This paper compares the performance of a relational database (PostgreSQL) and NoSQL database (MongoDB) in retrieving sets of IP intelligence data for information visualization. After an overview of the history of the two database paradigms, a study reporting the performance of an instance of each is described. Also described is database performance in conjunction with visualization software. Results are compared and discussed. Finally, limitations of the study are discussed and areas of future research are introduced.

Headings:

- Relational Databases
- PostgreSQL
- NoSQL
- MongoDB
- Open Database Connectivity (ODBC)
- Data Visualization
DATA VISUALIZATION SERVICE: SQL VS. NOSQL

by
Landon J Grindheim

A Master’s paper submitted to the faculty of the School of Information and Library Science of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in Information Science.

Chapel Hill, North Carolina
April 2015

Approved by

_______________________________________
Arcot Rajasekar
Table of Contents

1. Introduction ........................................................................................................... 2
2. Literature Review ................................................................................................... 3
   2.1 Introduction to Relational databases ................................................................. 3
   2.2 Introduction to The NoSQL movement ............................................................. 4
   2.3 Data Visualization ............................................................................................. 7
3. Anticipated Results ............................................................................................... 8
4. Methodology .......................................................................................................... 9
   4.1 Environment ...................................................................................................... 9
   4.2 Ingestion .......................................................................................................... 11
   4.3 Retrieval Testing ............................................................................................. 12
   4.4 Retrieval Results ............................................................................................ 15
   4.5 Tableau Testing ............................................................................................... 18
   4.6 Tableau Testing Results .................................................................................. 18
5. Discussion ............................................................................................................. 20
6. Limitations ........................................................................................................... 21
7. Future Study ......................................................................................................... 23
8. Conclusion ............................................................................................................ 23
9. References ............................................................................................................ 25
Appendix 1 ............................................................................................................... 28
Appendix 2 ............................................................................................................... 29
1. Introduction

The term ‘big data’ has become commonplace in our cultural vocabulary, yet the term remains puzzling for many. It is the apparent wave of the future: at once a force for scientific discovery and a threat to our privacy. A fitting description is that big data “is a popular term used to describe the exponential growth and availability of data, both structured and unstructured.” (SAS, n.d.) The growth and availability referred to here has been technology driven, particularly by innovations in the processing and storage of data.

At the heart of these innovations is a group of data storage solutions usually referred to as NoSQL databases. Initially developed by companies like Google, Facebook and Amazon, NoSQL databases have seemed to be the heir to relational databases, which have been popularly used for decades in the storage and retrieval of information. Relational databases have many limitations; among them are the reliance on a strong, predefined data model and the inability to scale well.

Just as the paradigm of database usage seemed to be shifting from relational databases to NoSQL databases, there has been an acknowledgement of the apparent shortcomings of NoSQL as well. Data consistency is not guaranteed, so sensitive information can be mishandled without the database operating improperly. Still, there exist many technical applications in which NoSQL databases, with their heightened scalability, flexibility and resulting performance, seem to make relational databases obsolete. Yet relational databases, with their ability to process transactions like a
withdrawal from a bank account or a password update without losing data, remain necessary.

Another quandary that has presented itself in the exponential growth of data is how to make sense of the growing information. One increasingly popular analysis method is data visualization. This paper will be an exploration of the performance of relational and non-relational databases, with an emphasis on their functionality in enabling data visualization. Upon completion of this study, the following question will have been answered: how do a relational database and a NoSQL database compare in serving data to visualization tools?

2. Literature Review

2.1 Introduction to Relational databases

Modern relational database management systems (RDBMS) have been widely used in a variety of settings for decades now. Based on the relational model of data proposed by E.F. Codd in 1970 (Codd, 1970), RDBMS allow lossless retrieval of structured data. In the time thereafter Codd developed relational algebra and relational calculus based on the model, both of which were mined for the creation of the Structured Query Language, or SQL.

SQL was designed for the query and management of structured data like employee and financial records. Within relational databases, data is stored logically in two-dimensional tables, with horizontal rows representing entities and vertical columns representing their attributes. “Relationships” are defined when equivalent data is stored in separate tables using primary keys and foreign keys. SQL uses these keys to perform
logic that joins tables and allows the retrieval, insertion, deletion and updating of data records (Gansen Zhao, Weichai Huang, Shunlin Liang, & Yong Tang, 2013). SQL has become such a ubiquitous database programming language that it has become more or less synonymous with relational databases.

While RDBMS are often referred to as SQL databases, a more fitting alias would be ACID databases. This refers to the properties, or guarantees, supported by RDBMS in processing transactions. These guarantees are atomicity (A), which implies that all parts of a transaction must be executed or the transaction does not take place; consistency (C), which implies that data which is inconsistent may not be introduced in a transaction; isolation (I), which implies that concurrent transactions will not affect the execution of each other; and durability (D), which implies that the transaction is persistent and the results will be stored physically (Frank, Pedersen, Frank, & Larsson, 2014). These guarantees are the reason that RDBMS have been relied upon for decades and are as widespread as they are today. They are also the reason that technologies other than RDBMS are being developed and used as the world embraces Big Data.

2.2 Introduction to The NoSQL movement

The term NoSQL first appeared as the name of an RDBMS, Strozzi NoSQL, developed by Carlo Strozzi in 1998. While the database was relational, it did not rely on SQL, hence the name NoSQL (Gansen Zhao et al., 2013). The term reappeared in 2009 when Eric Evans used it while promoting a meetup for people interested in distributed data storage systems (Evans, 2009). It has been used since then to refer to a group of non-relational, distributed datastores that typically shirk some or all of the ACID properties on which RDBMS rely.
NoSQL databases, such as Google’s BigTable, Amazon’s Dynamo and Facebook’s (now Apache’s) Cassandra, were built to scale easily. Scalability, more specifically horizontal scalability, in databases is the ability of a database system to add nodes, or partitions. Partitioning results in a distributed database, which can prove useful in many industries where storing data locally is preferable to using a monolithic, centralized storage system (Nicola & Jarke, 2000). However, as a database scales, the data stored therein may also become distributed, which requires replication.

When distributed databases rely on replication, the ACID properties of a transaction are threatened. Consider what happens when a node that contains replicated data temporarily loses its connection and a transaction occurs. Either the entire transaction must fail in order to guarantee atomicity or consistency becomes compromised. Because of this, RDBMS are not naturally partition tolerant. Despite this, the trajectory of societal data creation and consumption propels the need for distributed systems.

Eric Brewer of the University of California at Berkeley put forth a principle that described the three desirable properties of distributed data systems: consistency (C), high availability (A) and partition tolerance (P) (E. A. Brewer, 2000). Brewer’s principle, often referred to as the CAP theorem, states that any distributed data system can have at most two of these three desirable properties (E. Brewer, 2012). For example, if a database is partitioned it can either be highly available or consistent. It cannot be both. Likewise, in order to be highly available and consistent, a system cannot be partitioned (Gilbert & Lynch, 2002).
NoSQL databases are developed with these principles in mind, and their developers are aware of the shortcomings described in the CAP theorem. Contemporary NoSQL databases can largely be grouped into four general categories based on their data model. The categories are column-oriented, key-value, document and graph (Jing Han, Haihong, Guan Le, & Jian Du, 2011). Each category excels at tasks where other categories falter. Column-oriented databases like Google’s BigTable store records in tables like RDBMS, but with the primary focus being on the column, or attribute, rather than the row. Forsaking the row in favor of the column allows for much faster retrieval (Chang et al., 2008). In key-value databases like Amazon’s Dynamo each ‘row’ is simply a columnar value with a universally unique identifier and a timestamp. Dynamo shirks strict consistency in favor of a method that guarantees records to be ‘eventually consistent’ by distributing copies of each record to three nodes. As conflicts will be eventually resolved, this allows the database to be extremely highly available by storing records regardless of what has been written previously (DeCandia et al., 2007).

Document store databases like MongoDB and CouchDB specialize in the retrieval of semi-structured data, like webpages, using the ‘document’, rather than the relational row, as the focus of their design. This allows for great flexibility as the data model does not need to be predefined. Graph databases like Neo4j use graphs to store information about the relationships between entities. Whereas joining data is a costly operation in a relational database, and impossible in most NoSQL databases, graph databases are designed to excel at it (Hecht & Jablonski, 2011). To add to the diversity of the NoSQL movement, within each NoSQL category exists a host of individual applications, each with their own strengths and weaknesses. What these databases forfeit in eschewing the
general-purpose functionality of RDBMS, they gain in performance at specific tasks, which is how they are currently being used.

Comparing ACID and NoSQL databases is in a sense comparing apples to oranges. By design one operates best when processing a limited amount of data on a single node, while the other is meant to handle much greater loads over a network. Yet they are both referred to as databases. Because of this, some database-specific comparisons have been done. However, since NoSQL technologies are relatively new, and the discrepancy in structure between ACID and NoSQL systems is so great, there is still much room for evaluating systems in relation to each other (Parker, Poe, & Vrbsky, 2013; van der Veen, van der Waaij, & Meijer, 2012). One broad area of comparison in which there is room for study is in how the databases interact with applications. One such application is data visualization.

2.3 Data Visualization

Data visualization is the graphical communication of information, a logical extension of modern descriptive statistics. Done properly, it “can provide a qualitative overview of large and complex datasets, can summarize data, and can assist in identifying regions of interest and appropriate parameters for more focused quantitative analysis.” (Grinstein & Ward, 2002, p21) While the visual representation of information has been practiced for millennia, software-based data visualization has been around since the 1960s (Chen, Härdle, & Unwin, 2007).

Forces similar to those that have driven the expansion of database options have driven the growth and expansion of visualization tools. There exists now an abundance of software that powers the automatic visualization of information, many of which can
connect directly to database sources. Similar to NoSQL databases, data visualization tools are selected for their appropriateness to a situation. Some excel in presenting graphical representations of streaming data, while others boast the ability to represent network data. Still, others satisfy general situations. Among these are Tableau, Qlikview and Spotfire (Leishi Zhang et al., 2012).

Considering the relative youth of NoSQL databases and the abundance of visualization tools, knowing the current strengths and limitations of disparate database platforms in serving data to visualization software seems an appropriate and meaningful avenue of exploration. There is no known literature published that addresses this subject directly. To start a dialogue, I have tested and compared the performance of a single relational database and a single NoSQL database in conjunction with a single visualization software.

3. Anticipated Results

Prior to the execution of the experiment tests, it was my suspicion that retrieval results between the out-of-the-box PostgreSQL instance and the out-of-the-box instance of MongoDB, a popular NoSQL database, will be somewhat similar, with PostgreSQL slightly outperforming MongoDB. The reason behind this suspicion is that RDBMS automatically index records on a predefined unique attribute. MongoDB provides this functionality as well, but the primary key is added to the record rather than using the record’s unique attribute. I suspected that optimization of MongoDB will eliminate this edge, ultimately giving MongoDB an advantage.
It was also my suspicion that MongoDB and PostgreSQL will serve data to Tableau at the same rate, with the optimized instance of MongoDB gaining an advantage. However, due to the schemaless nature of NoSQL databases, I predict that the sets of data served by both MongoDB instances will not be as complete as the sets served by PostgreSQL.

Results can be found in Section 4.4 and Section 4.6. I discuss my findings in Section 5.

4. Methodology

4.1 Environment

All database processes were run on a single machine running the Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-44-generic x86_64) operating system with 450GB of disk space and 8GB of RAM. The machine was configured solely for the purposes of database testing. After the Ubuntu operating system was installed and configured, only PostgreSQL and MongoDB were added to the machine.

While a stable version of PostgreSQL 9.4 was released during the process of performing research, for the sake of continuity an upgrade was not made. Thus, all tests were done using PostgreSQL 9.3.6. Rather than allowing an application to interfere with the evaluation of performance of PostgreSQL, testing was done in the PSQL interactive terminal using SQL. Timing was performed by issuing the \ timing command.

The version of MongoDB used for testing was 2.6.7. Similarly to the PostgreSQL tests, all tests on MongoDB were done in the Mongo shell while an instance of the Mongod daemon ran the database server. In the Mongo shell Javascript was used to issue
database commands. Timing was initiated by setting the system profiler to record all actions. It was analyzed by querying the system profile, either over a timespan in the instance of complex queries or by viewing the last \( n \) actions in the instance of simpler queries.

Of interest to this study was performance gains that could result from optimizing MongoDB. In setting up an optimized instance of MongoDB, the same version (2.6.7) and environment were used. Following the guidelines laid out by 10gen, MongoDB’s creators, collections were capped, that is given a size limit that would enable sequential access to documents (Optimization strategies for MongoDB — MongoDB manual 3.0.0.). The size limit were relative to the size of the corresponding collections in the stock instance of MongoDB. Indexes were also created for the fields pertinent to query operations.

The version of Tableau Desktop, the visualization software used in this study, was 8.3, which was installed on a machine running a 64-bit version of Windows 7 Enterprise. The Windows machine had 465GB of disk space and 8GB of RAM. In an attempt to minimize variables that might affect performance, programs running in the foreground were limited to Tableau and a text editor.

Connections to the databases were made using 64-bit ODBC drivers. The PostgreSQL driver was obtained through Tableau, while the MongoDB driver was obtained through Simba Technologies. Both drivers allowed the databases to be interacted with using a relational-type interface. Because this alters the way in which data is retrieved from MongoDB, with the driver seeming to set up a separate connection for each collection, database timing methods become unreliable. In order to resolve this a
stopwatch was used to record timing of the amount of time taken by Tableau to complete a task.

4.2 Ingestion
In order to ensure accurate comparison, each database received the same data. Files containing IP intelligence data were obtained from MaxMind in the form of their GeoLite2 database. The database consists of two tables, one that contained ranges of IPv4 addresses which had been converted to a single integer value (using an inet_aton function) and a location identifier for each range. The other table held the location identifier and its corresponding location information. All location records contained country, latitude and longitude information. The following fields were populated with varying degrees of participation: region, city, area code, postal code and metro code.

In order to use the database files, a set of anonymized IP addresses was obtained from the Center of Applied Internet Data Analysis (CAIDA) which represent a portion of their global domain name server (DNS) traffic for the day of February 2nd, 2009 (CAIDA, 2014). IP addresses were transformed into integer values using the same function as the MaxMind tables prior to being inserted into either database. An entity-relationship diagram can be found in Figure 1 in Appendix 1.

Data from the files was inserted into PostgreSQL tables using the \COPY function. For MongoDB, data from the files were imported into separate collections using the mongoimport function. Though of lesser importance to the study, it is worthwhile noting that MongoDB reported insert rates more than four and a half times faster than that of PostgreSQL. It should be noted that the reporting of these times is inexact in the
case of MongoDB. They are reported as summaries outside the Mongo shell, whereas PostgreSQL reports exact times inside its shell.

Additionally, table size in PostgreSQL was larger than collection size in MongoDB for the corresponding files. This may be partially due to the use of primary key constraints, which trigger the creation of an index. However, MongoDB stores an identifier for each document upon insert. With these differences in mind, a summary of insertion performance can be found in Table 1. Queries used to create and populate tables/collections can be found in Appendix 2.

| Table/Collection | PostgreSQL | | | MongoDB | | |
|-------------------|------------|--|------------|-----------|
|                   | Create Time | Size (in MB) | Create Time | Size (in MB) |
| locations         | 19396 ms    | 142          | 3623 ms     | 74         |
| blocks            | 57198 ms    | 219          | 12495 ms    | 200        |
| ips               | 24549 ms    | 184          | 4636 ms     | 47         |

Table 1 - creation times and table/collection sizes

4.3 Retrieval Testing

During the testing of each database, the other database’s server was shut down in order to limit the possibility of other variables that could affect performance. During testing of MongoDB, the PostgreSQL server was stopped, and vice versa. Each separate query represented a test. For each retrieval test, the database server was restarted and the query was run five times. This allowed the database to move preliminary results into memory as one would expect to encounter in a production environment. The result of each query, as well as the time taken to execute the query, was recorded for analysis.

In reporting the results there was an assumption that equivalent sets would be retrieved by the two databases. While this was not always the case, it allowed for timing
comparisons to be communicated more clearly. As was stated, there were a few cases where sets were not equivalent. These came when handling the postal_code field, which varied in length and character type. Whereas PostgreSQL handled it as a string, MongoDB seems to have accepted the field as an integer in some cases and a string in others. Because MongoDB does not strongly type values, it was inconsistent in the way in which it handled the postal_code field. Instances in which sets were not equivalent have been noted with asterisks (*) in the summary tables.

4.3.1 - Single-Table Queries

Three groups of tests were formed. For each set, queries were written to reflect typical set retrieval operations used by Tableau. Equivalent queries were written in SQL for retrieval in PostgreSQL and in JavaScript for retrieval in MongoDB.

The first set of tests comprised of five queries designed to fetch results from single tables/collections. Queries focusing to the aggregation of fields; matching one, two and four clauses; and comparing ranges of integers and text were written to simulate the types of operations requested by Tableau. An example of one such query, which aggregates one field and matches four clauses, can be found below, first written in SQL, then in JavaScript.

**PostgreSQL (SQL)**

```sql
SELECT COUNT(id) FROM locations WHERE country = 'US' AND region = 'NC' AND postal_code = '27514' AND city = 'Chapel Hill';
```

**MongoDB (JavaScript)**

```javascript
// JavaScript code for MongoDB
```

4.3.2 - JOIN Queries
The second set of queries was created to compare the ability of each database to return data from multiple tables. The purpose of these queries was to compare database performance using one JOIN operation. However, there are several difficulties inherent to this task. Primary among them is the fact that MongoDB does not support JOINs (MongoDB, 2015). In fact, it was designed to eschew them in favor of storing and retrieving schema-less documents. Because of this, a workaround was used for comparison. Queries which included JOIN operations were written in SQL, then split apart and written equivalently in JavaScript. Results returned from the first query would be passed into the second query and so on. This resulted in multiple rather than single queries being issued to MongoDB. The results of each query and the time taken to execute the queries was recorded, with the sum of the timings used for comparison with PostgreSQL. Though problematic and not without its limitations, this method was only used for basic comparison.

4.3.3 – Denormalized Flat-Table “Document” Queries
The third set of queries is similar to the first set of queries, but was performed on a denormalized, or flattened, table. Records from all three tables were joined and exported to a single table containing all values. This introduced redundancy to the records, which is antithetical to the mission of relational databases. However, this table was intended to simulate the way in which MongoDB interacts with documents, that is
that all information pertaining to one record is found within that record. Because the operations involved were similar to those involved in the first set of queries, the queries were adapted for the flat table. All three sets of queries can be found in Appendix 2.

4.4 Retrieval Results

It was found that the execution of the first set of queries, designed to retrieve information from individual tables, was done more quickly by the out-of-the-box instance of PostgreSQL than the out-of-the-box instance of MongoDB in almost all cases. PostgreSQL was able to aggregate and match individual fields efficiently, and the addition of clauses did little to slow down the execution. This may not surprise those familiar with PostgreSQL’s query planning. Meanwhile, MongoDB (200ms) took almost three times longer than PostgreSQL (69ms) to aggregate and match a single field. Additional clauses affected the execution of the queries significantly, with a four-clause query taking almost 50% longer to execute (298ms vs 200ms). The one instance in which MongoDB outperformed PostgreSQL was in executing a range query on a field containing postal_codes. The out-of-the-box MongoDB instance was able to compare two fields against a single value in less than half (63ms) the time it took the PostgreSQL instance (139ms). However, as was described in Section 4.3, there was inconsistencies in the ways in which the databases handled the postal_code field. This resulted in sets which were not equivalent, making performance comparison difficult or inaccurate.

Upon optimizing MongoDB, which involved capping collection size for ordered reads and adding indexes, these trends began to change. The optimized instance of MongoDB (12ms) was able to aggregate and match one field more than five times more quickly than PostgreSQL (69ms) and more than 16 times faster than the out-of-the-box
instance of MongoDB (200ms). Adding clauses only improved the optimized instance’s query execution time. Matching four clauses took the optimized instance only 4ms, almost 19 times faster than PostgreSQL (71ms) and almost 80 times faster than the out-of-the-box instance of MongoDB (298ms). The greatest gains were in matching the postal_code field, where the optimized instance was able to execute the query in 0.58ms. This was 108 times faster than the out-of-the-box instance of MongoDB (63ms) and almost 240 times faster than PostgreSQL (139ms). There was one case in which the optimized instance performed worse than PostgreSQL, and worse than the out-of-the-box instance of MongoDB as well. This was in matching two fields to a single integer. As MongoDB doesn’t strongly type data, it is unclear why this occurred.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>PostgreSQL</th>
<th>MongoDB</th>
<th>MongoDB - Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select - one clause</td>
<td>69.43</td>
<td>199.62</td>
<td>12.45</td>
</tr>
<tr>
<td>Select - two clauses</td>
<td>69.01</td>
<td>258.66</td>
<td>63.33</td>
</tr>
<tr>
<td>Select - four clauses</td>
<td>70.67</td>
<td>298.04</td>
<td>3.73</td>
</tr>
<tr>
<td>Range – Integer</td>
<td>41.49</td>
<td>1237.50</td>
<td>1745.64</td>
</tr>
<tr>
<td>Range - ZIP – Int*</td>
<td>139.71</td>
<td>63.33</td>
<td>0.58</td>
</tr>
<tr>
<td>Range - ZIP – String*</td>
<td>n/a</td>
<td>522.12</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Table 2 - execution times for RDBMS-oriented single table queries

The second set of queries, which was meant to compare the ability of the databases to execute JOIN-type operations, saw different results. In both queries of the set PostgreSQL retrieved results more quickly than MongoDB, optimized or otherwise. PostgreSQL executed each query at least six times faster than the out-of-the-box instance of MongoDB. Interestingly, the gains seen by the optimized instance of MongoDB in the first set were not replicated here. One query saw a reduction of only one ms, while the
other query was executed significantly quicker, but not at the levels previously seen (1287ms to 877ms).

<table>
<thead>
<tr>
<th>Query Type</th>
<th>PostgreSQL</th>
<th>MongoDB</th>
<th>MongoDB - Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select – JOIN</td>
<td>214.71</td>
<td>1548.61</td>
<td>1547.80</td>
</tr>
<tr>
<td>Select - JOIN</td>
<td>194.20</td>
<td>1287.54</td>
<td>877.40</td>
</tr>
</tbody>
</table>

*Table 3* - execution times for RDBMS-oriented table queries

The third set of queries, designed to retrieve data from larger, flat tables, were perhaps unsurprisingly executed by the databases similarly to the first set of queries. Aside from the single-clause aggregation query and a range query, the optimized instance of MongoDB performed even better against both PostgreSQL and the out-of-the-box instance of MongoDB than it had with the first set. Again, PostgreSQL performed each query more quickly than the out-of-the-box instance, though this time on every query. The greatest disparities were in four-clause select query, with the optimized instance executing the query almost 560 times more quickly than PostgreSQL (0.22ms to 212ms), and the postal_code range query, with the optimized instance executing over 420 times more quickly (0.52ms to 218ms).

<table>
<thead>
<tr>
<th>Query Type</th>
<th>PostgreSQL</th>
<th>MongoDB</th>
<th>MongoDB - Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select - one clause</td>
<td>119.18</td>
<td>512.26</td>
<td>36.32</td>
</tr>
<tr>
<td>Select - two clauses</td>
<td>118.59</td>
<td>690.70</td>
<td>6.91</td>
</tr>
<tr>
<td>Select - four clauses</td>
<td>121.42</td>
<td>498.80</td>
<td>0.22</td>
</tr>
<tr>
<td>Range - Integer</td>
<td>93.37</td>
<td>432.05</td>
<td>54.55</td>
</tr>
<tr>
<td>Range – ZIP*</td>
<td>217.67</td>
<td>420.38</td>
<td>0.52</td>
</tr>
</tbody>
</table>

*Table 4* - execution times for MongoDB-oriented flat-table queries
4.5 Tableau Testing

Of particular interest to this study was the performance of the databases in serving data to visualization software. Tableau Desktop was chosen as it is one of the leaders in the market of commercial visualization software. It also supports SQL queries as a base for visualization, which provided more environmental control for accurate performance testing. Tableau uses their trademarked VizQL to translate actions into database queries.

In testing the databases with Tableau, a connection was established via an ODBC driver and an equivalent SQL query was given each database. In all, three SQL queries were used as a foundation on which to build the visualization, each representing different database interactions. The first query depended on a JOIN and column comparisons to retrieve data from all three tables. While visualization can be about discovery, the results of the queries were known beforehand. The second query used a JOIN to retrieve data from two tables. The last query pulled data from the single flat table which was intended to accommodate MongoDB’s strengths.

For each query, a set of predetermined actions was taken. As each interaction with the visualization triggers a query to the database, the amount of time taken to render changes was recorded in order to compare the databases. Timing was done using a stopwatch as Tableau does not support timing mechanisms.

4.6 Tableau Testing Results

In the testing process it became evident that there were significant limitations in the ability of MongoDB to be interacted with as a structured data source, even with the aid of an ODBC driver. Though all queries had been tested using Tableau’s Custom SQL
interface to ensure correct syntax and function, at runtime both queries which relied on the JOIN operator time either failed or ran for nearly an hour before being cancelled.

Error reporting by the three technologies was insufficient to determine the cause of the failure. It may have been that either MongoDB or Simba’s driver ran out of available memory to perform the operations. This is possible as the query used to pull data from three tables encountered errors within around 20 seconds of the same time frame, that is after around 15 minutes and 45 seconds. The query pulling data from two tables was aborted by the tester after 55 minutes using the non-optimized version of MongoDB and after 60 minutes using the optimized version. Because of this the exact numbers involved in the comparison are unavailable. To ensure that the setup was capable of rendering data using JOINs, a smaller dataset was queried without a customized SQL query. Tableau was able to render a visualization from that dataset.

Conclusions can still be drawn when comparing the performance of MongoDB with that of PostgreSQL. Where Tableau was waiting to render data from a single query from MongoDB after 55 minutes, the equivalent query was executed using PostgreSQL in under two seconds. Full results can be seen in Table 5, below.

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL</th>
<th>MongoDB</th>
<th>MongoDB-Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query Type</strong></td>
<td>execution time (Min:Sec.ms)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Three Tables</td>
<td>25.91</td>
<td>failed (~15:45)</td>
<td>failed (~15:45)</td>
</tr>
<tr>
<td>Two Tables</td>
<td>2.06</td>
<td>&gt; 55:00.00</td>
<td>&gt; 60:00.00</td>
</tr>
<tr>
<td>One Table</td>
<td>10.94</td>
<td>46.71</td>
<td>46.62</td>
</tr>
</tbody>
</table>

Table 5 - response times for query operations via Tableau

Similarly, where execution of a query issued to MongoDB failed after almost 16 minutes, Tableau was rendering the same requested data from PostgreSQL after just over 13 seconds. In the instance of querying a single table, both databases were able to render
the data. PostgreSQL was still the star performer here, rendering a visualization in less than one quarter the time it took either instance of MongoDB.

5. Discussion

In retrieving sets of data using queries that do not require JOINs, it was apparent that optimization increased MongoDB’s performance greatly. Comparisons of performance using queries that rely on JOINs becomes less straightforward. Because MongoDB does not support JOINs, the join must either be done outside of the database, or records must be denormalized in such a way to contain all pertinent information within a single document.

It seems a simple statement to say that while ODBC drivers have enabled Tableau to render data from MongoDB, the performance of MongoDB, the ODBC driver or a combination of both has left much to be desired. True, the ODBC driver used to connect to MongoDB is attempting to mimic the execution of a query in a relational database, so it could be expected that its ability to do so would fall behind that of a relational database’s ODBC driver. However, the disparity in performance between the two was unexpected. This could be due to a variety of reasons.

In the process of retrieving data from MongoDB, it was apparent that the MongoDB ODBC driver opened four database connections. There was no indication that more than one connection to PostgreSQL was opened by its ODBC driver. It is very possible that in the process of retrieving queries from MongoDB results were being returned as a Cartesian product of the tables before being culled for meeting the specific requirements of the query. If this were the case, a query which is meant to return one
record would be returning up to over two million records. This would offer an explanation as to why the query joining three tables failed in a similar timeframe in all tests. This would result in joining one record from the IP table with 1.2171041e+12 records from a product of the location and block tables. Whatever the case, the mechanisms involved in retrieving data from the two databases using Tableau are certainly not the same.

Of particular interest is that the significant performance gains due to optimization that were observed in Mongo shell were not observed when queries were executed via Tableau. This again seems to point to the operations of the ODBC driver. It is possible that driver was not accessing available indexes. Curiously, the gains made possible by capping collections were also not visible. However, because intermediate tests were not conducted, performance changes due to the capping of collections cannot be reported. These factors make it difficult to make inferences about the lack of performance gains.

6. Limitations

There are several limitations that exist within this study. Primary among them are that the findings of the comparison are applicable only to the tested versions of database and visualization software running in an environment similar to the one in which the tests were executed. Had the tests been executed using distributed instances of the databases, the results would likely be very different. It is understood that under these conditions MongoDB, being designed for such tasks, would very likely greatly outperform PostgreSQL.
As both MongoDB and Tableau exist within burgeoning fields of development, it is possible that the findings of this study will become outdated as new versions of the software are released. Improvements in the ability to connect the two will also likely arise in the near future. As of the writing of this paper, third-party ODBC drivers were required in order to connect Tableau to MongoDB. The general consensus seems to be that there are two ODBC drivers appropriate to the task, Simba Technologies’ driver which was used in this study, and Progress DataDirect’s driver. If my suspicion, that this specific field represents an area wherein new solutions will be developed, is correct, then the results of these tests will be made obsolete by new findings.

Though comparisons of performance in serving information to data visualization software may lose their usefulness somewhat quickly, the results of the database comparisons in retrieving sets of information will likely remain valid for some time. However, Just as PostgreSQL is a proven yet still developing technology (a new version with ‘big data’ capabilities was released during the course of this study), it is possible that MongoDB will develop new ways to ‘join’, or at least allow multi-step queries that enable SQL-like searching. However, at this point, this seems antithetical to MongoDB’s purpose. Because of this it is assumed that comparisons of shell-based set retrieval will hold true for some time.

The focus of this study is set retrieval. It is not necessarily generalizable to other domains of retrieval. Even within the context of this study, had there been a greater emphasis on geographic information systems (GIS) data and the functionality associated with it, the study setup and methodology would differ. PostgreSQL would have been paired with PostGIS and all instances of MongoDB would have been set up to optimally
retrieve such data. As someone who lacks expertise in GIS data and its use, it seemed inappropriate to execute tests on applications that would be easily misunderstood. Thus, these comparisons are only valid for set retrieval.

7. Future Study

With the above-listed limitations in mind, there exists plenty of avenues for future exploration of database and visualization software performance. The logical next step would be to test these applications at a larger scale and distributed over multiple machines. As MongoDB was created to perform in such circumstances it seems very probable that differing results would be observed. Scaling the dataset to near PostgreSQL’s limits and distributing the data over several nodes would give a better-rounded picture of how these databases can perform these tasks.

Another area of interest would be the testing of retrieving GIS data and testing GIS database extensions. In its appropriation of point data using latitude/longitude coordinates. This study approaches the periphery of the topic, but the study of the GIS capabilities of the databases exists outside the scope of this paper.

Of particular interest in light of these findings would be a comparison of ODBC drivers that are compatible with MongoDB. As the driver is marketed as a big data solution, it was surprising to observe such poor performance from the setup. I would like to know if separate ODBC drivers would affects performance, and if so, to what extent.

8. Conclusion
In basic set retrieval, optimizing MongoDB allows for significant performance improvement, both over a non-optimized instance of MongoDB and over PostgreSQL. PostgreSQL outperformed the non-optimized instance of MongoDB in set retrieval. This confirmed the expectations I had going into the experiment stage. Also confirmed was the expectation that optimization would give MongoDB the edge over the out-of-the-box instance of PostgreSQL. However, the expectation was that the edge would be slight, while the reality is that the improvements were much greater than anticipated, on occasion representative of an order of magnitude.

In serving data to Tableau visualization software, the current capabilities of MongoDB, facilitated by Simba Technologies’s ODBC driver, was found to be lacking. Though the ODBC driver enabled visualization capabilities, the resulting performance was poor rather relative to that of PostgreSQL. Under the current setup, optimization through capping collections and creating indexes has little to no effect on the retrieval process via Tableau. These results were not anticipated, and as such exist as a point of interest.
9. References


Appendix 1

Figure 1. Entity-relationship Model for database Paper (PostgreSQL)
Appendix 2

Schema Definition (SQL)

CREATE TABLE locations (  
id bigint NOT NULL,  
country character(2) NOT NULL,  
region character(2),  
city character varying(75),  
postal_code character varying(15),  
latitude numeric(6,4) NOT NULL,  
longitude numeric(7,4),  
metro_code integer,  
area_code integer,  
CONSTRAINT locations_pkey PRIMARY KEY (id));

CREATE TABLE blocks (  
startIp bigint NOT NULL,  
endIp bigint NOT NULL,  
locId bigint NOT NULL,  
CONSTRAINT blocks_pkey PRIMARY KEY (startIp, endIp, locId));

CREATE TABLE ips (  
ip_id bigint NOT NULL,  
ip_inet inet,  
message VARCHAR(250),  
ip_aton int,  
CONSTRAINT ips_pkey PRIMARY KEY (ip_id, ip_inet));

CREATE TABLE allData1 (  
ip_id bigint NOT NULL,  
ip_aton bigint NOT NULL,  
startip bigint,  
endip bigint,  
locid bigint NOT NULL,  
country VARCHAR(2),  
region VARCHAR(2),  
city VARCHAR(75),  
postal_code VARCHAR(15),  
latitude numeric(6,4),  
longitude numeric(7,4),  
metro_code int,  
area_code int,  
allData_id serial PRIMARY KEY);
Population Commands - PostgreSQL
\COPY locations (id, country, region, city, postal_code, latitude, longitude, metro_code, area_code) FROM '~/geoLite/GeoLiteCity-Location1.csv' WITH CSV HEADER;

\COPY blocks (startIp, endIp, locId) FROM '~/geoLite/GeoLiteCity-Blocks1.csv' WITH CSV HEADER;

\COPY ips (ip_id, ip_inet, message, ip_aton) FROM '/home/grindhei/IPdata/ipdata.tsv';

Note: allData table was populated using a series of commands. The timing of its population process was not considered in this study.

Collection Creation/Population Commands (MongoDB)
mongoimport --db paper --collection location --type csv --headerline --file
~/geoLite/GeoLiteCity-Location1.csv

mongoimport --db paper --collection blocks --type csv --headerline --file
~/geoLite/GeoLiteCity-Blocks1.csv

mongoimport --db paper --collection ips --type tsv --fields ip_id,ip_inet,message,ip_aton --file ~/IPdata/ipdata.tsv

mongoimport --db paper --collection allData --type csv --headerline --file
/home/grindhei/allData.csv

Collection Creation/Population Commands (MongoDB - Optimized)
// in shell — create capped collections
db.createCollection( "location", { capped: true, size: 145513440 } )
db.createCollection( "blocks", { capped: true, size: 224829808 } )
db.createCollection( "ips", { capped: true, size: 188732000 } )
db.createCollection( "allData", { capped: true, size: 192921744 } )

// outside the shell
mongoimport --db paperOpt --collection location --type csv --headerline --file
~/geoLite/GeoLiteCity-Location1.csv

mongoimport --db paperOpt --collection blocks --type csv --headerline --file
~/geoLite/GeoLiteCity-Blocks1.csv

mongoimport --db paperOpt --collection ips --type tsv --fields ip_id,ip_inet,message,ip_aton --file ~/IPdata/ipdata.tsv

mongoimport --db paperOpt --collection allData --type csv --headerline --file
/home/grindhei/allData.csv
Indexing Commands (MongoDB - Optimized)

db.location.createIndex( { country: 1 } )

db.location.createIndex( { region: 1 } )

db.location.createIndex( { postalCode: 1 } )

db.location.createIndex( { city: 1 } )

db.blocks.createIndex( { startIpNum: 1, endIpNum: 1 } )

db.location.createIndex( { locId: 1 } )

db.allData.createIndex( { country: 1 } )

db.allData.createIndex( { region: 1 } )

db.allData.createIndex( { postal_code: 1 } )

db.allData.createIndex( { city: 1 } )

db.allData.createIndex( { startip: 1, endip: 1 } )

Retrieval Queries - Set 1

PostgreSQL

# query 1 - Match Query (1 param)
SELECT COUNT(id) FROM locations WHERE country = 'US';

# query 2 - Match Query (2 params)
SELECT COUNT(id) FROM locations WHERE country = 'US' AND region = 'NC';

# query 3 - Match Query (4 params)
SELECT COUNT(id) FROM locations WHERE country = 'US' AND region = 'NC' AND postal_code='27514' AND city = 'Chapel Hill';

# query 4 - Range Query (int)
SELECT locId FROM blocks WHERE startIp < 2382516573 and endIp > 2382516573;

# query 5 - Range Query (string)
SELECT id, city FROM locations WHERE postal_code > '27500' AND postal_code < '27600';

MongoDB

# query 1 - Match Query (1 param)
db.location.find( { country: 'US' } ).count()

# query 2 - Match Query (2 params)
db.location.find( { country: 'US', region: 'NC' } ).count()

# query 3 - Match Query (4 params)

# query 4 - Range Query (int)
db.blocks.find( { startIpNum: { $lte: 2382516573 }, endIpNum: { $gte: 2382516573 } }, { locId: 1 } )

# query 5 - Range Query (string)
  # part 1 - retrieve based on int
  db.location.find( { postalCode: { $gt: 27500, $lt: 27600 } }, { locId: 1, city: 1 } )
  # part 2 - retrieve based on string
  db.location.find( { postalCode: { $gt: '27500', $lt: '27600' } }, { locId: 1, city: 1 } )

Retrieval Queries - Set 2 (JOIN attempts)

**PostgreSQL**
  # first query
  SELECT latitude, longitude FROM locations INNER JOIN blocks
  ON locations.id = blocks.locId WHERE blocks.startIp < 2382516573 and blocks.endIp > 2382516573;

  # second query
  SELECT latitude, longitude FROM locations INNER JOIN blocks
  ON locations.id = blocks.locId WHERE blocks.startIp < 1195815127 and blocks.endIp > 1195815127;

**MongoDB**
  # first query
  db.blocks.find( { startIpNum: { $lte: 2382516573 }, endIpNum: { $gte: 2382516573 } }, { locId: 1, _id: 0 } )
  db.location.find( { locId: 375056 }, { latitude: 1, longitude: 1 } )

  # second query
  db.blocks.find( { startIpNum: { $lte: 1195815127 }, endIpNum: { $gte: 1195815127 } }, { locId: 1, _id: 0 } )
  db.location.find( { locId: 861 }, { latitude: 1, longitude: 1 } )
Retrieval Queries - Set 3 (flat table)

**PostgreSQL**

# query 1 - Match Query (1 param)
SELECT COUNT(locid) FROM allData WHERE country = 'US';

# query 2 - Match Query (2 params)
SELECT COUNT(locid) FROM allData WHERE country = 'US' AND region = 'NC';

# query 3 - Match Query (3 params)
SELECT COUNT(locid) FROM allData WHERE country = 'US' AND region = 'NC' AND city = 'Chapel Hill' AND postal_code = '27514';

# query 4 - Range Query (int)
SELECT locid FROM allData WHERE startip <= 1270726656 and endip >= 1270728191;

# query 5 - Range Query (string)
SELECT locid, city FROM allData WHERE postal_code > '27500' AND postal_code < '27600';

**MongoDB**

# query 1 - Match Query (1 param)
```
db.allData.find( { country: 'US' } ).count()
```

# query 2 - Match Query (2 params)
```
db.allData.find( { country: 'US', region: 'NC' } ).count()
```

# query 3 - Match Query (4 params)
```
```

# query 4 - Range Query (int)
```
db.allData.find( { startip: { $gte: 1270726656 }, endip: { $lte: 1270728191 } }, { locid: 1 } )

db.system.profile.find().pretty()
```

# query 5 - Range Query (string)

# part 1 - retrieve based on int
```
db.allData.find( { postal_code: { $gt: 27500, $lt: 27600 } }, { locid: 1, city: 1 } )
```

# part 2 - retrieve based on string
```
db.allData.find( { postal_code: { $gt: '27500', $lt: '27600' } }, { locid: 1, city: 1 } )
```
Tableau Queries

**PostgreSQL**

```sql
# JOIN ALL DATA FOR CHAPEL HILL, NORTH CAROLINA
SELECT i.ip_id, i.ip_aton, b.locId, l.country, l.region, l.city,
    l.postal_code, l.latitude, l.longitude, l.metro_code, l.area_code
FROM ips AS i, blocks AS b, locations AS l
WHERE l.id = b.locid AND i.ip_aton BETWEEN b.startIp AND b.endIp
AND l.country = 'US' AND l.region = 'NC' AND l.city = 'Chapel Hill'

# JOIN BLOCKS AND LOCATIONS FOR NORTH CAROLINA
SELECT b.locId, l.country, l.region, l.city, l.postal_code, l.latitude,
    l.longitude, l.metro_code, l.area_code
FROM blocks AS b, locations AS l
WHERE l.id = b.locid
AND l.country = 'US' AND l.region = 'NC'

# PULL IN ALL DATA
SELECT * FROM allData
```

**MongoDB**

```sql
# JOIN ALL DATA FOR NORTH CAROLINA, CHAPEL HILL
SELECT i.ip_id, i.ip_aton, b.locId, l.country, l.region, l.city,
    l.postalCode, l.latitude, l.longitude, l.metroCode, l.areaCode
FROM ips AS i, blocks AS b, location AS l
WHERE l.locId = b.locId AND i.ip_aton BETWEEN b.startIpNum AND
    b.endIpNum
AND l.country = 'US' AND l.region = 'NC' AND l.city = 'Chapel Hill'

# JOIN BLOCKS AND LOCATIONS FOR NORTH CAROLINA
SELECT b.locId, l.country, l.region, l.city, l.postalCode, l.latitude,
    l.longitude, l.metroCode, l.areaCode
FROM ips AS i, blocks AS b, location AS l
WHERE l.locId = b.locId
AND l.country = 'US' AND l.region = 'NC'

# PULL IN ALL DATA
SELECT * FROM allData
```