Blekinge Institute of Technology

Bachelor thesis, computer science
DV1322

Document Oriented NoSQL Databases

A comparison of performance in MongoDB and CouchDB using a Python interface

Author:
Robin Henricsson

Supervisor:
Göran Gustafsson

June 13, 2011

Contents

Abstract

For quite some time relational databases, such as MySQL, Oracle and Microsoft SQL Server, have been used to store data for most applications. While they are indeed ACID compliant (meaning interrupted database transactions won't result in lost data or similar nasty surprises) and good at avoiding redundancy, they are difficult to scale horizontally (across multiple servers) and can be slow for certain tasks. With the Web growing rapidly, spawning enourmous, user-generated content websites such as Facebook and Twitter, fast databases that can handle huge amounts of data are a must. For this purpose new databases management systems collectively called NoSQL are being developed. This thesis explains NoSQL further and compares the write and retrieval speeds, as well as the space efficiency, of two database management systems from the document oriented branch of NoSQL called MongoDB and CouchDB, which both use the JavaScript Object Notation (JSON) to store their data within.

The benchmarkings performed show that MongoDB is quite a lot faster than CouchDB, both when inserting and querying, when used with their respective Python libraries and dynamic queries. MongoDB also is more space efficient than CouchDB.

Keywords

MongoDB, CouchDB, Python, pymongo, couchdb-python, NoSQL, Document database, JSON, DBMS, Database

Forewords

I would very much like to thank my supervisor Göran Gustafsson, who was kind enough to take on the role of supervisor even though he did not give any courses at the university during the first part of my work on the thesis. His helpful advice has helped me alot during my work.

I would also like to thank Dr Niklas Lavesson for giving me access to the university's amazing computer *beast* and for helping me install everything I needed despite his busy schedule.

Lastly, I want to thank my dear friend Peter Assmus for letting me install Linux on, and use, his computer when mine did not suffice.

1 Introduction

1.1 Glossary of terms

1.1.1 Data

Information and data are two words that are sometimes used interchangeably to describe bits of knowledge, or maybe rather the representation of that knowledge itself. However, the two words actually have different meanings.

Raw data is basically numbers (or some other symbols) that by themselves mean nothing. For example, "0" might be a piece of data. Of course the number zero has a certain meaning when in a context — it could be the number of times an individual has been married or perhaps the temperature in some unit of measure — but by itself it is insignificant.

1.1.2 Information

Information is data that has been given a meaning through some type of connection. It answers questions like "where?", "who?" and "how many?". For example, the number zero would be information if it is an answer to a question such as "how many times has Sweden been invaded by France?". In other words, information can be said to be data in a context.

Databases store data that the software can present in a meaningful (not necessarily useful) way, turning it into information that means something to us, the wielders of the human mind.

1.1.3 NULL

NULL is a value that is used in many databases and programming languages to represent the lack of a "real" value. It basicaly means "nothing".

1.1.4 Semi-structured data

Semi-structured data is data that is structured in some way that lets attributes be optional. Imagine a simple table, which is strictly structured data. Each column represents a specific attribute, and so every row in the table needs a value for every column (even if the value is *NULL*).

Now, imagine something like an XML document, which uses starting and ending tags to separate keys from values. If there is no value for a certain key, the entire element (starting tag, value, ending tag) is simply left out of the document. Structuring data in such a way that some attributes can truly be optional is making it semi-structured.

1.1.5 ACID

ACID, which is an abbreviation of Atomicity, Consistency, Isolation, Durability, is a set of properties that many relational database management systems possess. It basically means that transactions in databases are reliably processed. This might as an example mean that a bank transaction, which includes withdrawal from one account, insertion into another account and recording of the transaction itself into yet another table, is all treated as one atomic operation that either is fully completed, or not completed at all. That way, if the power goes out (or if the transaction fails for some other reason), no money is lost and no data corrupt.

1.1.6 URI

URI, Uniform Resource Identifier, is a string of characters used primarily on the Internet to address resources. An example of a simple URI is `http://robinhenricsson.se`.

1.1.7 Web 2.0

The Web 2.0 is a somewhat diffuse term that describe the new Web that is emerging, where more and more sites include user-generated content (Facebook, Twitter, Flickr) and where different sites and social medias can connect to eachother (think about it, how many blogs, newspapers etc. *don't* allow Facebook connectivity in one way or another?).

1.1.8 Plain text file

A plain text file, sometimes called flat file, is a file that contains only a stream of bytes which can be read without processing. Plain text files are human-readable, as opposed to binary files. They can be created and edited using a wide selection of text editors, such as Notepad for Windows, emacs or vim for UNIX, etc. Source code for software is almost always written in plain text.

1.2 Background

1.2.1 Databases today

There are several different ways to persistently store data and offer its efficient retrieval at later times. Probably the most common way is by using a *database management system* (*DBMS*), of which there are many, all differing from the others in some way. Some DBMS are very alike, and diverge only by small features, while others use entirely different data models. For some time, the standard model has been the *relational model* (which was presented in the 1970s by Edgar Codd), using tables for data storage with keys making every row uniquely addressable. These relational database management systems (*RDBMS*) typically use a query language called *Structured Query Language* (*SQL*) to fetch, modify and sava data.

Attempts at popularizing other kinds of databases, such as the object oriented database, which looks to simplify the transition between the widely used object oriented programming paradigm and the database world, have been made over the years, but have not been very successful at replacing the relational model. More recently however, with the rising of Web 2.0, databases need be able to scale vastly. RDBMS are easily able to do so vertically, by adding more hardware resources to the existing server containing the database, but scaling horizontally over multiple servers can be quite a challenge. To address this problem an array of new, non relational DBMS have been (and are still being) developed, and with big names like Google, Facebook and Amazon behind some of them, things are starting to get interesting.

1.2.2 Tomorrow's NoSQL databases

The new non relational DBMS that are being developed are called NoSQL databases. The name NoSQL implies a lack of SQL, which can be confusing since there are relational databases that do not use SQL. Indeed NoSQL DBMS do not use SQL, but this is just an effect of abandoning the relational model altogether. In other words, RDBMS that do not use SQL are not NoSQL. DBMS using models other than the relational one, and hence not SQL, are NoSQL. A NoSQL DBMS is simply a DBMS that does not use the relational model.

NoSQL databases often have in common the ability to easily scale horizontally as well as being non relational and not requiring fixed schemas. The lack of relations usually means that JOIN operations, which in SQL are common, here are superfluous. Other than that, different NoSQL implementations can look very different. There are, however, four main emerging subsets of NoSQL, catagorized by their different manners of storing data:

- Document database
- Graph database
- Key-value database
- Tabular database

This thesis will focus primarily on the document database model, investigating, benchmarking and comparing two different open source document DBMS; Apache Software Foundation's[1] *CouchDB* and 10gen's[2] *MongoDB*.

1.3 Research question

When using their respective Python drivers, which DBMS, MongoDB or CouchDB, is the fastest using ad-hoc queries, and which is more space efficient than the other?

1.4 Hypothesis

MongoDB is known for being fast, so the tests will likely show that CouchDB is a bit slower. What is interesting is how much slower it is. CouchDB might also be faster at certain tasks.

Since CouchDB uses revision control, its databases likely occupy more space than MongoDB's.

1.5 Question formulations

How fast is CouchDB and MongoDB respectively, using a single server setup with their respective Python libraries for executing ad-hoc queries, when

- inserting
  - 5,000 documents?
  - 50,000 documents?
  - 500,000 documents?
- runnings queries against
  - 5,000 documents?
  - 50,000 documents?
  - 500,000 documents?

How much disk space does CouchDB and MongoDB use respectively on a single server setup, when they contain

- 5,000 documents?
- 50,000 documents?
- 500,000 documents?

1.6 Goal and purpose

The goal is to present a clear picture of what NoSQL means, and to produce statistics about the performance and efficiency of CouchDB and MongoDB respectively and in comparison of eachother.

Comparisons of different DBMS have been made before, but a published comparison of performance between MongoDB and CouchDB is nowhere to be found. This is mainly what will make this thesis unique, and is of interest since MongoDB and CouchDB are direct competitors and the two major names of the document oriented model.

1.7 Target audience

The target audience for this thesis is anyone coming from the SQL world, trying to get a grasp on the diffuse term NoSQL or, more specifically, document oriented databases. It is also for anyone having a hard time deciding between using CouchDB and MongoDB for a specific task. Since everything of importance

[1] Apache Software Foundation is a foundation that legally and financially suppports software projects. It is perhaps most known for the Apache Web Server. Read more at `http://www.apache.org/foundation/`

[2] 10gen is an American company that develops, and offers support and training for, MongoDB. Read more at `http://www.10gen.com/about`

will be discussed and explained in the thesis, it could likely be read and understood by persons with no earlier experience in databases as well.

1.8 Delimitations

This thesis will not cover all document oriented databases. Instead, MongoDB or CouchDB, two of the largest names of the category, have been chosen. None of these systems will be reviewed "under the hood". In other words, the code behind the systems will not be evaluated or optimized in any way.

The testing of retrieval speeds for MongoDB and CouchDB will be performed using a single server setup, even though both allow for effective horizontal scaling.

Most information will be collected from Web resources, since the topic is new enough to not be the subject of more than a few articles and books.

1.9 Method

Information will be collected via BTH's library (both online, by searching different databases, and offline by books) and via Web searches.

The benchmarking of CouchDB and MongoDB will be performed using homebrewn applications written in the programming language Python, which has libraries available for both systems. The data used will be pseudo-randomly generated strings and numbers, and the same set of data will be used for both DBMS.

All software used for running the benchmarking tests herein is listed below. All software has been run under Ubuntu GNU/Linux 11.04, 32bit version.

- **Python** has been used to create scripts that generate random documents and communicate with MongoDB and CouchDB.
- **PyMongo** has been used to allow Python to communicate with MongoDB. `http://api.mongodb.org/python/1.11/`
- **couchdb-python** has been used to allow Python to communicate with CouchDB. `http://code.google.com/p/couchdb-python/`

2 Storing data

There are many different ways to store data on secondary storage. The easist way would be simply dumping all data into a plain text, or binary, file. Of course, simply storing data without offering flexible retrieval at later times is of little to no use. To fetch stored data one must know how the data is structured, and then use a readily available algorithm, or write a new one, designed for that particular way of storing. One could design own models of storage complete with own algorithms for retrieval, but doing so consumes time and makes life a lot harder for anyone else trying to fetch the stored data.

Thankfully there are several standardized ways of structuring data for storage and for interchange between information systems. It is important to note that these are merely models of structuring data for storage. Formatting data to fit one of these models and then fetching it has to be done separately. Many programming languages provide function for handling these ways of storage, however.

Plain text files are human-readable and easy to pass on networks and through firewalls, since they are plain text without any potentially dangerous executables. The most common data storage and interchange techniques using flat files are explained below.

2.1 CSV

Comma-Separated Values, or *CSV*, is a simple way of storing data in a plain text file. The data model is very easy to understand, as it is basically a table where every row is an object and is made up of columns, which represent the objects' attributes. Columns are separated by commas, just like the name implies. Depending on the implementation, however, any other character (such as the semicolon) could be used as separator. A simple example of a short CSV file follows below. Please note that the first line describing attributes is optional, but a good idea to include for making the file readable, and has to be taken into account when developing an algorithm for writing to, and reading from, the file. Note that this is the reason for the lack of commas between the values of the first line — they are not supposed to be parsed and are only there to make the file more readable for humans.

```
Product no.  Color   Cost
5347,        Blue,   250
5348,        Green,  35
5349,        Red,    1050
```

CSV is a simple and in many cases adequate way of representing data. It is used mainly for storing small amounts of data, and for transferring data from one information system to another. For example, a spreadsheet in an application such as Microsoft Excel can be saved as a CSV file, and then imported into Google Mail's contacts.[9] However, since the files are plain text, any text editor could be used to create them.

Even though CSV files are quite capable despite their simplicity, it does have some crucial restrictions. Let's use the above CSV file as an example again. Let's say we want to add a field to store the owner of the product. Doing so is straightforward and not a problem. However, what if certain products don't have an owner? We could use a value like *null* or *void* to mark the lack of a value. It's not pretty, but it works. The real problem arises if we want some products to be able to have more than one owner. The example below displays the problem.

```
Product no.   Color    Cost    Owner 1   Owner2
5347,         Blue,    250,    Robin,    Alex
5348,         Green,   35,     John,     NULL
5349,         Red,     1050,   NULL,     NULL
```

The file would be even more bloated and ridicilous if one product has a higher number of owners, like ten, and all other products only one.

The problem could possibly be solved by allowing semicolon separated lists as values. This, however, would not be standard CSV and would require an unnecessary amount of time spent on developing algorithms to handle such files.

2.2 XML

Extensible Markup Language (abbreviated *XML*) is another way of creating plain text files that are machine-readable, while still being perfectly human-readable. It is based on the *Standard Generalized Markup Language* (*SGML*) which is an ISO standard (ISO 8879:1986[8]). Different XML implementations are used for documents in applications such as Microsoft Office[10] and OpenOffice.org[16] but also for SVG images[6] in applications such as Inkscape.

XML documents consist of *elements*. An element is made up of a start tag, some contents (which may be plain text or other XML elements), and finally a closing tag. A tag is simply a keyword surrounded by less-than and greater-than signs. The closing tag of an element uses the same keyword as the starting tag, but with a slash between the less-than sign and keyword.

A minimal XML document might look some-

thing like this:

```
<?xml version="1.0"?>
<color>Blue</color>
```

The first line describes which version of XML that is being used. After this follows a single element which describes something as being blue. The fundamental building blocks of XML documents are easily grasped and by nesting them, complex documents can be created. A more realistic example of XML, this time in the form of a letter, follows:

```
<?xml version="1.0"?>
<letter language="english">
    <recipient>Linus Mostberg</recipient>
    <subject>About these pants..</subject>
    <message>
        Dear mr Mostberg,
        These pants are giving me hallucinations.
        Could I possibly return them to you?
        Thanks.
    </message>
</letter>
```

As the above example clearly shows, XML is very easy to understand. Due to its structure it is also easily parsed. Note that start tags can include attributes with values, such as `language="english"` in the letter. Multiple attributes for tags is also allowed. Also note that the `recipient`, `subject` and `message` elements are contained inside the `letter` element; elements may be nested, allowing for complex documents.

2.3 JSON

JSON, *JavaScript Object Notation*, is another easily understood way of storing and interchanging data, and is a lightweight way of doing so. It is based on the JavaScript scripting language, which is commonly used on the client-side for creating dynamic web pages.

The building blocks of JSON documents are *key-value pairs*. A key is something unique for which some information is available, and its corresponding value stores this information. For example, for the key `color` the value could be `blue`. A value can be a string, an integer, a floating point number and even a list of values or a nested JSON document.

A short JSON document modeling a blue car with four wheels:

```
{
    "color": "blue",
    "wheels": 4
}
```

As can be seen in the example above, different key-value pairs are separated by commas. Below is another example of a JSON document, this time representing a person, showcasing some of the different data types JSON has support for.

```
{
    "firstName": "John",
    "lastName": "Håkansson",
    "age": 20,
    "favouriteFoods": [
        "Banana",
        "Beer",
        "Haggis"
    ],
    "height": 1.93,
    "phone": [
        {
            "type": "Home",
            "number": "12345"
        },
        {
            "type": "Cellular",
            "number": "54321"
        }
    ],
    "married": false
}
```

The name of the person consists of two strings, the age is an integer, the favourite foods is a list of strings, the height is a floating point number, the phone numbers is made up of two self-contained JSON documents and the

marital status is a boolean value. JSON documents are easy to understand and most major programming languages have support for it.

3 Databases

A database is, simply put, structured and related data that can rapidly and easily be retrieved even when there is large amounts of it. A CSV, XML or JSON file (as described in previous chapter *Storing data*) can by this definition be considered to be databases in their own rights. These kind of databases are sometimes called *flat file databases*. Even a piece of paper with a drawn table, or a collection of papers in an archive, is a form of database, as long as the data written on the papers is structured in some way.

Today, when speaking of databases, usually this refers to digital databases, handled by *database management systems* (DBMS). This section covers the definition of DBMS and some of the data models used by different DBMS.

3.1 Database Management Systems

3.1.1 Relational model

DBMS utilizing the relational model has for some time been the most common choice for storing data.[12, p4] Examples of relational DBMS (RDBMS) are MySQL, PostgreSQL, Oracle, Microsoft SQL server and SQLite.

3.1.2 SQL

SQL (Structured Query Language) is a non-procedural, fourth generation language (as opposed to C, Java etc. which are third generation languages) that has become the standard language for use in managing data in relational database managament systems. It was developed in the 1970s by IBM, which is where Edger Codd was working as a researcher when he presented the relational model in his paper *A Relational Model of Data for Large Shared Data Banks* (1970).[5, p114]

Although SQL is not a single standard, but rather a set of standards such as SQL92, SQL:1999 and SQL:2003, it is the most widespread language for structuring, altering, adding and deleting data in relational databases. All major RDBMS use SQL implementations: MySQL, Microsoft SQL server, PostgreSQL, Oracle etc. This ensures portability; data can easily be migrated from one RDBMS to another, and a programmer with experience in a particular RDBMS can almost seamlessly start working with another one.

As previously stated, SQL is a fourth generation language. This means that programmers only need to specify *what* they want done, without having to specify the details of *how*.[3, pXXIV] It uses natural words such as *SELECT*, *UPDATE*, *ALTER* and *DELETE* which renders it fairly easy to understand. For example, the statement `SELECT name FROM persons WHERE age > 18;` fetches the names of all persons older than 18 from the table `persons`.

3.1.3 NoSQL

3.1.3.1 Definition NoSQL is a rather diffuse term. There is no organisation solely behind the term, there is no official website describing it and a good, substantial definition is hard to come by. From the name, one could possibly elicit the fact that NoSQL is not SQL, but not much more than this. So what exactly is NoSQL?

In 1998, a relational database management system, based on another database system called RDB, was developed by Italian Carlo Strozzi and released under the GNU General Public License. He chose to call his creation NoSQL, since it intentionally avoided the use

of SQL to structure and modify data. Instead, NoSQL stored data on a file system in the form of plain text files, where every file was considered a table, complete with columns and rows. Regular unix commands could then be used to modify and structure the files. To read data, UNIX input and output redirection and pipes were used on the files.[14]

The subject of NoSQL, when discussed today, has very little to do with Strozzi's RDBMS. The term was introduced again in 2009, when the company Last.fm arranged an event to discuss distributed, structured storage in the form of new DBMS using other data models than the traditional, relational one. They called the event NOSQL (all capitalized).[7] The term (with the spelling NoSQL, however) has since been used to bundle databases that use other models than the relational model, and that hence do not use SQL. In other words, NoSQL does not describe what a database management system *is*, but rather what it *is not*.

Apart from not using the relational model, NoSQL databases are often open source (which to be fair many RDBMS are, too), schemaless, avoiding *JOIN*-like operations thanks to self-contained data, easily replicated and able to scale to humongous sizes. They typically do not guarantee *ACID* support, which is a common feature of RDBMS.

The name NoSQL might seem like a strange choice, partly because there is already a DBMS out there with the same name, which ironically is relational, and partly because it does not express what the term really means. Strazzo himself has commented on this on his webpage, stating:

> While the former [Strazzo's DBMS] is a well-defined software package, is a relational database to all effects and just does intentionally not use SQL as a query language, the newcomer is mostly a concept (and by no means a novel one either), which departs from the relational model altogether and it should therefore have been called more appropriately "NoREL", or something to that effect.[14]

3.1.3.2 Why NoSQL? When discussing if there really is a need for new, non-relational DBMS, the expression *one size doesn't fit all* comes to mind. Relational databases have been used for a large array of very different applications for a long time. Database courses usually revolve around relational databases and books covering subjects such as PHP often include chapters about connecting to MySQL (or some other RDBMS). The relational model is often used regardless of the application, programming language and nature of the data.

Using RDBMS, and hence tables and relationships, to store data might not always be the optimal solution. Data that is not normalized and that does not conform to the relational model may have to be restructured. Developing code to do this takes time. It also slows down the code since reformatting is needed every time data is to be saved to, or read from, the RDBMS. Information about real world objects are not stored optimally in relations.[5, p809]

One kind of data that obviously does not fit naturally into tables is unstructured data such as pictures or sound files. This type of data can often be stored in RDBMS in the form of *BLOBs*, or *Binary Large Objects*. However, the DBMS knows nothing of the data's internal structure since it usually stores a reference to the file and not the file itself, rendering it impossible to fetch parts of or modify a file. An alternative is storing files on a file system outside of the confinements of the database, and letting the database contain paths to the files. This works, but since the security mecha-

nisms of DBMS do not apply to the underlying file system, one has to make sure that is has the correct permissions set. This is undesirable since it means extra work for the database manager.[5, p811]

The relational model uses a static schema. This means that homogenous data is required. If the data to be stored is of no such nature, relations have to be split into new relations and JOIN operations have to be used to puzzle them together (problems with JOIN operations are discussed further later on). If the schema is to be changed, ALTER operations have to be performed along with changes to any applications working against the database. The ALTER statement has to be performed by a database administrator and is a slow operation.[5, 813]

Today many popular programming languages, such as Java, C++ and Python, are at least partially OO (*object oriented*). The set of problems related to mapping objects in an OO programming language to tables in an RDBMS is known as *object-relational impedance mismatch*. One of the problems relates to the encapsulation of objects that is recommended by the object oriented programming philosophy. An object may have private and public members and the only way to reach the private members is by using an interface. For example, the class Person may have a private member called *name*, which from outside of the class and its subsequent objects can only be accessed by public function *getName()*. The relational model does not include the same kind of private/public access, which results in the encapsulation of data being lost while in a relational database. A column in the database called *access* or something else to that effect is one possible solution to this, even if the access of attributes is not really part of the data itself, but rather information about the data (so-called *metadata*) and might thus be awkward to store this way.

Data types may differ in RDBMS and OO. Some types are handled differently while others exist only in one of the two. For example, strings in OO often include whitespaces as the last character(s) when compared, while RDBMS do not, instead stripping the trailing whitespace.

Relational databases also suffer from scaling problems. Scaling a relational database is commonly a matter of buying more powerful hardware, upgrading the server the database is running on. Upgrading "upwards" in this way, on a single server, is known as *vertical scaling*. There comes a point where the biggest and baddest hardware is in place, but where the database needs to be able to scale even more. At this point the database needs to be spread out across multiple servers. This is called *horizontal scaling* and is one particular area where RDBMS are lacking.[12, p7] Oracle has a solution for this called Real Application Clusters, but Oracle can be very expensive and is therefore not a real alternative for everyone.

The relational model, with its references to primary keys for records in other tables and germane JOIN operations, is excellent for avoiding redundancy, but is also one of the reasons RDBMS do not scale (horizontally) well; its performance optimizations assume that all data is in one place[12, p7]. Another inherent problem of JOIN operations is its costly nature. JOINs are the most time-consuming operations in SQL.[5, p654]

Many of the problems of RDBMS can be avoided or overcome, but doing so in a graceful way is not always possible. NoSQL databases have been designed to offer scalable alternatives. Object oriented databases such as Db4o[3], for example, solve the problems of object-relational impedance mismatch while MongoDB solves the scaling problems. There

[3]DB4o is an object DBMS for use with Java and .NET. Read more at `http://www.db4o.com/`

are many different NoSQL databases, each specialized in certain areas. When a particular problem arises when using an RDBMS, it might be worth searching for a NoSQL alternative that aims at solving that particular problem.

To sum it up, RDBMS is not always the best choice of database. RDBMS are not directly compatible with the popular object oriented programming paradigm and are difficult to scale horizontally. The very common JOIN operations, which are necessary for the relational model to link relations together, are slow. If RDBMS problems are encountered, there is probably a NoSQL alternative that excels where the RDBMS lack.

3.1.3.3 Tabular NoSQL databases

Tabular databases, just like the relational model, use tables to store data. One notable tabular database is Google's BigTable, which is used by a long list of Google products. BigTable uses a sorted, multidimensonal map and is made for scaling up to petabyte (one petabyte is 10^{15} bytes) sizes. Even though it uses tables, BigTable is *not* a relational database.[4]

Since BigTable is proprietary software not released outside of Google, open source tabular DBMS inspired by BigTable have been developed. Two such examples are HBase[4] and Hypertable[5].

3.1.3.4 Graph NoSQL databases

In a graph database management system, entire databases are viewed as more or less complex graphs where every node, and every edge between them, may have attributes. The edges (or relationships) may be of different types, such as *knows*, *has*, *loves* etc.

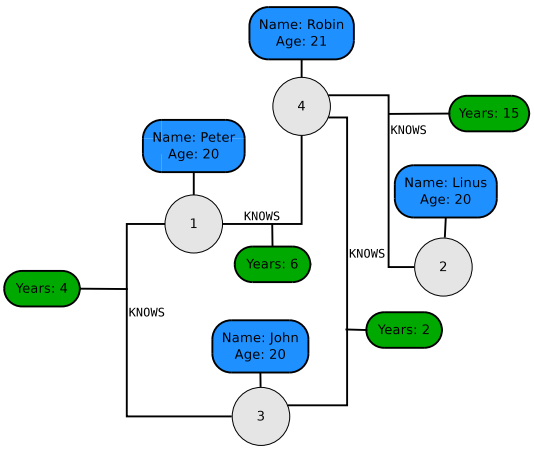


Figure 1: A simple example of a social network stored using the graph model

These types of databases are well suited for huge social networks (think Facebook) due to their scalability and ability to store information about relations between nodes.

A noteworthy graph DBMS is neo4j[6], which is written in Java, weighs under 500kB and can, according to its developers, handle complex graphs of billions of nodes and relationships on a single server setup.[15]

3.1.3.5 Document databases

Document-oriented databases use *documents* to store data. The word document in this context rather confusingly does not mean what one might think. A document database gives the impression of a database that stores something like Microsoft Word documents. However, this is not the case. According to Oxford Dictionaries online, a document is "a piece of written, printed, or electronic matter that provides information or evidence or that serves as an official record."[11] This is very

[4]HBase is maintained by the Apache Software Foundation. Its official website is `http://hbase.apache.org/`

[5]Hypertable is another tabular DBMS modeled after Google's Bigtable. Its website can be found at `http://www.hypertable.org/about.html`

[6]neo4j is an open source NoSQL object DBMS developed by Swedish company Neo Technology. Read more at `http://neo4j.org/`

much in accordance to what a document in a document database is. How these documents are structured depends on the implementation, but common choices are XML, JSON and YAML. What these three have in common is their semi-structured nature — the documents do not have to conform to any static schemas or tables. Instead, they use tags and other methods that allow related documents to contain different keys and values.

In some document DBMS, references like the ones thoroughly used in SQL databases are possible to use. Documents are, however, most often self-containing and lack reference to other documents. This results in redundancy, since data might repeat itself in multiple documents, but also means no costly JOIN operations have to take place.

The main advantage of document databases is their simplicity; pretty much anyone could probably understand a simple document. Since they contain semi-structured data, the databases are also a breeze for developers, who can easily use lists and associative arrays in their programming language of choice to insert data into the database with no need for further formatting.

Two of the more popular document database management systems are MongoDB and CouchDB, which will both be presented and benchmarked later.

4 MongoDB

MongoDB is a document-oriented NoSQL DBMS written in C++ and developed by 10gen. The word *mongo* in its name comes from the word *humongous*[1], hinting at its vastly scalable potential. It focuses on ease of use, performance and high scalability. MongoDB is available in 32-bit and 64-bit versions for Windows and Unix-like environments.

MongoDB's default way of managing databases is the interactive JavaScript shell that it provides out of the box. In addition, there exist bindings for many programming languages. Since many programming languages have data types that are built up of key-value pairs — some even support JSON directly — using these languages is excellent for creating documents for entering into MongoDB.

Some attributes of MongoDB are described in the ensuing subchapters.

4.1 Data model

MongoDB uses a binary form of JSON called Binary JSON, or BSON, to store data. BSON is designed to be easily and efficiently traversed and parsed. When users enter data into MongoDB they use regular JSON, which is then converted into the BSON format. When data is retrieved, it is again converted into regular JSON. In other words, the user never has to see any BSON; it is strictly used for internal purposes. The specification for BSON can be found at its official site, `http://bsonspec.org`. Although documents are stored as BSON, MongoDB documents will be referred to as JSON documents for the rest of this thesis.

A JSON document is zero or more key-value pairs, and a MongoDB document is simply a JSON document. Since MongoDB uses JSON, it is schemaless which means that there is no grouping of documents that share exactly the same keys, like in the relational model where the relation roughly fills this purpose. Instead, similar documents that contain data about the same thing, but with different key-value pairs, are bundled together in what is called *collections*. A database, in its turn, can be seen as a collection of collections.

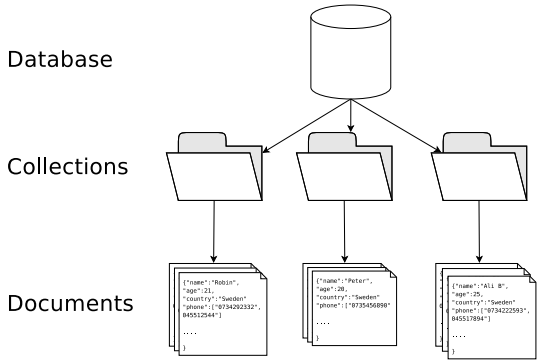


Figure 2: MongoDB's data model showing database, collections and documents

4.2 Indexing

MongoDB supports indexing on any attribute of a document, similar to how RDBMS offer indexing on any column. Indexes are implemented as B-Trees and can in many cases, assuming the right indexes are created, drastically increase the performance of queries.[12, p41]

Indexes in MongoDB are created from its JavaScript shell using the `ensureIndex()` function. It is possible to create indexes on simple keys, embedded keys and even entire embedded documents.

4.3 Sharding and replication

MongoDB has inherent support for replication, which means that a database is cloned and synchronized on two or more computers. Replicating is good for increasing redundancy, and for increasing raw performance.[12, p242] The simplest form of replication is *Single Master/Single Slave* replication, which as the name implies uses one master server, and one slave. However, it also has support for multiple masters, multiple slaves, master/master replication etc.

While replication lets data be synchronized between servers, *sharding* is a way of using a cluster of servers as one big database, with the data spread across the machines. A single server setup of MongoDB (or pretty much any other DBMS, really) is capable, but only to a certain degree. For example, the website Flickr, which lets users upload photos, have about ten *billion* photos.[12, p278] Storing that much data on one server is unwise, inefficient and likely very difficult to achieve.

4.4 Querying

4.4.1 JSON-style queries

MongoDB provides ad-hoc queries in a similar fashion to what most RDBMS offer with the help of SQL. Instead of using SQL, MongoDB accepts JSON documents that state what is to be searched for. For example, in a collection called *persons*, using the JavaScript shell and entering `db.persons.find({"name":"Robin"})` would return all documents containing the key `names` with the corresponding value `Robin`. There are a number of special operators that can be used in the documents to perform comparisons, find ranges of values and so on. For example, `db.persons.find({"age":{$gt:25}})` returns all persons that are over 25 years old.

4.4.2 Map/reduce

MongoDB additionally has support for *Map/Reduce* querying. Since this will not be used with MongoDB in this thesis, it will not be further explained. It is, however, the primary way of querying in CouchDB, and will hence be explained in section 5, *CouchDB*.

4.5 GridFS

GridFS is a specification for storing files in MongoDB. It consists of two collections, *files*, which stores the files' metadata, and *chunks*, which stores the actual files, by default split into 256Kb chunks.[12, p86] Since the files are

stored in chunks, and not as references like BLOBs in RDBMS, it is possible to retrieve parts of files without the need of first loading the file in its entirety.

4.6 Lack of transactions

While major RDBMS support atomic transactions to ensure data consistency and storage, MongoDB does not. This functionality has been left out in favor of speed and scalability. However, with the help of replication, it is possible to make the master server wait for the replica server to confirm a successful transaction with a form of receipt. Using two servers, this can serve as a kind of replacement for traditional transaction management.

4.7 In-place updating

Many DBMS use *Multiversion Concurrency Controller* (*MVCC*) which means that information is shown in different version for different users. Every user gets a "snapshot" of the current state of the database to work with. Changes made to the data will not be seen by other users until the transaction is fullfilled.

MongoDB does not use MVCC; it instead updates all information in-place, relinquishing the need for keeping track of different versions which increases performance. It also allows for lazy writes, meaning that MongoDB writes to disk only when it has to. Since primary memory is many times faster than secondary memory, MongoDB groups changes together and writes them to disk together. If more than one change affect the same value, the changes are treated as one and therefor only one update has to be made to the value on disk.[12, p14]

5 CouchDB

CouchDB is another document-oriented NoSQL DBMS, developed and maintained by the Apache Software Foundation and written in the functional programming language Erlang. The name CouchDB is derived from its developers' idea of it being easy to use; when a CouchDB server is started, the phrase "It's time to relax" is printed on the console.

What makes CouchDB special is to a large degree its RESTful API, which lets any environment that allows HTTP requests to access data from the database. It also uses a different kind of system for querying data than traditional DBMS, as will be discussed further on.

The default way of managing databases in CouchDB is through its Web based interface called *Futon* (a futon is a kind of couch, which matches the name of the DBMS itself). Futon is (by default) accessed through the URL `http://localhost:5984/_utils`.

5.1 Data model

Just like MongoDB, CouchDB stores JSON documents in a binary format. The file extension of its database files is *.couch*.

CouchDB stores documents directly inside of its databases. There are no collections like the ones found in MongoDB.

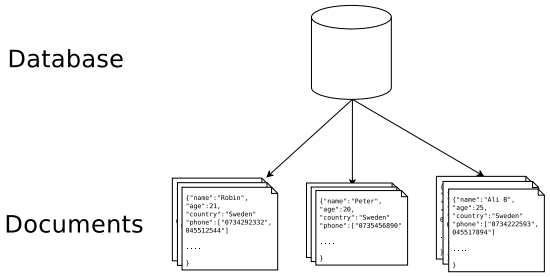


Figure 3: CouchDB's data model showing database and documents

Each document has a unique ID which can be assigned manually when inserting documents, or automatically by CouchDB. There is no maximimum number of key-value pairs for documents and there is no miximum size;

the default max size is 4GB, but this can be changed by editing CouchDB's configuration file.

5.2 RESTful API

One of the, if not *the*, most prominent features of MongoDB is its RESTful API. REST (Representational State Transfer) is an architecture which describes how services can be provided for machine-to-machine communications. It urges developers to use HTTP methods to perform CRUD (Create, Read, Update, Delete) operations. HTTP methods are mapped to CRUD in the following way:

- **POST** - Create a resource
- **GET** - Read a resource
- **POST** - Update a resource
- **DELETE** - Delete a resource[13]

In a RESTful architecture, resources (databases, documents, attachments etc.) get unique identifiers in the form of URIs. Imagine you want to create a database called *cars* on a local CouchDB setup. Using CouchDB's standard port (5984) and the command-line utility *curl* (which is an application that lets users perform raw HTTP requests), doing so would look like this:

```
curl -X PUT http://localhost:5984/cars
```

REST also urges developers to send information in XML or JSON, which fits CouchDB like a glove. The above request would be answered by CouchDB with a simple JSON document, to inform the user of success:

```
{"ok":true
```

Similar requests using curl can be made to perform all of the CRUD operations. Since CouchDB comes with a graphical Web interface, and since there are libraries available for many programming languages, the user seldom needs to make raw HTTP requests himself.

Because Web browsers use HTTP, they can also be used to read JSON documents from CouchDB.

5.3 Revisions

CouchDB supports strong versioning. In this case, it means that when a value in a document is to be updated, the data is not updated in-place on the underlying storage. Instead, the document in its current state is copied from the database, and the modified document is saved as a new revision. This revisioning system works very much like versioning systems such as *Apache Subversion*[7] or *CVS*.[8] Just like these systems, CouchDB does not allow an outdated document to be inserted as a new revision. Data always has to be up to date, and then possibly modified, before being inserted as a new revision.[2, p39]

5.4 Scaling and replication

Replicating databases in CouchDB is easy. All it takes to trigger replication is one simple HTTP request that specifies the *source* database and the *target* database:[2, p149]

```
POST /_replicate HTTP/1.1 {"source":"database",
"target":"http://somewhere.com/db"}
```

It is also possible to replicate from a remote server to the local server by switching the values of the `source` and `target` keys. This can be used to enable a form of two-way replication; simply trigger two replications sequentially, and switch the parameters.

[7] More info about Subversion can be found at its official website, `http://subversion.apache.org/`

[8] Concurrent Versions System. Read more at http://www.nongnu.org/cvs/.

To make replication even easier, it can be performed from the graphical Web interface Futon.

Scaling out databases by splitting them into an array of servers in a cluster is not as much of a trivial matter as replication. While MongoDB has inherent support for scaling horizontally, CouchDB does not. It can, however, be done with the help of an application called *CouchDB Lounge*[9], which is a partitioning and clustering framework for CouchDB.[2, p165] Another alternative for scaling CouchDB is *BigCouch*[10].

5.5 Querying

Relational database managament systems typically use static data and dynamic queries; schemas are fixed, and SQL queries are dynamic. CouchDB, however, has turned this upside down. Since it uses JSON documents, the data is dynamic.

Querying data in CouchDB is done through *views*. There are two kinds of views: permanent views, which are static, and temporary views, which can be provided ad-hoc. Views show the results of *Map/Reduce* functions. Map functions are written by the user, and iterate over all documents in the database to check if the documents match the criteria specified in the function by the user. If everything matches, and a result is hence found, the document (or selected parts of it) are *emitted* using the `emit()` function. A simple example follows:

```
function(doc) {
   if(doc.age && doc.age > 15 && doc.name)
      emit(doc.name,doc.age);
}
```

In the above example, all documents that have the key *age*, with a corresponding value that is over 15, are emitted. Since CouchDB does not use static schemas, it is important to always check if a certain key-value pair exists before trying to use it.

After a list of emitted documents has been generated by the map function, a reduce function may be used to further operate on the data.

In the benchmarking later on in this thesis, temporary views using only map functions are used. When static views are used, map and reduce functions are stored inside something called *design documents*, which is a special kind of document that contains executable code.

5.6 Indexing

When a data set is queried for the first times, the above explained map functions are run on all documents. CouchDB uses the view results from the map functions to build a *B-tree*, which is a data structure that offers logarithmic insertion, deletion and access.

Whenever a document is modified into a new revision, or deleted, CouchDB automatically updates the index B-tree.

5.7 Attachments

CouchDB supports binary attachments to documents, very much like how e-mails support attachments. Any kind of file can be attached to a document. Every attachments gets its own URI, which is the URI to the underlying document, followed by a slash and the name of the attached file (for example http://localhost:5984/persons/6e1295ed6c29495e54cc05947f18c8af/image.jpg).

[9] Read more about Lounge at `http://tilgovi.github.com/couchdb-lounge/`

[10] More information about BigCouch can be found at `http://github.com/cloudant/bigcouch`.

6 Benchmarking

6.1 Generating test data

To test the performance of document retrieval in MongoDB and CouchDB respectively, a set of data to store in the databases first needs to be created. For this purpose the Python module *randPersons* has been developed. The module include functions that generates a given number of documents with different key-value pairs.

Python has a data type known as dictionary, which is analogous to what is in other programming languages called associative arrays (as in PHP) or maps (as in Java, C++). A dictionary is a list of key-value pairs and is hence an easy and effective way of creating JSON style documents, which is how both MongoDB and CouchDB store their data. Even the syntax of JSON documents closely resemble that of Python dictionaries.

The pseudo-randomly generated data represents persons and their different attributes. The pseudo-randomness decides not only the values of keys, but also which key-value pairs every person possesses. To ensure that the same set of data is generated every time the code is run, a constant seed with the value of 1337 is used.

Note that no manual indexing on the data has been performed. This is because the benchmarks are supposed to test CouchDB and MongoDB under a primitive state, without any manual optimization techniques being used. Both DBMS have their own ways of speeding operations up (such as manual indexing in MongoDB), but these are outside the scope for this thesis and have hence not been tested.

6.1.1 Document outline

All persons share the following keys:

Key	Value	Range of values
Name	Arbitrary string	5-15 characters (a-z)
Income	Integer	9000-40000
Age	Integer	18-90
Telephone	Arbitrary string	10 characters (0-9)

Some persons also have pets:

Key	Value	Range of values
Pet	Pre-defined string	Albatross, horse, piranha, slow lori, panda, penguin

Other people still use fax machines to send documents:

Key	Value	Range of values
Fax	Arbitrary string	10 characters (0-9)

Last, but certainly not least, some people are blessed with the gift of Jedi powers:

Key	Value	Range of values
Jedi Power	Pre-defined string	Force lighting, Jedi mind trick, force choke, force throw, force insanity

When a person is generated, the first shared attributes are created. One, and only one, of the optional attributes might then (based on pseudo-randomness) be appended to the dictionary of that particular person. There are, however, basic persons that have neither a pet, nor a fax or Jedi power. This flexibility is possible thanks to the schemaless nature of JSON documents.

The resulting documents are somewhat strange; who would store this kind of information about persons, for what purpose, and who actually has Jedi powers? Keep in mind that these documents are for benchmarking purposes only, so they don't need to make sense.

6.2 Inserting test data

Since the same data is needed in both MongoDB and CouchDB, the script has to continu-

ally insert document by document until the desired number of documents has been reached. Two Python scripts, one for MongoDB called *mongoInsert*, and one for CouchDB called *couchInsert*, have been developed. Both scripts utilize the previously described module randPersons, which generates random persons.

Both mongoInsert and couchInsert are command-line Python scripts that let the user provide the number of documents to be inserted and whether composite values are to be generated for the optional keys *jedi power*, *pet* and *fax*. The scripts use a Python *for* loop, and call the `randomPerson()` function from the randPersons module once every iteration.

Note that the scripts in their current form only work for a certain number of documents, since the *for* loop is only able to iterate a certain number of times. For the numbers used in this experiment the script works fine, but when inserting documents in the range of millions or billions, the loops need to be replaced by *while* loops. The reason for maintaining the for loops, despite their lackluster scalability, is that they are seemingly a bit faster than the while loops.

Connecting to MongoDB from Python is done with the help of the library *pymongo*[11]. The corresponding library for CouchDB is called *couchdb-python*[12].

6.3 Querying data

After data has been inserted and insertion times have been measured, two scripts called *mongoBench* and *couchBench* have been used to benchmark MongoDB and CouchDB respectively. Both scripts let the user choose which of five pre-defined queries that are to be run.

[11] `http://api.mongodb.org/python/current/`
[12] `http://code.google.com/p/couchdb-python/`

6.3.1 Querying MongoDB

mongoBench utilizes MongoDB's ad-hoc, document-based queries using pymongo's `find()` function. The function returns a pointer to a result set on which the number of documents is counted and printed. A *for-each* loop is then used to iterate over all elements of the result set.

6.3.2 Querying CouchDB

couchBench utilizes CouchDB's temporary views. Map functions are stored as JavaScript functions in regular Python strings, and then used as parameters in couchdb-python's `query()` function, which returns a view result. The amount of documents in the view result is counted and printed, and the view result is then iterated over.

No reduce functions are used, since the queries are of a rather simple nature.

6.3.3 Query 1

The first query retrieves all persons with a salary of between 18500 and 35000.

6.3.4 Query 2

The second query retrieves all persons with a salary of between 18500 and 35000, and with an age of over 25.

6.3.5 Query 3

The third query retrieves all persons with a salary of between 18500 and 35000, with an age of over 25 and who have a fax number.

6.3.6 Query 4

The fourth query retrieves all persons with a salary of between 18500 and 35000, with an age of over 25 and who own an animal which is a penguin.

6.3.7 Query 5

The fifth query retrieves all persons with a salary of between 18500 and 35000, with an age of over 25 and who have a jedi power, which is either force choke or force insanity.

6.4 Measuring time

For the purpose of timing the insertion and retrieval Python scripts, no code inside the files have been used. Instead, the scripts have been run in a Linux command-line environment through the utility *time*, which is described in its manpage as being able to "run programs and summarize system resource usage". An example of the output of running a simple Python script, which prints 0-9 on the console, through time follows:

```
  robinproNook: $ time ./time.py
0 1 2 3 4 5 6 7 8 9

   real  0m0.034s
   user  0m0.028s
   sys   0m0.004s
```

- **real** shows the actual "wall clock" time.
- **user** shows the CPU time spent in user-mode.
- **sys** shows the CPU time spent in the kernel.

Adding user time and sys time results in the total CPU time the process has used.

The inserts have been measured directly. Every query, however, has been run three times. The spent time has been added and divided by three, thus generating the average time the queries used.

7 Benchmarking hardware and software information

For the sake of enabling duplication of the benchmarking, lists of software versions and hardware used is found below.

7.1 Software

- Ubuntu GNU/Linux 11.04, kernel 2.6.38-7-generic
- Python 2.7.1+ (for Python 3+, the print operations in all scripts need to be changed to functions)
- Apache CouchDB v1.01.1
- 10gen MongoDB v1.6.3
- pymongo v1.10.1
- couchdb-python v0.8.

7.2 Hardware

- HP ProBook (laptop)
- Intel Celeron P4500, dual core (1.86Ghz total), i386
- 2GB RAM

8 Benchmarking results

The ensuing subchapters present the results of the benchmarkings, and answer the questions formulated in the chapter *question formulations*.

Note that all graphs are available in large versions in the appendix section.

8.1 Insert speeds

This chart answers the question formulated in the introduction of this document — how fast is MongoDB and CouchDB respectively when inserting 5,000 documents, 50,000 documents and 500,000 documents?

As the graph clearly shows, MongoDB is a whole lot faster than CouchDB at inserting documents. The increase in time for both DBMS seem to be linear.

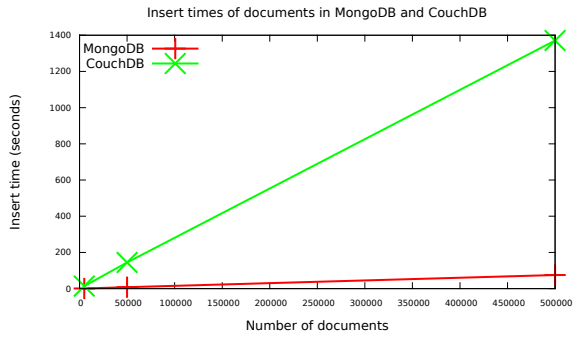


Figure 4: Graph visualizing insert speeds of MongoDB and CouchDB

8.2 Database sizes

This chart answers the question of how much space MongoDB and CouchDB use respectively for the same data set, when containing 5,000 documents, 50,000 documents and 500,000 documents.

Note that CouchDB contains only one revision for every document. The higher sizes for CouchDB thus is not because of multiple revisions of files.

The increase in time grows in a linear fashion as the size increases.

8.3 Read speeds

The following subesections showcase the results of the read speed benchmarks performed. They answer the questions of which DBMS,

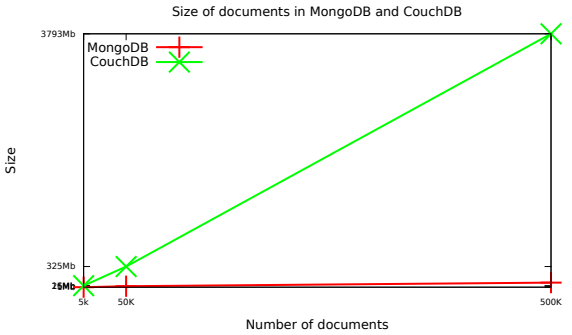


Figure 5: Graph visualizing sizes of MongoDB and CouchDB

CouchDB or MongoDB, is the fastest using their respective Python libraries at three different sizes — 5,000 documents, 50,000 documents and 500,000 documents.

8.3.1 Query 1

MongoDB is faster than CouchDB at processing the first query. Linear increase in time.

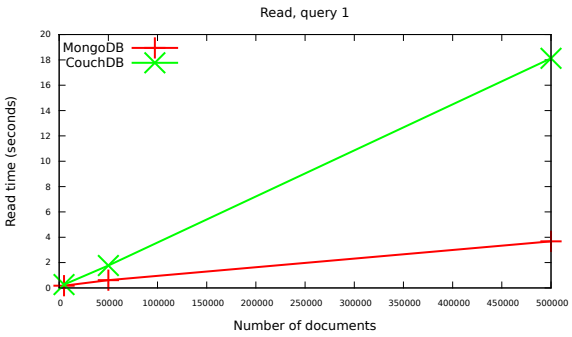


Figure 6: Graph visualizing the average results of read speeds for query 1 in MongoDB and CouchDB

8.3.2 Query 2

MongoDB is faster than CouchDB at processing the second query. Linear increase in time.

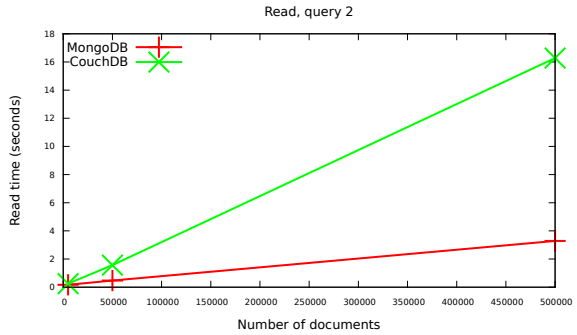


Figure 7: Graph visualizing the average results of read speeds for query 2 in MongoDB and CouchDB

8.3.3 Query 3

MongoDB is faster than CouchDB at processing the third query. Linear increase in time.

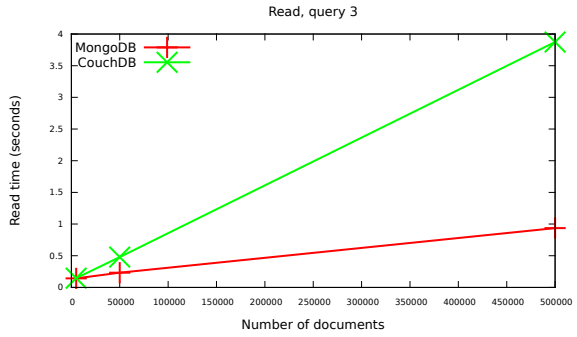


Figure 8: Graph visualizing the average results of read speeds for query 3 in MongoDB and CouchDB

8.3.4 Query 4

MongoDB is faster than CouchDB at processing the fourth query. Linear increase in time.

8.3.5 Query 5

MongoDB is faster than CouchDB at processing the fifth query. Linear increase in time.

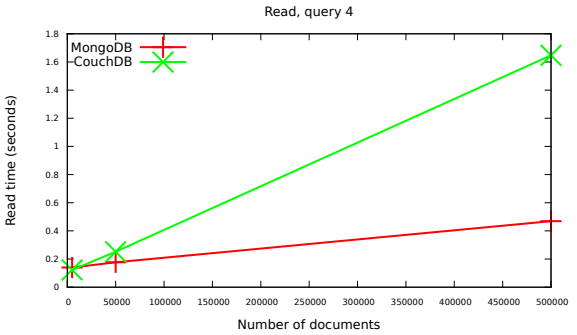


Figure 9: Graph visualizing the average results of read speeds for query 4 in MongoDB and CouchDB

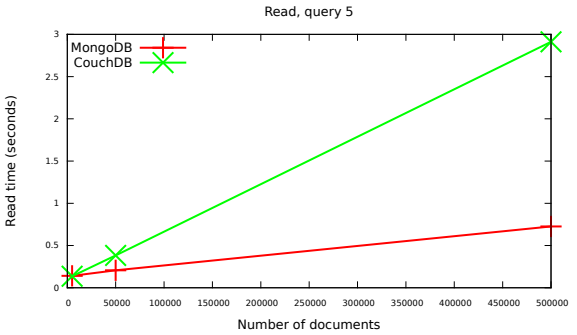


Figure 10: Graph visualizing the average results of read speeds for query 5 in MongoDB and CouchDB

9 Conclusions

9.1 Benchmarking results

MongoDB is undoubtedly a lot faster than CouchDB under the circumstances they were tested in for this thesis. Both DBMS show linear increase in time for inserts and reads, and linear increase in size. However, it is important to note that MongoDB and CouchDB are like apples and oranges — they're both fruits (or document databases, in our case), but they're still very different. MongoDB is built with efficiency in mind, while CouchDB focuses on providing ACID-like properties, a RESTful API and strong versioning; the RDBMS excel in different areas. If a lightning fast DBMS is needed, MongoDB is a good choice, at least if

the data stored therein is not of extremely vital importance, such as bank balances and transactions. If a robust and fault-tolerant DBMS is needed, CouchDB might be the better choice. Both DBMS are able to gracefully scale up to enourmous sizes, and back down again if needed. However, as proven by the disk usage test, CouchDB uses quite a bit more secondary memory than MongoDB.

One probable reason for CouchDB apparently being much slower than MongoDB in writing documents is that CouchDB writes every single document to disk, one by one, to ensure that data is consistent. MongoDB, on the other hand, groups documents together in primary memore and writes them to disk in batches.

An issue worth mentioning is the temporary views used by CouchDB in the benchmarking experiments. Temporary views are supposedly slower than permanent views. Temporary views were still used because ad-hoc queries from Python scripts is what was to be tested. Permanent views would probably have increased CouchDB's performance, but likely not to the degree of outmatching the performance of MongoDB.

Another issue that has not been taken into account is the efficeny of the Python libraries used themselves. It is possible that couchdb-python is not as optimized as pymongo. However, it could just as likely be the other way around. No matter the case, this has not likely affected the results much. For the disk usage comparison this naturally has not been an issue at all.

9.2 Problems encountered

The experiments have not been carried out without problems arising. The main problems encountered will be described in this section.

The read benchmarking scripts written in Python at first showed incorrect results, which unfortunately was not acknowledged until relatively close to the deadline. Both CouchDB and MongoDB have functions for performing queries — `find()` in MongoDB and `query()` in CouchDB. Both functions return a pointer to a result set, where individual records can be addressed and fetched. At first, the benchmarking scripts were returned such a pointer, and that was the end of the script. The assumption here was that the databases had processed the query and generated the entire result, but in reality it seems that the databases keep working in the background, even after the pointers are returned. 120ms retrieval speeds for five billion documents (which was tested just for fun) in MongoDB really sounded, and indeed was, too good to be true. The problem was solved by iterating over the resultset and doing nothing (which in Python is done using the keyword `pass`) on every element. This produced more accurate and fair results.

The problem mentioned above was encountered at first when experimenting on a rather weak laptop, using a 32-bit CPU. Since the maximum size of a database in the 32-bit version of MongoDB is 2GB, which in this case meant about 6 million documents, this lead to the conclusion that a computer using a 64-bit CPU was necessary because of the need for more documents in order to actually get any substantial time to measure. The university at first could not provide one; noone seemed to have one to lend for the experiments. After many hard tries, supervisor Göran Gustafsson took contact with Niklas Lavesson PhD, who set up an account on one of the university's really quite powerful computers using dual Xeon CPUs and 64GB RAM and with a 64-bit version of Ubuntu GNU/Linux installed. Since no administrative permissions were granted, Dr Lavesson, who at the time had a busy schedule, was the one to install all necessary components for the tests. Nothing seemed to work like it should have: different versions of the

Python libraries than had been programmed for were installed, and changes to configuration files in CouchDB, trying to change on what partition database files were saved, did not take effect. At this time a friend, Peter Assmus, lent his workstation with a 64-bit AMD CPU within. Unfortunately, that computer kept disconnecting the CouchDB insertion script, which was disastrous since all data needed to be inserted in one continual loop to ensure that the same data was used in both DBMS. At this point, the result set pointer background problem mentioned earlier became clear, and the realization that the laptop, which had been used at the very beginning, was actually adequate came. A lot of time was spent on trying to find a computer, when the first one used was the best fit for the experiments after all.

The writing of this thesis has proven to be quite time-consuming, as well. A good deal of time that could have been spent on actual writing was instead spent on trying to get LaTeX to behave in desirable manners. In the end it was worth it however, since a lot of things, such as cross-referencing, bibliography and table of contents are managed pretty much by themselves when using LaTeX.

9.3 Future work

The experiments presented in this thesis have touched upon only a fraction of what CouchDB and MongoDB are capable of. The maximum document size used has been 500,000 documents, when the databases could very well handle *billions* of documents. Most likely the insertion and read speeds would still be increasing linearly if tried on a single server setup, but with data sharded across a cluster, things might look different. This would make for an interesting expermiment.

Another interesting topic to investigate is how well, and how rapidly, CouchDB and MongoDB handle replication when used on the same two servers on a network.

Since all the tests performed have been in MongoDB's favor, CouchDB has been shed in a rather negative light, which is unfair. While MongoDB is indeed faster, CouchDB is more robust and handles failures more gracefully. The benchmarks performed in this thesis focused only on speed and size, but what would the results be if an experiment that focused more on CouchDB's strengths was to be performed? It would be interesting to see what would happen if one were to "pull the plug" on a server performing a transaction, both on a single server and on a network.

Something that has not been mentioned until now is the fact that MongoDB has installable RESTful API extensions available. When one is installed, how well does it work compared to CouchDB's inherent RESTful nature?

MongoDB uses lazy writes and handles much data in primary memory before saving it to secondary storage. CouchDB saves to disk everytime something is added, modified or deleted. It would be interesting to mount a RAM disk drive and perform the benchmarks there, to see if the secondary storage (a hard drive, in this case) is truly the bottleneck of CouchDB.

References

[1] 10gen Inc. *Agile and Scalable*. http://www.mongodb.org/, 2011. Visited 22th May, 2011.

[2] J.K. Anderson and N. Slater. *CouchDB: The Definitive Guide*. O'Reilly, 2010.

[3] K. Bhamidipati. *SQL Programmer's Reference*, 1998.

[4] F et al. Chang. *Bigtable: A Distributed Storage System for Structured Data*, November 2006.

[5] T. Conolly and C. Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*, 2005.

[6] O. Andersson et al. *Scalable Vector Grap hics (SVG) Full 1.2 Specification*. http://www.w3.org/Graphics/SVG/, April 2005. Visited 22 May, 2011.

[7] E. Evans. *NOSQL 2009*. http://blog.sym-link.com/2009/05/12/nosql_2009.html, May 2009. 22 May, 2011.

[8] International Organization for Standardization. *ISO 8879:1986*. http://www.iso.org/iso/catalogue_detail.htm?csnumber=16387, March 2011.

[9] Google Inc. *Creating or Editing CSV files*. http://mail.google.com/support/bin/answer.py?answer=12119, March 2011. Visited 22 May, 2011.

[10] Microsoft. . https://www.microsoft.com/presspass/features/2005/nov05/11-21Ecma.mspx, November 2005. Visited 22 May, 2011.

[11] Oxford Dictionaries Online. *document*. http://oxforddictionaries.com/view/entry/m_en_gb0235770#m_en_gb0235770, 2011. Visited 22 May, 2011.

[12] Membrey P. Plugge E. and Hawkins T. *The Definitive Guide to MongoDB*. Apress, 2010.

[13] A. Rodriguez. *RESTful Web Services: The basics*. https://www.ibm.com/developerworks/webservices/library/ws-restful/, November 2008. Visited 22th may, 2001.

[14] C. Strozzi. *NoSQL, A Relational Database Managament System*. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Visited 22 May, 2011.

[15] Neo Technology. *Why Neo*. http://neotechnology.com/why-neo. Visited 22 May, 2011.

[16] R. Weir and M. Brauer. *OASIS Open Document Format for Office Applications (OpenDocument) TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office, January 2011. Visited 22 May, 2011.

A Appendices

A.1 Source code, randPersons.py

The code has been run under Ubuntu GNU/Linux using kernel release 2.6.38-7-generic, using Python version Python 2.7.1+.

```
#!/usr/bin/python
import random;
from collections import defaultdict;
#"Constants" are put here for ease of modification
SEED = 1337;
ALPHA = 'abcdefghijklmnopqrstuvwxyz';
DIGITS = '0123456789';
ANIMALS = ["Albatross","Horse","Piranha","Slow loris","Panda","Penguin"];
JEDI_POWERS = ['Force Lightning', 'Jedi Mind Trick','Force Choke', 'Force Throw', 'Force Insanity'];
MIN_NAME_LENGTH = 5;
MAX_NAME_LENGTH = 15;
MINIMUM_SALARY = 9000;
MAXIMUM_SALARY = 40000;
MINIMUM_AGE = 18;
MAXIMUM_AGE = 90;
PHONE_LENGTH = 9;
MIN_NESTED_ELEMS = 1;
MAX_NESTED_ELEMS = 5;
#Data set versions
DEFAULT = 0;
WITH_PET = 1;
WITH_FAX = 2;
JEDI = 3;

random.seed(SEED);

def randomName():
    nameArray = random.sample(ALPHA,random.randint(MIN_NAME_LENGTH,MAX_NAME_LENGTH));
    nameString = '';
    for char in nameArray:
        nameString += char;
    return nameString;

def randomPet():
    return ANIMALS[random.randint(0,len(ANIMALS)-1)];

def randomJediPower():
    return JEDI_POWERS[random.randint(0,len(JEDI_POWERS)-1)];

def randomIncome():
    return random.randint(MINIMUM_SALARY,MAXIMUM_SALARY);

def randomAge():
    return random.randint(MINIMUM_AGE,MAXIMUM_AGE);

#This function is also used for generating fax numbers
def randomPhone():
    phoneArray = random.sample(DIGITS,10);
    phoneString = '';
    for digit in phoneArray:
        phoneString += digit;
```

```
    return phoneString;

#This function generates documents.
#Depending on what value the version parameter has, new key-value pairs
#are appended to the default document.
def randomDoc(version, nested):
    person = {
        "name":randomName(),
        "salary":randomIncome(),
        "age":randomAge(),
        "telephone":randomPhone()
        };
    if version == WITH_PET:
        if nested:
            nrOfPets = random.randint(MIN_NESTED_ELEMS,MAX_NESTED_ELEMS);
            pets = [];
            for i in range(nrOfPets):
                pets.append(randomPet());
            person['pet'] = pets;
        else:
            person['pet'] = randomPet();
    elif version == WITH_FAX:
        if nested:
            nrOfFaxNrs = random.randint(MIN_NESTED_ELEMS,MAX_NESTED_ELEMS);
            faxNrs = [];
            for i in range(nrOfFaxNrs):
                faxNrs.append(randomPhone());
            person['fax'] = faxNrs;
        else:
            person['fax'] = randomPhone();
    elif version == JEDI:
        #Remember, a Jedi's strength flows from the Force.
        #But beware. Anger, fear, aggression. The dark side are they.
        if nested:
            nrOfPowers = random.randint(MIN_NESTED_ELEMS, MAX_NESTED_ELEMS);
            powers = [];
            for i in range(nrOfPowers):
                powers.append(randomJediPower());
            person['jedi power'] = powers;
        else:
            person['jedi power'] = randomJediPower();
    return person;

def flatPerson():
    return randomDoc(random.randint(0,3), False);

def nestedPerson():
    return randomDoc(random.randint(0,3), True);
```

A.2 Source code, mongoInsert.py

The code has been run under Ubuntu GNU/Linux using kernel release 2.6.38-7-generic, using Python version Python 2.7.1+.

```
#!/usr/bin/python
#This script generates a number of persons according to the randPerson module.
#How many persons are generated is determined by a command line argument.
#If the parameter "drop" is provided, the collection is dropped.
import sys;
from pymongo import Connection;
from randPersons import flatPerson;
from randPersons import nestedPerson;

if len(sys.argv) == 2 or len(sys.argv) == 3:
    connection = Connection(); #host address and port can optionally be provided here
    db = connection.benchmarking; #benchmarking is the name of the database
    collection = db.persons; #persons is the name of the collection
    if sys.argv[1] == 'drop':
        print "Dropping collection.";
        db.drop_collection(collection);
    else:
        if len(sys.argv) == 3 and sys.argv[2] == 'nested':
            nested = True;
        else:
            nested = False;
        nrOfPersons = sys.argv[1];
        print "Inserting", nrOfPersons, "documents. Please wait...";
        if nested:
            counter = 1;
            for i in range(int(nrOfPersons)):
                collection.insert(nestedPerson());
                print (float(counter)/float(nrOfPersons))*100,"\b%\r",;
                counter = counter + 1;
        else:
            counter = 1;
            for i in range(int(nrOfPersons)):
                collection.insert(flatPerson());
                print (float(counter)/float(nrOfPersons))*100,"\b%\r",;
                counter = counter + 1;
else:
    print "Please provide the number of persons to insert using format", sys.argv[0], "[NR OF PERSONS] [MODE]";
    print "You may also drop the 'person' collection by using", sys.argv[0], "drop";
```

A.3 Source code, couchInsert.py

The code has been run under Ubuntu GNU/Linux using kernel release 2.6.38-7-generic, using Python version Python 2.7.1+.

```
#!/usr/bin/python
#This script generates a number of persons according to the randPerson module.
#How many persons are generated is determined by a command line argument.
#If the parameter "drop" is provided, the collection is dropped.
import sys;
import couchdb;
from randPersons import flatPerson;
from randPersons import nestedPerson;

if len(sys.argv) == 2 or len(sys.argv) == 3:
    connection = couchdb.Server(); #Server and port can optionally be specified here
    db = connection['benchmarking']; #Make sure there is a database named 'benchmarking' before running this script!
    if sys.argv[1] == "drop":
        print "Dropping database.";
        connection.delete('benchmarking');
        print "Recreating database.";
        connection.create('benchmarking');
    else:
        if len(sys.argv) == 3 and sys.argv[2] == 'nested':
            nested = True;
        else:
            nested = False;
        nrOfPersons = sys.argv[1];
        print "Inserting", nrOfPersons, "documents. Please wait...";
        if nested:
            counter = 1;
            for i in range(int(nrOfPersons)):
                db.create(nestedPerson());
                print "\t\r",; #Clear line
                print (float(counter)/float(nrOfPersons))*100,"\b\r%",;
                counter = counter + 1;
        else:
            counter = 1;
            for i in range(int(nrOfPersons)):
                db.create(flatPerson());
                print "\t\r",; #Clear line
                print (float(counter)/float(nrOfPersons))*100,"\b%\r",;
                counter = counter + 1;
else:
    print "Please provide the number of persons to insert using format", sys.argv[0], "[NR OF PERSONS]";
    print "You may also drop the 'person' collection by using", sys.argv[0], "drop";
```

A.4 Source code, mongoBench.py

The code has been run under Ubuntu GNU/Linux using kernel release 2.6.38-7-generic, using Python version Python 2.7.1+.

```
#!/usr/bin/python
#This script is used to benchmark a MongoDB database.
#It retrieves valued inserted by the module randPersons.py

import sys;
from pymongo import Connection;

connection = Connection();
db = connection.benchmarking;
collection = db.persons;

if len(sys.argv) != 2:
    print "Please choose a test number (0-5).";
elif len(sys.argv) == 2:
    option = sys.argv[1];
    if option == '1':
        cursor = collection.find({"salary":{"$gt":18500, "$lt":35000}});
    elif option == '2':
        cursor = collection.find({"salary":{"$gt":18500, "$lt":35000},
                                  "age":{"$gt":25}})
    elif option == '3':
        cursor = collection.find({"salary":{"$gt":18500, "$lt":35000},
                                  "age":{"$gt":25},
                                  "fax":{"$exists":True}});
    elif option == '4':
        cursor = collection.find({"salary":{"$gt":18500, "$lt":35000},
                                  "age":{"$gt":25},
                                  "pet":{"$exists":True},
                                  "pet":"Penguin"});
    elif option == '5':
        cursor = collection.find({"salary":{"$gt":18500, "$lt":35000},
                                  "age":{"$gt":25},
                                  "jedi_power":{"$exists":True},
                                  "$or":[{"jedi_power":"Force Choke"},
                                       {"jedi_power":"Force Insanity"}]});

#Print the number of matching documents
print cursor.count();

#Iterate through all results
for person in cursor:
    pass;
```

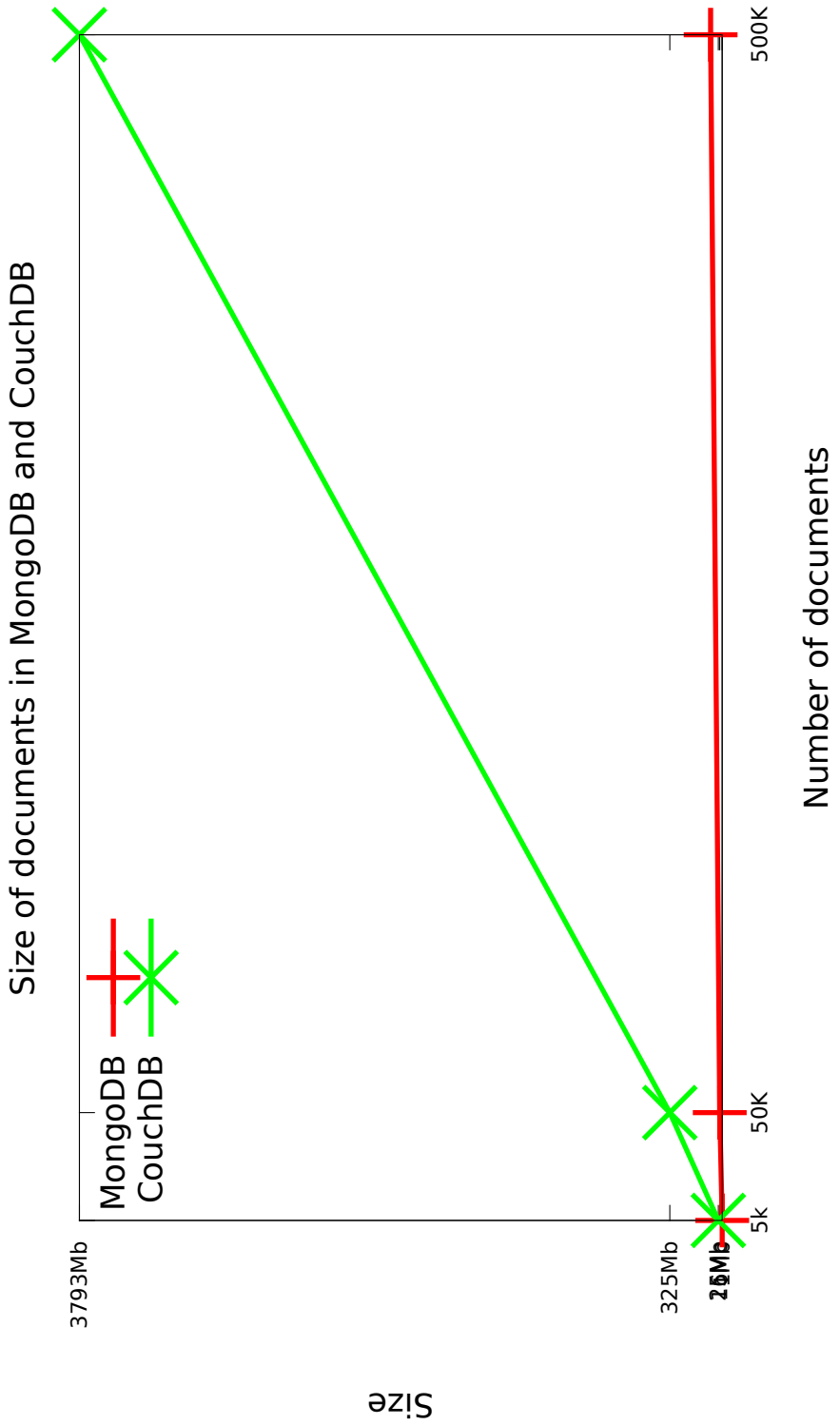
A.5 Source code, couchBench.py

The code has been run under Ubuntu GNU/Linux using kernel release 2.6.38-7-generic, using Python version Python 2.7.1+.

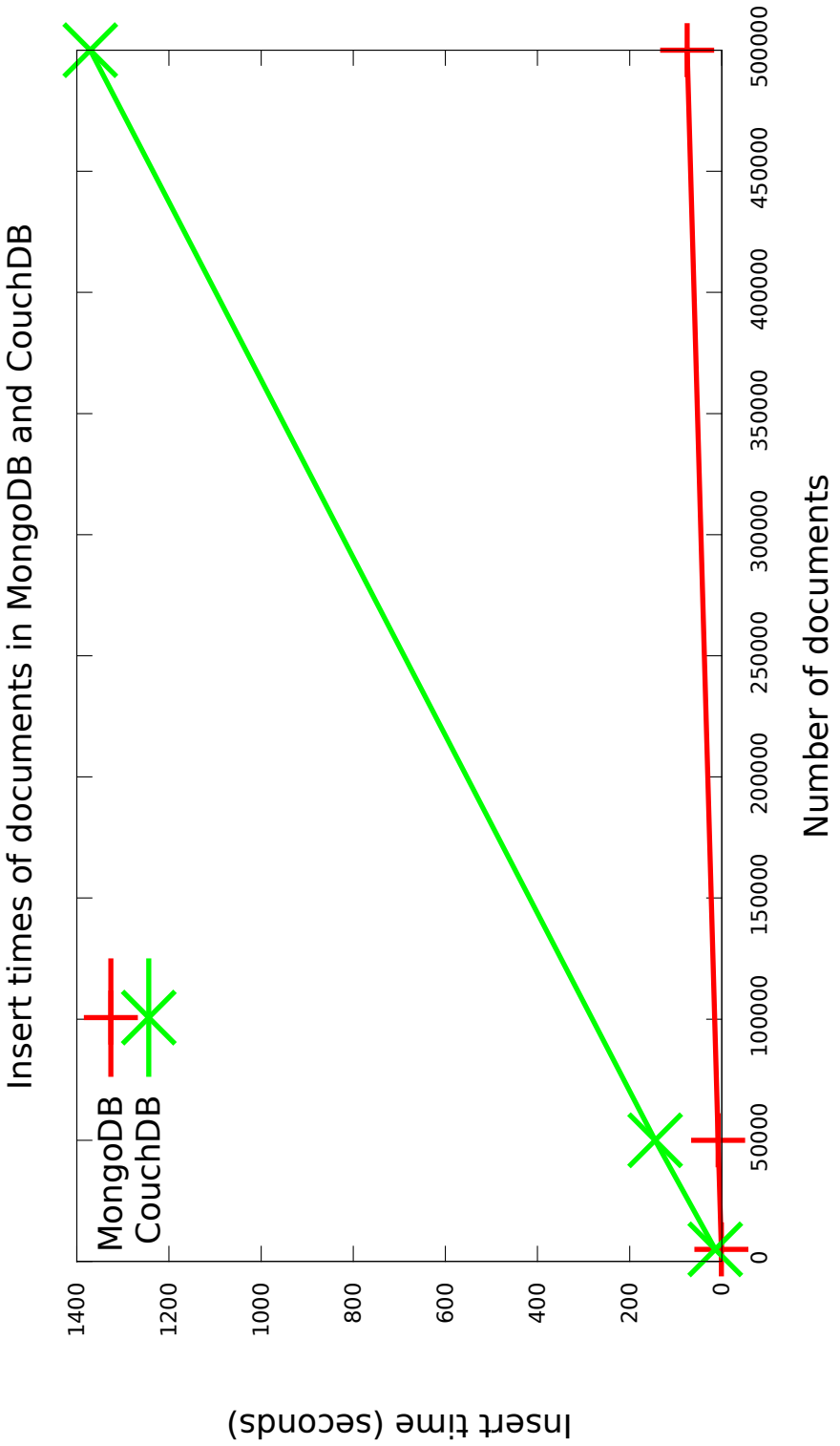
```
#!/usr/bin/python
import sys;
import couchdb;

connection = couchdb.Server(); #Server and port can optionally be specified
db = connection['benchmarking']; #Make sure there is a database named 'benchmarking'

#Map function definitions
#This is pretty ugly - javascript functions
#stored as Python strings. It is possible to
#use native Python functions as map functions,
#but doing so requires additional tweaking.
mapOne = '''function(doc) {
if(doc.salary > 18500 && doc.salary < 35000)
  emit(doc._id,doc);
}'''

mapTwo = '''function(doc) {
if(doc.salary > 18500 && doc.salary < 35000
    && doc.age > 25)
  emit(doc._id,doc);
}'''

mapThree = '''function(doc) {
if(doc.salary > 18500 && doc.salary < 35000
    && doc.age > 25
    && doc.fax)
  emit(doc._id,doc);
}'''

mapFour = '''function(doc) {
if(doc.salary > 18500 && doc.salary < 35000
   && doc.age > 25
   && doc.pet)
{
   var found = false;

  for(var i = 0; i < doc.pet.length; i++)
  {
    if(doc.pet[i] == 'Penguin')
      found = true;
  }

   if(found)
      emit(doc._id,doc);
}
}'''

mapFive = '''function(doc) {
if(doc.salary > 18500
   && doc.salary < 35000
   && doc.age > 25
   && doc.jedi_power)
{
```

```
var found = false;

for(var i = 0; i < doc.jedi_power.length; i++)
{
   if(doc.jedi_power[i] == 'Force Choke'
      || doc.jedi_power[i] == 'Force Insanity')
     found = true;
}

if(found)
  emit(doc._id,doc)};
}'''

rows = 0
if len(sys.argv) != 2:
    print "Please choose a test number (0-5).";
elif len(sys.argv) == 2:
    option = sys.argv[1];
    if option == '1':
        rows = db.query(mapOne);
    if option == '2':
        rows = db.query(mapTwo);
    if option == '3':
        rows = db.query(mapThree);
    if option == '4':
        rows = db.query(mapFour);
    if option == '5':
        rows = db.query(mapFive);

#Print number of documents
print len(rows)

#Iterate over all results in the View Result
for row in rows:
    pass;
```

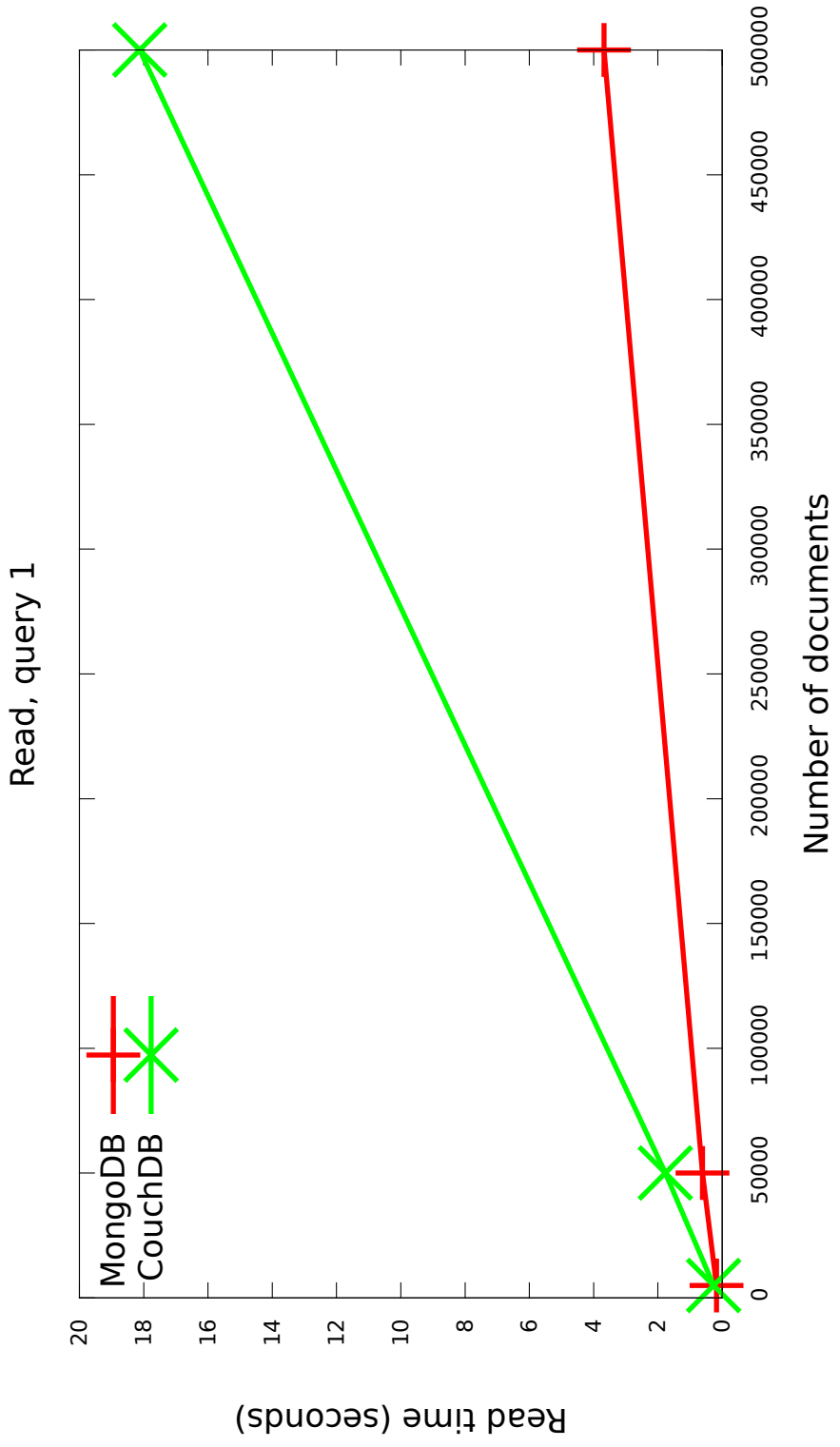
A.6 Graph, sizes



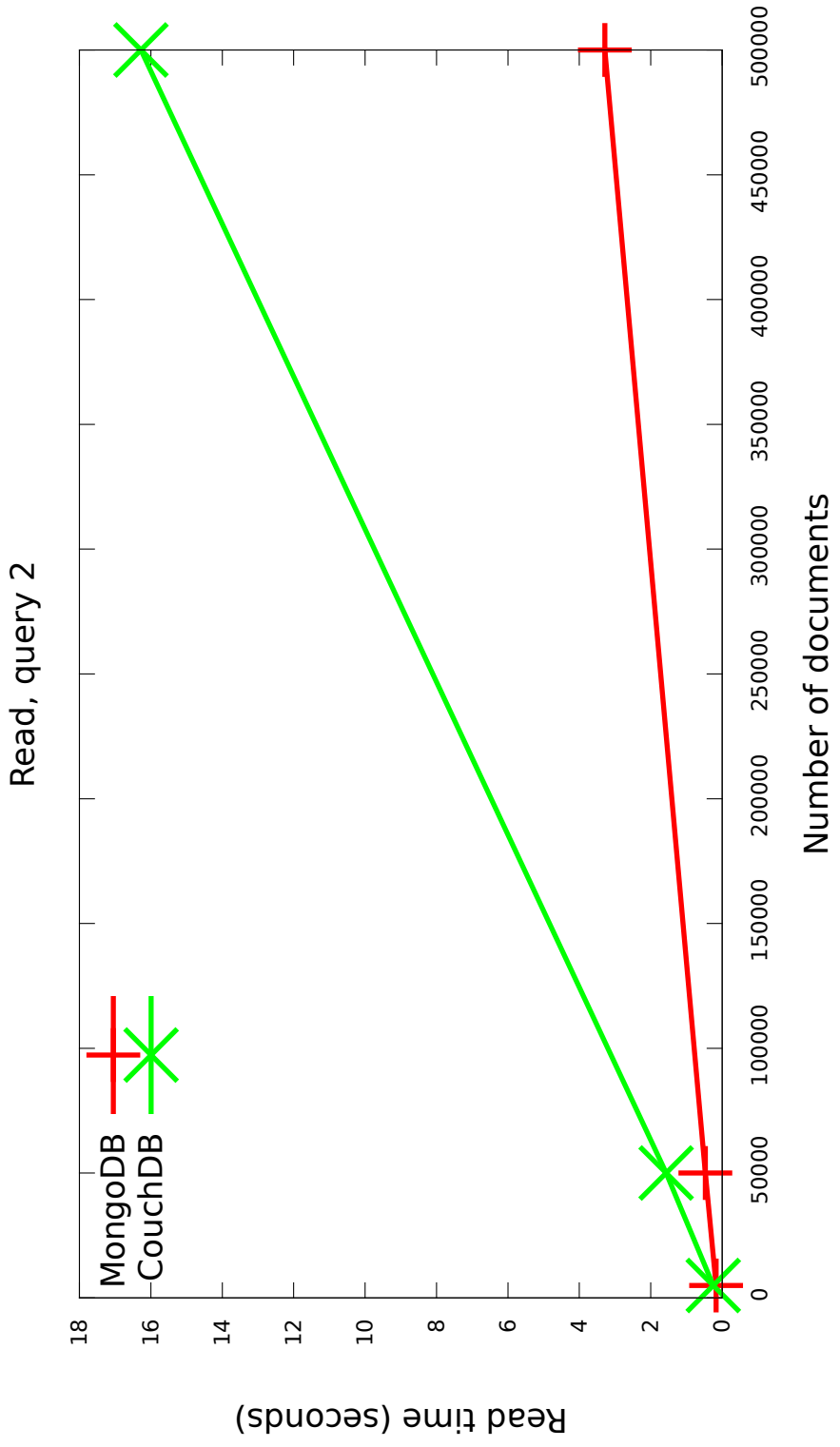
A.7 Graph, reads



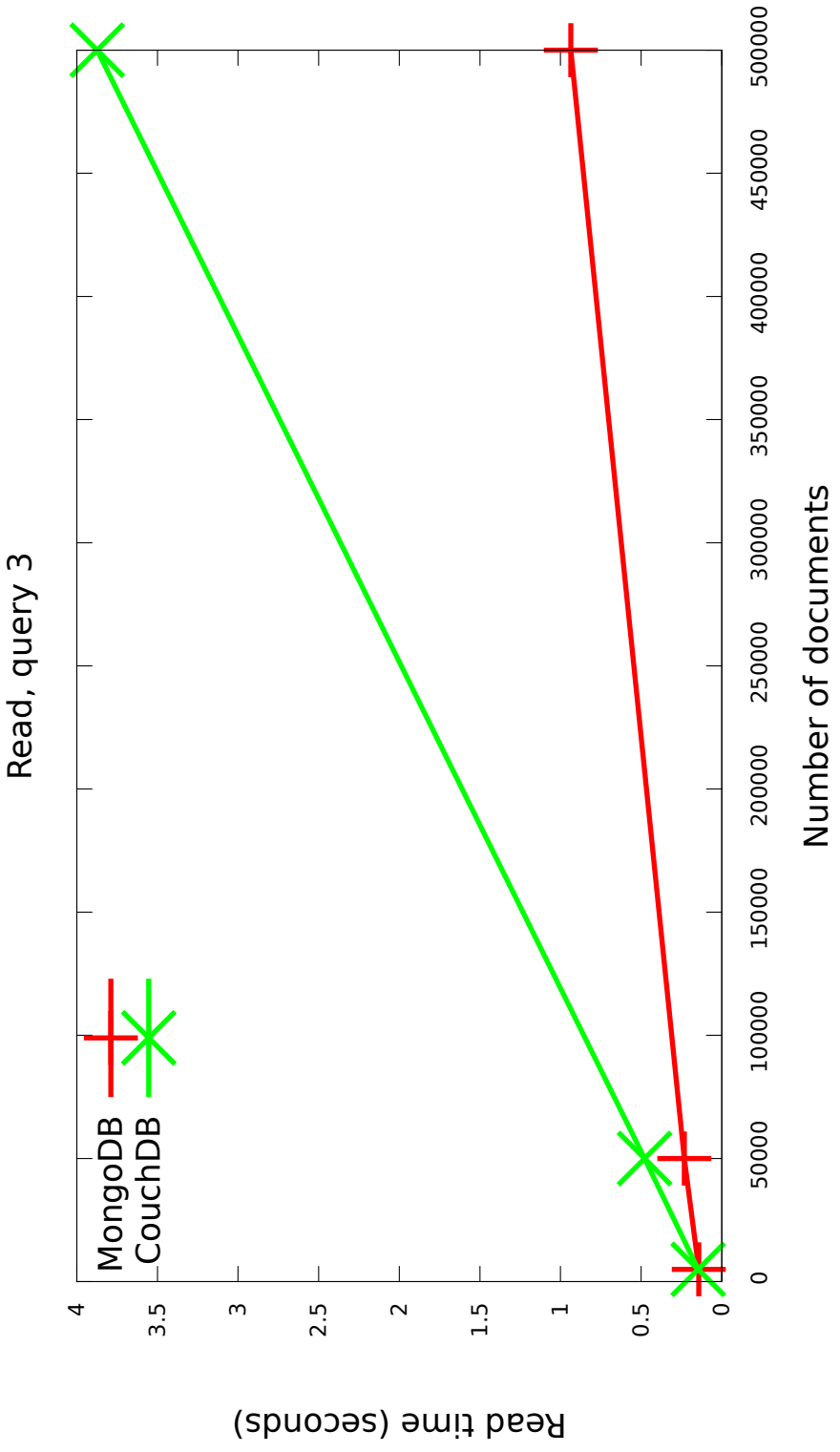
A.8 Graph, query 1



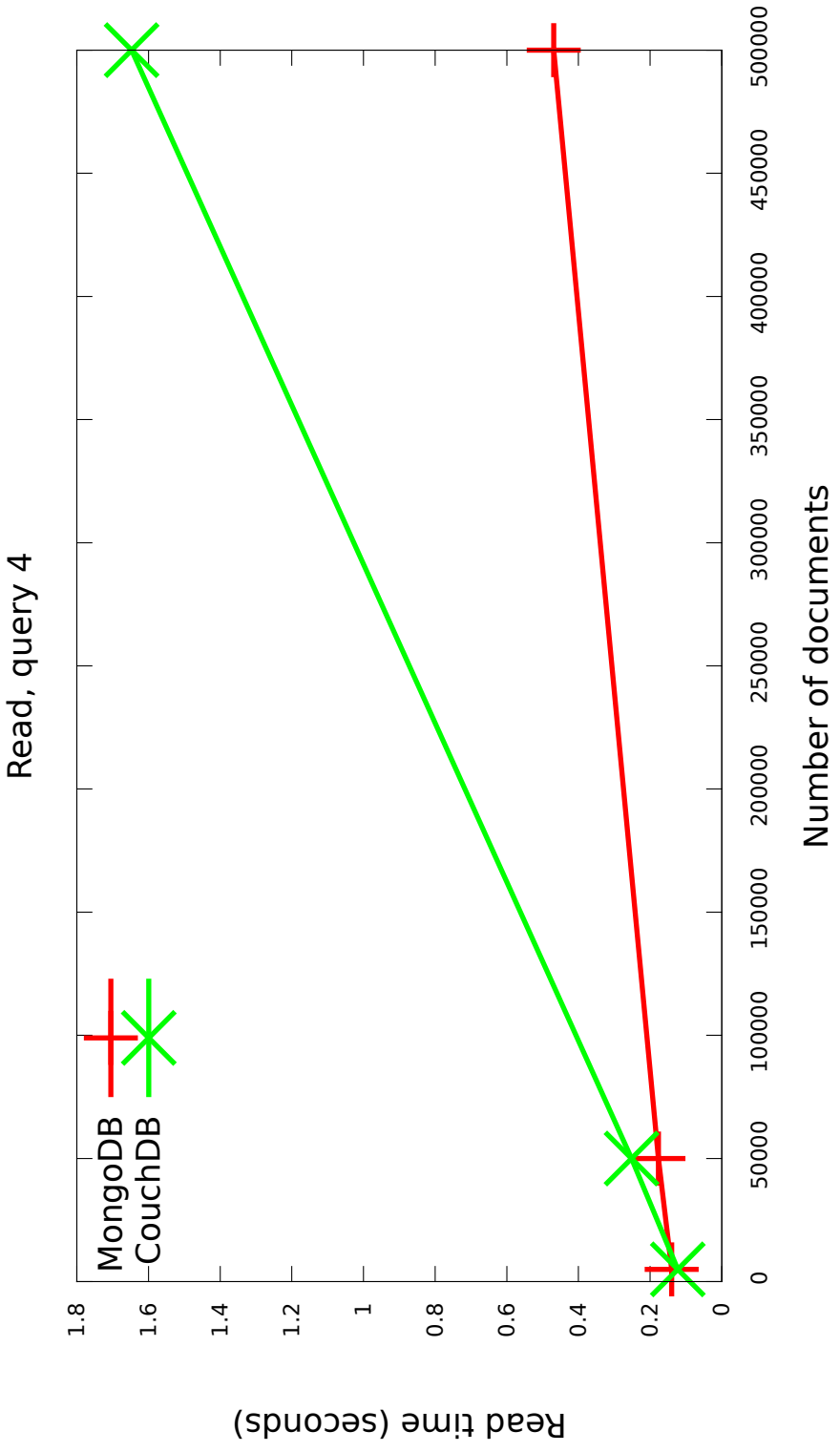
A.9 Graph, query 2



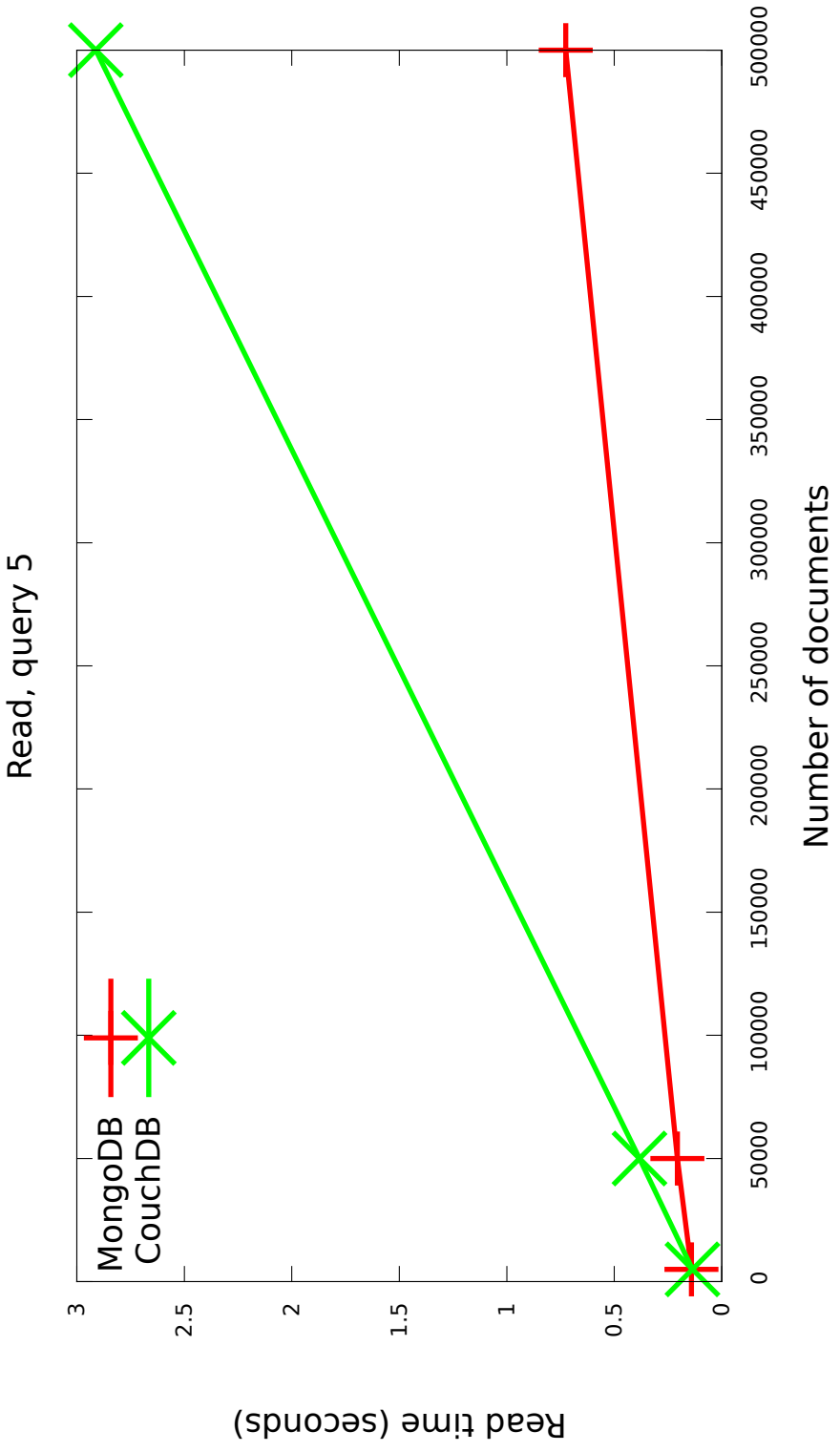
A.10 Graph, query 3



A.11 Graph, query 4



A.12 Graph, query 5



A.13 Techniques used for creating thesis

- **LaTeX** has been used for compiling the thesis from plain text files.
  `http://www.tug.org/texlive/`
- **Pygments** has been used to create syntax highlighted code in LaTeXformat for the thesis.
  `http://pygments.org/`
- **Dia** has been used to create diagrams and drawings present in the thesis.
  `http://live.gnome.org/Dia`
- **gnuplot** has been used to create the graphs that visualize the benchmarking of MongoDB and CouchDB.
  `http://www.gnuplot.info`