its URL. As the importance of ensuring that the DBMS’s are given the same chances to perform as that the DBMS’s are not misused in their implementation. Since the two DBMS’s are implemented differently the only aspect that will be measured are the response time of the queries themselves. Since the response time stems from the web browser the TamperMonkey tool may be implemented in order to extract the query’s exact response time. In order to extract data from CouchDB a view has to be created. The view structures the data by specifying the conditions of the query and also what should be returned in the result. Figure 31 demonstrates the particular view used in this pilot study.

```javascript
function(dbo) {
  if(dbo.Filetype == 'html'){
    emit(dbo.HTTP_Archive_Time, {ID: dbo._id});
  }
}
```

**Figur 31** CouchDB view

Depending on what type of data that is needed it might be necessary to add further specification in the URL. Figure 32 demonstrates the difference between a CouchDB query that does not include a further specification and a CouchDB query that do include a further specification.

```
// Ordinary query
http://localhost:5984/db/_design/testqueries/_view/query2

// Ordinary query
http://localhost:5984/db/_design/testqueries/_view/query2?startkey=3222005-01-01T00:00:00Z&endkey=3222005-12-31T00:00:00Z
```

**Figur 32** CouchDB view query

The CouchDB script is initiated when it senses that the current URL coincides with the URL that is declared in the TamperMonkey header. When the view has been created its URL may be entered into the web browser which in turn will display the results. In order to extract the desired data from the web browser TamperMonkey is used. The TamperMonkey script is written in the JavaScript programming language and in that regard it works similar to the MongoDB script which is written in NodeJS. The initiation of the TamperMonkey script is started when the correct page is loaded by the web browser.
The TamperMonkey script is implemented in the TamperMonkey tool which essentially is a tool to modify web pages. The TamperMonkey script implemented in this study is programmed to loop through the same query ten times before increasing the variable containing the targeted year. Figure 33 demonstrates the event chain in that the TamperMonkey script loops through when querying the databases.

![CouchDB event chain](image)

**Figure 33** CouchDB event chain

In the TamperMonkey script there is a need for special headers which conveys what attributes and resources are required by the script so that it will work as intended. In Figure 34 all the required TamperMonkey headers are displayed.

```javascript
// ==UserScript==
// @name New Userscript
// @namespace http://tampermonkey.net/
// @version 0.1
// @description Generation of statistical data for CouchDB's query 2
// @include http://localhost:5984/db0/design/testqueries/view/query2
// @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author Mathias Kinnander
// @match https://chrome.google.com/webstore/category/extensions
// @grant GM_xmlhttpRequest
// ==/UserScript==
```

**Figure 34** CouchDB headers

Since the TamperMonkey script changes the current web page each time a new query is executed all local variables and data stored in the script get destroyed. The solution to this problem is using the local storage method which enables web application to store data locally within the user’s web browser so that the script can act appropriately depending in which stage it finds itself in. The variables listed in Figure 35 enables the script to loop through the queries by year.
The variables startYear and currentYear are defined in the code snippet below:

```javascript
var startYear = 1997;
var currentYear = localStorage.getItem("LScurrentYear");
var endYear = 2017;
```

**Figure 35** CouchDB year variables

The startYear variable defines the earliest possible year to measure from and gets assigned to the currentYear variable if it currentYear is undefined. First the currentYear variable gets assigned the value from local storage and if local storage were empty it will will later get assigned the startYear value. The currentYear variable stores its value into local storage in the end of the script which then allows it to reuse the same value of the variable on the next web page. The endYear is the variable that is used in comparison to the currentYear variable and if the currentYear value is equal to the endYear value the script is stopped.

When measuring the performance of CouchDB it must be taken into account that it is only a few aspects that are interesting to measure. One of these are `window.performance.timing.requestStart` which measures when the request was sent from the web browser to the DBMS, the other aspect that is interesting is the `window.performance.timing.responseStart` which measure the current time when the response were starting to be received from the DBMS. In Figure 36 there is a demonstration of the window.performance structure and what is measured in the study as highlighted by the red marker.

**Figure 36** Window.performance structure

Subtracting the requestStart value from the responseStart value will then convey the total time it took for the DBMS to start replying after the browser was done sending the request. Figure 37 demonstrates how the performance is extracted and Figure 38 demonstrates how it is subtracted in CouchDB.
function collectResponseTime() {
    var requestStartTime = window.performance.timing.requestStart;
    var responseStartTime = window.performance.timing.responseStart;
    var responseTotalTime = responseStartTime - requestStartTime;
    console.log("Response time collected: " + responseTotalTime);
}


var responseTimeSum = 0;
for (var i = LSallResponseTimes.length; i--; ) {
    responseTimeSum = responseTimeSum + LSallResponseTimes[i];
}
var avgResponseTime = responseTimeSum / LSallResponseTimes.length;

**Figur 37** Collecting the response time

Each type of query is executed 100 times in order to generate a greater amount of data instead of just querying once per query type. In turn this will arguably make the calculated average time performance for each query more reliable. In order to execute the same query 100 times while also changing the webpage local storage may be implemented in a similar way using a local storage variable. The local storage variable used for the query number is queryNr for the local variable and LSqueryNr for the local storage variable. The different functions is run within If statements since the script is required to do different things depending on what query number it is. The IF statements are conditions regarding the queryNr variable and its current value. When a query has been successfully made the queryNr variable gets incremented by one and stored into the localstorage so that it may be used on the next web page. Depending on what is needed by the script different functions are called. Figure 39 demonstrates a function that extracts the number of results found by the query and the calculated average response time.

function endOfquery() {
    var count_data = '';
    function getCountData() {
        count_data = document.body.innerText.split('key').length;
    }
    function ajaxCall(data) {
        try {
            var XMLHttpRequest = QM/XMLHttpRequest;
            var url = localURL;
            var data = 'get' + encodeURIComponent(data) + '\n\n',
            headers = ["
                'Content-Type': 'application/x-www-form-urlencoded'
            ];
        }
        catch (ex1) {
            console.log(ex1);
        }
    }
    getCountData();
    scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = " + count_data);
    ajaxCall(scrapedData);
}

**Figur 38** Calculating the average response time

**Figur 39** A function within TamperMonkey
All of the functions implements AJAX calls to transfer the extracted search time data onto the remote script. Figures 40 contains the code of the PHP script that transfer the data into the text file.

```php
<?php
    $fp=fopen("couchDB_query2.txt","a");
    fputs($fp, $_POST['str']);
    fclose($fp);
?>
```

**Figure 40** CouchDB store data to text file

When the data has been stored correctly into the file it should be similar to the results displayed in Figure 41.

```
Query nr: 0. Elapsed time = 6
Query nr: 1. Elapsed time = 2
Query nr: 2. Elapsed time = 2
Query nr: 3. Elapsed time = 2
Query nr: 4. Elapsed time = 2
Query nr: 5. Elapsed time = 2
Query nr: 6. Elapsed time = 3
Query nr: 7. Elapsed time = 5
Query nr: 8. Elapsed time = 2
Query nr: 9. Elapsed time = 1
Query nr: 10. Elapsed time = 2
Average response time = 2.3. Number of results = 5676
```

**Figure 41** Results of a CouchDB query

When querying CouchDB the usage of views are needed. Views can be created using the Futon administration tool. Futon is a web GUI administration panel which enables the user to interact with CouchDB in a convenient way. Creating a view can be done when saving a temporary view which in turn will turn it into a permanent one. Figure 42, Figure 43 and Figure 44 demonstrates the creation process of a permanent query which is used in pilot study.

```javascript
function(dbo) {
    if(dbo.Filetype == 'html'){
        emit(dbo.HTTP_Archive_Time, {ID: dbo._id});
    }
}
```

**Figure 42** Writing the code in the temporary view
Saving the temporary view as a permanent view

When a permanent view is created the indexation process starts. When the indexation process has been completed the view is ready for querying. When accessing the result from the view all the returned data is listed on the CouchDB’s API. A demonstration of this is shown in Figure 45.

The view lists only the keys from documents and the attributes that were included in the view. If further specification is required the URL on CouchDB’s webpage can be altered so that it includes a filtration of the results. Figure 46 demonstrates how such the URL includes a filtration based on a particular set of years.
The displayed data can then be extracted and implemented in a solution that would for example display the returned results in a statistical fashion. More documentation regarding views can be found at Couchdb.org (2018) Find Your Data with Views. http://guide.couchdb.org/draft/views.html.

5.1.5 CouchDB indexation

Similar to MongoDB CouchDB also implements indexation in order to increase its search performance. CouchDB however implements B-tree indexation which is the structure of the indexation. B-tree structures are made up of values stored in the root node and with for example the maximum of four leaf nodes. This set up can be seen in Figure 47 where the values 1, 2, 5, 8, 10, 12, 15, 19, 20 and 25 are stored.

![B-tree first example](image1)

**Figure 47** B-tree first example

If the maximum number of values in a box is set to four in this scenario the box will then be split up into two. If a new a new value such as 30 would be inserted into the collection it would get sorted into the lower right box but the values 15 and 19 would be placed in a new separate box. The median number of the ones in the lower right box and the new number becomes part of the root node box and as such b-tree would balance itself out by creating a new structure as can be seen in Figure 48.

![B-tree second example](image2)

**Figure 48** B-tree second example
More documentation regarding CouchDB’s implementation of b-tree can be found at Couchdb.org (2018) The Power of B-trees. http://guide.couchdb.org/draft/btree.html. In the setup of this experiment Futon was used in order to create views. When a view is created the indexation process begins automatically. Depending on the size and quantity of the data stored within the database the indexation process time varies. When dealing with a large data set the indexation time for CouchDB is significantly greater than that of the indexation time for MongoDB. When evaluating which DBMS would be the more efficient solution to implement for the Swedish Internet it would also be necessary to keep the total indexation time in mind. Figure 49 demonstrates an instance where CouchDB is indexing the database for the view used in the main experiments.

**Figure 49** CouchDB indexation process on Futon
5.2 Work Progression

In the beginning of the project there was an intention of implementing a visualization tool and together with the chosen DBMSs. This idea however was abandoned due in part to time constraints but also since it was deemed to be more interesting to evaluate the performance of the DBMSs when storing data from the Swedish Internet. One of the greatest problem in this study was the lack of access to the data itself in the early stages of the study. This made the preplanning of the implementation difficult and only allowed for speculation. The main hindrance from this is the amount of times the aspects and goals in study has been altered or completely changed.

The plans of implementing CassandraDB were also canceled due to time constraints. The premise with CassandraDB was to provide an alternative to the document oriented databases since CassandraDB is a wide column database. Three different types of queries were developed for both of the DBMSs, each targeting specific documents with the correct attributes. The following queries were:

- Select all websites content length in order to calculate the average value of the content length by year.
- Select all HTML websites from a specific date in order to group them by year in order to calculate the number of HTML websites
- Select all subtypes and group them by each subtype in order to see the frequency of different formats contained within the Swedish Internet.

During the setup of MongoDB and CouchDB the idea was to implement them in the exact same way with the difference being the structure of the queries themselves. This however was not plausible since the default way to use CouchDB is through the web browser and the default way to use MongoDB is using a MongoDB client or driver. The way to circumvent these differences is to measure the same aspects in the search time performance execution and disregarding any other irrelevant data.

In the beginning of the study there was an idea to do each measurement manually. However this quickly became apparent that a measurement that automatization of the process were needed to conduct the study in an efficient way. The solution were to use the queryNr function to loop through the querying a specific amount of time. The code was implemented in the Github (2018) https://github.com/b15matki/examensarbete/commit/4b09f9e84bbaa1547727724de92e732c1ff8f3f292a commit. Prior to this implementation the work was working correctly but were not automatized.

Initially the storing of the results were intended to be saved onto the local hard drive. However since there should be as little strain on the local system as possible the storing of the data should be outsourced. In order to store the data onto a text file on another server a PHP script was implemented in Github (2018) b15matki/examensarbete@edfce08 https://github.com/b15matki/examensarbete/commit/edfce08999e28993e0a24caef5e83e74ab12c5e6.
6 Experiment Setup, Results and Analysis

6.1 Experiment Description
The testing was conducted by executing different queries against the databases that contain the Swedish Internet. The different queries vary in order to determine how the structure or the number of returned documents from the query might impact the overall search performance. In the pilot study the queries searches for HTML documents grouped by year. The queries implemented in the evaluation are similar queries that searches for the different types of documents stored within the Swedish Internet. The other queries implemented in the statistical evaluation examine how the complexity of the queries might affect the performance of the DBMS.

Identical queries were implemented for each DBMS which in turn yields the same results. The averaged query response time is based on 100 queries from one of query types. They are in turn compared with the other query type’s averaged response time from the other DBMS.

6.1.1 Hardware & Software
In this study all of the experiments were run on the same computer with the same type of settings and versions of the programs implemented. The hardware specifications are listed in Table 1 and software specifics are listed in Table 2.

**Tabell 1** Hardware specifications

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Ubuntu 16.04 LTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7-3770k CPU @ 3.50GHz x 8</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GiB</td>
</tr>
<tr>
<td>Graphics</td>
<td>NV64</td>
</tr>
<tr>
<td>Storage</td>
<td>Western Digital WD10EZEX 1 TB</td>
</tr>
</tbody>
</table>

**Tabell 2** Software specifications

<table>
<thead>
<tr>
<th>Web browser</th>
<th>Google Chrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>64.0.3282.167 (Official build) (64-bit)</td>
</tr>
<tr>
<td>Tampermonkey version</td>
<td>4.6 (2018-05-07)</td>
</tr>
<tr>
<td>MongoDB version</td>
<td>v3.2.19</td>
</tr>
<tr>
<td>CouchDB version</td>
<td>1.6.0</td>
</tr>
<tr>
<td>Linux version</td>
<td>4.13.0-41</td>
</tr>
</tbody>
</table>


6.1.2 Pilot study

Before the testing began the computer was rebooted and effort was made to minimize the amount of unnecessary programs that were running. The first measurement began with the MongoDB script which was executed through the terminal on Ubuntu. The script saved the data generated into the text files located on the University of Skövde’s server. Only 14,418 documents were inserted into the database during the pilot study. The process was repeated for CouchDB: the computer was rebooted and unnecessary program were terminated. The web browser Google Chrome opened with the TamperMonkey script enabled. The script was then initiated by entering the following URL http://localhost:5984/dbo/_design/testqueries/_view/query2 which triggers the execution of the query. The script proceeded to parse through all of the queries and storing the data onto the text files hosted on the University’s server.

When the measurements had been executed they returned the search performance results of each DBMS and also the number of returned articles for each DBMS. The number of articles returned from the DBMSs’ queries were identical for that of each DBMS. Extracting the averaged values from each year along with the number of returned results and inserting them into a chart creates the following result as seen in Figure 50.

![Pilot study query results](chart)

**Figure 50** Results Chart

The result indicates that the performance of CouchDB (in red) is decisively better than the performance of MongoDB (in blue) when it comes to this particular query and data set. The results (in green) seem to be centered between the years 2006 to 2011 with the exception of 2009. Another conclusion that can be drawn from the pilot study is that there are no apparent correlation between the numbers of returned results and search performance.

- The longest measurement time for MongoDB was 42 ms and for CouchDB 15 ms.
- The fastest measurement time for MongoDB was 22 ms and for CouchDB 1 ms.
- The average measurement time for MongoDB was 27.4 ms and for CouchDB 2 ms.
6.1.3 Experiment 1 – Content type

**Figure 51** Experiment 1’s result chart
The first conducted experiment which can be seen on Figure 51 evaluated one specific query where the DBMSs were asked to find and return all the different files of a particular content type. All of the content types except “unidentified” were involved in the querying process. Similarly to the experiment that was conducted in the pilot study this experiment retrieved all instances of the 71 content types stored within the database. Each content type were queried 100 times. In total 14 200 individual queries were executed in the experiment. The database contained 888164 documents and were 32.2 GB in size.

### 6.1.4 Experiment 2 – Query Complexity

![Diagram](image)

**Figure 52** Experiment 2’s result chart

The second experiment that was conducted which results is demonstrated in Figure 52 was a series of queries with an ascending degree of complexity. This was done in order to evaluate how the increase of complexity in queries affected the performance of the DBMSs. Similar to the previous experiment the database contained 888164 documents and were 32.2 GB in size. The different queries implemented in the experiment were the following:

1. Find and retrieve all HTML documents
2. Find and retrieve all HTML documents with the content length size between 1000 bytes and 20 000 bytes
3. Find and retrieve all HTML documents with the content length size between 1000 bytes and 20 000 bytes that has been published sometime between the years of 2000 and 2005
4. Find and retrieve all HTML documents with the content length size between 1000 bytes and 20 000 bytes that has been published sometime between the years of 2000 and 2005 from “www.skolverket.se”
6.2 Result analysis

6.2.1 Experiment 1 – Content type

The result from the first experiment showed that CouchDB performed considerably better in some instances than MongoDB with MongoDB sometimes being hundreds of times slower than CouchDB. The average search time performance for both DBMSs however is significantly more similar to each other. MongoDB seems to perform worse when more results are returned from the queries while CouchDB performs consistently around the same execution time. The response time, the average response time and number of returned documents from the queries were extracted and stored into a text folder. The general results of the queries can be demonstrated in Figure 53.

![Comparison of search performance from the 1st experiment (without indexation time)](image)

**Figure 53**  Experiment 1’s result comparison

- The longest measurement time for MongoDB was 456 ms and for CouchDB 25 ms.
- The fastest measurement time for MongoDB was 0 ms and for CouchDB 1 ms.
- The average measurement time for MongoDB was 9.21 ms and for CouchDB 2.44 ms.
- The average measurement time for the query with the most returned results were 410.9 ms for MongoDB and 2.31 ms CouchDB.
- The average measurement time for the queries with the fewest returned results were 0 ms for MongoDB and 2.23 ms CouchDB.

As demonstrated in Figure 53 MongoDB performed considerably worse in some measurements than CouchDB did, especially when it comes to the prospect of querying data where the returned result were large. One aspect that must be taken into account is the time it takes for CouchDB to index the data before enabling any execution of queries which in this case was 47 minutes and 5 seconds. In comparison MongoDB’s indexations which were all finished within 30 seconds.

Dividing the time it took for CouchDB to index the database (2 825 000 ms) by the number of queries that were executed (7100) and adding the result to the calculation with the other
averaged times for a CouchDB queries outlines the impact of the indexation time over each individual query. This result can be seen on Figure 54.

**Figure 54** Experiment 1’s result comparison with indexation time

- **Longest CouchDB measurement time:** \((2,825,000 \text{ ms} / 7100) + 25 \text{ ms} = 422.88 \text{ ms}\)
- **Fastest measurement time:** \((2,825,000 \text{ ms} / 7100) + 1 \text{ ms} = 398.88 \text{ ms}\)
- **Average measurement time:** \((2,825,000 \text{ ms} / 7100) + 2.44 \text{ ms} = 400.32 \text{ ms}\)
- **Average CouchDB measurement time for the query with the most returned results:** \((2,825,000 \text{ ms} / 7100) + 2.31 \text{ ms} = 400.19 \text{ ms}\)
- **Average measurement time for the queries with the fewest returned results:** \((2,825,000 \text{ ms} / 7100) + 2.23 \text{ ms} = 400.11 \text{ ms}\)

With the new averaged values from CouchDB when taking into account the time it took for the database to be indexed and queried the averaged results from both DBMSs are much more even. A graphical comparison between the averaged results are demonstrated in Figure 40. Despite the added execution time to CouchDB’s time measurements it still performs better than MongoDB in the measurements that had a larger query result although the differences between the two are notably less different. MongoDB is a more efficient implementation when it comes to queries that yields fewer results.
6.2.2 Experiment 2 – Query Complexity

The longest measurement time for MongoDB was 999 ms and for CouchDB 94 ms.

The fastest measurement time for MongoDB was 397 ms and for CouchDB 1 ms.

The average measurement time for MongoDB was 625.52 ms and for CouchDB 2.65 ms.

The average measurement time for the query with the most complexity and least results were 980.02 ms for MongoDB and 3.32 ms CouchDB.

The average measurement time for the queries with the least complexity and most results were 416.13 ms for MongoDB and 2.62 ms CouchDB.

As demonstrated earlier Figure 55 shows that there is a significant correlation between the level of complexity and the time it took for MongoDB to retrieve the information from the database, with a total difference of 563.62 ms in between the averages in time performance on the first level of complexity and the fourth level of complexity. The performance impact of complexity on CouchDB is minimal in comparison, with an average difference of 0.7 ms. The search response time of CouchDB is diminutive compared to MongoDB.

Dividing the time it took for CouchDB to index the database (3 092 000 ms) by the number of queries that were executed (400) and adding the result to the calculation with the other averaged times for a CouchDB queries outlines the impact of the indexation time over each individual query. This result can be seen on Figure 56.
Experiment 2’s result comparison with added indexation time

- Longest CouchDB measurement time: \((3 092 \text{ ms} / 400) + 94 \text{ ms} = 7782 \text{ ms}\)
- Fastest measurement time: \((3 092 \text{ ms} / 400) + 1 \text{ ms} = 7731 \text{ ms}\)
- Average measurement time: \((3 092 \text{ ms} / 400) + 2.65 \text{ ms} = 7732.65 \text{ ms}\)
- Average CouchDB measurement time for the query with the most returned results: \((3 092 \text{ ms} / 400) + 2.62 \text{ ms} = 7732.62 \text{ ms}\)
- Average measurement time for the queries with the fewest returned results: \((3 092 \text{ ms} / 400) + 3.32 \text{ ms} = 7733.32 \text{ ms}\)

With the addition of the indexation time onto the CouchDB performance results it is clear that the total amount of time it took in order to execute a CouchDB query was remarkably higher than it would have been to implement a MongoDB query that would have yielded the same results. This is in due part to the low number of executed queries in the experiment which renders the implementation of CouchDB as the inferior choice.

**6.2.3 Evaluation conclusions**

Comparing the response time of each DBMSs through the experiments made it clear that CouchDB consistently performs better than MongoDB. An exception to this statement can be made however when the returned result is relatively small. MongoDB is greatly impacted by the size of the returned results while CouchDB is significantly less impacted. This is due to the efficient indexation done by CouchDB and the fact that CouchDB uses B-trees which is as a self-balancing tree data structures as defined by Comer, D. (1979). The B-tree data structure enables CouchDB to index huge amount of data and guarantess fast and consistent performance. MongoDB also implements indexation but leaves it up to the user to define what and how the indexation should be done. Multiple indexes can be implemented when using MongoDB which improves its search performance. However despite implementing multiple indexes matching the queries’ needs, MongoDB’s search performance cannot match the performance of CouchDB.
When also accounting for the CouchDB’s indexation time, it is clear that the average time it took for a query to be completed for CouchDB is massively longer than that of MongoDB under a certain number of queries threshold. The more CouchDB is executing queries based on a particular view the more a user gains in time compared to if a user would use MongoDB which often has a higher search time. A general demonstration of when CouchDB actually becomes the more efficient alternative based on the second experiment is shown in Figure 57.

![Approximation of the implementation effectiveness for the DBMSs](image)

**Figure 57** Line diagram of implementation effectiveness

Using the returned performance statistics from the second experiment where only 400 queries were executed the approximate breaking point where the time it took for CouchDB become more cost effective is around the 5000th query if MongoDB has a search time performance of 625.52 ms. Purposefully querying a particular view 5000 times requires that the view is well thought out since changing the view itself requires a new indexation of the database. There is however some flexibility with the CouchDB’s views since further specifications can be added and altered in the view’s URL to filter out unwanted results. Depending on how the DBMSs are intended to be used e.g., using the same view frequently or updating the view structure regularly, may have a significant impact on whether or not one particular DBMS is the superior choice.
7 Concluding Remarks

7.1 Discussion

The study has been conducted in such a way that would reflect a real life implementation of the DBMSs and makes an effort to come to a conclusion that is as close to their actual performances as possible, with a subset of data from the Swedish Internet. The queries used in the study are meant to reflect queries that could potentially yield interesting and valuable results for potential researchers and scientists as motivated by Ratanaworabhan, P. et al (2010). When querying the data, the search performance measurements are extracted from the time it took for the particular DBMS to execute the query and therefore disregards any other aspects that might play a role in the total performance output.

One important factor that should be kept in mind from this study is that the different DBMSs benchmarking has been realized using different programming languages. It might be the case that if the querying of the DBMSs used other programming languages than the ones implemented in the study, the search performance of the DBMSs could change. The study conducted by Bhardwaj, N. (2016) implemented a similar study using the same programming languages for the setup and the querying. Their study however concluded that MongoDB had a faster response time than CouchDB. This possible performance impact because of the choice of programming language is unfortunately not addressed in this study. Furthermore the study does not take into account for the performance output of a DBMS with other applications and claims only to evaluate the isolated search performance of the chosen DBMSs as previously mentioned. Although the quality of the technical artifact is not optimal the results that it produces are reliable results that is concurrent with DBMSs actual performance.

The results from this study might be valueable for individuals and organisations that are evaluating different DBMS options for their particular needs. If their dataset and the queries they implement are similar to the used subset data and the study queries, their search performance output would be similar too. The study shows that the specific usage (e.g., a website where hundreds of users summons multiple requests per second or researchers extracting data a few times a day) of a DBMS significantly impacts the general effectiveness of the implementation.

Websites and organisations that retains large data archives such as Wikipedia, Stackoverflow, Facebook and Amazon etc could potentially extract some useful information from this study that could assist in their decisionsmaking that concerns future DBMS selections and implementations. One other possibility is that the developers of the DBMSs evaluated in the study could benefit from the feedback provided. Future versions of the DBMSs will most likely improve their search performance and their overall effectiveness. Subsequently similar studies should be conducted in order to keep the conclusions current and relevant. If the study needs to be expanded or redone in the future the technical artifacts could be adapted to suit new requirements and testing goals such as other measurements that could be implemented into the artifact to then increase the span of the study.
7.2 Conclusion

The hypothesis of this study was that MongoDB’s search performance will be superior to that of CouchDB when it comes to querying and retrieving results from databases that contain the Swedish Internet which has been proven to be false. The result and the evaluations from the study concludes that CouchDB has a superior and a greater consistency in performance than MongoDB when it comes to querying and retrieving data from a database that stores the Swedish Internet. This outcome is consistent with the conclusion from the study conducted by Bhardwaj, N. (2016) where they established that CouchDB had a superior READ performance than MongoDB. CouchDB consistently shows a query performance around 1 – 3 ms per query independent of the magnitude of the returned result or the query’s level of complexity which is contrary to the results from the study by Henricsson, R. (2011). The main reason why CouchDB had such a superior performance is the implementation of permanent views instead of the temporary views that were used by Henricsson, R. (2011). Furthermore the measurements from our study also demonstrates that MongoDB's query performance varies greatly depending on the amount of data that is being queried, the extent of the returned results and the level of complexity of the query itself.

The conclusion that CouchDB has a superior search performance was reached by performing experiments on sample data from the Swedish Internet and using the response time from the DBMSs in empirical measurements. When indexing the views in CouchDB time measurements were taken of the total indexation time which were later used in the evaluation. The sample data were inserted into both DBMSs' respective databases and then were subject to a number of identical experiments that implemented different types of queries in order to find out how the DBMSs performance varied depending on each type of query.

When other aspects other than just the search performance are taken into account, as it is done in the study by Bhardwaj, N. (2016) the results from the evaluation are more contrasted. The strength of CouchDB's indexation however can also become its weakness since the time it takes for CouchDB to index data is vastly greater than that of MongoDB. This drawback which MongoDB lack is also what makes it the superior choice for queries that would only be executed a few amount of times. MongoDB is more suited for dynamic querying and fast prototyping, while CouchDB excels at performing under high loads and static querying. When keeping the indexation time in mind there is scenarios where MongoDB would be the more efficient option to implement despite its relatively long and inconsistent search time.
7.3 Future work

There are several aspects that this study has not touched upon that may be relevant in order to be assured that the particular DMBS would be the most effective alternative of the two. For example implementing both DBMSs with the type of application that would be used in practice might yield further insight. The performance output with the complete solution might differ from the performance output of the DBMSs alone because of the composition and quantity of the queries.

One aspect that should be investigated further is the impact of NodeJS, Javascript and PHP which were used for the implementation of the querying in the study. The selection of language and the final implementation of the language itself may have a performance impact on the search performance of the DBMSs. In order to evaluate this aspect one possible scenario would be to implement the DBMSs and the databases using the exact same language or even set up multiple pair of DBMSs with different languages.

Since big data sets grows exponentially and quite frequently it would be a good idea to conduct studies to see how the different DBMSs deals with new data being continuously added to the data set. It might be the case that a CouchDB solution would be impractical on such a large data set as the Swedish Internet. Alternatively it could be the case that the performance of MongoDB deteriorates as more and more data is added and stored into the database.

As time progresses new versions of the DBMSs will become available for implementation. New similar studies should be conducted to evaluate the then current search performance of the DBMSs in order to conclude which alternative is the more efficient one. The performance of a particular DBMS might differ greatly in the future from how it performs today. One comparable study akin to conducting experiments with new versions, would be to conduct the same type of experiments with older versions of the DBMSs. This could indicate potential trends in search performance gain over multiple previous versions of the DBMSs.

The data subset given in this study cannot yield the same measurement results as if the entirety of the Swedish internet was tested, because of the quantity of data considered and because of the differences of implementation: one could hardly store all the Swedish Internet data on a single DBMS node, a distributed architecture would be required. Therefore it might be a good idea to try to implement the solution on a larger or preferably the entire data set from the Swedish Internet or a similar data set. Only when the entirety Swedish Internet is implemented into the study the actual performance be measured. The data in the Swedish Internet might need to include different types of data in the future in order to accommodate new technology, laws or because of any other reason. It would be wise to test how the chosen DBMSs would handle and perform with other types of data, even though NoSQL DBMSs are well suited for unstructured data.

Furthermore in order to assure that the DBMS chosen in the final implementation efficient similar studies should be conducted with other DBMSs. It may be the case that neither MongoDB nor CouchDB is the one DBMS most suitable for this task and therefore other alternatives should also be taken into consideration.
8 Research Ethics

The ethical aspects of this research is to study the different DBMS under the same opportunities consideration, and tries to minimize all interference and bias that might occur. One minor possibility that might occur because of this thesis would be that the reader will develop a bias favoring a particular DBMS that in turn would make the reader discriminate towards other DBMS alternative. Since this study only focuses which DBMS that has the fastest search performance for which situations, between the two candidate DBMSs when big data sets such as the Swedish Internet are considered. Other DBMS alternatives might be a more suitable choice for similar data sets. Furthermore undoubtedly new updates and versions of the selected DBMS as well as other DBMS will be updated that may improve the performance of a particular DBMS so the thesis may lose its relevance in the future and make the bias irrelevant or even incorrect.

Since the implementation of the study will not involve any test subjects there is essentially no possible effect on any individual in the surrounding environment other than the conductor and reader. The study is therefore a technology-oriented experiment as defined by Wohlin, C. et al (2012) which focuses more on the empirical measurement of technical aspects rather than the human aspects.

Handling of the data that is saved within the Swedish Internet collection is pivotal to ensure the confidentiality of the data. The study only concerns itself of the structure and format of used by webpages in the Swedish Internet. Results from the data migration into the databases will not include any content in the form of text, images, audio etc.

One of the aspects why the integrity of the data set is so important is that it may contain certain sensitive data that has been extracted from government, commercial, organizational as well as other private websites. The usage of the data and the results produced in the testing has to be handled with great care and caution ensuring that no unauthorized viewing or usage occurs. One way to minimize the risk of displaying sensitive data in the results would be not to store any type of text, audio, image related data on public servers, domains and other storage alternatives. In order to ensure that the test will be repeatable the source code and scripts used in the study will also be available to any individual that wishes to conduct a similar study. However the data used in the study was lent by the Swedish Royal Library (Kungliga biblioteket) and might therefore be in some circumstances unavailable. Similar data sets with the same structure and size could substitute the Swedish Internet in a recreation of the study. The data does not necessarily need to be web related nor in the format of the Swedish Internet. Furthermore, the studied DBMSs are distributed freely and are part of open-source projects, guaranteeing their availability for future evaluation.
References


Fix for query 2 · b15matki/examensarbete@4b09f9e (2018). Available at: https://github.com/b15matki/examensarbete/commit/4b09f9e84baa1547727724de92e732c1ff8f392a (Accessed: 15 April 2018).

Added php file · b15matki/examensarbete@edfce08 (2018). Available at: https://github.com/b15matki/examensarbete/commit/edfce08999c28993e0a24caef5e83e74ab12c5e6 (Accessed: 15 April 2018).

Appendix A - Insertdata.js

// process.env.DEBUG = 'mimemessage*'; // enables error logging for mimemessage
var fs = require('fs'); // import the fs library to read files on the disk
var mimemessage = require('mimemessage'); // imports the mimemessage library
const targetFolder = '/home/mathias/Documents/kungliga-biblioteket/deflated/';
console.log("Document insertion initiated!");

var nrFile = 0;
var numberOfRejectedFiles = 0;

/************************** Progress initialization **************************/
const progress = require('cli-progress');
const insertProgress = new progress.Bar({etaBuffer: 1000},
progress.Presets.shades_classic);

/************************** MongoDB initialization **************************/

var Mongo = require('mongodb');
var MongoClient = Mongo.MongoClient;
var Mongo_url = "mongodb://localhost:27017/";

/************************** CouchDB initialization **************************/

var nano = require('nano')('http://localhost:5984');
var dbcouch = nano.db.use('dbo');

var Mongo_client;
var dbmongo;
var gridfs;

async function connectMongo() {

    var settings = {
        poolSize: 10
    }

    Mongo_client = await MongoClient.connect(Mongo_url, settings);
    dbmongo = await Mongo_client.db("mongomydb");
    gridfs = await new Mongo.GridFSBucket(dbmongo);
}

/* Loops through the files */
var files = fs.readdirSync(targetFolder);

async function insertFiles(files) {

    await connectMongo();

    insertProgress.start(files.length, 0);
for (let file of files) {

try {
    /* Reading of targetet file */
    var content = fs.readFileSync(targetFolder + file, 'utf8'); // reads from the file which path is in argument and puts its content into the content variable
    content = content.replace(/\n/g, '\r\n'); // fixes lines ending so that mimemessage works: REALLY IMPORTANT
    content = content.replace(/Content-Type: null/g, 'Content-Type: text/plain'); // fixes malformed mime messages with a null content type
    content = content.replace(/Content-Type: unknown/g, 'Content-Type: text/plain'); // fixes malformed mime messages with an unknown content type
    content = content.replace(/Content-Type: text/html; iso-8859-1/g, 'Content-Type: text/html; charset=iso-8859-1'); // fixes missing charset=

    var msg = mimemessage.parse(content);

    /****************** Generation of data ******************/
    /* Extract the correct element containing the subtype & the other body */
    var filetype;
    var filecontent;
    var http_headers;
    for (var k in msg.body) {
        if (msg.body[k].header('HTTP-part') == 'Content') {
            filetype = msg.body[k].contentType()['subtype'];
            filecontent = msg.body[k].body;
        } else if (msg.body[k].header('HTTP-part') == 'Header') {
            http_headers = msg.body[k].body;
        }
    }

    /* Converting the epoch time into dates */
    var utcSeconds = msg.header('HTTP-Archive-Time');
    var d = new Date(0); // The 0 there is the key, which sets the date to the epoch
    d.setUTCSeconds(utcSeconds);

    //Standard headers
    var myobj = {
        //First Header
        MIME_Version: msg.header('MIME-Version'),
        Type: msg.contentType()['type'],
        Subtype: msg.contentType()['subtype'],
        Params: msg.contentType()['params'],
        HTTP_Part: msg.header('HTTP-part'),
        HTTP_Collection: msg.header('HTTP-Collection'),
        HTTP_Harvester: msg.header('HTTP-Harvester'),
        HTTP_Header_Length: parseInt(msg.header('HTTP-Header-Length')),  
        HTTP_Header Md5: msg.header('HTTP-Header-MDS'),
        HTTP_Content_Length: parseInt(msg.header('HTTP-Content-Length')),  
        HTTP_Content Md5: msg.header('HTTP-Content-MDS'),
        HTTP_Url: msg.header('HTTP-URL'),
        HTTP_Archive_Time: d,
        HTTP_Headers: http_headers,
    }
}
// Second section: file
Filetype: filetype,
Filecontent: null
}

/********************** MongoDB data insertion **********************/
try {

// Mongo file insertion
if (filecontent != null || filecontent != undefined || filecontent != "") {
    try {
        var upload = gridfs.openUploadStream(myobj.HTTP_Content_Md5);
        await upload.write(filecontent);
        await upload.end();
        myobj.Filecontent = "gridfs";
    } catch (err) {
        console.log("Error uploading file content to gridfs");
    }
}

// Mongo article insertion
await dbmongo.collection("mongoarticles").insertOne(myobj);

} catch (err) {
    console.log("Mongo error" + err);
}

/********************** CouchDB data insertion **********************/
myobj.Filecontent = filecontent;
await dbcouch.insert(myobj);
nrFile++;

} catch (err) {
    numberOfRejectedFiles++;
    console.log("Error on file " + file + ": " + err);
}

insertProgress.update(nrFile + numberOfRejectedFiles);
}

insertProgress.stop();
console.log("Document insertion process completed! " + nrFile + " of files scanned in total!");
console.log("Number of rejected files: " + numberOfRejectedFiles);
} insertFiles(files);
Appendix B - mongoDB_query2.js

```javascript
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = "";

// Time variables
var currentYear = 1997;
var endYear = 2017;

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';

async function searchMongoDB() {
  while (currentYear <= endYear) {
    var samples = [];
    console.log("Querying: " + currentYear);
    for (var queryNr = 0; queryNr < 100; queryNr++) {
      // Use connect method to connect to the server
      var client = await MongoClient.connect(url);
      console.log("Connected, executing " + queryNr + "/" + 100);
      const db = client.db(dbName);
      var articles = db.collection("mongoarticles");

      // Get the documents collection
      // Find some documents
      var req = await articles.find({
        "Filetype": 'html',
        "HTTP_Archive_Time": {
          "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
          "$lt": new Date(currentYear.toString().concat("-12-31T00:00:00.000Z"))
        },
      }, { _id: 1 });

      // Adds the explain specification onto the req variable
      var explain = await req.explain();

      // Puts the executionTimeMillis in the end of the array
      samples.push(explain.executionStats.executionTimeMillis);

      // Save the ordinary response time
      await client.close();
      await request.post({
```

58
headers: { 'content-type': 'application/x-www-form-urlencoded' },
url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_query2.php',
body: "str=Query nr: " + queryNr + ". Elapsed time = " +
        explain.executionStats.executionTimeMillis + "\r\n"};
}

// Use connect method to connect to the server
var client = await MongoClient.connect(url);
const db = client.db(dbName);
var articles = db.collection("mongoarticles");

//Storing the number of results into an array
var res_data = await articles.find(
    { "Filetype": 'html',
      "HTTP_Archive_Time": {
        "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
        "$lt": new Date(currentYear.toString().concat("-12-31T00:00:00.000Z"))
      }
    },
    { _id: 1 }).toArray();

var count_data = res_data.length;
console.log("Number elements: " + count_data);

//Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--;) {
    responseTimeSum += samples[i];
}
var res_avg_data = responseTimeSum / samples.length;

//Storing the average response time and number of results when one query
//loop has been completed
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_query2.php',
    body: "str=Average response time = " + res_avg_data + ". Number of results=" + count_data + "\r\n"
});

//Resetting the array
res_data = "";

//Incrementing the current year variable so that the next query searches the
//following year
currentYear++;}
}

searchMongoDB();
Appendix C - mongoDB_query3.js

const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');

// Type variables
currentType = 0;
var query = "";

async function searchMongoDB() {
  while (currentType < allTypes.length) {

    var samples = [];
    console.log("Querying: " + currentYear);
    for (var queryNr = 0; queryNr < 100; queryNr++) {

      // Use connect method to connect to the server
      var client = await MongoClient.connect(url);

      // REMOVE WHEN DONE
      console.log("Connected, executing " + queryNr + "/" + 100);

      const db = client.db(dbName);
      var articles = db.collection("mongoarticles");

      // Get the documents collection
      // Find some documents
      var req = await articles.find({
        "Filetype": "html",
        "HTTP_Archive_Time": {
"$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
"$lt": new Date(currentYear.toString().concat("-12-31T00:00:00.000Z"))}
}, { _id: 1 });

// Adds the explain specification onto the req variable
var explain = await req.explain();

// Puts the executionTimeMillis in the end of the array
samples.push(explain.executionStats.executionTimeMillis);

// Save the ordinary response time
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_query2.php',
    body: "str=Query nr: "+ allTypes[currentType] + ". Elapsed time = " + explain.executionStats.executionTimeMillis + "\r\n" });

// Use connect method to connect to the server
var client = await MongoClient.connect(url);
const db = client.db(dbName);
var articles = db.collection("mongoarticles");

//Storing the number of results into an array
var res_data = await articles.find({
    "Filetype": 'html',
    "HTTP_Archive_Time": {
        "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
        "$lt": new Date(currentYear.toString().concat("-12-31T00:00:00.000Z"))
    },
}, { _id: 1 }).toArray();

var count_data = res_data.length;
console.log("Number elements: " + count_data);

//Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--; ) {
    responseTimeSum += samples[i];
}
var res_avg_data = responseTimeSum / samples.length;

//Storing the average response time and number of results when one query
loop has been completed
await client.close();
await request.post({

headers: { 'content-type': 'application/x-www-form-urlencoded' },
url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_query2.php',
body: "str=Average response time = " + res_avg_data + " Number of results= " + count_data + "\r\n"
});

//Resetting the array
res_data = "";

//Incrementing the current year variable so that the next query searches the following year
currentYear++;
}
}
searchMongoDB();
Appendix D - mongoDB_complexityQuery_lvl1.js

const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = "";

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';

async function searchMongoDB() {
  var samples = [];
  for (var queryNr = 0; queryNr < 100; queryNr++) {
    // Use connect method to connect to the server
    var client = await MongoClient.connect(url);
    console.log("Connected, executing "+ queryNr + "/" + 100);
    const db = client.db(dbName);
    var articles = db.collection("mongoarticles");

    // Get the documents collection
    // Find some documents
    var req = await articles.find({
      "Filetype": 'html',
    }, { _id: 1 });

    // Adds the explain specification onto the req variable
    var explain = await req.explain();

    // Puts the executionTimeMillis in the end of the array
    samples.push(explain.executionStats.executionTimeMillis);

    // Save the ordinary response time
    await client.close();
    await request.post({
      headers: { 'content-type': 'application/x-www-form-urlencoded' },
      url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl1.php',
      body: "str=Query nr: " + queryNr + ". Elapsed time = " + explain.executionStats.executionTimeMillis + "\r\n"
    });
  }

  // Use connect method to connect to the server
  var client = await MongoClient.connect(url);
  const db = client.db(dbName);
  var articles = db.collection("mongoarticles");
// Storing the number of results into an array
var res_data = await articles.find({
  "Filetype": 'html',
},
  { _id: 1 }).toArray();

var count_data = res_data.length;
console.log("Number elements: " + count_data);

// Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--;){
    responseTimeSum += samples[i];
}
var res_avg_data = responseTimeSum / samples.length;

// Storing the average response time and number of results when one query loop has been completed
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl1.php',
    body: "str=Average response time = " + res_avg_data + " Number of results= " + count_data + "\n"
});

// Resetting the array
res_data = "";
}
searchMongoDB();
Appendix E - mongoDB_complexityQuery_lvl2.js

```javascript
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = "";

// Time variables
var currentYear = 2000;
var endYear = 2005;

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';

async function searchMongoDB() {
    var samples = [];
    for (var queryNr = 0; queryNr < 100; queryNr++) {
        // Use connect method to connect to the server
        var client = await MongoClient.connect(url);
        console.log("Connected, executing "+ queryNr + "/" + 100);
        const db = client.db(dbName);
        var articles = db.collection("mongoarticles");

        // Get the documents collection
        // Find some documents
        var req = await articles.find({
            "Filetype": 'html',
            "HTTP_Content_Length": {
                "$gte": 1000,
                "$lt": 20000
            }
        }), { _id: 1 });

        // Adds the explain specification onto the req variable
        var explain = await req.explain();

        // Puts the executionTimeMillis in the end of the array
        samples.push(explain.executionStats.executionTimeMillis);

        // Save the ordinary response time
        await client.close();
        await request.post({
            headers: { 'content-type': 'application/x-www-form-urlencoded' },
            url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl2.php',
        });
    }
}
```
```javascript
body: "str=Query nr: " + queryNr + ". Elapsed time = " +
        explain.executionStats.executionTimeMillis + "\r\n"
});
}

// Use connect method to connect to the server
var client = await MongoClient.connect(url);
const db = client.db(dbName);
var articles = db.collection("mongoarticles");

//Storing the number of results into an array
var res_data = await articles.find({
    "Filetype": 'html',
    "HTTP_Content_Length": {
        "$gte": 1000,
        "$lt": 20000
    }
},
    { _id: 1 }).toArray();

var count_data = res_data.length;
console.log("Number elements: " + count_data);

//Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--;)
    responseTimeSum += samples[i];

var res_avg_data = responseTimeSum / samples.length;

//Storing the average response time and number of results when one query loop
//has been completed
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl2.php',
    body: "str=Average response time = " + res_avg_data + ". Number of results=
        " + count_data + "\r\n"
});

//Resetting the array
res_data = "";
}
searchMongoDB();
```
Appendix F - mongoDB_complexityQuery_lvl3.js

const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = ""

// Time variables
var currentYear = 2000;
var endYear = 2005;

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';

async function searchMongoDB() {
    var samples = [];
    for (var queryNr = 0; queryNr < 100; queryNr++) {
        // Use connect method to connect to the server
        var client = await MongoClient.connect(url);
        console.log("Connected, executing " + queryNr + "/" + 100);
        const db = client.db(dbName);
        var articles = db.collection("mongoarticles");

        // Get the documents collection
        // Find some documents
        var req = await articles.find(
            {
                "Filetype": "html",
                "HTTP_Content_Length": {
                    "$gte": 1000,
                    "$lt": 20000
                },
                "HTTP_Archive_Time": {
                    "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
                    "$lt": new Date(endYear.toString().concat("-12-31T00:00:00.000Z"))
                }
            }, { _id: 1 });

        // Adds the explain specification onto the req variable
        var explain = await req.explain();
        // Puts the executionTimeMillis in the end of the array
        samples.push(explain.executionStats.executionTimeMillis);
        // Save the ordinary response time
        await client.close();
        await request.post({

```
headers: { 'content-type': 'application/x-www-form-urlencoded' },
url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl3.php',
    body: "str=Query nr: " + queryNr + ". Elapsed time = " +
            explain.executionStats.executionTimeMillis + "\r\n"
    });
}

// Use connect method to connect to the server
var client = await MongoClient.connect(url);
const db = client.db(dbName);
var articles = db.collection("mongoarticles");

//Storing the number of results into an array
var res_data = await articles.find(
    "Filetype": 'html',
    "HTTP_Content_Length": {
        "$gte": 1000,
        "$lt": 20000,
    },
    "HTTP_Archive_Time": {
        "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
        "$lt": new Date(endYear.toString().concat("-12-31T00:00:00.000Z"))
    }
), { _id: 1 }).toArray();

var count_data = res_data.length;
console.log("Number elements: " + count_data);

//Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--;)
    responseTimeSum += samples[i];
var res_avg_data = responseTimeSum / samples.length;

//Storing the average response time and number of results when one query loop
//has been completed
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl3.php',
    body: "str=Average response time = " + res_avg_data + ". Number of results="
            + count_data + "\r\n"
    });

//Resetting the array
res_data = "";
}
searchMongoDB();
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
var request = require('request');
var query = "";

// Time variables
var currentYear = 2000;
var endYear = 2005;

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'mongomydb';

var query = {
    "Filetype": 'html',
    "HTTP_Url": {
        "$regex": '.*skolverket\.se.*'
    },
    "HTTP_Content_Length": {
        "$gte": 1000,
        "$lt": 20000
    },
    "HTTP_Archive_Time": {
        "$gte": new Date(currentYear.toString().concat("-01-01T00:00:00.000Z")),
        "$lt": new Date(endYear.toString().concat("-12-31T00:00:00.000Z"))
    }
}

async function searchMongoDB() {
    var samples = [];
    for (var queryNr = 0; queryNr < 100; queryNr++) {

        // Use connect method to connect to the server
        var client = await MongoClient.connect(url);

        console.log("Connected, executing " + queryNr + "/" + 100);

        const db = client.db(dbName);
        var articles = db.collection("mongoarticles");

        // Get the documents collection
        // Find some documents
        var req = await articles.find(query, { _id: 1 });

        // Adds the explain specification onto the req variable
        var explain = await req.explain();

        // Puts the executionTimeMillis in the end of the array
        samples.push(explain.executionStats.executionTimeMillis);
    }
}

Appendix G - mongoDB_complexityQuery_lvl4.js
```javascript
// Save the ordinary response time
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl4.php',
    body: "str=Query nr: " + queryNr + ". Elapsed time = " + explain.executionStats.executionTimeMillis + \n"
});

// Use connect method to connect to the server
var client = await MongoClient.connect(url);
const db = client.db(dbName);
var articles = db.collection("mongoarticles");

// Storing the number of results into an array
var res_data = await articles.find(query, { _id: 1 }).toArray();
var count_data = res_data.length;
console.log("Number elements: " + count_data);

// Calculate the average from the samples array
var responseTimeSum = 0;
for (var i = samples.length; i--; ) {
    responseTimeSum += samples[i];
}
var res_avg_data = responseTimeSum / samples.length;

// Storing the average response time and number of results when one query loop has been completed
await client.close();
await request.post({
    headers: { 'content-type': 'application/x-www-form-urlencoded' },
    url: 'http://wwwlab.iit.his.se/b15matki/query_results/mongoDB_complexityQuery_lvl4.php',
    body: "str=Average response time = " + res_avg_data + ". Number of results= " + count_data + \n"
});

// Resetting the array
res_data = "";
}
searchMongoDB();
```
Appendix H - couchDB_query2.js

/**
 * @namespace http://tampermonkey.net/
 * @version 0.1
 * @description Generation of statistical data for CouchDB's query 2
 * @include http://localhost:5984/dbo/_design/testqueries/_view/query2*
 * @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
 * @author Mathias Kinnander
 * @match https://chrome.google.com/webstore/category/extensions
 * @grant GM_xmlhttpRequest
*/

console.log("Start");
var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_query2.php';
var scrapedData = [];
var responseTotalTime = "";
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

//Query specification

// Main function that calls the other functions
$(document).ready(function () {
    collectResponseTime();
    nextQuery();
});

// Saving response times
function collectResponseTime() {
    var requestStartTime = window.performance.timing.requestStart;
    var responseStartTime = window.performance.timing.responseStart;
    var responseTotalTime = responseStartTime - requestStartTime;
    console.log("Response time collected: " + responseTotalTime);
}
function nextQuery() {
  if (currentType === null || currentType == "undefined" || currentType == "NaN" || currentType == "") {
    currentType = 0;
    localStorage.setItem("LScurrentType", currentType);
    console.log("currentYear initialized");
  }
  console.log("Current Type: " + currentType);
  console.log("End Year: " + endYear);
  if (currentType < allTypes.length) {
    console.log("startYear < currentType");
    if (queryNr === null || queryNr == "undefined" || queryNr == "NaN" || queryNr == "") {
      //Increase the query number
      queryNr = 1;
      //Store the current query number
      localStorage.setItem("LSqueryNr", queryNr);
      localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
      console.log("queryNr === null || queryNr == undefined || queryNr == NaN || queryNr == ");
      responseTime();
      window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query2?startkey=%22" + currentYear + "-01-01T00:00:00Z%22&endkey=%22" + currentYear + "-12-31T00:00:00Z%22");
    } else if (queryNr < 100) {
      //If the current date is equal to the minimum start year proceed as usual
      LSallResponseTimes.push(responseTotalTime);
      localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
      //Increase the query number
      queryNr++;
      localStorage.setItem("LSqueryNr", queryNr);
      responseTime();
      //Replacing the current window
      window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query2?startkey=%22" + currentYear + "-01-01T00:00:00Z%22&endkey=%22" + currentYear + "-12-31T00:00:00Z%22");
    } else if (queryNr == 100) {
      //Summarize average times and
      var responseTimeSum = 0;
      for (var i = LSallResponseTimes.length; i--; ) {
        responseTimeSum = responseTimeSum + LSallResponseTimes[i];
      }
    }
  }
}
avgResponseTime = responseTimeSum / LSallResponseTimes.length;

endOfquery();

//Resetting the query
queryNr = "";
localStorage.setItem("LSqueryNr", queryNr);

//Incrementing the year variable
currentYear++;
localStorage.setItem("LScurrentYear", currentYear);

//Replacing the current window to the new query
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query2?startkey=%22" + currentYear + "-01-01T00:00:00Z%22&endkey=%22" + currentYear + "-12-31T00:00:00Z%22");
console.log("queryNR == 100");

//Store which year is being queried
function startOfQuery() {
    console.log("Querying: "+ currentYear);
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
    scrapedData.push("Querying: " + currentYear);
    ajaxCall(scrapedData);
}

//Extract the response time to the textfile
function responseTime() {
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
}
scrapedData.push("Query nr: " + queryNr + ". Elapsed time = " + responseTotalTime);
ajaxCall(scrapedData);

//Extract the average response time and the number of results
function endOfquery() {
  var count_data = "";
  function getACount() {
    count_data = document.body.innerText.split('key').length;
  }
  function ajaxCall(data) {
    try {
      GM_xmlhttpRequest({
        method: 'POST',
        url: localURL,
        data: 'str=' + encodeURIComponent(data) + '\r\n',
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded'
        },
      });
    } catch (ex1) {
      console.log(ex1);
    }
  }
  getACount();
  scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = " + count_data-1);
  ajaxCall(scrapedData);
}
Appendix I - couchDB_query3.js

```javascript
// ==UserScript==
// @name CouchDB query script 3
// @namespace http://tampermonkey.net/
// @version 1.0
// @description Generation of statistical data for CouchDB's query 3
// @include http://localhost:5984/dbo/_design/testqueries/_view/query3*
// @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author Mathias Kinnander
// @match https://chrome.google.com/webstore/category/extensions
// @grant GM_xmlhttpRequest
// ==/UserScript==

console.log("Start");
var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_query3.php';
var scrapedData = [];
var responseTotalTime = ""
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

// Query specification

// Main function that calls the other functions
$(document).ready(function () {
    collectResponseTime();
    nextQuery();
});

// Saving response times
function collectResponseTime() {
    var requestStartTime = window.performance.timing.requestStart;
    var responseStartTime = window.performance.timing.responseStart;
    responseTotalTime = responseStartTime - requestStartTime;
    console.log("Response time collected: " + responseTotalTime);
}
```
// TODO Run 10 times for each query, save to file if time i
function nextQuery() {
    if (currentType === null || currentType == "undefined" || currentType == "NaN" || currentType == "") {
        currentType = 0;
        localStorage.setItem("LScurrentType", currentType);
        console.log("CurrentType initialized");
    }
    console.log("Current Type: ", currentType);
    if (currentType < allTypes.length) {
        console.log("currentType < allTypes.length");
        if (queryNr === null || queryNr == undefined || queryNr == NaN || queryNr == ") {
            // Increase the query number
            queryNr = 1;
            // Store the current query number
            localStorage.setItem("LSqueryNr", queryNr);
            localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
            console.log("queryNr === null || queryNr == undefined || queryNr == NaN || queryNr == ");
            responseTime();
            window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query3?key=%22" + allTypes[currentType] + "/%22");
        } else if (queryNr < 5) {
            // If the current date is equal to the minimum start year proceed as usual
            LSallResponseTimes.push(responseTotalTime);
            localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
            // Increase the query number
            queryNr++;
            localStorage.setItem("LSqueryNr", queryNr);
            responseTime();
            // Replacing the current window
            window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query3?key=%22" + allTypes[currentType] + "/%22");
            console.log("else current is not equal to NULL");
        } else if (queryNr == 5) {
            // Summarize average times and
            var responseTimeSum = 0;
            for (var i = LSallResponseTimes.length; i--; ) {
                responseTimeSum = responseTimeSum + LSallResponseTimes[i];
            }
            avgResponseTime = responseTimeSum / LSallResponseTimes.length;
endOfquery();

//Resetting the query
queryNr = "";
localStorage.setItem("LSqueryNr", queryNr);

//Incrementing the year variable
currentType++;
localStorage.setItem("LScurrentType", currentType);

//Replacing the current window to the new query
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/query3?key=%22" + allTypes[currentType] + "%22");
console.log("queryNR == 5");
}

//Store which year is being queried
function startOfQuery() {
  console.log("Querying: " + currentType);
  function ajaxCall(data) {
    try {
      GM_xmlhttpRequest({
        method: 'POST',
        url: localURL,
        data: 'str=' + encodeURIComponent(data) + '\r\n',
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded'
        },
      });
    } catch (ex1) {
      console.log(ex1);
    }
  }
  scrapedData.push("Querying: " + currentType);
  ajaxCall(scrapedData);
}

//Extract the response time to the textfile
function responseTime() {
  console.log("query number response time");
  function ajaxCall(data) {
    try {
      GM_xmlhttpRequest({
        method: 'POST',
        url: localURL,
        data: 'str=' + encodeURIComponent(data) + '\r\n',
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded'
        },
      });
    } catch (ex1) {
      console.log(ex1);
    }
  }
  console.log(ex1);
scrapedData.push("Query nr: " + queryNr + ". Elapsed time = " + responseTotalTime);
ajaxCall(scrapedData);
}

// Extract the average response time and the number of results
function endOfQuery() {
  console.log("Average response time initiated");
  var count_data = "";
  function getACount() {
    console.log("Count the number of keys");
    count_data = document.body.innerText.split('key').length;
  }
  function ajaxCall(data) {
    console.log("Try to get an ajax call");
    try {
      GM_xmlhttpRequest({
        method: 'POST',
        url: localURL,
        data: 'str=' + encodeURIComponent(data) + '\r\n',
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded'
        },
      });
    } catch (ex1) {
      console.log(ex1);
    }
  }
  getACount();
  scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = " + count_data);
  ajaxCall(scrapedData);
}
Appendix J - couchDB_complexityQuery_lvl1.js

// ==UserScript==
// @name New Userscript
// @namespace http://tampermonkey.net/
// @version 0.1
// @description Generation of statistical data for CouchDB's complexity query level 1
// @include http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl1*
// @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author Mathias Kinnander
// @match https://chrome.google.com/webstore/category/extensions
// @grant GM_xmlhttpRequest
// ==/UserScript==

console.log("Start");
var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_complexityQuery_lvl1.php';
var scrapedData = [];
var responseTotalTime = "";
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

//Query specification
var startYear = 2000;
var endYear = 2005;

// Main function that calls the other functions
$(document).ready(function () {
    collectResponseTime();
    nextQuery();
});

// Saving response times
function collectResponseTime() {
    var requestStartTime = window.performance.timing.requestStart;
    var responseStartTime = window.performance.timing.responseStart;
    responseTotalTime = responseStartTime - requestStartTime;
    console.log("Response time collected: " + responseTotalTime);
}

//TODO Run 10 times for each query, save to file if time i
function nextQuery() {
    if (queryNr === null || queryNr == "undefined" || queryNr == "NaN" ||
        queryNr == "") {
        //Increase the query number
        queryNr = 1;
// Store the current query number
localStorage.setItem("LSqueryNr", queryNr);
localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
console.log("queryNr === null || queryNr == undefined || queryNr == NaN || queryNr == ");

responseTime();
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl1");
}
else if (queryNr < 100) {
  // If the current date is equal to the minimum start year proceed as usual
  LSallResponseTimes.push(responseTotalTime);
  localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
  // Increase the query number
  queryNr++;
  localStorage.setItem("LSqueryNr", queryNr);
  responseTime();

  // Replacing the current window
  window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl1");
  console.log("else current is not equal to NULL");
} else if (queryNr == 100) {

  // Summarize average times and
  var responseTimeSum = 0;
  for (var i = LSallResponseTimes.length; i--;)
  {
    responseTimeSum = responseTimeSum + LSallResponseTimes[i];
  }
  avgResponseTime = responseTimeSum / LSallResponseTimes.length;

  endOfquery();

  // Resetting the query
  queryNr = "";
  localStorage.setItem("LSqueryNr", queryNr);

  // Incrementing the year variable

  // Replacing the current window to the new query
  console.log("queryNR == 100");
}

// Store which year is being queried
function startOfQuery() {
  function ajaxCall(data) {
    try {
      // AJAX call code here
    } catch (error) {
      console.log("Error: ", error);
    }
  }

  // AJAX call code here
}

// AJAX call code here

GM_xmlhttpRequest({
    method: 'POST',
    url: localURL,
    data: 'str=' + encodeURIComponent(data) + '\n',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
    },
});
} catch (ex1) {
    console.log(ex1);
}
}
scrapedData.push("Querying years 2000 - 2005 ");
ajaxCall(scrapedData);

//Extract the response time to the textfile
function responseTime() {
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
        scrapedData.push("Query nr: "+ queryNr + ". Elapsed time = " + responseTotalTime);
        ajaxCall(scrapedData);
    }
}

//Extract the average response time and the number of results
function endOfquery() {
    var count_data = "";
    function getACount() {
        count_data = document.body.innerText.split('key').length;
    }
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
}
console.log(ex1); } 

getACount(); 
scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = " + count_data-1); 
ajaxCall(scrapedData); 
}
Appendix K - couchDB_complexityQuery_lvl2.js

```
// ==UserScript==
// @name New UserScript
// @namespace http://tampermonkey.net/
// @version 0.1
// @description Generation of statistical data for CouchDB's complexity query level 2
// @include http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl2*
// @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author Mathias Kinnander
// @match https://chrome.google.com/webstore/category/extensions
// @grant GM_xmlhttpRequest
// ==/UserScript==

console.log("Start");

var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_complexityQuery_lvl2.php';
var scrapedData = [];  
var responseTotalTime = "";
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

//Query specification
var startYear = 2000;
var endYear = 2005;

// Main function that calls the other functions
$(document).ready(function () {
  collectResponseTime();
  nextQuery();
});

// Saving response times
function collectResponseTime() {
  var requestStartTime = window.performance.timing.requestStart;
  var responseStartTime = window.performance.timing.responseStart;
  responseTotalTime = responseStartTime - requestStartTime;
  console.log("Response time collected: " + responseTotalTime);
}

//TODO Run 10 times for each query, save to file if time i
function nextQuery() {
  if (queryNr === null || queryNr == "undefined" || queryNr == "NaN" ||
      queryNr == "") {
    //Increase the query number
    queryNr = 1;

```
// Store the current query number
localStorage.setItem("LSqueryNr", queryNr);
localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
console.log("queryNr === null || queryNr == undefined || queryNr == NaN");
responseTime();
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl2");
}
else if (queryNr < 100) {
    // If the current date is equal to the minimum start year proceed as usual
    LSallResponseTimes.push(responseTotalTime);
    localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
    // Increase the query number
    queryNr++;
    localStorage.setItem("LSqueryNr", queryNr);
    responseTime();
}
else if (queryNr == 100) {
    // Summarize average times and
    var responseTimeSum = 0;
    for (var i = LSallResponseTimes.length; i--; ) {
        responseTimeSum = responseTimeSum + LSallResponseTimes[i];
    }
    avgResponseTime = responseTimeSum / LSallResponseTimes.length;
    endOfquery();
    // Resetting the query
    queryNr = "";
    localStorage.setItem("LSqueryNr", queryNr);
    // Incrementing the year variable
    // Replacing the current window to the new query
    console.log("queryNR == 100");
}
else if (queryNr == NaN) {
    // Store which year is being queried
    function startOfQuery() {
        function ajaxCall(data) {
            try {
                // ...
GM_xmlhttpRequest({
    method: 'POST',
    url: localURL,
    data: 'str=' + encodeURIComponent(data) + '\n',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
    },
});
} catch (ex1) {
    console.log(ex1);
}

scrapedData.push("Querying years 2000 - 2005");
ajaxCall(scrapedData);

// Extract the response time to the textfile
function responseTime() {
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
    scrapedData.push("Query nr: " + queryNr + ". Elapsed time = "+
    responseTotalTime);
    ajaxCall(scrapedData);
}

// Extract the average response time and the number of results
function endOfquery() {
    var count_data = "";
    function getACount() {
        count_data = document.body.innerText.split('key').length;
    }
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
}
console.log(ex1);
}

getACount();
scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = " + count_data-1);
ajaxCall(scrapedData);
Appendix L - couchDB_complexityQuery_lvl3.js

// ==UserScript==
// @name     New Userscript
// @namespace http://tampermonkey.net/
// @version 0.1
// @description Generation of statistical data for CouchDB's complexity query level 3
// @include  http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl3*
// @require  http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author   Mathias Kinnander
// @match    https://chrome.google.com/webstore/category/extensions
// @grant    GM_xmlhttpRequest
// ==/UserScript==

console.log("Start");
var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_complexityQuery_lvl3.php';
var scrapedData = [];
var responseTotalTime = ";
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

//Query specification
var startYear = 2000;
var endYear = 2005;

// Main function that calls the other functions
$(document).ready(function () {
  collectResponseTime();
  nextQuery();
});

// Saving response times
function collectResponseTime() {
  var requestStartTime = window.performance.timing.requestStart;
  var responseStartTime = window.performance.timing.responseStart;
  responseTotalTime = responseStartTime - requestStartTime;
  console.log("Response time collected: " + responseTotalTime);
}

//TODO Run 10 times for each query, save to file if time i
function nextQuery() {
  if (queryNr === null || queryNr == "undefined" || queryNr == "NaN" || queryNr == "") {
    //Increase the query number
    queryNr = 1;
  }
}
//Store the current query number
localStorage.setItem("LSqueryNr", queryNr);
localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
console.log("queryNr == null || queryNr == undefined || queryNr == NaN || queryNr == ");

responseTime();
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl3?startkey=%222000-01-01T00:00:00Z%22&endkey=%222005-12-31T00:00:00Z%22");
}
else if (queryNr < 100) {
    //If the current date is equal to the mimimum start year proceed as usual
    LSallResponseTimes.push(responseTotalTime);
    localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
    //Increase the query number
    queryNr++;
    localStorage.setItem("LSqueryNr", queryNr);

    responseTime();

    //Replacing the current window
    window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl3?startkey=%222000-01-01T00:00:00Z%22&endkey=%222005-12-31T00:00:00Z%22");
    console.log("else current is not equal to NULL");
} else if (queryNr == 100) {
    //Summarize average times and
    var responseTimeSum = 0;
    for (var i = LSallResponseTimes.length; i--; ) {
        responseTimeSum = responseTimeSum + LSallResponseTimes[i];
    }
    avgResponseTime = responseTimeSum / LSallResponseTimes.length;

    endOfquery();

    //Resetting the query
    queryNr = "";
    localStorage.setItem("LSqueryNr", queryNr);

    //Incrementing the year variable

    //Replacing the current window to the new query
    console.log("queryNR == 100");
}

//Store which year is being queried
function startOfQuery() {
function ajaxCall(data) {
    try {
        GM_xmlhttpRequest({
            method: 'POST',
            url: localURL,
            data: 'str=' + encodeURIComponent(data) + '\n',
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded'
            },
        });
    } catch (ex1) {
        console.log(ex1);
    }
    scrapedData.push("Querying years 2000 - 2005 ");
    ajaxCall(scrapedData);
}

//Extract the response time to the textfile
function responseTime() {
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
        scrapedData.push("Query nr: " + queryNr + ". Elapsed time = " + responseTotalTime);
    } ajaxCall(scrapedData);
}

//Extract the average response time and the nume of results
function endOfquery() {
    var count_data = ";
    function getACount() {
        count_data = document.body.innerText.split('key').length;
    }
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    } ajaxCall(scrapedData);
```javascript
})
   } catch (ex1) {
       console.log(ex1);
   }

} getACount();
scrapedData.push("Average response time = " + avgResponseTime + "). Number of results = " + count_data-1);
ajaxCall(scrapedData);
```
Appendix M - couchDBComplexityQuery_lvl4.js

a // ==UserScript==
// @name New Userscript
// @namespace http://tampermonkey.net/
// @version 0.1
// @description Generation of statistical data for CouchDB's complexity query level 4
// @include http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl4*
// @require http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js
// @author Mathias Kinnander
// @match https://chrome.google.com/webstore/category/extensions
// @grant GM_xmlhttpRequest
// ==/UserScript==

console.log("Start");
var localURL = 'http://wwwlab.iit.his.se/b15matki/query_results/couchDB_complexityQuery_lvl4.php';
var scrapedData = [];
var responseTotalTime = "";
var queryNr = localStorage.getItem("LSqueryNr");
var currentType = localStorage.getItem("LScurrentType");
var avgResponseTime = 0;
var LSallResponseTimes = JSON.parse(localStorage.getItem("LSallResponseTimes"));

// Query specification
var startYear = 2000;
var endYear = 2005;

// Main function that calls the other functions
$(document).ready(function () {
    collectResponseTime();
    nextQuery();
});

// Saving response times
function collectResponseTime() {
    var requestStartTime = window.performance.timing.requestStart;
    var responseStartTime = window.performance.timing.responseStart;
    responseTotalTime = responseStartTime - requestStartTime;
    console.log("Response time collected: " + responseTotalTime);
}

function nextQuery() {
    if (queryNr == null || queryNr == "undefined" || queryNr == "NaN" || queryNr == ") { //Increase the query number
        queryNr = 1;
    }
// Store the current query number
localStorage.setItem("LSqueryNr", queryNr);
localStorage.setItem("LSallResponseTimes", JSON.stringify([]));
console.log("queryNr == null || queryNr == undefined || queryNr == NaN || queryNr == ");

responseTime();
window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl4?startkey=%222000-01-01T00:00:00Z%22&endkey=%222005-12-31T00:00:00Z%22");

} else if (queryNr < 100) {
    // If the current date is equal to the minimum start year proceed as usual
    LSallResponseTimes.push(responseTotalTime);
    localStorage.setItem("LSallResponseTimes", JSON.stringify(LSallResponseTimes));
    // Increase the query number
    queryNr++;
    localStorage.setItem("LSqueryNr", queryNr);

    responseTime();
    // Replacing the current window
    window.location.replace("http://localhost:5984/dbo/_design/testqueries/_view/complexityQuery_lvl4?startkey=%222000-01-01T00:00:00Z%22&endkey=%222005-12-31T00:00:00Z%22");
    console.log("else current is not equal to NULL");
} else if(queryNr == 100) {

    // Summarize average times and
    var responseTimeSum = 0;
    for (var i = LSallResponseTimes.length; i--; ) {
        responseTimeSum = responseTimeSum + LSallResponseTimes[i];
    }
    avgResponseTime = responseTimeSum / LSallResponseTimes.length;

    endOfQuery();

    // Resetting the query
    queryNr = "";
    localStorage.setItem("LSqueryNr", queryNr);

    // Incrementing the year variable

    // Replacing the current window to the new query
    console.log("queryNR == 100");
}

// Store which year is being queried
function startOfQuery() {
    function ajaxCall(data) {

try {
    GM_xmlhttpRequest({
        method: 'POST',
        url: localURL,
        data: 'str=' + encodeURIComponent(data) + '\r\n',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        },
    });
} catch (ex1) {
    console.log(ex1);
}

scrapedData.push("Querying years 2000 - 2005 ");
ajaxCall(scrapedData);

//Extract the response time to the textfile
function responseTime() {
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\r\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
    scrapedData.push("Query nr: " + queryNr + ". Elapsed time = " + responseTotalTime);
    ajaxCall(scrapedData);
}

//Extract the average response time and the number of results
function endOfquery() {
    var count_data = "";
    function getACount() {
        count_data = document.body.innerText.split('key').length;
    }
    function ajaxCall(data) {
        try {
            GM_xmlhttpRequest({
                method: 'POST',
                url: localURL,
                data: 'str=' + encodeURIComponent(data) + '\r\n',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded'
                },
            });
        } catch (ex1) {
            console.log(ex1);
        }
    }
}
catch (ex1) {
    console.log(ex1);
}

getACount();
scrapedData.push("Average response time = " + avgResponseTime + ". Number of results = "+ count_data-1);
ajaxCall(scrapedData);