University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Strategies on optimizing graph database querying services

Lifion by ADP

New York City, New York, USA

Prepared by
Arthur Chun-Yin Leung
ID 20601312
userid ac7leung
4A Computer Engineering
25 June 2020

200 University Ave W
Waterloo, Ontario, Canada
N2L 3G1

25 June 2020

Vincent Gaudet, Chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir,

This report, entitled ‘‘Strategies on optimizing graph database querying services’’ was prepared as my 4A Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 401. The purpose of this report is to compare solutions and strategies for improving the performance of reader and writer services that interface with graph database systems, and make them scalable and reliable at the same time.

Lifion by ADP provides software solutions to Human Capital Management and Human Resources. The software engineering and tech lead of the team that I worked in was led by Brett Cohen, who oversaw the testing and deployments of the web application. We also worked closely with other internal teams of Lifion, which use the platform to build features for clients.

I’d like to thank my supervisor Brett Cohen on providing guidance on the design process of the project outlined on this report. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Arthur Chun-Yin Leung
ID 20601312

Contributions

For the duration of this co-op term, I worked in Lifion's Flex Structures team, which consisted of 10 members. The team was mainly organized into the Software Engineering side and Product side, which includes a Project Manager and Solutions Architect (similar to a Product Owner or planner).

The Flex Structures team's main goal was to provide a reliable and fast graph data source to other teams utilizing the platform. Lifion's parent company is ADP, which at its core is a human resources company. Most of the data involved in these transactions pertain to employee information and organizational structures. A common problem is to retrieve an employee's record and status in an organization, as well as their managers and peers in a timely manner. As any given organization can comprise of tens or hundreds of thousands of associates, with their positions and statuses changing on a daily basis, the need for an extensible and high performance database to store this information is evident. The main web application is implemented in Javascript using Node for the microservices and React for the user interface (which will not be discussed in depth in this report). Tasks for each member of the team ranged from active development with the graph database, monitoring and performance testing, as well as operations tasks to ensure compliance with security and privacy regulations.

My primary role in the team was to investigate and implement a ''Data Contract'' design pattern to reduce coupling between data providers and consumers, with the provider being a graph database/ API and the consumer being the rest of the platform. This effort was part of a company wide effort to standardize the means in which teams serve data from the backend to the frontend UI, and was also meant to improve performance by discouraging poor API implementation or undocumented APIs.

Additionally I was to assist in efforts to identify defects such as memory or connection leaks and performance bottlenecks in the reader and writer services of the graph database, which is the main focus of this report. Some responsibilities in support and operations such as configuration of testing environments and continuous integration/ deployment systems were also delegated to me. In a few instances, I also helped other teams automate miscellaneous tasks such as generation of changelogs when there is a new version released.

Users of the Flex Structures team's services are actually developers of the platform, who build applications called ''miniapps'' for clients. In a way, it is analogous to developing for a Code Editor or IDE for other developers to use. The Lifion ''miniapps'' could then be delivered to clients who are subscribed to the platform as modular features that are packaged and tailored to the client's needs and regional regulations. For example, a module might run payroll or calculate taxes and deduction for an organization in a particular country. These projects often involve a great deal of cross team collaboration to complete a feature, as each team is responsible for a service (ex. Persistence team owns the Database Abstraction Layer, Security team owns authentication and access control to the system).

This report was written to describe the efforts involve in improving the performance of the Flex Structures graph database read service during my co-op term at Lifion, and to outline the process through which these performance defects were identified. My responsibilities were varied but this topic is relevant since modern cloud microservices utilize similar architectures, and companies that use managed services such as AWS are ubiquitous. The approaches shown should generally apply to analyzing any type of service that retrieves from a database, not just a graph one.

The effort I have invested in preparing this report has also furthered my communication and analytical skills; skills of which I believe are essential to developing professionalism, and are indispensable in all engineering workplace and academic settings.

In the broader scheme of things, the improvements as a result of implementing this solution can help encourage better practices in the future. The findings in this report could also be used as a guide to benchmarking other services.

Summary

The scope of this report is to document the improvement of performance of an existing graph database read service and write service, with considerations such as scalability and reliability as the software is subject to heavy workloads.

The major points covered in this report are the requirements and instrumentation performed on the graph database services, the importance of each criteria in this scenario, and comparison of design alternatives. A recommended solution optimized for the problem described would then be constructed from these available design options.

The major conclusions in this report are that scalability and redundancy should place highest in the list of considerations when designing any type of microservices, and that technical debt incurred could have unforeseen impacts when these services work in a distributed system. The benefits of utilizing a managed cloud computing service such as AWS almost always outweigh managing instances manually. However, the software engineer should also be aware of limitations and potential difficulties in testing and deployment, as the associated drawbacks and costs may not be always justified.

The major recommendations in this report are that development teams should consider alternate mechanisms to guarantee consistency when working with distributed databases. Where message queues are employed, retry policy of message queues should be designed carefully such that starvation does not occur. Real time logging and monitoring tools are indispensable in helping identify bottlenecks in the system architecture, open source tools such as Kibana are suitable for these purposes. It is also highly recommended that solutions for intelligent load balancing be explored in addition to the autoscaling feature for Kubernetes deployments, to prevent potential spikes in processing time during heavy workloads.

Table of Contents

List of Figures

List of Tables

1 Introduction

This section details the background information and motivation leading up to the optimization effort. In addition, the expected scope of the project will also be outlined, as a preface to the nature of the problem.

1.1 Background

Data modelling is the foundation of database design that is highly specific to the requirements and types of information being stored and retrieved from a database. The most widely used relational database system would be MySQL, where information is stored in tables; each entry is a row and the columns are defined by a schema. In recent years, databases have rapidly evolved alongside cloud computing to accommodate more business use cases, scalability, and performance requirements. These databases fall under the NoSQL category, which stands for ‘‘Not Only SQL’’. As the name implies, this encompasses databases other than the tabular/ relational system. Examples of these include key-value storage (Redis), document (MongoDB), and graph databases (Neo4j, AWS Neptune).

Data models should be intuitive to understand and reflect real world use cases. While there are many paradigms on how data should be modelled and represented (such as normalization in SQL databases), this report will mainly cover use cases for graph databases. The fundamental concept of a graph database is that information is stored in vertexes, with edges representing relationships between connected vertexes. A simple example might be that persons Alice and Bob are represented as vertexes with a ‘‘name’’ property; if Alice knows Bob, then a directed edge with this ‘‘knows’’ relationship can be inserted into the graph. Figure 1 shows an additional example of how a manager-employee relation might be imagined in both databases; it happens that in this case, a row/ entry in the MySQL database corresponds to a vertex in the graph.

In order to query the graph structure, one must perform a ‘‘traversal’’, which is simply a walk over the graph structure and selecting relevant properties. Likewise, to insert data, a new vertex is created with the relevant edges well. This report will focus on the Apache TinkerPop graph computing framework, an open source project that provides the Gremlin

query language. Compared to tabular relational databases, graph databases exhibit faster query speeds for searching irregularly structured data. This is possible since it does not operate by joining tables, but instead by establishing a path for the traversal: Only vertexes connected by the relevant relationships/ edges are traversed and selected, so query speed remains fairly constant, irrespective to data volume stored. Compare this to relational databases, where query times increase exponentially as more join operations are performed. The caveat for using a graph database would be that write (create, update, and delete) operations take longer due to having to update the relevant relationships and edges whenever a new vertex is inserted.

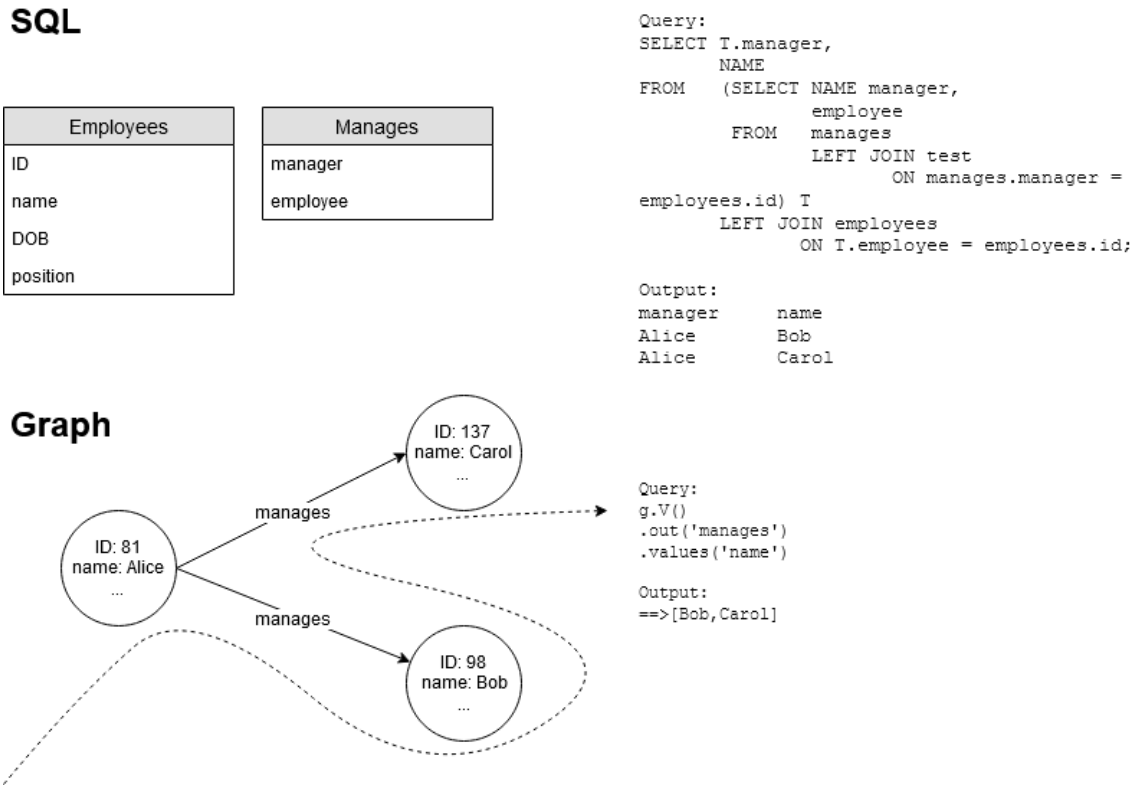

Figure 1. SQL vs Graph (gremlin) database query

The use case for Flex Structures services are mostly read workloads, where a business structure, usually consisting of employees (Organizational Chart) is queried. The nature of this data is intrinsically suited as a use case for graph database, and is more intuitive to visualize as a graph. Examples of more complex queries could be querying all open positions within a business unit, or the hierarchy/ lineage of a particular employee. At a high level, use cases can be separated into the following two categories Figure 2:

1. Frequently accessed queries: simple/ hardcoded queries (i.e. get my manager), 85-90% of use cases, highly optimized
2. Dynamically generated queries: can be used to create complex filtering logic, but queries are generated on the fly, 10-15% of use cases, may not be the most efficient

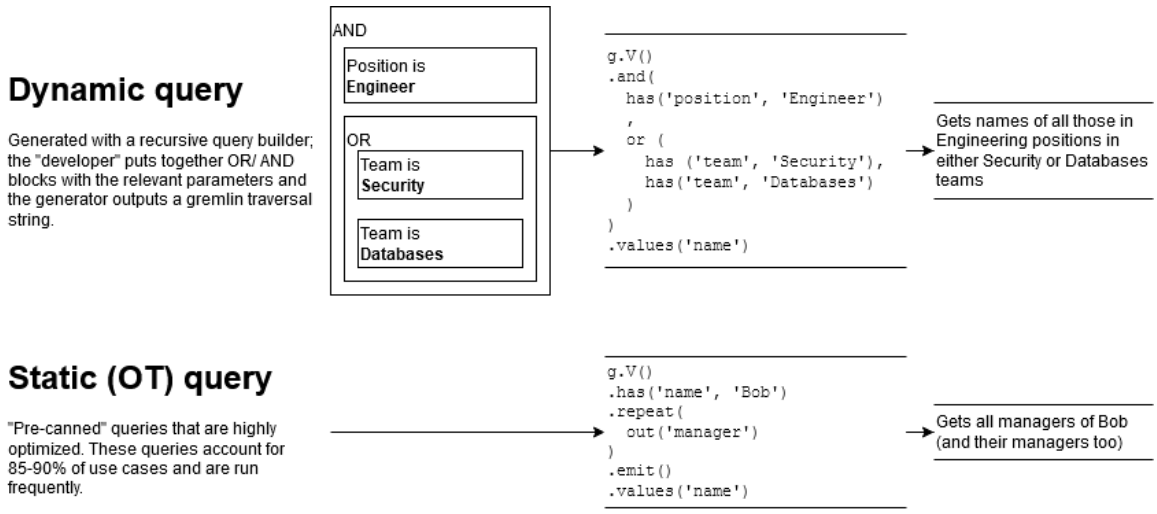

Figure 2. Static v. Dynamic queries

The Flex Structures team currently supports two graph database providers in the backend: DataStax Enterprise Graph (DSE) and Amazon Web Services (AWS) Neptune Graph. Both databases can be accessed through the Gremlin console as part of the Apache TinkerPop framework. It is planned that the team will migrate from DSE to Neptune entirely over the next year, so efforts are being taken to improve performance for interfacing with Neptune. The reason for this move was also due partly to security and overall effort in maintenance; since AWS Neptune is a fully managed database, the team has less to worry about uptime and managing clusters of the distributed graph database.

Figure 3 illustrates the services that read and write to the graph database:

On the read side there are the Flex Org Traversal (OT) and Flex Dynamic Read services, which correspond to the Static and Dynamic query use cases as discussed before, and read data from the AWS Neptune graph database.

On the write side there are the Reactor and Reconciliation services. The Reactor is responsible for monitoring messages originating from Maxwell, and writes these changes to the AWS Neptune graph database. The Reconciliation service is a secondary service that performs a

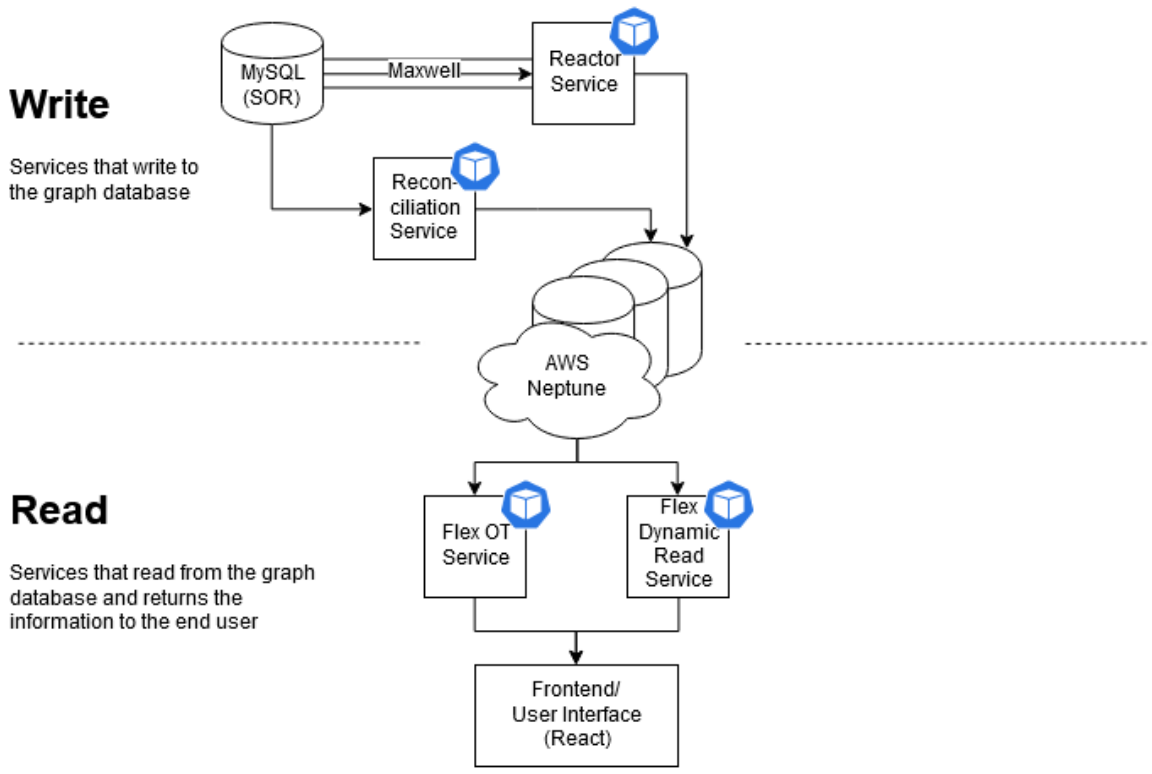

Figure 3. High level architecture of the Flex Structures graph database services

synchronization between the MySQL database and the Neptune graph database, and is run manually for support purposes. It does so by calculating the differential between the MySQL data and that of the graph. The key here is that the writer services replicates organizational data from the MySQL database into the graph database to enable fast querying as mentioned before, since reading from the graph avoids join operations[1].

1.2 Motivation

With more users adopting Flex Structures as a graph database provider, it is important to optimize frequently accessed queries. This not only includes the improvement of latency or response time, but also in resource usage such as memory and CPU. While most of the services are running in Dockerized containers, and managed by Kubernetes, occasionally spikes in memory usage were observed. In a few observed instances, these spikes even caused the read services to stop responding entirely. This had an impact on user experiences of consumers downstream, so it is an issue that must be addressed.

For the services that wrote to the graph database, performance degradation was noted during heavy loads placed on the Maxwell/ Kinesis message queue. Sometimes, operations to synchronize the graph database would time out for clients that have a large number of employees (40-50 thousand).

1.3 Scope and considerations

This report is mainly concerned with strategies and methods to improve the efficiency of the graph read (Flex OT and Dynamic read services) and graph write services (Flex Reactor and Reconciliation service). Figure 3 illustrates the services that read and write to the graph database, and only the technical implementation relevant to performance will be discussed. Since microservice architectures are ubiquitous in modern day cloud computing and use, most of these strategies are likely applicable to other microservices.

At the time of writing, the solution had to work for both DSE and Neptune graph databases. However, since support for DSE was soon to be phased out in favor for a fully managed environment in AWS, the scope of this report will cover the latter. Aside from that there are many similarities between interfacing with the DSE and Neptune graph databases, since they both support the Gremlin query language.

Almost all of the microservices in Lifion are Node.js web services run in Dockerized containers, with virtualization resources managed by Kubernetes. The amount of resources (CPU, memory) allocated to each service and specific parameters (timeouts, environment variables, feature flags) are configurable via Helm charts, but will not be discussed in detail in this report.

2 Requirements

There are two main requirements that need to be met. Firstly, raw read and write performance must be improved in terms of latency. Secondly, the service must be able to handle concurrent loads effectively as to support multiple requests, and failovers should cause minimal degradation to the end user's experience.

2.1 Summary and Criteria selection

The main goal was to improve the performance and stability of the read service under heavy load, while trying to identify any defects. An ideal solution should be reliable and scalable enough to accommodate dynamic loads. The maximum number of reads in a short period of time based on real life workloads were about 2000 requests within 5 minutes.

The requirements, or characteristics of the ideal solution are as follows:

- Low latency; service should process requests without significant delay
- Scalable; service should be able to handle dynamic workloads
- Fault tolerant; service should be reliable in the event of failures encountered during processing
- Informative monitoring; the service should detect and alert whenever heavy workloads are experienced, so that automated scaling and provisioning of responses is possible

From these requirements we chose the following criteria which will be used to select a solution:

- Ease of implementation: Is the architecture complex enough to require significant time in development? (Estimated in man hours, or equivalent metric)
- Ease of testing: How simple is integration testing for the chosen solution?
- Performance Gain: How much is the solution expected to improve latency?
- Compatibility: Would this solution be compatible with the existing system (schema, formats, data model)?
- Resource cost: Does this solution require more computational resources (CPU and memory, network bandwidth)

2.2 Available solutions

During preliminary investigation, several problems in the codebase of the services on the read and write sides were found. While these are not bugs or defects themselves, they largely contributed to poor performance and scalability under heavy loads. Lifion already

has a pipeline in place which involves the use of Jenkins and Terraform to deploy services to different environments; this leaves very little room for configuration other than tuning specific parameters (scaling resources, CPU, memory limits) as the process is largely automated. Therefore to measure fault tolerance, we simply record the number of times a service has restarted due to errors since the last deployment, using a particular solution.

The following weighing scheme shown in Table 1 was used for the qualitative analysis of each solution, based on the considerations of this project. An integer score ranging from 1 to 5 was assigned, with the highest score representing that it is the most effective solution of all the ones considered. For example, if a score of 5 is assigned to ‘‘Implementation’’ that would mean the solution is the best in terms of ease of implementation. This integer score is multiplied by the weight to obtain the normalized score. The final score for each solution was obtained by summing all of the normalized category scores.

Table 1. Criteria weighing scheme

Criteria	Impl.	Perf. Gain	Testing	Compat.	Resource
Weight	25%	30%	15%	20%	10%

2.3 Improving read performance

It was found that read performance suffered most when there were multiple concurrent requests, and was agnostic of however many vertexes were in the graph database. The following three places were identified as potential bottlenecks that could hinder the read performance under this scenario:

Caching results: The read services return the traversal/ query results as soon as they are

Data modelling: It is possible that operations on the current edge centric data model is inefficient. The alternative vertex centric model could be used instead.

Request handling: When requests are made to the read service, they are processed sequentially.

For the read services, the following optimizations are proposed. The scores for each solution are shown in Table 2:

1. **Caching results:** If a request is a recurrent traversal or utilizes a similar traversal, intermediate results can be cached.
2. **Vertex-centric modelling:** Changing from Edge-centric data model to a Vertex-centric model could simplify traversals and remove the need for string processing. Previously the edge centric model required string processing which could impact performance.
3. **Parallelized reads:** Using the Javascript/ NodeJS Promise API, it is possible to parallelize read operations as shown below:

```
//In parallel:
await Promise.all(requests.map(async request => { await executeGremlinQuery(
    request) }))

//In sequence:
for (let request of requests) {
    await executeGremlinQuery(request)
}
```

Table 2. Qualitative evaluation of read optimization solutions

	Implementation	Perf. Gain	Testing	Compatibility	Resource	**Score**
1. Caching results	4 → **1.0**	3 → **0.9**	4 → **0.6**	3 → **0.6**	3 → **0.3**	**3.4**
2. Vertex centric	1 → **0.25**	3 → **0.9**	5 → **0.45**	1 → **0.2**	5 → **0.5**	**2.3**
3. Parallelized reads	5 → **1.25**	4 → **1.2**	3 → **0.45**	3 → **0.6**	4 → **0.4**	**3.9**

For caching query results, it received a good score in Ease of Implementation and Testing. To prevent unnecessary graph database accesses, a simple cache library such as Node cache can be used to improve performance for recurring requests. For further optimization, a probabilistic data structure called a bloom filter can be used with a cache filtering algorithm[2]. The benefit of using this type of filter is that it knows an element either definitely is not in the set or may be in the set, shown in Figure 4. Performance wise it is inexpensive due to its probabilistic nature, and is often used to filter elements that do not match a query, but will still consume more space in memory. When used in tandem with a caching system, it can increase cache hit rate and theoretically speed up response times; in other words, lower latency for each request. Instrumentation can also be used to benchmark the cache hit and miss rates.

For Vertex-centric modelling over Edge-centric modelling, it received the lowest score

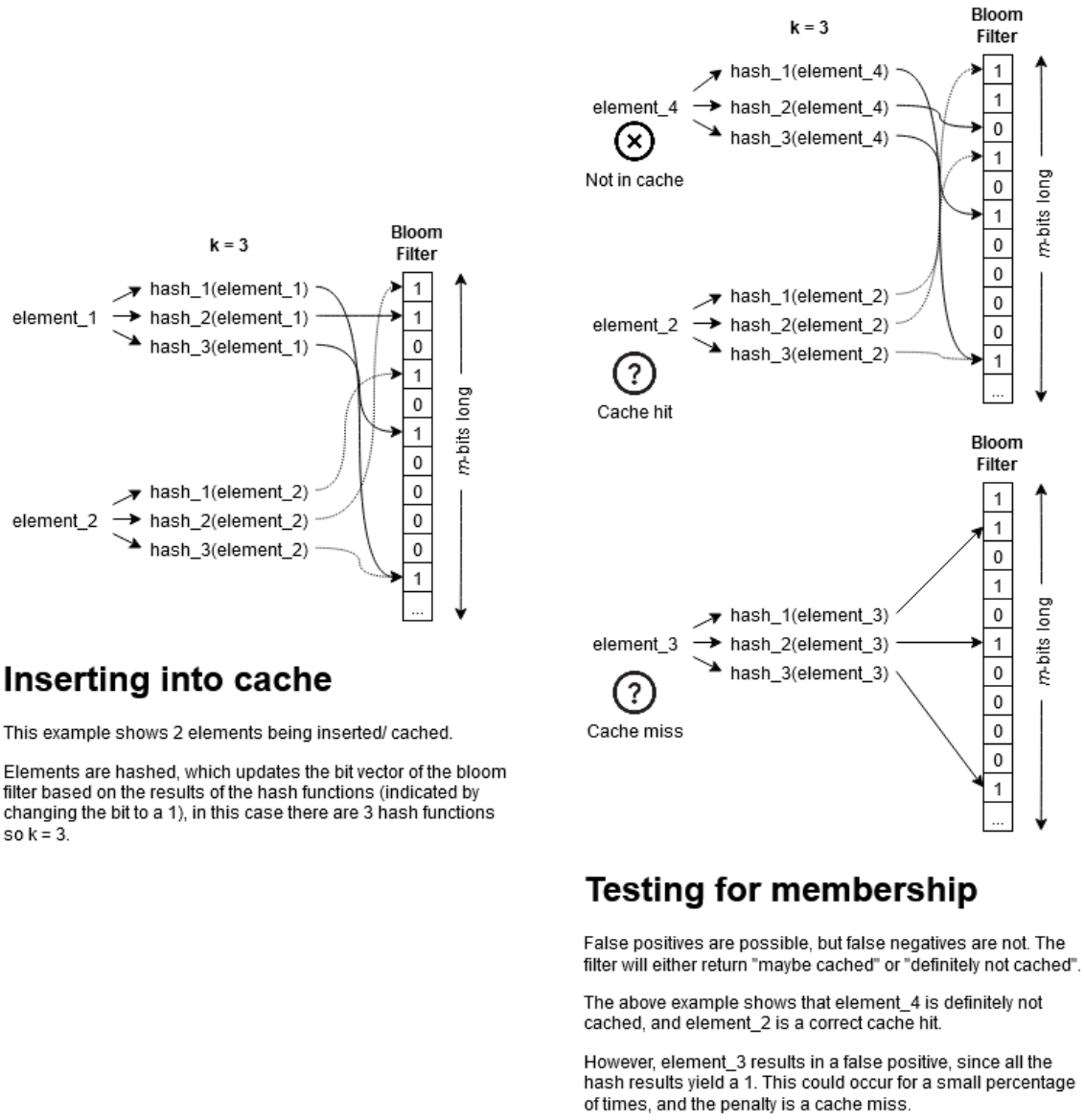

Figure 4. A bloom filter used to filter requests, used with a cache mechanism

in Ease of Implementation and Compatibility. Since the fundamental data model is being changed, logic across multiple services would need to be rewritten and rigorously tested to ensure correct behaviour. The existing edge centric model requires processing to be done on a ‘‘derived edge’’. A ‘‘derived edge’’ is but an edge label with the following naming convention: ‘‘sourceTable|targetTable|sourceField|targetField’’, and to traverse this edge one must concatenate this string and perform a string comparison to see if the edge/ relation exists on the current vertex. The key idea is that since string operations are expensive and can impact performance, this solution itself merits special consideration.

For parallelized reads, it received the best score in Ease of Implementation as it is a simple code change, but will utilize more computational power on the graph database host (by increasing read replicas). However, this is desired, since it was found that the graph database host's multi-core CPU was underutilized even during high loads. Using this solution, there should be no conflicts since the workloads here are reads, so testing would also take minimal effort.

2.4 Improving write performance

The primary workload is to replicate data from the SQL database, also known as the ‘‘Source of Truth’’, into the graph database. These change records are sent to the Kinesis stream by Maxwell, which is a daemon that monitors the MySQL database for changes.

Similarly, the following places were identified as potential bottlenecks that could hinder the write performance:

Data Abstraction Layer: The database connections are a shared resource (pool) managed by the MySQL2 Javascript framework[3]. Previous attempts to increase the pool size did not result in significant performance increase.

Differential calculations: Before writing to the graph database, there is a step to calculate data parity (difference between the MySQL and graph data) and only sending the necessary changes. It was a mechanism intended to save bandwidth, at the expense of some processing power and time.

Kinesis stream: The record stream does not implement any retry policy itself. If a message from Kinesis fails to get processed by the Reactor service, it remains in the message queue. Currently it is possible for starvation[4] to occur if the message is not removed from the queue.

For the write services, the following optimizations are proposed. The scores for each solution are shown on Table 3:

1. **Caching connections:**

Table 3. Qualitative evaluation of write optimization solutions

	Implementation	Perf. Gain	Testing	Compatibility	Resource	**Score**
1. Caching connections	3 → **0.75**	4 → **1.2**	4 → **0.6**	5 → **1.0**	3 → **0.3**	**3.85**
2. Bulk loading	4 → **1.0**	5 → **1.5**	3 → **0.45**	4 → **0.8**	4 → **0.4**	**4.15**
3. Retry policy	4 → **1.0**	3 → **0.9**	2 → **0.3**	4 → **0.8**	5 → **0.5**	**3.5**

The database connections are managed by the MySQL2 Node library, and it was found that the pool object is cached in memory. Previous implementations of the read service used Node cache to accomplish this, but it could be susceptible to a memory leak.

2. **Bulk loading:**
   AWS Neptune provides a mechanism to import bulk data in the CSV format. Instead of manually calculating the differential between MySQL and Neptune graph manually and writing it (in the case of the Reconciliation service), it is possible to dump the MySQL data into a CSV file and use this method.
3. **Retry policy:**
   If a write query fails, it should retry up to 3 times, then remove the message from the Kinesis stream (then place it in a failed queue).

For caching database connections, the existing implementation stores the database pool in memory and deletes it if no new requests arrive after 10 minutes. This design decision was made to free up resources and memory when the service is not in use, but the time spent initializing a new connection pool (cold start) could contribute to poor latency. It is simple to reverse this change to yield better latency during during sparse workloads but will consume more resources.

For bulk loading, it is an option made available as a feature by the AWS Neptune graph database. This solution requires setting up an AWS S3 storage node to store the CSV files, which can be accomplished trivially, but requires more network bandwidth.

For implementing a retry policy, this was the simplest solution to provide recoverability, but at best is a stop gap measure. If the application logic was flawed and a write query were to fail because of it, it only ensures that subsequent queries can be written to the graph database (after retrying 3 times). Due to this fact, complexity in testing increases, and additional logging or monitoring should be put in place to ensure errors are caught and are not left unreported.

2.5 Possible solutions

After performing the qualitative analysis of the optimization strategies, the following viable solutions for the read and write services were considered:

1. **Caching read results while bulk loading periodically for writes:**
   Implementing a cache layer and bloom filter would certainly increase application complexity, but yield better performance. However, going with this solution may also warrant additional effort in experimenting with different cache eviction algorithms, in order to yield the best performance under real-world work loads. The bulk loading ensures minimal overhead incurred due to network delays. This solution makes the most sense when network infrastructure is the bottleneck, or when the data center is located very far from the user, since both strategies' goal here is to minimize the time spent accessing the database.
2. **Parallelizing reads while implementing a retry policy for writes:**
   Utilizing the Javascript Promise API to send requests in parallel makes full use of the database host's processing capability. When it comes to writing, the retry policy prevents starvation of the service and so tries to minimize disruption to the quality of service, while providing a means for recovery (by retrying the failed write query).
3. **Parallelizing reads while bulk loading periodically for writes:**
   This alternative is the result of striking a compromise between complexity of implementing additional logic and minimizing overhead for keeping the graph database consistent. Although there is not a recovery mechanism in place like the previous solution, if the bulk loading takes a relatively short amount of time, it can be run frequently instead to supersede the need of a recovery mechanism.

Out of these, the best scoring option was to parallelize reads while Bulk loading periodically for writes, with a combined score of 8.05:

3 Analysis of proposed solution

The simplicity of implementation and testing means that this solution could be realized in a short time frame with relatively less effort, and it is expected to yield a noticeable performance gain. It is also the least risky solution to implement, in that the underlying data model remains the same, and most of the application logic changes are isolated to that of the read services.

With the parallel read query processing, it is expected that concurrent requests can scale with the number of CPU threads available on the graph database host. This should be reflected with increased CPU utilization while latency is kept low at the same time.

On the write queries, sending them in bulk to the graph eliminates the network round trip delay incurred by processing each message one-by-one instead. A more advanced implementation may parallelize this process as well, but logic to check for read-write and write-write conflicts and linearization anomalies is required when using this strategy for distributed databases. One might accomplish this through the use of a distributed checkpointing system, but the technical implementation is out of scope for this solution.

Altogether, this solution is estimated to take 1.5 sprints (3 weeks) to fully implement and test by a single software engineer. It is ideal since the Flex team is small and cannot afford to spend extended periods of time rewriting large amounts of logic (at least without external aid or a contractor), which is required in changing the base data model. At the same time, the implementation should be simple for most developers to understand and be able to debug, when compared with something like the bloom filter. There were also concerns of security and for the bulk loading API, but as AWS is a fully managed cloud host, it also provides Identity Access Management (IAM) for enforcing security policies and access rules, and can be integrated with this solution.

4 Performance

For the read services, a simulated load was applied for 5 minutes using Autocannon[5], an automated testing tool that is designed to stress test HTTP services. Peak values of response

latency and CPU usage of the graph database host were recorded after each trial, and at each level of intensity. The results showed that by parallelizing read queries, it is possible to achieve full CPU utilization of the graph database host, while capping the latency below 100 milliseconds Table 4. Although it was benchmarked against DSE in this case, the same principle should also apply for AWS Neptune.

Benchmarking of the write services was done by performing a full synchronization between the MySQL ‘‘Source of Truth’’ database and the AWS Neptune graph database. The client sizes (number of vertexes in the graph) varied from 5,000 to 40,000, and the runtimes of each trial is recorded in Table 5. It is clear that the method of bulk loading improved times by an order of magnitude. A noteworthy observation here is that the lower bound for times seems to be around 2 minutes; upon further investigation it was found that most of the time was spent copying the CSV file to the AWS S3 storage node. In other words, it is limited by network speed. Nonetheless, this optimization proved to be a significant improvement over the previous implementation.

Table 4. Read performance benchmarks peak values

Requests per minute	Sequential		Parallelized	
	Latency (ms)	CPU (%)	Latency (ms)	CPU (%)
100	54	7	52	8
300	110	10	64	30
500	140	12	60	42
1k	205	16	67	81
3k	1530	25	81	99

Table 5. Performance for a full synchronization compared to CSV bulk loading

Client size	Normal synchronization	CSV Bulk loading
5k	10 minutes	2 minutes
7k	18 minutes	2 minutes
15k	32 minutes	3 minutes
40k	DNF	5 minutes

The implementation of this solution successfully fulfilled all the requirements described in section 2.1 with the following characteristics:

Low latency: Parallelized read requests helped reduce latency under heavy loads, the results are shown in Table 4. For write workloads, the bulk CSV import option is far superior in terms of runtime.

Scalable: For read workloads, CPU utilization was much better compared to sequential processing, and latency did not increase significantly even at the highest simulated workloads.

Fault tolerance: AWS Neptune provides the option of creating read-only replicas to offload requests from the primary replica. In the event that the primary is unreachable, the failover is handled automatically, which is another benefit of using a fully managed (graph) database.

Self recovering: The bulk loading will overwrite any outdated or otherwise corrupted data in the graph database with the latest copy. As far as consistency goes, it only needs to be consistent with the MySQL ‘‘Source of Truth’’ database.

5 Conclusions

Based on the findings from the previous analysis and benchmarks, it was determined that the solution utilizing parallelized reads while bulk writes would be the most suitable for performance improvements. Although a less sophisticated solution, it required the least amount of effort by the team members, and performed well for this purpose. The bulk write strategy was also successful in effectively reducing time taken to populate the graph database by an order of magnitude. In conclusion, this solution served well for the requirements that were initially set forth to optimize read and write performance.

6 Recommendations

Based on the analysis and conclusions in this report, it is recommended that additional testing be conducted on both read and write services, under a variety of scenarios such as unstable network conditions. Additional monitoring tools should also be explored to help catch degradation in quality of service before it reaches the end user. In addition, the following tasks are worth considering as next steps, if more time and effort can be afforded:

- Collect sufficient data to form a predictive load balancing model
- Streamline the process in which bulk data is loaded into AWS Neptune, or create a user friendly interface for other teams to use (as a self-serve tool)
- Implement the bloom filter and cache solution to further improve query latency
- Investigate the long term scalability of the Vertex-centric data model, and whether it is worth the effort to rewrite application logic for the conversion

Glossary

API: Application Programming Interface, a defined set of subroutines of a software application that allows external components or other applications to interact with the application.

AWS: Amazon Web Services, a fully managed cloud computing and hosting platform provided by Amazon.com, Inc.

CSV: Comma Separated Values, a data format in which fields values are delimited by commas. Also formalized as the RFC 4180 standard.

DSE: DataStax Enterprise Graph, a graph database provider capable of providing scalability and analytics for massive datasets.

WKRPT: Work-term report; the acronym used by the University of Waterloo Undergraduate Calendar.

References

[1] Rik Van Bruggen. *Demining the "Join Bomb" with graph queries*. 2013. URL: http://blog.bruggen.com/2013/01/demining-join-bomb-with-graph-queries.html.

[2] Matei Ripeanu and Adriana Iamnitchi. ''Bloom Filters Short Tutorial''. In: (Sept. 2001).

[3] Andrey Sidorov. *MySQL2*. URL: https://github.com/sidorares/node-mysql2#readme.

[4] Sandro Fiore. *Grid and cloud database management*. Springer, 2011, p. 183.

[5] Matteo Collina. *Autocannon*. URL: https://github.com/mcollina/autocannon.