EECS 433 - Database Systems
Project Final Report

Yigit Kucuk, yxk368@case.edu
Devin Schwab, dts34@case.edu

November 30, 2014

1 Abstract

As the semantic web grows bigger it is necessary to come up with more scalable database technologies. Graph databases are a different technique for storing complex and dynamic data (exactly the type of data that the semantic web is composed of). Graph databases can often take better advantage of parallelism through techniques like map/reduce. Graph databases also lend themselves to analyzing the structure of data through algorithms like Page Rank and Single Source Shortest Path (SSSP). However, within the area of graph databases there are many different architectures including: relationally backed databases, bitmap graph databases, and map/reduce graph databases. Understanding the differences between the architectures and how they perform over a standard set of typical queries is important for any database designer. For this project we examined these results and reproduced them over freely available graph databases. For this purpose we chose four databases with different implementations also mentioned in [3] to examine and one data set (LinkedMDB) [7] to use for all 4 databases. Neo4j [9] , Sparksee(or formerly known as DEX) [15] , Giraph [16] and Filament [5].

2 Introduction

Big Data has to deal with two key issues: the growing size of the datasets and the increase of data complexity. One approach is to move away from

the traditional relational databases and store the data as a graph. In many problem domains such as semantic web RDF data, and scientific information graphs provide a more intuitive easy to use model of the data. For example, it is quite natural to represent a social network as a graph.

In addition to reducing impedance mismatch in some problem domains, graph-based models can often be leveraged to increase scalability and perform more complex queries. There are many graph partitioning algorithms that allow a database to span multiple servers. This makes data size easier to deal with and provides opportunity for parallelism. Additionally a lot of interesting information in a graph-based model is **not** in the data itself, but in the structure of the data. Consider a map database, optimal traffic routing and route planning can be calculated based only on the edges between the nodes.

For these reasons there are multitude of commercial and open source graph databases available today. While its easy to find a large list of these databases it is not apparent which one to choose in which circumstances. Different storage schemes, APIs, and licensing can affect the choice of the best database for a particular problem domain. As such this project focuses on benchmarking and comparing different graph databases. As mentioned there are two main types of queries on graph databases: pattern matching, like in SPARQL queries, and structural queries, such as minimum distance between vertices. This project focuses on the latter as they are a unique advantage of graph databases. We build off of prior benchmark results for different databases by repeating some of their methodologies and tests. However, we try to capture a wide range of different implementations to compare and contrast the different classes of graph databases.

3 Project Goals and Related Work

There have been many different benchmarking papers for graph databases. To avoid reinventing the wheel we have mimicked the benchmarks used in these previous papers. This allows our work to be compared with existing results, making the contributions from this project more relevant. *A Performance Evaluation of Open Source Graph Databases* by McColl, et al. gives a large number of results comparing many different graph and non-graph databases. This paper also suggests a number of useful benchmarking algorithms, some of which were used in this project. [12] *An Experimental*

Comparison of Pregel-like Graph Processing Systems by Han, et al. provides a set of benchmarks and results for Bulk Synchronous Parallel Style Databases (BSP). [8]. These two papers formed the bulk of the inspiration for the algorithms and methodology used in this project.

Besides just repeating experiments, this project also set out to add databases using architectures not in the above two papers. *Survey of Graph Database Models* by Angles, et al. provided a wide summary of the different graph database architectures. It detailed data models, data structures used, query languages supported and integrity constraints supported. [1] *The Current State of Graph Databases* provided even more databases to choose from, albeit with much less detail than the Angles paper. We chose our four databases to test from these two papers.

Finally, being a class project we had the additional goal of gaining hands on experience with real world graph database systems. Our goal was to become comfortable with a few of the systems and be aware of what else exists.

4 Databases

4.1 Database Types

The Current State of Graph Databases gives an overview of the main categories of graph databases. There are the mainstream graph databases, that are specifically designed for graph models and have been tuned to support them. There are also distributed graph databases. These databases focus on parallelization and large datasets. Key-value databases use dictionaries to store the mappings between vertices, edges and their various properties. Document graph databases have nodes that are a blob of data. They generally model edges as pointers between the different documents. As relational databases are mature, well understood and offer reasonable performance there are a number of projects that attempt to store graphical models in a relational database. Map/Reduce databases and Bulk Synchronous Parallel (BSP) databases are designed to scale massively and perform well on highly parallel computations. This data model was inspired by a number of projects at Google.

In the next section we explain which databases, we chose, why they were chosen and what category they fall under.

4.2 Chosen Databases

4.2.1 Neo4j(Graph Database)

Neo4j is a disk-based transactional graph database. It works on a network oriented model with relations as first class objects. The API is in Java, and supports Java object storage. The system is very efficient in graph traversals, however currently requires the full dataset on each node (work is being done on transparent partitioning). Neo4j also has partial ACID support and lends itself well to transactional enterprise solutions.

4.2.2 Sparksee (DEX)(Graph Database)

Sparksee is a very efficient, bitmaps-based graph database model written in C++. The focus of Sparksee is performance in the management of very large graphs, and even allows for the integration of various data sources. In addition to the large data capacity, Sparksee has a good integrity model for management of persistent and temporary graphs. Operation or core functionality like link analysis, social network analy- sis, pattern recognition and keyword search is done through their Java API.

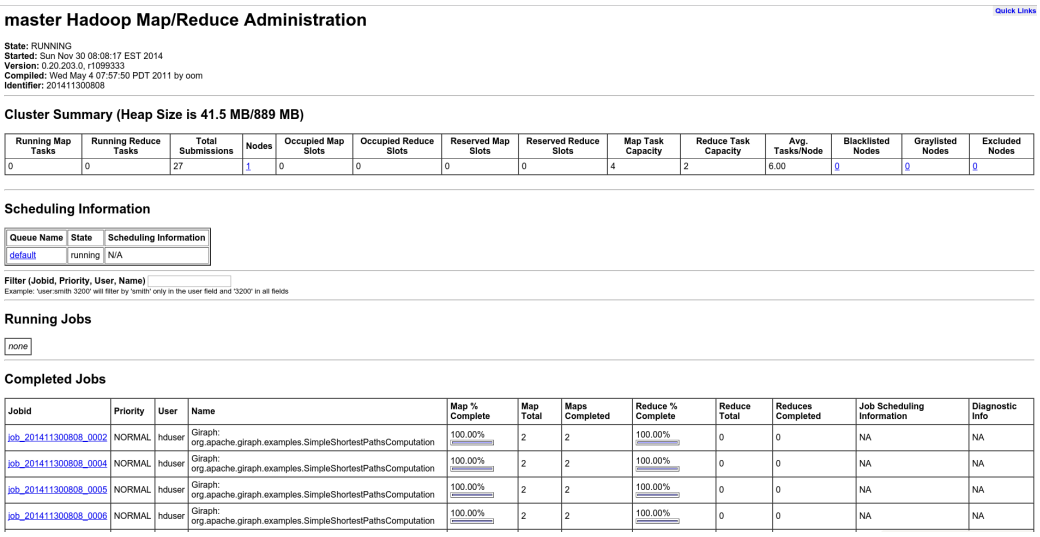
4.2.3 Giraph (Map-Reduce Database/BSP)

Giraph is a Bulk Synchronous Parallel (BSP) database by the Apache Software Foundation. It is an open source project inspired by Google's Pregel system. [11] Rather than having a special master node that directs a group of machines to perform a large scale computation, BSP systems decompose the problem into a set of *supersteps*. Each of the workers is treated as a vertex in a large graph that performs the computation. During a superstep each vertex performs an independent computation over some data. Because the computations are independent each computation is done in parallel. As each vertex computes they can send messages to the other vertices in the computation graph. Each vertex votes based on their internal state and the messages they have received whether to halt. Eventually all of the vertices halt. When this occurs the algorithm has completed its computation and the user can collect the results from each vertex.

This framework is attractive because it does not require locking mechanisms which make reasoning about an algorithm complicated. Having no locks also eliminates the chance of a deadlock or livelock like in normal

asynchronous programming. It is also easy to scale computations by increasing the number of graph nodes. Finally it overcomes a shortcoming of map/reduce frameworks in that there is no single point of failure in the special master computer. For all of these reasons Giraph and other BSP databases are becoming more prevalent and useful for large scale cloud computing.

Giraph also comes with a number of useful features including a built in redundant network file storage layer. It also includes a web interface that can be used to monitor the current status of any long running queries. A screenshot of this interface can be seen in figure 1.



master Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sun Nov 30 08:08:17 EST 2014
Version: 0.20.203.0, r1099333
Compiled: Wed May 4 07:57:50 PDT 2011 by oom
Identifier: 201411300808

Cluster Summary (Heap Size is 41.5 MB/889 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
0	0	27	1	0	0	0	0	4	2	6.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201411300808_0002	NORMAL	hduser	Giraph: org.apache.giraph.examples.SimpleShortestPathsComputation	100.00%	2	2	100.00%	0	0	NA	NA
job_201411300808_0004	NORMAL	hduser	Giraph: org.apache.giraph.examples.SimpleShortestPathsComputation	100.00%	2	2	100.00%	0	0	NA	NA
job_201411300808_0005	NORMAL	hduser	Giraph: org.apache.giraph.examples.SimpleShortestPathsComputation	100.00%	2	2	100.00%	0	0	NA	NA
job_201411300808_0006	NORMAL	hduser	Giraph: org.apache.giraph.examples.SimpleShortestPathsComputation	100.00%	2	2	100.00%	0	0	NA	NA

Figure 1: Screenshot of the Giraph Web UI provided by Hadoop

4.2.4 Filament (SQL Database)

Filament is a java library that can connect to a JDBC database to store and manipulate a graph. It is designed to work with PostgreSQL, but any other relational database with a JDBC driver can be used. The nodes, edges and their properties are stored in three special tables in the relational database. The idea is that the Filament API translates the graph operations to a standard relational database.

Filament stores its data in three tables: edges, nodes, and def_props. The nodes table simply stores a list of all of the nodes in the database which the other two tables reference. edges contains 4 columns: id, field1, field2,

and field3. id is the primary key and is generated by a sequence. field1 is the head of the edge and field3 is the tail of the edge. field2 specifies the edge value. In RDF terms field1 is the subject, field2 is the property and field3 is the object. def_props has 5 columns: id, field1, field2, field3 and lg_field3. id is the same as the other tables. field1 is a foreign key to the nodes table. field2 is the name of the property of this node. field3 is the value of that property. lg_field3 is available for the value stored in a node exceeds a standard VARCHAR datatype.

4.3 Redis Graph Database (Key- Value Graph Database)

Redis graph is a python library that takes advantage of Redis's set capabilities. [13] [14] Redis is a key/value store that can work as both an in-memory map or as a persistent data store. [4] Redis supports many different value types including sets. By storing the edges as a concatenation of node names in a set of strings in Redis one can treat the key/value store as a graph database. Redis graph was the first database that we installed and loaded with data. It was extremely easy to setup and write a program to load the RDF triples. Unfortunately when we ran the data loading program it made it clear that Redis graph would not be capable of keeping up with the other graph databases. Its API only supports adding nodes and edges. To access a node or an edge one needs to know the node name or the nodes the edge connects. Because Redis graph is built on top of a key-value store there is no way to enumerate and filter the node keys efficiently. Furthermore while its python API makes it extremely easy to use, it adds an additional overhead. So we have decided not to implement the previously described benchmarks for Redis graph.

5 Benchmarks

The paper *A Performance Evaluation of Open Source Graph Databases* by McColl et al. contains a number of different benchmarks for graph databases. We based our project's benchmarks off of these. We compared the different database's computational efficiency as well as their storage efficiency. Each of the databases have been setup in docker containers to keep the test environments pristine and aid in repeatability. Docker utilizes the linux kernel LXC (LinuX Containers) to isolate different programs. [6] LXC is like a virtual

machine, but rather than loading a separate copy of every single OS file a single OS kernel is loaded and namespaced so that each program thinks it is running on its own machine. Docker has very little computation and memory overhead compared to a virtual machine and is in use at a number of large companies. Docker also uses an immutable file system so after a container is restarted it will be in the exact same that as it was at the start of the last run. Finally Docker contains built in memory and computation measurement tools. Docker also guarantees that the caches are flushed between runs.

For the computational benchmarks we tested two of the graph algorithms from the McColl et al. paper: Single Source Shortest Path (SSSP), and the PageRank algorithm. Single Source Shortest Path (SSSP) uses Dijkstra's algorithm for calculating the minimum path between a source node and every other node in the graph. For the BSP databases the algorithm is implemented so that the next hop nodes are computed in parallel. After a level is finished all the nodes progress to the next level. The second algorithm is PageRank (PR). PageRank is a measure of a node's importance in the graph. The node's page rank is equal to the probability of being in that particular node given the stationary distribution of a random walk on the graph. The values can be computed by finding the principle eigenvalues for the system of equations, however, this is a very large and involved matrix calculation. So instead an iterative approach is used. Each node updates its weight based on the current weight of the neighbors. In the BSP type database each superstep every node sends out its rank to its neighbors. After a vertex updates its own page rank, if it has changed by a small enough amount then that node votes to halt.

For the non-BSP databases we also evaluated the performance of random edge insertions and random edge deletions. This was designed to highlight which databases would work best for dynamic data vs static data.

Our main metric was the amount of time taken for each test. However, we also looked at memory usage and disk usage where possible.

We additionally measured throughput for random edge insertions and deletions. This gave us an idea of how these databases would perform if used in a production system.

All of the tests were run on standardized hardware. Each set of docker containers was run on a separate VPS provided by DigitalOcean. Each VPS had 2 GB of RAM, a 40 GB SSD, and 2 CPUs. For the giraph cluster tests a private network internal to the datacenter was used to maximize available bandwidth and minimize latency.

All of our experiments used a dump of LinkedMDB. Some of the papers used artificially generated data. We wanted to compare the artificial results with results from a real data set. This also prevented us from having to write a data generator in the limited amount of time we had. LinkedMDB is an RDF database. When loading the data we ignored all RDF meta-data edges.

6 Experimental Results

We ran the different databases with three different subsets of the LinkedMDB dataset. The smallest contained 10K edges and 11494 vertices. The medium dataset contained 100K edges and 58871 vertices. The largest dataset contained 1000K edges and 371919 vertices. The subsets were selected by adding the first 10K, 100K, and 1000K edges from the data dump respectively.

Our results are summarized in figures 2 through 6. Notice that Dex is not included in any of them. This is for two reasons. The first is that the free version of Dex has a limit on the number of edges you can load into the database. This prevented us from testing the largest dataset with Dex. The second reason is that even for the small dataset Dex would segfault partway through loading the data. We are not sure why this was happening, so unfortunately we couldn't resolve the problem. Also notice that in some graphs Neo4j is missing a datapoint for the largest dataset. We found in our testing that Neo4j uses a large amount of RAM. Given the limited 2 GB on the test machines Neo4j would run out of heap space and crash.

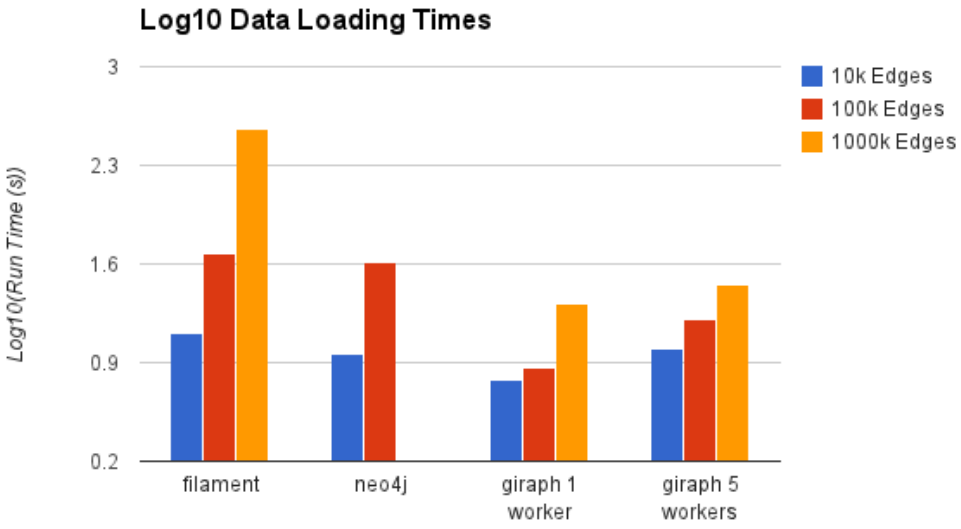


Figure 2: Amount of time to load each of the datasets into the databases. Note that times are log scaled because of drastically different time scales

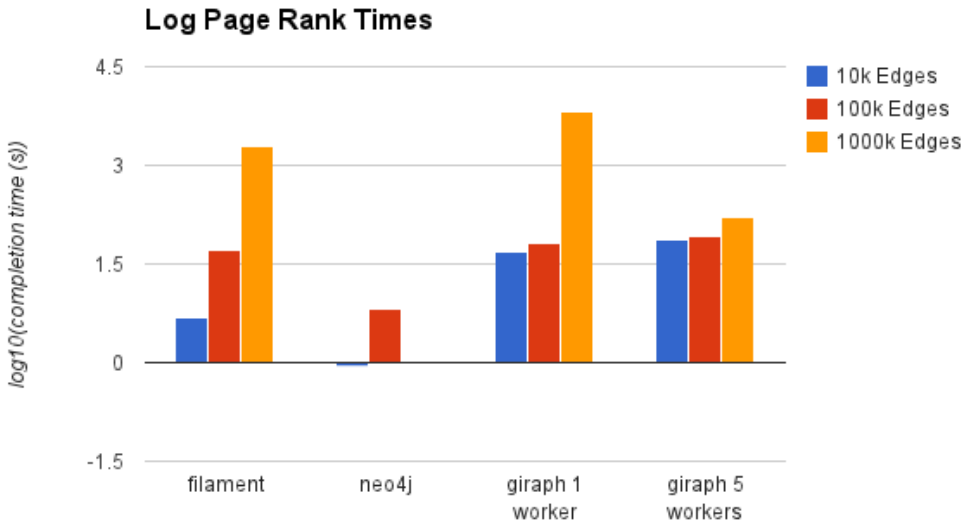


Figure 3: Amount of time to complete Page Rank algorithm. Note that times are log scaled because of drastically different time scales

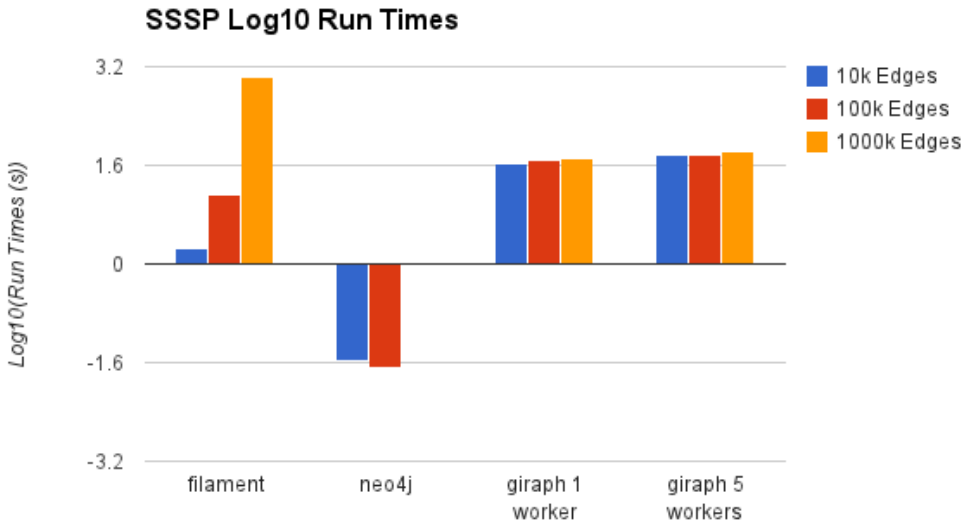


Figure 4: Amount of time to complete SSSP algorithm. Note that times are log scaled because of drastically different time scales

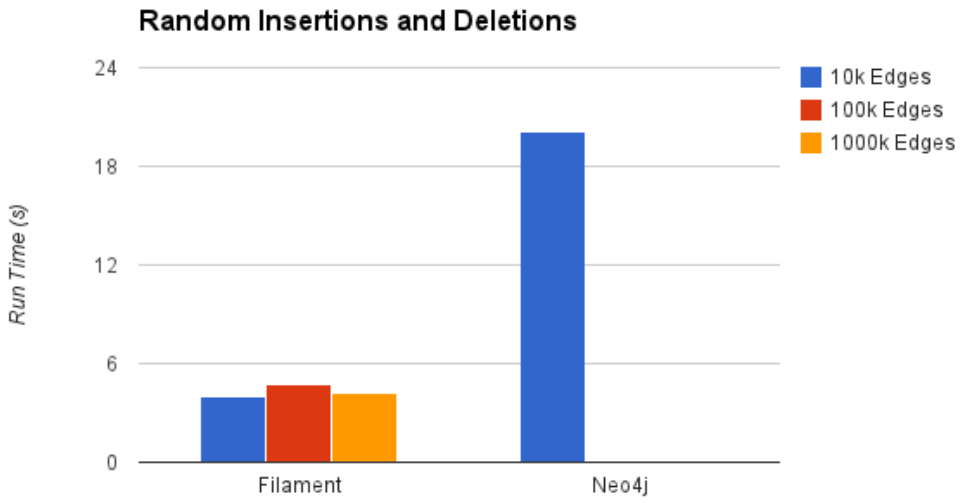


Figure 5: Amount of time to perform 1000 random insertions and 1000 random deletions interleaved

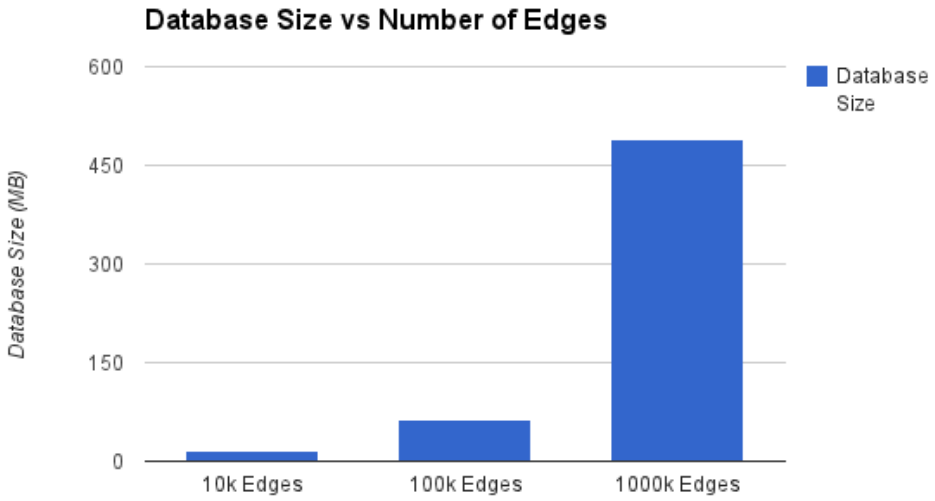


Figure 6: Growth of storage space required for Filament database

As expected all of the databases have an increase in the time required to load data as the number of edges and vertices being bulk loaded increases. However, there was such a large disparity in the scales of these times that it was necessary to plot these times on a log scale in figure 2. All of the databases were approximately the same for the small dataset. However, filament was far worse than the others for the 1000k edges dataset. This indicates that filament does not do well with many inserts compared to the other databases. There is not much difference in loading times between a Giraph cluster of size 1 and a Giraph cluster of size 5. The majority of that time was spent in converting the raw data to a format Giraph could handle. Only a small percentage was actually spent copying the data onto the network file store.

For page rank and SSSP filament starts out competitively again and then quickly increases in time taken. Again these two graphs are shown on a log scale because of the wide range of scales between the different run times. It makes sense that filament would not scale as well as the others given its architecture and API. For page rank a simple select query can get all of the nodes to iterate over and update. But for each node another query must

be made for incoming edges. That means there are $O(|V|)$ select queries run every iteration. In practice we found that for the larger datasets page rank took about 40 iterations. Hence the large jump in time taken between the different dataset sizes. It might be possible for Filament to use a more optimized query, but limiting ourselves to the Java API prevents this.

An oddity in the page rank data is that Giraph with 1 worker performs approximately the same for 10k edges and 100k edges, but significantly worse for 1000k edges. In plain seconds the first two take approximately 55 seconds to complete, but the biggest dataset takes almost 2 hours! We are not sure what causes this large decrease in performance, however, it appears that simply having a bigger cluster fixes the issue. As Giraph with 5 workers has approximately the same run time across all three datasets.

It is interesting to note that while Neo4j could not handle the largest dataset due to memory limitations, for the datasets it could handle it was extremely fast. We believe this is because Neo4j is a commercial database. Both SSSP and Page Rank are useful in many real world applications so it seems likely that Neo4j is highly optimized for these types of queries.

While Neo4j is good at analyzing data it appears to be optimized for static datasets. As performing 1000 random interleaved insertions and deletions is 3 times slower on Neo4j then Filament. The random insertions and deletions were not performed on Giraph so no comment can be made about how fast Giraph can mutate its graph structure.

Finally figure 6 shows how the size of the PostgreSQL database behind Filament grows as the number of edges and vertices grows. The raw data with all of the RDF metadata properties is 429 MB, however, even with all of the metadata properties removed and only including 1 million out of the 2 million triples, Filament requires 490 MB. This indicates that Filament requires a lot of overhead. We believe most of this overhead comes from the various indices that the PostgreSQL database contains. Looking at the schema used by Filament there are a number of B+-trees in use.

7 Experiences with Databases

Testing the different databases in a limited time-line required us to be efficient in learning the usage of databases and how to implement or create our first databases. In this section we give information about our experiences as the users and developers of databases.

Since Neo4j and Sparksee(DEX) are commercialized databases, they supported a number of popular programming languages with mature APIs. However, Among both commercial and open source software packages, we find a lack of consistency in approaches to documentation. It is often difficult to quickly get started. System requirements and library dependencies are not always clear. Quickstart guides were helpful to installing the databases, but most projects lacked an intermediate stage of documentation between the quickstart and the raw API documentation. It is often possible to store large graphs on disk, but the working memory required for the computation can be too much for a modest sized server. For example Neo4j was one of the faster databases to complete the PageRank and SSSP algorithms, but it was too memory intensive to run on the 1000k edges dataset.

Giraph and Hadoop were rather complicated to install. Requiring many different configuration files spread about the system and many different environment variables controlling the operation of the system. However, once installed it was fairly easy to use and the web interface made it easy to monitor and interact with the programs that were running. The API for Giraph was also rather easy to use, although in the end we used the example implementations because they included better file parsing and other nice features that made it easier to run the benchmarks.

8 Conclusion

Our benchmarks tested only a very small subset of the existing graph databases. Even so we can draw some interesting conclusions from the data. Filament degrades with data size quite quickly and consistently across all of the tests. Therefore we would not recommend using a relationally backed graph database unless it was necessary for compatibility with an existing relational database or only small datasets were to be used with it. Additionally, Filament could most likely provide more optimized implementations of common algorithms like PageRank by combining many of the small queries so that the database can handle the optimization and avoid repeatedly querying the same table.

Neo4j appears to be the fastest, but only if you can afford a lot of memory. While its possible to get machines with far more memory than our test machines, there is a limit to how much can exist in one machine so Neo4j is not recommended for databases that are expected to scale to large numbers.

Instead Giraph or a similar BSP database is recommended. Giraph did not slow down that much when loading larger datasets. Additionally while it did perform poorly for 1000k edges when running as a single node cluster, it improved drastically by adding just a few more machines. This indicates that Giraph can handle extremely large datasets efficiently.

Dex unfortunately did not work at all in our tests. A combination of a restrictive license, closed source and an unexplained bug prevented us from doing much work. However, this does imply that at least for research open source databases should be preferred. Open source allows for greater understanding of the data structures and algorithms in use and could allow a research to work around bugs, by simply fixing them.

9 Future Work

This project focused on structural queries as these were what were focused on in our reference papers. However, there is the open question of how well a graphical model would work for SPARQL type queries. A future project could create and run benchmarks designed for these types of queries.

Additionally more databases and more types of databases could be added to the benchmark suite. For instance no document type databases were tested in this project and the key-value database that was originally to be tested was dropped for reasons stated earlier. It would be interesting to see how these two types of models compare to models specifically designed for graph data.

References

[1] Renzo Angles. A Comparison of Current Graph Database Models. (Gdm), 2012.

[2] Renzo Angles, Claudio Gutierrez, and De Alberto Mendelzon. Survey of Graph Database Models. pages 1–65, 2005.

[3] Mike Buerli. The Current State of Graph Databases. 2012.

[4] CitrusByte. Redis. `http://redis.io/`.

[5] Filament Developers. Filament. `http://filament.sourceforge.net`.

[6] Inc. Docker. Docker. https://www.docker.com/.

[7] Linked Movie Database (LinkedMDB) Group. Linked movie database (linkedmdb). http://linkedmdb.org/.

[8] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. http://www.vldb.org/pvldb/vol7/p1047-han.pdf, 2014.

[9] Neo Technology Inc. Neo4j. http://neo4j.com//.

[10] Salim Jouili and Valentin Vansteenberghe. An Empirical Comparison of Graph Databases. *2013 International Conference on Social Computing*, pages 708–715, September 2013.

[11] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135, New York, New York, USA, June 2010. ACM Press.

[12] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David a. Bader. A performance evaluation of open source graph databases. *Proceedings of the first workshop on Parallel programming for analytics applications - PPAA '14*, pages 11–18, 2014.

[13] Amir Salihefendic. Redis graph. https://pypi.python.org/pypi/redis_graph/1.0.

[14] Amir Salihefendic. redis_graph: Graph database for Python. http://amix.dk/blog/post/19592#redis-graph-Graph-database-for-Python.

[15] Sparsity Technologies. Sparksee (dex). http://www.sparsity-technologies.com//.

[16] The Apache Software Foundation. Giraph. http://giraph.apache.org/.

[17] Bryan Thompson. Literature Survey of Graph Databases. (January):1–40, 2013.