Calculating Apriori in Real-Time using NoSQL Databases

Devlin Cronin
rdcronin@calpoly.edu

Austin Dworaczyk Wiltshire
adworacz@calpoly.edu

Therin Irwin
tcirwin@calpoly.edu

Stephanie Krapil
skrapil@calpoly.edu

Halli Meth
hmeth@calpoly.edu

ABSTRACT
Many database applications are written such that they require extensive and computationally-heavy analysis of the data stored; however, this can serve as a bottleneck in many cases. We examine the benefits of converting a SQL database to a NoSQL database for the implementation of an Apriori calculation to determine the most common "baskets" within a data set. After testing the system with BerkeleyDB, MongoDB, and CouchDB, using the original MySQL implementation as a baseline, we find that all NoSQL implementations are faster than the RDBMS counterpart, and that the BerkeleyDB implementation runs orders of magnitude faster than all others.

1. INTRODUCTION
The Animal Science Department at California Polytechnic State University, San Luis Obispo records experiments using a database containing information on animal genes. Specifically, the database stores information about the genes in the mammary glands of different animals, including the transcription factors and regulatory sequences, and information about the frequency and location of these within the gene. The database is currently implemented entirely in MySQL, and can take a prohibitive time to retrieve the answers to certain queries needed by Animal Science researchers.

This paper documents our results of attempting to speed up one query, an apriori calculation, with three different NoSQL implementations: CouchDB, MongoDB, and BerkeleyDB. This paper will discuss background, related work, the current system, implementations of the current system, CouchDB, MongoDB, and BerkeleyDB, our results, and a conclusion.

2. BACKGROUND AND RELATED WORK
2.1 MySQL vs. NoSQL
Database Management Systems are constructed by weighing various factors and choosing a subset of benefits that optimize the goals of the system, while sacrificing as little as possible. These factors include efficiency, availability, the capability for a multi-user system, ACID properties (atomicity, consistency, isolation, durability), scalability, integrity, rich queries, and rich data models. Each system is implemented to fully accommodate, partially accommodate or not accommodate each factor [11].

MySQL is optimized for transactional systems such as financial processing. The query language is optimized for rich querying and strict consistency. However, MySQL only works well with structured data. The user must specify the schema for all of their data ahead of time. Alternatively, NoSQL works with unstructured or semi-structured data. In the simplest form, a key-value store maps a Key "String" to a value "String". NoSQL makes compromises with the level of querying provided to the user out-of-box. Some NoSQL systems are also limited in that they provide eventual, rather than immediate consistency [3]

2.2 Apriori Algorithm
A common problem in the field of data mining is to determine frequent itemsets for a series of market baskets. A market basket is a large quantity of lists of items that occur together. The problem is then to figure out whether certain items frequently occur together in subsets. The solution uses the apriori algorithm.

The apriori algorithm consists of a series of breadthwise searches over the entire dataset [8, 9]. The first pass through searches for frequent itemsets of size 1, the second pass through searches for frequent itemsets of size 2, and the kth pass through searches for frequent itemsets of size k [4]. The complexity of the algorithm would grow exponentially if not for pruning. If on step i itemset y is found to not be frequent, any itemset in step i+1 that includes itemset y as a subset is guaranteed to not be a frequent itemset [4]. We are able to further limit the algorithm by defining minimum and maximum supports. By defining supports, you are finding the itemsets that occur in min to max percent of the market baskets. This is useful to weed out frequent itemsets that occur in every market basket or only occur in a small percentage of market baskets.

2.3 NoSQL Systems
In order to decide which NoSQL DBMSs to utilize in our new implementations, we looked at a variety of the most widely-used databases of varying types. In our decisions, we

attempt to find significantly different architectures so that we could more likely associate the cause of significant differences in speed to the architecture, rather than specific implementations. We selected three different types of NoSQL DBMSs to contrast with the current MySQL implementation: a key-value store, a highly-available document store, and a highly-consistent document store.

2.3.1 BerkeleyDB

For the key-value store, we decide to use BerkeleyDB. This is because it is widely-used, freely available, and has been an industry standard since its release more than fifteen years previously. [14] BerkeleyDB has been used prevalently by consumers, and has also been used as the persistent store for larger, more complex systems like Amazon's Dynamo and Project Voldemort. [7, 2] Additionally, BerkeleyDB is extremely lightweight, and is thus a good candidate for our key-value store, for which we want something which will cause as little overhead as possible. [14] We want this small amount of overhead so that we can analyze the speed with which we can perform the calculations externally, rather than through built-in database functionality.

2.3.2 MongoDB

We also chose MongoDB. MongoDB is a well constructed and well tested JSON-based document store. It is in use at a number of high profile companies, including Craigslist, Disney, Intuit and Foursquare. Additionally, MongoDB has a built in MapReduce framework, which lends itself well to the implementation of the Apriori algorithm to determine significant genetic transcription factors. MongoDB has an excellent amount of documentation, as well as a variety of client modules for a multitude of programming languages, including Python, C++, JavaScript (Node.js), Perl, and many more.

2.3.3 CouchDB

We chose Couchbase to as another Document store, in addition to Mongo. We wanted to be sure that any performance issues we encountered were the result of using MapReduce, not a specific type of database. Its management console was the deciding factor in it being a choice. One of the big benefits was being able to work with a MapReduce query in the browser console for debugging and getting results quickly. Additionally, the browser console informed the user when a view's index was building, which was also very important since we made significant use of views. Although their client support is lacking for views in many languages, they have SDKs for Java, Node.js, Ruby, .net, and C.

3. CURRENT SYSTEM

3.1 Brovine

Brovine is the Cal Poly Animal Science Department's Mammary Gland Gene Database. The database is designed "for storing information about the regulatory sequences of proteins found in the promoter regions of the genes responsible for the synthesis of these proteins" [10]. These proteins are also called transcription factors because they bind to regulatory sequences and translate sections of DNA into RNA, which leads to gene expression. The customers of the database are interested in studying transition phases of mammary glands, such as the process of going from being pregnant to lactating. To do this, the customers perform experiments on mammary gland cells in each phase to discover the genetic factors present in the cells. These experiments are performed through an "application that can analyze the similarities and differences between the collections of regulatory sequences for different proteins" [10]. The customers wish to examine the probability that each regulatory sequence actually exists and the relationships between the transcription factors of different phases. The database currently contains data on bovine and murine (cow and mouse, respectively) genes.

The current system is a PHP application built using the CodeIgniter framework. The backend is a MySQL database. Although the size of the database is currently not large, it is the customer's intent to eventually store several orders of magnitude more data. Approximately 156 unique genes and 400 unique transcription factors are currently in the database, so we tested our implementations on up to 15,600 unique genes. We did not, however, test using an order of magnitude more unique transcription factors - this number does not necessarily affect the run-time complexity of the algorithm because we're counting the total number of factors (that is, the number of transcription factors in each bucket and the support percentage is what counts - not the number of total uniques). Additionally, the customers do not expect the number of unique transcription factors to scale linearly with the number of unique genes, as there are typically many duplicates within each gene.

3.2 Apriori Query

Brovine applies the apriori algorithm to determine which transcription factors and sets of transcription factors occur frequently in animal genes. According to the database owners, "With each gene stored in the database, a list of transcription factors found to bind to its promoter region is associated. It is of interest to the users of the system to be able to find the lists of transcription factors that are commonly shared, and be able to view the list of genes that share these transcription factors" [12].

We apply the apriori algorithm using genes as market baskets and the transcription factors contained in them as items. The Animal Science Department is interested in determining the groups of transcription factors that frequently occur in genes [12]. However, they are uninterested in any groups of transcription factors that occur in every gene, so for their case the apriori algorithm will take in a minimum and maximum support value as seen below.

For this experiment, we are executing the apriori query on four different systems: the original MySQL system and implementations in CouchDB, MongoDB, and BerkeleyDB. The query will be run with a minimum support of .835 and a maximum support of .95 (this corresponds to baskets which appear in between 130 and 148 of the 156 total genes). We will be testing this with both the current dataset and an expanded dataset to analyze how well the system will perform when more data is added.

4. BROVINE'S EXISTING IMPLEMENTATION

The existing frequent transcription factor generation is performed using PHP, with a MySQL view hosting the buckets. Each tuple in this table has two attributes, gene ID and transcription factor ID. A bucket is defined as all of the records with a single gene ID. All of the data processing was done using PHP, including the candidate generation and item counts. Currently this implementation is about 50 times slower than the worst NoSQL implementation we completed over the course of this project. However, it is ridiculous to say that this is because of the MySQL database. Much of this speedup is attributed to the move away from PHP as an implementation language, because the array operations it offered were exceedingly slow. The point of this project, however, was more to the effect of examining different implementations and attempting to determine which one was most feasible for the current environment.

5. COUCHDB IMPLEMENTATION

The Couchbase implementation examined the feasibility of using MapReduce to efficiently tackle the Apriori algorithm. The MapReduce query to do this was simple, but not trivial. We broke the algorithm into two parts, the "candidate generation" and the "item counting." Unfortunately, only item counting could be done using MapReduce.

There were several hurdles to overcome before Couchbase was able to process itemset candidates. One of these was the partial lack of API support for saving views and the complete lack of support for extra "scoped" variables, which would have negated the need for saving views through the API [6]. MongoDB, on the other hand, provides widespread support for both features [1]. To avoid having to examine every single combination of subsets (which would require exponential time), Apriori "trims" sets which have any infrequent subsets. So, when determining the frequency of the next set of items, a set of candidates must be provided to the MapReduce functions. These candidates are the only possible sets that could be frequent. This necessitates the need to either re-save the MapReduce functions before each iteration, or pass a different "scope" each time. Fortunately, each run of the Apriori algorithm in our case produced a maximum of 6 or 7 iterations, so the first option was feasible.

Another frustration was the default "show stale view" option. Because we updated the MapReduce functions for each iteration, at first we saw stale data each time the view was queried. This is because Couchbase must build an index around each view before it can be queried. The majority of time spent on the database was building this index. However, we imagine that this indexing is simply building a cache for each object's emit(..)s using the map function, so we doubt that having support for extra "scoped" variables would affect the runtime cost.

6. MONGODB IMPLEMENTATION

MongoDB bills itself as a pure document store, where all documents are stored in a JSON-like format. The official storage format is actually that of BSON, so it technically supports the storage of binary objects such as png or jpeg images. MongoDB sports a number of "niceties" that make it easier to transfer over from a relational database, such as aggregation support, grouping, distinct document retrieval, and limiting. One feature that MongoDB has that is definitely not built into a standard relational database is that of MapReduce.

MongoDB's MapReduce framework is provided in JavaScript, which allows for a seamless integration with it's JSON-style document objects. The Map function is given every document in a collection by MongoDB's architecture, and then the Map results are funneled into the Reduce function, whose output can be stored in an alternative collection or an output dictionary (in the case of the Python API) [1]. This built-in MapReduce support makes MongoDB a prime candidate for Apriori testing.

The MongoDB client was written in Python using the PyMongo module, version 2.4 at the time of this writing. Emphasis was placed on pushing as much work on to the database as possible, and avoiding any extraneous work in Python. We made this decision because with a system such as MongoDB, we wanted to evaluate whether or not the extra functionality of the database was beneficial. If the extra functionality is not beneficial for analysis such as this, then the simplistic key-value store will be faster.

MongoDB proved to be an excellent database overall, and provides a wide array of features that expedite development. In terms of overall speed, it comes in second place when compared against BerkeleyDB and Couchbase. Its performance is detailed in the following charts and graphs.

7. BERKELEYDB IMPLEMENTATION

BerkeleyDB is a simple key-value store based on the dbm architecture created by ATT [?]. The main advantage of BerkeleyDB is its speed and lightweight presence, allowing relatively simplistic databases to be stored efficiently and accessed quickly. Given the nature of the Apriori algorithm analysis which we are performing, if we load all the records of the genes and the transcription factors present in each gene in one read (this is reasonable for the current dataset size, and even for a database 1000 times larger in size), we can have the algorithm only require database access at one point. By doing so, we effectively put the burden of the calculations on the programmatic implementation of the algorithm, rather than on the database.

We use Python bindings for the Oracle's BerkeleyDB 5.1.0 release. The choice to use Python was determined by a number of factors. Most importantly, the development was easier (in our opinion) than if we had used a different language. We find that the production time for the database and the Apriori algorithm in Python is significantly lower than the same in a language such as C, C++, or Java. Additionally, crashes and exceptional behavior are easier to resolve. Finally, we have heard anecdotal evidence that Python's implementation of BerkeleyDB may in fact be faster than Java's. As such, Python was chosen.

Our implementation takes advantage of this property by implementing the Apriori algorithm purely in Python, and using BerkeleyDB only to retrieve the information about the genes and transcription factors, rather than attempting to perform complex joins or calculations with database operations. This is an effective route, as it allows the relatively-

efficient high-level language programming to take on the majority of the necessary work, and has the database used rarely. This may not be optimal in all possible workloads; however, for a workload such as the Apriori algorithm, the ability to use a high-level implementation of the calculation does improve performance significantly.

With this implementation, all data is loaded in one operation. The current size of the database is 405KB, and scales nearly linearly, such that even a database with all the data replicated 1000 times is 406MB in size. With these sizes, loading the entire database into RAM is easily achievable, even on a commodity machine. Therefore, we do not do any sophisticated handling for if the database is greater than the size of RAM, and we can rely on operating system protocols to handle this gracefully.

The end result is that the database is hit for only a small fraction of the total calculation time, and that the total run time for this implementation is far lower than the run time for others, which attempt to offload some of the calculations to the database itself.

8. RESULTS

8.1 System Specs

Our experiments were run on a MacBook Pro with Retina Display. The system has a 4 core 2.3 Ghz core-i7, 8 gigabytes of RAM, and a solid state drive. [5]

8.2 Raw Data

8.2.1 Original Data (1x)

Implementation	DB Access(s)	Query Time (s)
Brovine	<no data>	1434.122
CouchDB	31.939	33.205
MongoDB	9.971	10.146
BerkeleyDB	0.0058	1.589

8.2.2 Replicated Data (10x)

Implementation	DB Access(s)	Query Time (s)
Brovine	<no data>	<no data>
CouchDB	99.912	101.011
MongoDB	82.309	85.798
BerkeleyDB	0.0522	11.652

8.2.3 Replicated Data (100x)

Implementation	DB Access(s)	Query Time (s)
Brovine	<no data>	<no data>
CouchDB	955.025	956.606
MongoDB	811.560	846.286
BerkeleyDB	0.515	117.759

8.2.4 Replicated Data (1000x)

Implementation	DB Access(s)	Query Time (s)
Brovine	<no data>	<no data>
CouchDB	<no data>	<no data>
MongoDB	<no data>	<no data>
BerkeleyDB	6.229	1166.505

8.3 Analysis

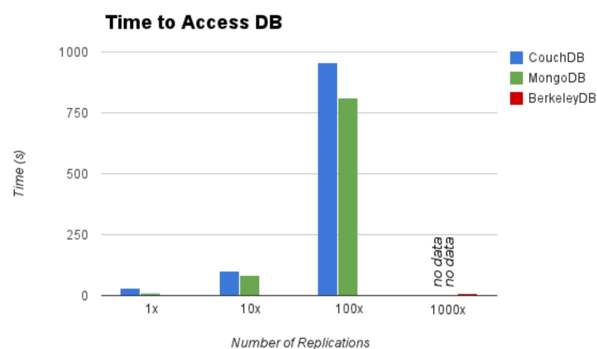
All three NoSQL implementations improved the speed of our Apriori query. The fastest implementation was that of BerkeleyDB. This is due to the fact that the BerkeleyDB implementation required only one read from the database file per apriori run, whereas both the MongoDB and the CouchDB implementations required multiple reads per run. Instead of performing calculations through the database functionality, the BerkeleyDB implementation performs all such calculations in a Python script. Through this experiment, we found that, for computationally-heavy but relationally-sparse data such as this, it is better to not use database functionality when it is avoidable, due to the overhead which is incurred.

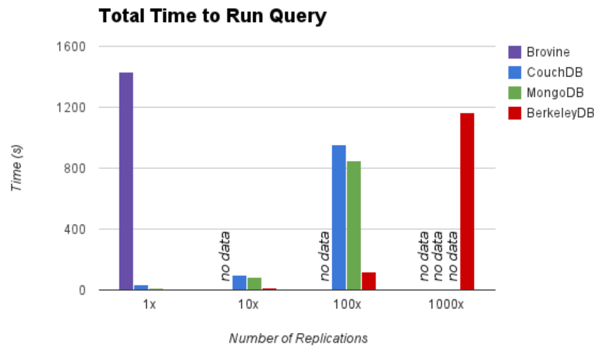
With the increased speed of the BerkeleyDB implementation, it was able to run the query on an increased data set with replication factors of 10, 100, and 1000. It was the only system which could run all of these datasets in a reasonable amount of time, as it was multiple orders of magnitude faster than the other two implementations.

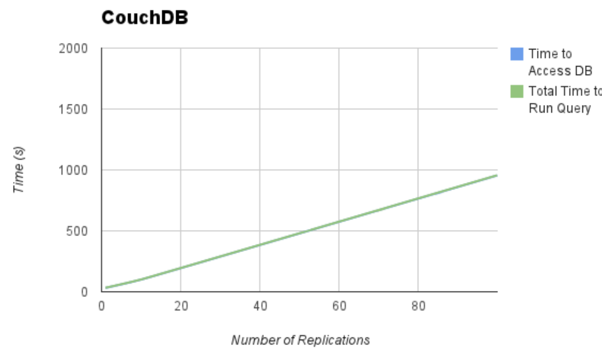
The MongoDB implementation aimed to offload as much of the apriori work onto MongoDB's MapReduce framework as possible. MongoDB's MapReduce framework is written in JavaScript, with all Map and Reduce functions written in JavaScript to operate on a collections stored JSON documents. MongoDB offers a more limited framework called the "aggregation" framework that is written in C++, but its currently available feature set is not enough to perform the operations necessary to determine the most significant factors in our genetic data. That being said, MongoDB's MapReduce framework still proved to perform well enough, beating out CouchDB's implementation by a significant margin.

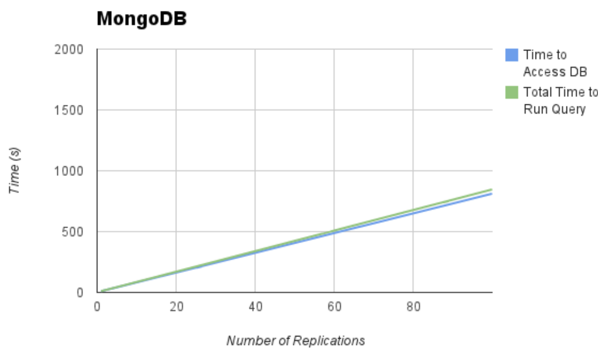
The Couchbase implementation also aimed to offload as much work as possible to the database. This implementation was the slowest of the three implementations we tried on with replication factors of 1, 10, and 100. This is partially due to the overhead provided by the HTTP wrapper for Couchbase, because the 1x replication was significantly better for Mongo than for Couchbase, whereas the 10 and 100x replication were closer.

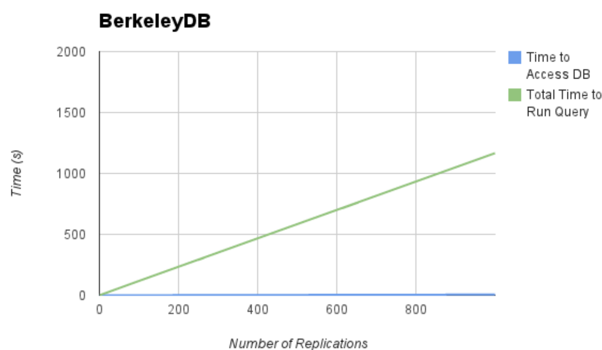
8.4 Graphs











9. FUTURE WORK

The Apriori algorithm could also be replaced by the FP-Growth algorithm to increase speed. In the future we could attempt work with the FP-Growth algorithm and compare it to the current implementations. FP-Growth encodes the data set using a compact structure called an FP-tree and extracts frequent itemsets directly from this tree. Once data has been encoded into the tree, subsequent updates to the data set are easier because the previous tree can be kept and built upon. Changes in the minimum and maximum support values require no modifications to the tree, which make it advantageous for a system like Brovine where a user may be frequently changing the min and max support. However, the tree could require a lot of memory to store based on the initial ordering of sets, but sorting the sets in some specific order first does not necessarily lead to a small tree. Little information is available on the proper sorting order for an optimal FP-tree, and it is likely that it would require the exact information that the FP-tree is used to generate, namely, the frequent items in the set [13].

Another improvement that could be applied to either the Apriori or the FP-growth algorithm is to only use skyline itemsets. Skyline itemsets are any set where there are no supersets that contain any subsets within the skyline set. This could significantly decrease the runtime of either algorithm because it would limit the number of sets that need to be considered on each pass through the data.

10. CONCLUSION

All in all, we have learned that it is not necessarily advantageous to use built-in MapReduce for the Apriori algorithm. Performing the algorithm is much faster on the client-side, using a language like Python. However, it would have been useful to see how the Mongo and Couchbase implementations scaled to a large cluster; but it is unlikely that they would match the speed of the BerkeleyDB implementation, even for a much larger dataset.

11. REFERENCES

[1] Collection level operators.

[2] Project voldemort.

[3] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, H. V. Jagadish, A. Labrinidis, S. Madden, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, K. Ross, C. Shahabi, D. Suciu, S. Vaithyanathan, and J. Widom. Challenges and opportunities with big data, 2011-2012.

[4] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD '93 Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993.

[5] AppleInc. Macbook pro with retina display.

[6] CouchBase. Chapter 9. views and indexes.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[8] A. Dekhtyar. Data mining: Mining association rules.

[9] A. Dekhtyar. Data mining: Mining association rules examples.
[10] A. Dekhtyar. Database requirements analysis and design.
[11] A. Dekhtyar. Dbms: Challenges.
[12] A. Dekhtyar, D. Peterson, A. Conly, and L. Strand. Regulatory sequences database use cases.
[13] P. ning Tan. *Introduction to Data Mining*. Pearson Education, 2006.
[14] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999.
[15] P. W. Purdom, D. V. Gucht, and D. P. Groth. Average-case analysis of the apriori algorithm.