Comparing NoSQL MongoDB to an SQL DB

Zachary Parker
The University of Alabama
Center for Advanced Public Safety
Tuscaloosa, AL 35487-0290
(205) 348-6363
zparker@cs.ua.edu

Scott Poe
The University of Alabama
Center for Advanced Public Safety
Tuscaloosa, AL 35487-0290
(205) 348-6363
spoe@cs.ua.edu

Susan V. Vrbsky
The University of Alabama
Department of Computer Science
Tuscaloosa, AL 35487-0290
(205) 348-6363
vrbsky@cs.ua.edu

ABSTRACT

NoSQL database solutions are becoming more and more prevalent in a world currently dominated by SQL relational databases. NoSQL databases were designed to provide database solutions for large volumes of data that is not structured. However, the advantages (or disadvantages) of using a NoSQL database for data that is structured, and not necessarily "Big," is not clear. There are not many studies that compare the performance of processing a modest amount of structured data in a NoSQL database with a traditional relational database. In this paper, we compare one of the NoSQL solutions, MongoDB, to the standard SQL relational database, SQL Server. We compare the performance, in terms of runtime, of these two databases for a modest-sized structured database. Results show that MongoDB performs equally as well or better than the relational database, except when aggregate functions are utilized.

Categories and Subject Descriptors

H.2.4 [Information Systems]: Systems – *relational databases*

General Terms

Performance

Keywords

MongoDB, NoSQL, performance, relational, structured data, SQL

1. INTRODUCTION

The most common database implementation today is based on the relational model [1] which uses SQL as its query language. However, NoSQL (**N**ot **O**nly **SQL**) database solutions [2,3] are becoming more prominent as massive amounts of rapidly growing data are being collected today. This data is typically non-structured, complex and does not fit well into the relational model. Examples of this type of data are smart phone records in which the location is broadcast every few seconds, feed from video cameras in public spaces and even the enormous number of pages and documents on the web. However, there also exists a tremendous amount of modest-sized databases with structured data that still needs to be stored and processed in a database.

There is currently much discussion in the database world about whether or not NoSQL databases will replace relational databases. Given the move towards NoSQL databases and the availability of open-source NoSQL databases, a natural question for a database designer is whether or not to choose a NoSQL database. If the database is non-structured and extremely large, then a NoSQL database is a good choice. However, it is not clear if using a NoSQL database instead of a relational database is a viable solution for a modest-sized database with structured data.

Data in the relational model is usually represented by a database schema [1], in order to capture the semantics of the database. Objects in the database with the same number of characteristics, type and format are grouped together, making it structured data. The relational model is built on this assumption of structured data, with its data stored in the rows and columns of a table, whereby each row has the same number and type of data columns.

Tables in relational databases are typically normalized, which results in the creation of multiple tables. Querying those tables requires fetching and combining information from the many different tables. Combining information based on a matching value for a primary key and foreign key across multiple tables in the relational model requires using a join operation. The larger the schema and the more tables that need to be joined, the longer it takes for the relational database to fetch the data.

NoSQL can help deal with data that is not structured. Data can be semi-structured, such that similar data objects can be grouped together, but the objects may have different characteristics. Schema information may also be mixed in with data values in semi-structured data, such as found in XML data. Unstructured data can be of any type and may have no format. This data cannot be represented by any type of schema, such as web pages in HTML.

The typical features of SQL databases, such as the ACID properties, require a certain amount of overhead, and are relaxed or eliminated in NoSQL databases to maximize performance. Many NoSQL databases organize the data into key-value pairs. The key is used to uniquely identify a particular data item and the value can be a simple word, number or a complex structure with unique semantics. The development of queries is more complex, there is no standard query language, and there are limits to the operations. Specifically, there is no join operation. However, in general processing is simpler, more affordable and more flexible.

In this paper we compare one of the NoSQL solutions, MongoDB [4,5], to a standard SQL relational database, SQL Server. We run experiments using a modest-sized structured database in order to determine the performance of the relational database to the NoSQL database. We note that we do not consider any ACID properties in our study. We compare three main performance aspects of these databases: insert speed, update speed, and select operation speed. Results from our experiments provide insight into the viability of using a NoSQL database for modest sized structured data.

This paper is organized as follows. In Section 2 we discuss related work. We present our experimental setup in Section 3 and in Section 4 we describe the results of our experiments. Conclusions and future work appear in Section 5.

2. RELATED WORK

There are many blogs, white papers and online comments about the benefits of NoSQL databases versus SQL databases, but few academic papers comparing NoSQL to SQL. Instead, there have been several papers recently evaluating the different NoSQL databases available [6-8]. In [6] the authors perform a case study of 14 different NoSQL Databases over such features as their data models, query possibilities, concurrency control, partitioning and replication opportunities. Their recommendation is to use NoSQL databases for operations that are very fast and simple for very large datasets.

NoSQL databases are compared to SQL databases in [9]. The overhead that results from online transaction processing is purposed to not be related to SQL, but instead to other components. The author considers the four overhead components of logging, locking, latching and buffer management. Elimination of the overhead associated with one or more of these can provide a speedup of two. The author concludes that one can improve SQL databases to compete with NoSQL databases.

When using the NoSQL implementation MongoDB, there are no database schemas or tables. MongoDB [4,5] instead uses a “collection” which is similar to a table and “documents” which are similar to rows, to store the data and schema information. MongoDB automatically generates a primary key (id) to uniquely identify each document. The id and document are conceptually similar to a key-value pair. MongoDB attempts to hold most of the data in memory so simple queries take less time by eliminating the need to retrieve data from the hard disk. One caveat to this is once the data set becomes larger than the available memory, then MongoDB will have to start querying the hard disk for results. Since MongoDB is a NoSQL implementation, it is highly scalable and does not require a rigid schema definition. This can sometimes make queries more difficult to write because the user may not know exactly which parts of the document to query.

There is no need for a join operation in MongoDB. Storing data in MongoDB can be done in one of two ways [4]. The first way is by nesting documents inside each other. This option can work for one-to-one or one-to-many relationships. Note this option cannot be used for many-to-many relationships since it could cause infinite nesting loops. The second option is to store a reference to the other document rather than nesting the entire document. With this second option MongoDB will need to retrieve the referenced document only when the user requests data inside the referenced document. The downside to this feature is that MongoDB has no built in way to retrieve an object based on reference. This means that the user must define their own method for retrieving the data on reference. It is implemented as querying the collection of objects in MongoDB for the reference based on its primary key which makes MongoDB similar in its behavior and purpose to a SQL join.

Some of the main problems with MongoDB and other NoSQL databases are the lack of many features standard relational databases have. MongoDB only provides atomic operations within a single document. Another major feature MongoDB lacks is most aggregate functions; MongoDB does not have many of the simple aggregate functions built in like standard relational databases do. MongoDB does have a way to get around this though and that is allowing the user to define their functions via the MapReduce [4] method.

MapReduce is a programming abstraction proposed in [10] which is used today to process big data. MapReduce consists of two stages: the Map stage in which the user specifies computation to be applied over all the input and the Reduce phase, in which the output from the Map phase is aggregated using another user specified computation. The classic example used to illustrate MapReduce is word count, in which the occurrence of each word in a given document is computed. In the Map phase, for each word encountered in the document, a key-value pair is emitted consisting of the word as the key and 1 as the value, e.g. <hello, 1>. In the Reduce phase, all of the occurrences of the same word are combined and their values of ‘1’ summed, which give the resulting word count.

In the next section we describe our experiments to study the performance of SQL and MongoDB considering inserts, updates, and select operations that require join and MapReduce.

3. EXPERIMENTS

For our initial setup we installed both MongoDB and Microsoft SQL Server Express on a machine running an Intel i7 quad core 3.4GHz processor with 8GB of DDR3 Ram. Both databases were installed and stored their data on a SSD (Solid State Drive) for the fastest possible reads and writes. We wrote both test applications in C# using Visual Studio 2012 under the .Net 4.0 framework. For the SQL Server application we used entity framework for inserts and object data binding, but for the updates and the queries we used straight SQL so the results would not be skewed. On the MongoDB side we used the latest version 1.7 of the MongoDB C# driver and all queries were written directly using MongoDB’s query writer.

Our tests consisted of four separate tests with 100 runs for each test. Figure 1 shows the object schema used for the tests. As shown in Figure 1, the data consisted of 3 tables/collections representing department, project and user. There are relationships between these data objects, as a user has a particular department and an array of projects. Similarly, a project is associated with a particular department, a manager that is a user and also an array of users of the project.

In the relational model implementation of the database in Figure 1, queries involving more than one table require a join operation. For example, a query involving the User and the Department requires a join between the user and department tables based on the common value for Department and id. In MongoDB, such a query requires either copying the Department data in the User collection or the retrieval of data based on the key. We also note that an implementation of the database in Figure 1 requires the 3 tables for each of the data objects, as well as an additional table representing the M:N relationship between the User and Project objects. This additional table replaces the array of projects in the User table and the array of users in the Project table.

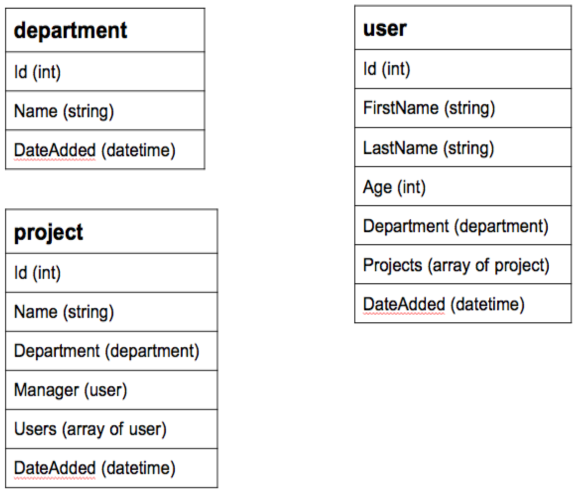

Figure 1. Database Schema

Figure 2 illustrates the number of Departments, Users and Projects for each of the four test cases. The total number of tuples used in the test cases ranges from 148 for Test Case 1 to 12,416 for Test Case 4. The Number of Selects is discussed later.

Test Case	Departments	Users	Projects	Number of Selects
1	4	128	16	100
2	4	256	64	200
3	16	1024	512	100
4	128	4096	8192	200

Figure 2. Test Cases

For the experiments we built separate console applications in C# that attempted to use the same code for both the SQL and MongoDB when possible. The code for the inserts differed only in their references to the different databases. The methods for performing updates and queries are very different between the two implementations. The relational database uses SQL syntax while the MongoDB database uses JavaScript Object Notation (JSON). Aggregate functions differ even more between the two implementations due to the added complexity of using MapReduce functions in MongoDB.

For the Insert portion of our testing, we inserted each of the data items as shown in Figure 2. The relationships between the data objects, such as the Department in the User object was implemented in MongoDB by storing a reference to the other document rather than nesting the entire document. For example, a reference to a Department was stored in a User object.

For the Update portion of our testing, we used three different types of updates. The first type involved updating the name of one-tenth of the departments based off their primary key. The second type updated the last name of any user whose first name matched a chosen name (the chosen first name for our test was Jacob). The third type updated the name of one-fourth of the projects based on their key. We tested each of these three types of updates on all four test cases.

We also tested seven different Select queries. The selects were divided into two categories: simple and complex. The simple queries involved selecting data of only one object type. The complex queries involved multiple object types, nested queries, and aggregate functions. The number of selects run for Test Cases 1 and 3 was 100 and the number of selects run for Cases 2 and 4 was 200.

The four different simple selects we generated for the tests are as follows. The first select retrieved a department by its primary key. The second retrieved a department by a randomly chosen name. All departments had unique names, and the name was guaranteed to match one result in the department's collection. The third select retrieved a user by its primary key. The fourth select retrieved all users whose first name matched a randomly chosen first name; this randomly chosen name may not match any users in the collection.

The three different complex selects used in the tests involved a join operation and are as follows. The first complex select retrieved all departments that contain one or more users that have a randomly chosen first name. Since first names were randomly assigned this may or may not return any matches. The second complex select retrieved all users who work on any of the three randomly chosen projects. Since users are randomly assigned to projects this query may also not return any valid matches. The final complex select returned the average age of the users who work on any of the three randomly chosen projects, and obviously, contains the aggregate function average(). This final query may return zero if no users were assigned to the projects.

4. RESULTS

The results of all of our experiments appear in Figures 3 through 13. Each figure compares the average time, in milliseconds, of the SQL implementation versus the MongoDB implementation of the given job for each of the 4 Test cases. As stated earlier, we ran 100 and 200 runs for each test to reduce the skewing effect any outliers would have on the average time.

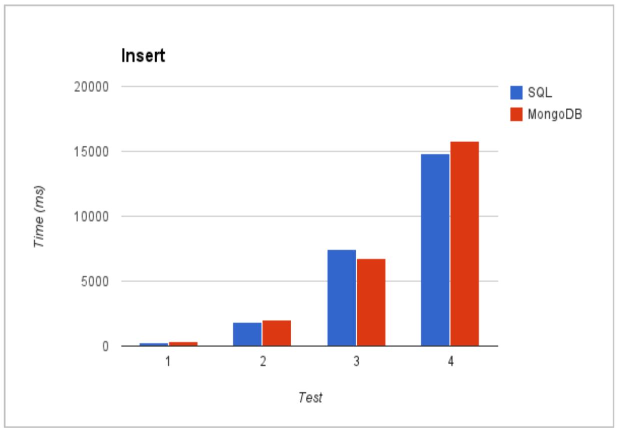

Figure 3. Insert Speed

Figure 3 shows the difference in insert speed between the two database systems. Neither database performs consistently better than the other. SQL is faster than MongoDB in Test Cases 1, 2 and 4, while MongoDB is slightly faster than SQL in Test Case 3. As noted earlier, the inserts in MongoDB implemented relationships between data objects using a reference. This is did not require inserting additional data in MongoDB. MongoDB was 34% slower for Test Cases 1 than SQL and 7% slower for Test Cases 2 and 4, while SQL was 10% slower for Test Case 3 than MongoDB.

Figures 4, 5 and 6 present the times in ms for each of the different updates we ran. For the update in Figure 4, we performed an update on users based on first name and the first name was not indexed in any way. In Figure 4, MongoDB performs consistency worse than SQL Server for each of the four Test Cases. Since Figure 4 was an update on a non-indexed column, and MongoDB's unstructured nature forces any unindexed queries to perform complex lookups on each data item, it took MongoDB much longer to run on larger data updates than SQL. For Test Case 1, MySQL and MongoDB has almost identical performance. As the size of the database increases, the time for MongoDB to update Users by their First Name ranges from 3 times slower than SQL for Test Case 2, to more than 5 times slower for MongoDB for Test Case 3.

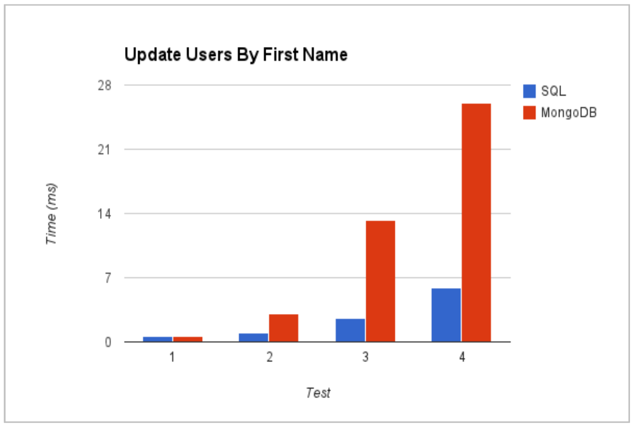

Figure 4. Update - First Name

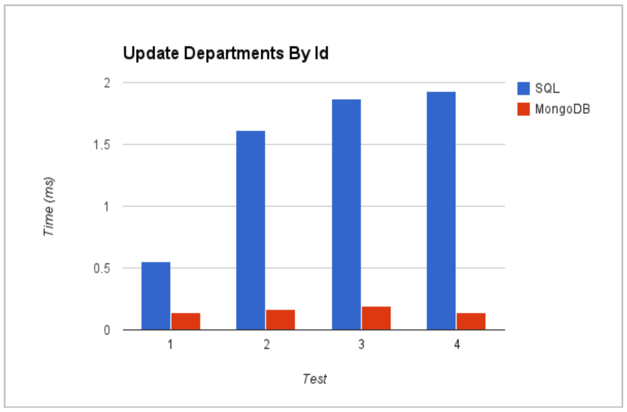

Figure 5. Update – Department ID

MongoDB outperformed SQL in all cases in which the update involved using the primary key, as shown in Figures 5 and 6. The update in Figure 5 involves updating Department using its department ID, while the update in Figure 6 involves updating Projects using the project ID. The variation in update time is small across the Test Cases, particularly for MongoDB, which is less than 0.2 ms for all four Test Cases. SQL Server ranges from a low of 0.55 ms for Test Case 1 to 1.93 for Test Case 2. In Figure 6, SQL Server is 13 times slower than MongoDB for Test Case 4 when updating by Department ID. We believe this performance gap is due to MongoDB having a pre-built index on the primary key of the document which is faster than SQL Server's primary key clustered index.

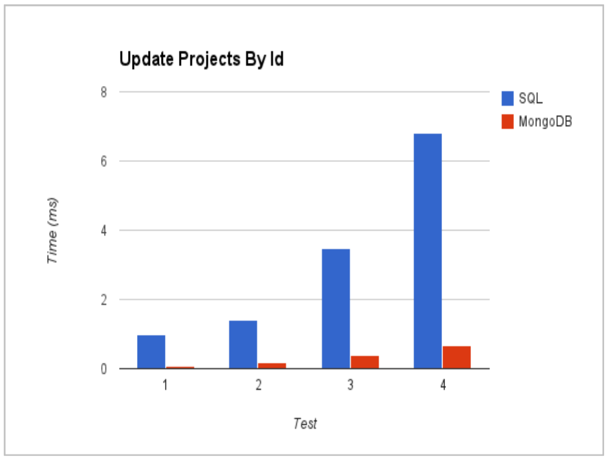

Figure 6. Update – Project ID

Figures 7 through 13 illustrate the time taken to perform the select queries, with Figures 7-10 showing the results for the simple queries and Figures 11-13 showing the results for the complex queries. The simple selects were either on the primary key of the table (document) or one of the other values. The complex selects involved grouping, M:N relationships and aggregation. We note that although the sizes of the tables increases for Test Cases 1-4, each test has a variable number of selects. Test Case 2 has a higher number of selects than Test 1, while Test Case 3 has the same number of selects as Test 1, but with more data. Similarly, the selects for Test Cases 2 and 4 are the same, but Test Case 4 has more data. Hence, it is not always useful to compare the runtimes of the tests to each other, but rather to compare the runtime of MongoDB to SQL.

In Figure 7, the time to select the Departments by ID is about 3.5 times higher for SQL than MySQL for all of the Test Cases. Even though the size of the data increases from Test Cases 1 to 4, the runtime for this experiment is also affected by the number of selects, as demonstrated by the decrease in runtime for Case 3 versus Case 2.

Figure 8 illustrates the time to select the Departments by Name. SQL was 1.5 times slower than MongoDB for Test Cases 1 and 2, and more than 3 times slower than MongoDB for Test Cases 3 and 4. There is a similar trend in Figure 9, in which Departments were selected by Name, but it is more dramatic. MongoDB was 3 times faster than SQL for Test Cases 1 and 2 and 8 times faster for Test Cases 3 and 4. It is noteworthy that in Figures 8 and 9, the runtimes for SQL increased as the amount of data increased, while runtimes for MongoDB were related to the amount of data as well as the number of selects performed. Test Cases 1 and 3 involved 100 selects, while Test Cases 2 and 4 involved 200 select, and have a higher runtime.

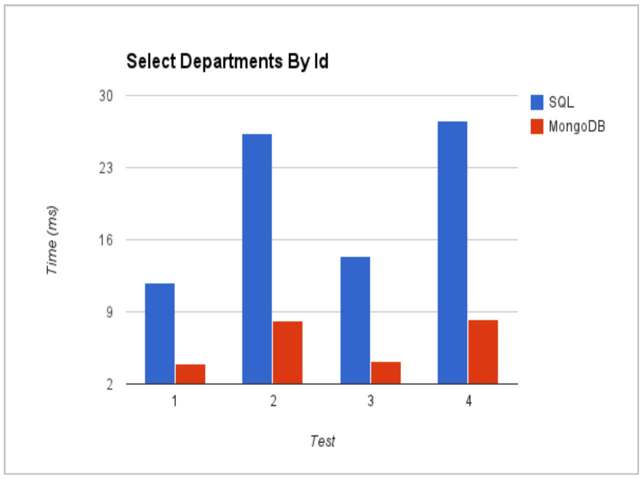

Figure 7. Select – Department ID

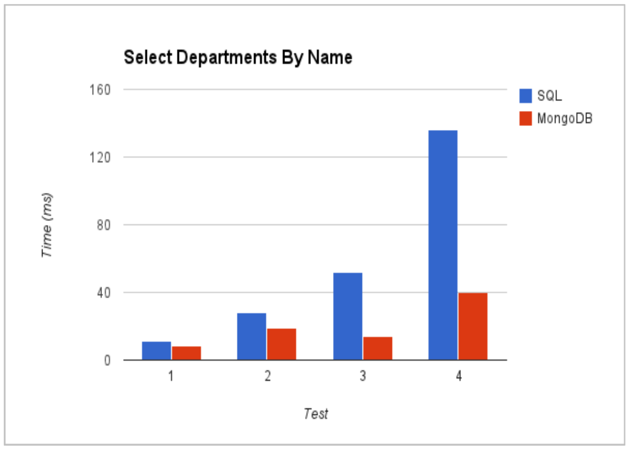

Figure 8. Select – Department Name

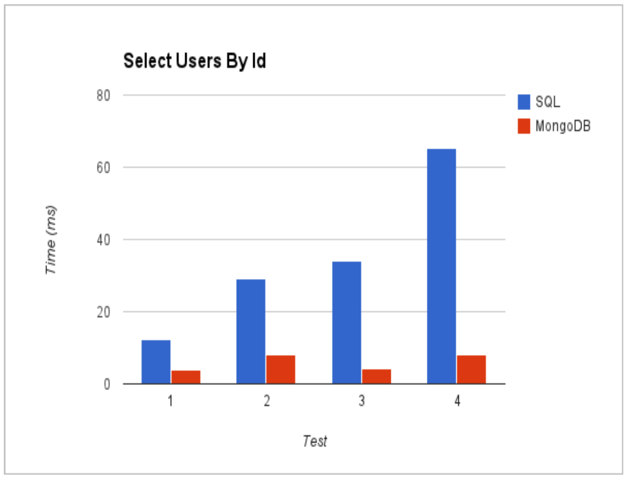

Figure 9. Select – User ID

The query in Figure 10 involves selecting Users based on their First Name. The results in Figure 10 are consistent with the results in Figure 4 for Test Cases 1-3, in which Users are updated based on their First Name. SQL has faster running times than MongoDB for Test Cases 1-3 because it is able to choose the individual attribute in the relational table more easily than locating the value in MongoDB's collection. However, when the data grows large enough in Test Case 4, the runtime of SQL is 23% higher than the runtime of MongoDB. The SQL processing time of this query becomes slower than the lookup of MongoDB as the amount of data increases.

Figures 7-9 illustrate that MongoDB outperformed SQL in all tests by a large amount. We believe this is due to the combination of the index used by MongoDB and its use of memory. MongoDB uses memory mapped files to store all its documents in memory rather than on disk (as long as the system has the available free memory). Since SQL Server has to retrieve all its data from disk and disk speed is slower than memory fetch time, MongoDB outperforms SQL.

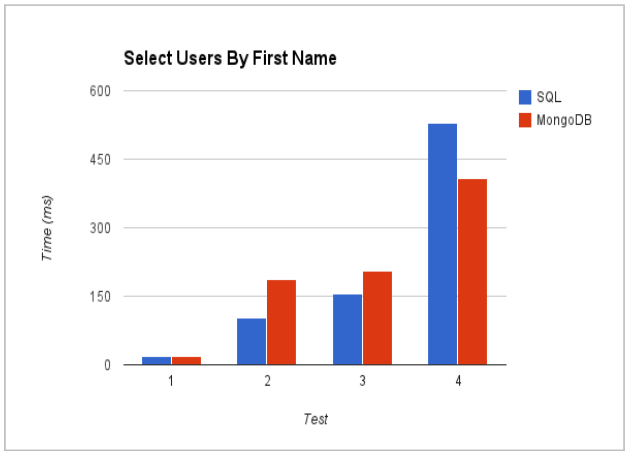

Figure 10. Select – User First Name

For the more complex queries in Figures 11 and 12, MongoDB has a dramatically faster running time than SQL. The runtimes for MongoDB are so low, that they are not always visible in the figures. The runtimes for MongoDB in Figure 10 range from 0.13 for Test Case 1 to 0.19 for Test Case 2. The runtimes for MongoDB in Figure 11 range from 0.24 for Test Case 1 to 0.42 for Test Case 4.

As shown in Figure 13, MongoDB performs well on the complex queries, except when it comes to using aggregate functions. In Figure 13, for Test Case 4, SQL has a runtime of 1,836 ms while MongoDB has a runtime of 41,754ms. MongoDB is almost 23 times slower than SQL. We believe this is due to MongoDB having no true aggregate functions defined for most methods, such as the SQL Average function. Since MongoDB has to use the MapReduce method to allow the developer to create their own aggregate functions, this slows down the performance by a considerable amount.

Unlike using SQL, additional implementation decisions were required which had an effect on the performance of MongoDB. Copying the data in each collection instead of a reference may have resulted in faster performance, although it would have required more storage. In addition to determining how to implement the relationships, an implementation of aggregate functions was also needed.

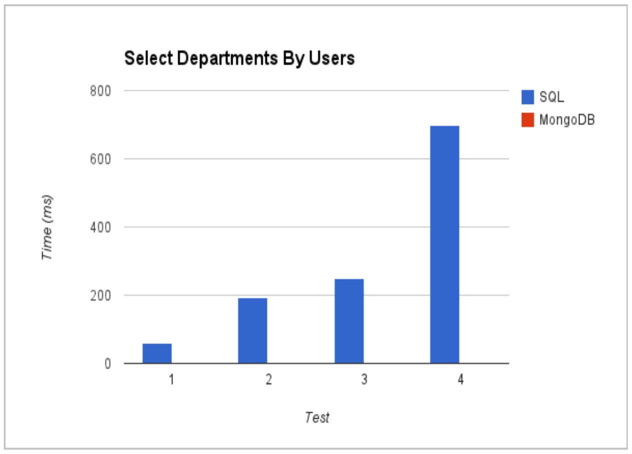

Figure 11. Select Departments by Users

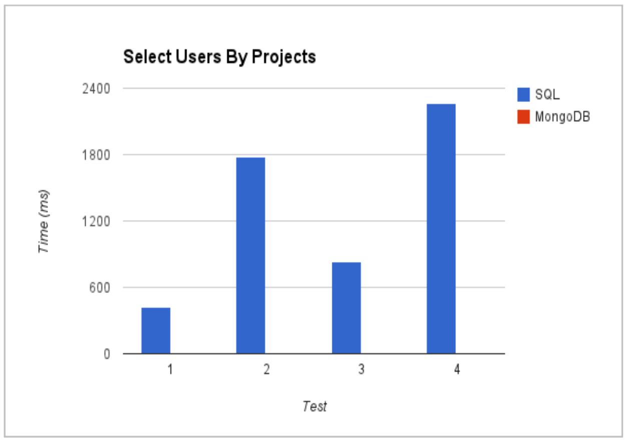

Figure 12. Select Users by Project

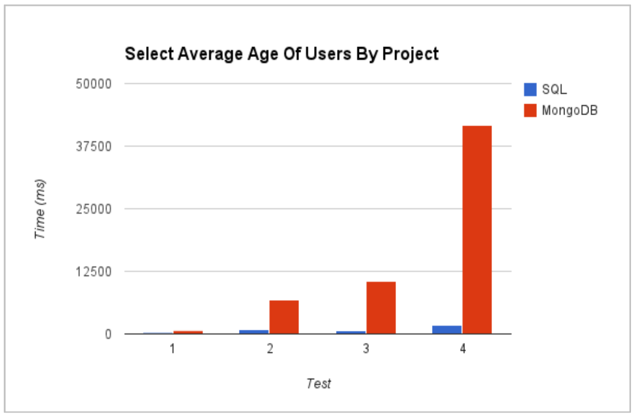

Figure 13. Select Average Age of Users by Project

5. CONCLUSIONS AND FUTURE WORK

Overall, when comparing SQL to MongoDB, MongoDB has better runtime performance for inserts, updates and simple queries. SQL performed better when updating and querying non-key attributes, as well as for aggregate queries. MongoDB could be a good solution for larger data sets in which the schema is constantly changing or in the case that queries performed will be less complex. Since MongoDB has no true schema defined and SQL requires a rigid schema definition, MongoDB would easily handle a dynamic schema such as a document management system with several dynamic fields and only a few well known searchable fields. In conclusion, MongoDB is definitely the choice for users who need a less rigid database structure. MongoDB could be a good solution for larger data sets in which the schema is constantly changing or in the case that queries performed will be less complex.

For those users that have a strict schema defined and a modest amount of structured data we also found MongoDB to perform better than SQL in general. However, there were downsides to MongoDB, such as its poor performance for aggregate functions and querying based on non-key values. Also, MongoDB required additional effort in implementation compared to SQL and required decisions that affected its performance. Lastly, SQL is the industry standard and much more widely supported over MongoDB.

One thing we would like to see as future work from this paper would be running MongoDB and SQL as a distributed database. MongoDB is known to work best as a distributed database so the performance for its complex queries should increase when it is used in this fashion. Other future work would involve running similar experiments on both MongoDB and SQL with a larger and much more complex schema. In theory MongoDB should outperform SQL due to it not requiring a true schema to be defined. Since the relational SQL database has considerable overhead and requires additional joins in a more complex schema, we theorize that SQL's performance would continue to decrease compared to MongoDB.

6. REFERENCES

[1] Elmasri, R. and Navathe, S. 2011. *Fundamentals of Database Systems*, Addison-Wesley.

[2] Strauch, C. NoSQL Databases. *Selected Topics on Software-Technology*, Stuttgart Media University.

[3] NoSQL. http://nosql-database.org/

[4] MongoDB. http://www.mongodb.org/

[5] BSON. http://bsonspec.org/

[6] Hecht, R. and Jablinski, S. 2011. NoSQL Evaluation A Use Case Oriented Survey. *Proceedings International Conference on Cloud and Service Computing*, pp. 12-14.

[7] Han, J. 2011. Survey on NOSQL Databases, *Proceedings 6th International Conference on Pervasive Computing and Applications*, pp. 363-366.

[8] Leavitt, N. 2010. Will NoSQL Databases Live up to Their Promise?. Computer Magazine, Vol. 43 No. 2, pp. 12-14.

[9] Stonebreaker, M. 2010. SQL Databases v. NoSQL Databases. *Communications of the ACM*, Vol. 25, No. 4, pp. 10-11.

[10] Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings Symposium on Operating Systems Design and Implementation*.