Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Database Group

Efficient Graph Traversals Using AWS S3

Master Thesis

Aditya Chandla

Supervisors:
dr. Nikolay Yakovets

Eindhoven, July 2024

Abstract

As the size of graphs being queried by graph databases increases, coupling storage and compute increases the cost and limits the flexibility of traditional graph database management systems (GDBMS). As part of this thesis, we explore the viability of an architecture where the compute and storage are independent. This independence is achieved using a distributed cloud storage service (AWS S3) which provides bottomless storage and theoretically unlimited throughput. Using this distributed cloud storage solution, we evaluate the latency of running two common graph traversal algorithms i.e breadth first search (BFS) and depth first search (DFS). We then compare this latency with some other systems which may be used to perform BFS and DFS on large graphs.

Preface

Please write all your preface text here. If you do so, don't forget to thank your supervisor, other committee members, your family, colleagues etc. etc.

Contents

Chapter 1

Introduction

The size of graphs being used in both academia and industry is increasing at a rapid rate, and as a result, it has become common for graphs to have tens of billions of edges[22]. Furthermore, this trend is expected to continue as the amount of data collected and processed increases. A recent survey shows that scalability is the primary concern of graph database users[22]. As this scale increases, decoupling storage from graph query evaluations using distributed cloud storage services like AWS S3[4] may be more cost-efficient. These systems provide durable data storage, high throughput, and a pay-as-you-go model where you only pay for the data you store and the number of requests you make. Making use of these characteristics would enable graph databases to provide more flexible scaling and alleviate the costs related to redundancy. Therefore, it is worthwhile to explore the feasibility of such database architectures.

The idea of having separate storage and compute has been studied in the context of relational databases. In 2008, Brantner et al[7] described how a relational database could be built on top of AWS S3 while still providing atomicity, isolation, and durability. Snowflake[24] was one of the first databases which fully realized the idea of having separate storage and compute layers. This idea was adopted by other databases like AWS Aurora[27] in 2017 and more recently by Neon[20]. Despite the existence of so many relational databases which have separate storage and compute layers, this idea remains unexplored for native graph databases.

Although there are many native graph databases that enable scaling out using sharding[5], the compute and storage are still tied together. Unlike such architectures, this paper explores an architecture where storage of raw graphs is delegated to a distributed cloud store and, memory of compute instances is only used to process queries and store intermediate results.

There are two main challenges when considering databases with separate compute and storage: Distribution transaction management and access latency. The first challenge, although more intricate, has been studied in the context of relational databases[7] and can be extended for other systems that follow this general architecture. Therefore, in this paper, we explore how to reduce the access latency for graph-specific operations in the context of graphs stored on distributed file systems.

To study the impact of our techniques on access latency, we use Breadth-first search (BFS) and Depth-first search (DFS) as two algorithms to benchmark our performance. These two traversal algorithms are the most commonly used algorithms for performing graph traversals[22]. These traversal algorithms will have a bounded depth and thus only access a small part of the graph. On the spectrum of graph algorithms from OLTP to OLAP as described by Besta et al.[5], these traversal algorithms lie more on the OLTP side as the portion of the graph explored by a single traversal would be much smaller than the size of the graph.

To minimize the latency, we will first propose a graph storage format that would provide a gain granular access over the graph stored in AWS S3. Then, we would extend some of the existing

caching techniques to lower the latency of graph access. Finally, we will describe the cases where it is viable and more cost-efficient to use an architecture that separates storage from compute.

1.1 Research Objectives

In this paper, we provide an initial analysis of how distributed storage services can be used to query large graphs. While using these distributed storage services, the primary issue that needs to be addressed is latency. The latency of accessing data in a networked distributed system involves communication over the network which is at least three orders of magnitude more than accessing data from local storage. In return for this increased latency, we get virtually unlimited throughput as read operations are distributed across a cluster of thousands of physical machines if not more. Therefore, in this paper, we evaluate ways to reduce the latency of accessing graphs. We will focus on the performance of two commonly used graph traversal algorithms: Breadth-first search (BFS) and Depth-first search (DFS). More formally, the first research objective is as follows:

> **RO 1:** Gauge the effectiveness of caching and prefetching techniques on the latency of graph access for graph traversals.

Apart from reducing the latency, we will also discuss how the cost model of distributed storage engines differs from the traditional model of coupled compute and storage. For storage engines like S3, we are charged for every gigabyte of storage and we are charged for every request that we make on that storage. On the other hand, in the case of an SSD/HDD storage unit, you pay for a certain amount of storage and there is no additional cost for accessing the storage. Therefore, we aim to provide a model to help users choose between one form of storage over the other. More formally, the second research objective is as follows:

> **RO 2:** Provide a cost model to help users decide whether using S3 instead of SSD/HDDs might be more cost effective.

1.2 Research contributions

In line with the research objectives mentioned in the previous section, the unique contributions of this research are as follows:

1. We provide the details of how we can perform traversals on large graphs using a distribution cloud storage service. The architecture for separating compute and storage was already proposed by Brantner et al.[7] in 2008, however, there has been no research on the viability of this architecture for graph databases. We answer this question in Chapter 3 and discuss its performance in Chapter 4.

2. We provide a cost model to enable users, and developers of graph databases to check if using S3 or similar tools would fit their use case. The authors are not aware of any research which answers this question. We answer this question in Section 5.1.

1.3 Structure of the thesis

Chapter 2, provides the necessary background for this thesis. Section 2.1 talks about the history, architecture, and characteristics of distributed storage systems. We also give a brief overview of the capabilities of AWS S3, which is the service we use in this thesis. Then Section 2.2 discusses the advantages of database architectures which separate compute and storage. Finally, Section 2.3 discusses the caching and prefetching strategies that were employed in this thesis to lower the latency of graph traversals.

Chapter 3, introduces our architecture for performing traversals. We start by providing a description of each component and its responsibilities in Section 3.1. Then Section 3.2 provides a baseline implementation which will be useful for comparing the impact of the improvements made in the subsequent sections. Finally, Sections 3.3 and 3.4 describe the details of our proposed architecture which enables low-latency traversals for large graphs.

In Chapter 4, we present benchmarks showcasing the effectiveness of our implementation and the underlying system architecture. We begin this chapter by comparing the performance with the baseline solution in Section 4.1. Then in Section 4.2, we compare the performance of our solution with Neo4j and Apache Flink. This section highlights various characteristics of different types of tools(GDBMS, RDBMS, Big Data tools, and Custom Solutions) and the areas in which they are suitable.

Chapter 5 elucidates the reasoning for various choices made throughout the thesis. Section 5.3 discusses other possible architectures for performing traversals on large graphs using S3 and their advantages and disadvantages over the proposed architecture. Then, Section 5.1 discusses the use cases where this architecture can be more cost-efficient and flexible compared to other tools and where users would be better off avoiding this architecture. Finally, Section 5.2 considers the threats to the credibility of this work.

Finally, Chapter 6 concludes the thesis and suggests possible directions for future work. This section contains information about how we may be able to extend this work to reach a point where we have a fully functioning graph database whose storage resides in S3.

Chapter 2

Preliminaries

The purpose of this thesis is to reduce the latency of graph access in cases where the graph is stored on a distributed file system like AWS S3. In order to put this work into context, we start by providing an overview of distributed storage services and more details about the particular storage service which will be used in this thesis. Then, we provide background on the idea of separating compute and storage, the evolution of this idea, and its applicability. Finally, we describe the current literature on how others have approached the idea of reducing access latency by using caching algorithms.

2.1 Distributed storage services

The idea of accessing files via the network can be traced back to the early days of the internet. However, the first widely used implementation of a networked file-system was developed by SUN Microsystems[18]. Their implementation is widely referred to as the networked file-system(NFS). Their implementation provided users with the ability to mount a file-system present on a remote machine. However, this system did not provide any distribution transparency as the users had to be aware of where and how each file is stored on remote machines.

The first widely used implementation of a distributed file-system is considered to be the Hadoop Distributed File-system (HDFS)[23]. This implementation provided distribution transparency and consistency guarantees on various operations on the files stored in HDFS. This implementation was inspired by the Google File-system[13] which was developed by Google in 2003 almost seven years before HDFS.

Although file-systems like HDFS were widely used, file-systems like AWS S3[4] provided an additional feature of being multi-tenant. In other words, with file-systems like AWS S3, a user could reap the benefits of a distributed file-system without having to manage or pay for an entire cluster. The cost of managing, scaling, and maintaining the distributed cluster was delegated to cloud providers like AWS. Now, users could pay for the amount of data that they store, and the number of requests that they make to their data. This has enabled the usage of such distributed file-systems in applications like video streaming, database storage[24], web content delivery, and backup storage.

The exact architecture of AWS S3 or any similar service provided by competing cloud providers is not known, however, we can glean some information about S3 based on how other distributed file-systems were designed. We will explain some basic ideas related to AWS S3 using Figure-2.1. Let us assume we need to store four files in AWS S3. In order to do that, we first create a storage unit called a 'bucket' which is similar to a directory in a file-system. After uploading our files to this bucket, the files get replicated over the entire AWS S3 cluster in that particular region. Figure-2.1 assumes that this cluster has a replication factor of two and therefore every file is stored on two physical servers. Now, if users send requests to access these files, their requests can be

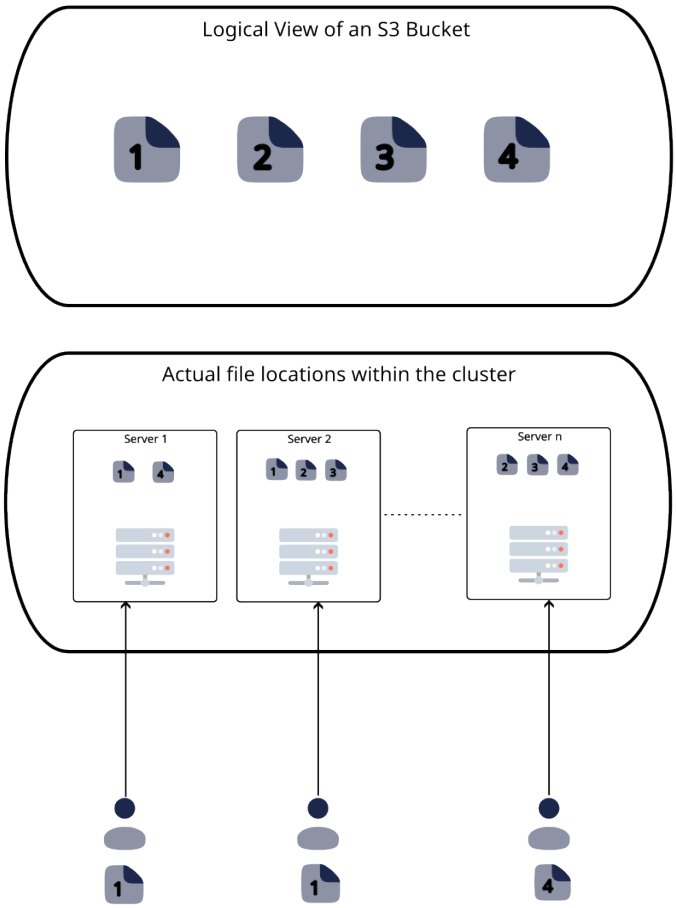


Figure 2.1: Distributed File Store

redirected to any one of the servers containing the desired file. For example, if the first and second user both want to access file number 1, their requests can be served by two different servers since the same copy will be saved on two different servers. For simplicity, the figure shows that users make direct requests to the physical machines containing these files, however, in reality they all send requests to a single endpoint and their requests are then redirected to a server based on some load balancing scheme. In later chapters, we will use this abstract model of a distributed file-system to argue about the applicability of various techniques.

Apart from the basic understanding of AWS S3 as it related to a distributed file-system, we also want to draw the reader's attention to the following characteristics of AWS S3 as mentioned in its documentation:

1. **Throughput SLA**: At the time of writing this paper, AWS guarantees that S3 can handle at least 5500 requests per object stored in a bucket. Note that there is no restriction on the total number of objects that can be stored in a bucket.

2. **Storage Classes**: S3 offers various storage classes that have different cost and performance characteristics. These storage classes determine how much you pay for storage and for making requests. In this thesis, we make use of the 'S3 Standard' and 'Express One Zone' storage classes.

3. **Costs**: The cost of using AWS S3 depends on two factors: total storage and number of

requests. The storage cost for 'S3 Standard' storage class is $0.023/GB and the cost of GET requests is $0.0004/1k requests. However, the storage cost for the lower latency 'Express One Zone' storage type is higher with $0.16/GB but the cost of making requests is lower at $0.0002/1k requests.

4. **Artificial sub-directories**: Although AWS CLI and other tools provide the capability to upload and navigate a bucket like a directory, but the underlying architecture only supports having files inside a bucket. Therefore, if you have a folder 'f1' with two files 'one' and 'two', unlike a file-system, the bucket would only contain two files namely: 'f1/one' and 'f1/two'. The name of the folder is simply prepended to the file names.

5. **Hashing object names**: In order to decide which partitions an object is assigned to, S3 internally hashes the first few characters of the filename. Therefore, if we have object names whose first few characters are the same, it is possible they will all be assigned to a single partition which may lead to lower performance. Although, it is mentioned that S3 automatically splits partitions in case of overload, there is some overhead and time delay in performing the split.

6. **Consistency**: Despite being a distributed system, AWS S3 provides a strong 'Read-After-Write' consistency. This means that any updates to an object are immediately visible to all clients.

2.2 Separating compute and storage

One of the first proposed implementation of a database with separate compute and storage can be traced back to 2008[7], where Brantner et al. proposed an architecture of a relational database which used AWS S3 as its storage layer. The authors of this paper primarily focused on how to handle updates under various consistency models when the data is stored in S3. This idea was further developed over the years by AWS and culminated with the release of AWS Aurora[27]. Although Aurora separated compute and storage, the write throughput was still limited due to the fact that all write traffic was directed to a single instance. The first database to support multiple read and write instances on top of distributed data is considered to be Snowflake[8]. Their architecture was built upon the storage layer of AWS S3 on top of which users could create as many 'virtual warehouses' as they needed. Since then, various databases with independent storage and compute planes like Neon[20] and AWS Aurora Serverless[2] have been released.

The term 'Serverless databases' is often used for databases with separate compute and storage, however, these two concepts are different. A 'serverless database' has a pay-as-you-go model where you pay for the amount of storage that your databases uses and the compute resources that were used in a fixed amount of time. In this model, the users simply have to define the minimum and maximum compute resources that the database can use and the databases scales according to the load. Note that the definition of serverless database does not necessitate separation between compute and storage and there are indeed databases like CockroachDB[25] which offer a serverless model although the underlying architecture couples storage and compute together. Therefore, although separating compute and storage may be amenable to a serverless design, it is not a requirement for it. The term serverless, therefore, defines a type of payment model whereas having separate storage and compute for a database is more of an system architecture choice.

A database which separates compute and storage offers better scalability, elasticity, and modularity. When the compute and storage are separated, both of them can be scaled independently. A heavily used service which has relatively low amount of data can scale the database's compute without wasting any storage. On the other hand, a service which has a lot of data but has a low query workload can scale the database's storage without wasting any compute. Furthermore, this separation also provides more elasticity so that the database can seamlessly scale up when the workload increases and can scale down during low traffic hours. Finally, separating storage and

compute can lead to more modularity since the storage layer and the compute layer can be upgraded and changed based on the system's requirements. For example, if a cloud provider launches a new instance type which is better suited for your workload, you can switch to the newer instance type without being concerned about data migration. These aforementioned characteristics make databases with separate compute and storage attractive to a variety of use-cases.

Separating compute and storage in databases often comes at the cost of increased latency. This latency is a result of the network communication between the storage layer and compute layer that needs to take place to complete any database operation. Although the throughput of networks has massively increased in the past few years (AWS offers instances with up to 200 Gbps bandwidth), the latency of network communication is limited by the speed of light. Therefore, a database with a local SSD containing all the base data will always be faster than a database whose storage needs to be accessed over a network. The actual difference between performance depends on the physical distance between storage and compute nodes. If the entire deployment is in a single data-center, the latency is around 500μs which is still around 10 times higher than access latency for an SSD[14]. This advantage becomes less pronounced in distributed databases with coupled storage and compute since they may also require some communication between instances in order to respond to user queries. Although there are various caching and prefetching mechanisms to reduce latency for databases with separate storage and compute, there still remain cases where coupled storage and compute yields lower latencies.

2.3 Caching and Prefetching

The concept of caching and prefetching in case of graphs enable us to take advantage of spatial and temporal locality to reduce the latency of future graph accesses. The terms spatial and temporal locality come from literature on processor caches. In the context of a processor cache, temporal locality refers to the tendency of a memory location that is accessed now to be accessed again in the near future. Similarly, spatial locality refers to the tendency of a memory location close to a recently accessed memory location to be accessed in the near future. These concepts can be translated to graphs as follows:

1. **Spatial Locality** in case of graphs means that if a node is accessed, then it is likely that this node's neighbours will be accessed in the near future.

2. **Temporal Locality** in case of graphs refers to the existence of nodes that are central according to some centrality metric. There are various centrality measures[15] which may help us understand which nodes are central for a particular graph algorithm. The main idea is that there exist some nodes which have higher probability of being accessed while running a particular algorithm on a graph.

In this sections, we will discuss some background research on caching and prefetching which would help us exploit spatial and temporal locality to lower the data access latency for graph traversals.

Any caching scheme consists of two fundamental operations: admission of data to the cache and eviction of data from the cache. There are various cache eviction policies like Least Recently Used (LRU), Least Frequently Used (LFU) which define ways to find out what data to evict from the cache when it becomes full. There also exist eviction policies which rely on metadata related to the data in order to decide which data elements to evict. Such metadata might include size of the data element, time of last data access, and access latency of the element in case of a cache miss. Apart from eviction, a cache also needs an admission policy which, in case of most caches, is simply to add a piece of data in case of a cache miss. However, there do exist more sophisticated admission algorithms like TinyLFU[10] which make the admission decision based on metadata related to a data object, which in case of TinuLFU is the access frequency of a data item. Storing metadata related to an object often results in increased memory and maintenance overhead. These admission and eviction policies determine the effectiveness of a cache algorithm for a given use-case.

We will first consider schemes which do not require additional information about the data being stored like size or access latency in case of a cache miss. In this case, LRU and LFU are two of the most commonly used algorithms. It has been shown that if the access pattern follows Zipf's law[28], then LFU outperforms LRU. However, if the access pattern has high temporal locality, then LRU can outperform LFU. Therefore, the choice between LRU and LFU depends on the underlying access pattern of the data. This choice, however, does not have to be binary as we can also have an LRFU cache[17] which provides us with a way to have an eviction policy which takes both recency and frequency into account for eviction. This algorithm subsumes both LRU and LFU because its functioning depends on a factor λ which dictates the weight that is given to recency versus frequency while evicting an item. As a result, we can tune this parameter to have the cache behave like LRU or LFU. Thus, with LRFU cache, we can fine-tune our eviction policy in case we do not have any other additional information about the items.

In the last paragraph we talked about caching policies when we do not have any additional information about the items being stored, however, while processing graphs, we do have some information about the graph topology when we access a node's neighbours. This information can be useful if there is a high likelihood that the neighbours of this node will be accessed. This idea of graph aware caching algorithms was introduced by Aksu et al. in a paper where they introduced a graph aware caching policy named Clock-based graph aware caching (CBGA)[1]. Although the algorithm contain many important contributions, the main idea of this algorithm is to expect that a node's neighbours will be accessed if a node is accessed. This idea helped their algorithm outperform all other competing cache algorithms which did not exploit the structure of the graph. Their idea was further extended by Bok et al.[6] where they added a used-cache in addition to a prefetcher cache for speeding up sub-graph lookups. This idea helped them outperform CBGA for the task of sub-graph matching where they were able to achieve a hit ratio of between 50% and 65%. These contributions highlight the potential for graph aware caching for reducing the latency of graph access.

Chapter 3

System Architecture

3.1 Component Overview

Figure 3.1 contains the high-level overview of the system that we will use for query evaluation. We will now give a brief explanation of the components present in this figure:

1. **LDBC**: The Linked Data Benchmark Council(LDBC) publishes datasets for various types of graph processing workloads. In this thesis, we make use of the Social Network Benchmark (SNB) datasets in order to evaluate the performance of our traversals.

2. **LDBC Converter**: The LDBC dataset are in CSV format and contain additional information about nodes and edges. As we will explain in Section 3.3.1, we need the data to be in a particular binary format and we do not have any use for node and edge properties. This component of the system removes the unnecessary information from the LDBC datasets and maps the data into the desired binary format.

3. **AWS S3**: Once the LDBC converter converts the data into a binary format, this data is added to an AWS S3 bucket. A 'bucket' in AWS S3 is a container for files.

4. **Graph Access Service**: This service provides an interface for accessing a graph stored in AWS S3. This interface provides its users the ability to get a node's neighbours by providing the source node, edge label, and edge direction. This service contains all the caching mechanisms that we will use to reduce the latency of traversals. We will discuss this service in more detail in Section 3.3.

5. **Graph Algorithm Service**: This service is responsible for using the interface provided by the graph access service to perform traversals and measure their performance. We discuss this component in more detail in Section 3.4.

Looking at the architecture in Figure 3.1, it is natural to wonder why we have two separate services for accessing the graph and performing traversals on that graph. There are two reasons for this separation: first, it separates the logic for accessing graphs from the logic for performing traversals with minimal overhead; second, this type of architecture is particularly amenable to creation of a multi-tenant storage layer, as used in the Neon database [20]. This is why we have chosen to separate the service responsible for accessing the graph and caching from the service responsible for running traversal algorithms.

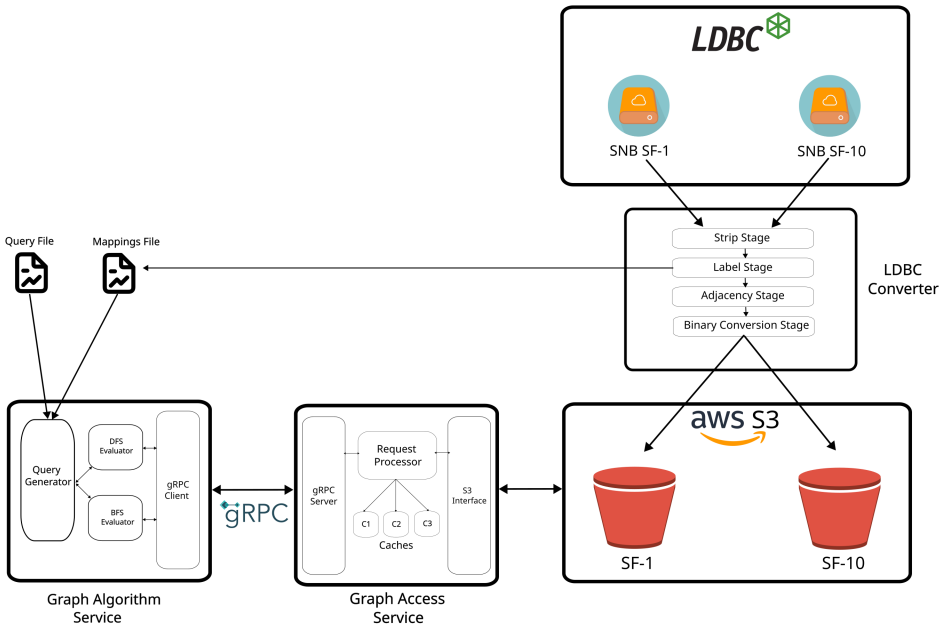


Figure 3.1: High Level System Architecture

3.2 Baseline Implementation

Since there are no existing tools that we can use to compare our solution with, we present a baseline implementation to gauge the effectiveness of our caching techniques and algorithmic improvements that we will present in the following sections. The main idea of the baseline implementation is to provide the most straightforward way to perform traversals on a graph that is stored in AWS S3.

As per the workflow we presented in Section 3.1, we first need to convert the graph from the LDBC dataset into a format that enables us to fetch parts of the graph that we need for processing. In order to do this, we convert the graph into an adjacency list, then we partition that adjacency list based on size, and finally convert it to compressed sparse row (CSR) format [9]. With this partitioned CSR format stored in AWS S3, our aim is to load the desired file whenever we want to fetch a node's neighbours.

Figure 3.2 shows the design of graph access service and graph algorithm service for this baseline implementation. The graph algorithm service, is responsible for performing the traversals and recording the results. This service uses the textbook implementation of DFS and BFS, with the only caveat that a node's neighbours are requested from the graph access service. The graph access service contains an index which maps node ranges to AWS S3's file names. With this index, we can get the file name for the AWS S3 file which contains a particular node's neighbours. Additionally, this service also contains an LRU cache of some files that were fetched from AWS S3. The size of this cache is limited because each cache entry contains an entire file with adjacency information for hundreds of nodes. With these components in place, whenever the graph access service gets a request, it first checks if that request can be served from the cache; if not, we fetch the file corresponding to the requested node and add that file to the cache.

The approach described in this section is quite straightforward but there are various improvements that can be done to improve the performance and to better leverage the features provided by AWS S3. In the next few sections, we will discuss how we can improve the implementation for both the graph access service as well as the graph algorithm service.

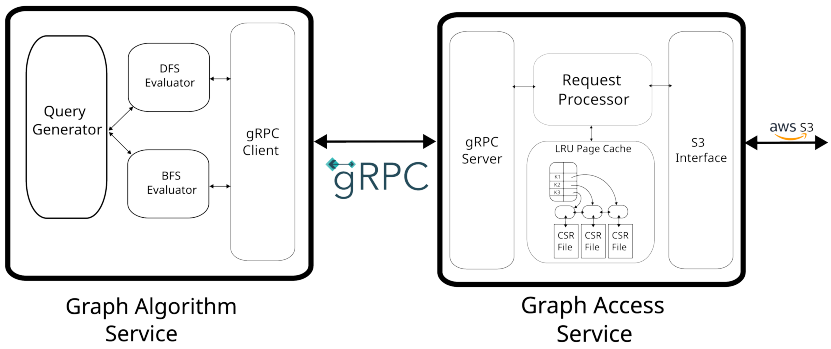


Figure 3.2: Baseline Implementation

3.3 Graph access service

In this section, we first describe a modification of the CSR format in Section 3.3.1 which will help us improve our cache performance. In Section 3.3.2, we describe how we leverage this updated file format and employ the caching schemes mentioned in Section 2.3.

3.3.1 Modified CSR structure

In the baseline implementation of graph access service described in Section 3.2, we had to download entire files to access a node's neighbours. This approach suffers from a couple of problems, which makes fetching inefficient and restrictive. The first problem is that we have to fetch and parse the information for an entire file in order to access a single node's neighbours which is wasteful. Furthermore, this approach hinders our ability to achieve a higher level of granularity for better cache performance. Second, this approach makes it harder to utilize the fact that AWS S3 replicates files to various servers. Due to these problems, we decided to modify the underlying binary format and the way we fetch a node's neighbours.

Figure 3.3 shows the binary format we use for fetching a node's neighbour. This format can logically be divided into three layers. The first layer, which is always of a constant size, contains the first and last node identifier present in this file. The second layer contains the byte offset for the first incoming edge for a node and the first outgoing edge for a node. Since these offsets are of a constant size and the fact that the first layer of the header indicates how many nodes are present in this file, we can calculate the size of this second layer. The third and final layer contains the edge information, which in our case consists of the edge label and the edge destination. This type of structure closely resembles a traditional CSR format except for the fact that we store both incoming and outgoing edges in the same array and that we store byte offsets instead of array indices.

The aforementioned format gives us the ability to fetch a node's neighbours without having to download an entire file. With this modified structure, we can store the first layer of every file in memory and fetch the second layer of a file when it is first accessed. After doing so, we would be able to fetch a node's incoming or outgoing neighbours without any additional overhead.

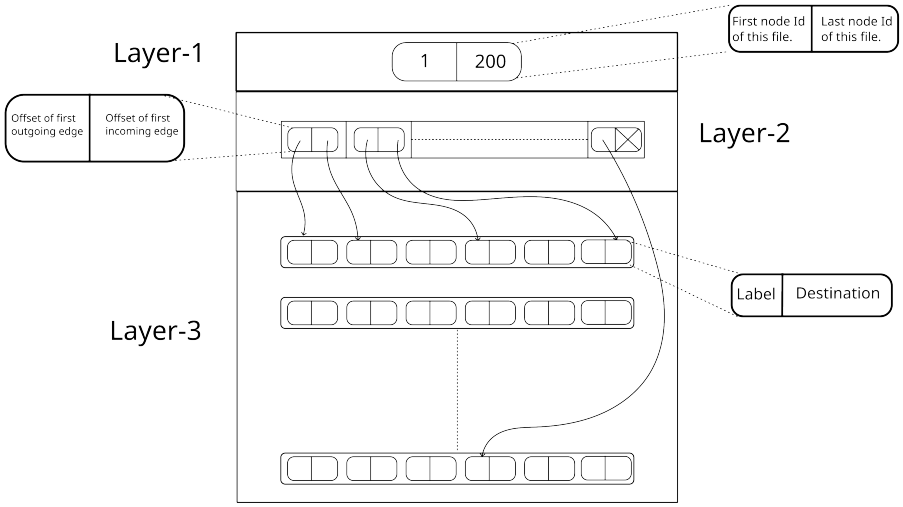


Figure 3.3: Updated CSR Format

Furthermore, we can now parallelize requests to a single file, which would lead to better throughput since these requests would likely be distributed to different instances within AWS S3. As an additional benefit, we can now perform caching on a more granular level since we do not incur the overhead of fetching and parsing an entire file whenever we need to access a node's neighbours.

3.3.2 Caching and Prefetching

Figure 3.4 shows the final architecture of the graph access service. The service consists of three basic parts:

1. **Interfaces for external communication**: There are two components which provide an interface with external resources and are colored grey in the figure. The first component, on the left, exposes a gRPC interface which receives requests for accessing a node's neighbours, starting traversals, and ending traversals. The second component, on the right, is responsible for accessing AWS S3. This component takes a filename along with starting and ending file offsets and returns the requested file content.

2. **Orchestration**: There are two components that help with orchestration of request and are colored blue in the figure. The first component is the 'Request Processor' which is responsible for interacting with the caches, moving data between caches, and fetching data from AWS S3 in case it is not present in the caches. The second component titled 'Vertex offsets' contains metadata related to the CSR format that we described in Section 3.3.1.

3. **Caches**: There are two different types of caches that are used in this service and are colored green in the figure. The first of these components is an LRFU cache which is used to store nodes that were previously accessed. The second component is a per-query prefetcher which fetches the nodes that are likely to be accessed by a client in the near future. The rest of this section is devoted to the description of these two components.

The first major component of our cache in the **LRFU cache**[17]. As mentioned in Section 2.3, the main idea of this cache is to take both recency and frequency into account while making the decision about which item to evict from the cache. While initializing this cache, we make a decision about how much weight should be given to an objects recency versus its frequency. Note

that unlike perfect-LFU schemes, we only store the frequency of the elements present in the cache. After initializing the cache, it maintains a heap of elements where each element is assigned a score based on the last time it was accessed along with its frequency. The key to maintaining this heap is the realization that the relative ordering of the heap remains constant if none of the elements are accessed. This is because the degradation of score due to ageing happens at the same rate for every element. This means that only a change in frequency can alter the relative ordering of elements, and since the frequency only changes when an object is accessed, we only need to change the position of the accessed node within the heap. This enables us to perform access and insertion operations in $\mathcal{O}(\log(n))$ time. Although simple LRU cache implementations have an access and insertion time complexity of $\mathcal{O}(1)$, this cache provides us with a good tirade between time complexity and flexibility for different workloads. We will talk more about the empirical evidence supporting our choice in the next chapter.

The next component, **the Prefetcher**, is inspired by the work done by Bok et al. [6]. This component fetches the neighbours of nodes that were accessed by a particular traversal. Every time a node's neighbours are fetched, these neighbours are added to the 'Prefetch Queue'. The elements from this prefetch queue are read by the worker threads that are attached to every prefetcher cache. These worker threads send our requests to fetch the neighbours of nodes in the prefetch queue and add the nodes to the 'In-flight queue' while they wait for the response. Finally, once the neighbours for a node are available, they are added to the prefetcher cache which is a standard LRU cache. We will now discuss some of the design choices that we have made for this cache.

First, if there are multiple traversals happening, we would create a separate prefetcher for each traversal. This choice ensures that there is minimal interference on the performance of a traversal depending on other traversals that may be happening. This comes with the added cost of having more worker threads. However, using a green-thread model or asynchronous programming can help mitigate this issue to a large extent.

Apart from separate prefetcher caches, we also have an 'In-flight Queue' within each prefetcher. This is useful because of the time gap between sending a request and getting a response. This time, as we will see in the next chapter, is an order of magnitude greater than the time scale at which the graph access service and the graph algorithm service work. Therefore, it makes sense to have a small amount of extra memory to ensure that if a request is already in progress, we do not make a duplicate request, which would almost always take longer to execute.

Finally, we note that the size of the prefetch queue is finite. Therefore, we need to have some strategy to handle the case when this prefetch queue is full. We argue that this strategy needs to be different for both BFS and DFS. This is due to the relevance of nodes that are added to this prefetch queue. The node at the front of the queue in BFS will be accessed earlier than the neighbours of a node we are trying to add to the queue. However, in case of DFS, the opposite is true because the node we are trying to add is likely the node at the top of the stack and would be accessed next. This leads us to having two different strategies for dealing with a full prefetcher queue for BFS and DFS:

1. **BFS**: In case of BFS, the prefetcher queue behaves like a queue data structure and if the queue is full, the newer elements are discarded.
2. **DFS**: In case of DFS, the prefetcher queue behaves like a stack data structure and if the queue is full, the newer elements are added to the top of the stack, and the elements at the bottom of the stack are discarded.

In this section, we described a file format that enables us to perform caching at a more granular level, and we described how this granular caching helps us design caches that may be better suited for masking AWS S3's access latency. In the next section, we turn our attention to graph algorithm service where we will modify our traversals to better fit the capabilities of AWS S3.

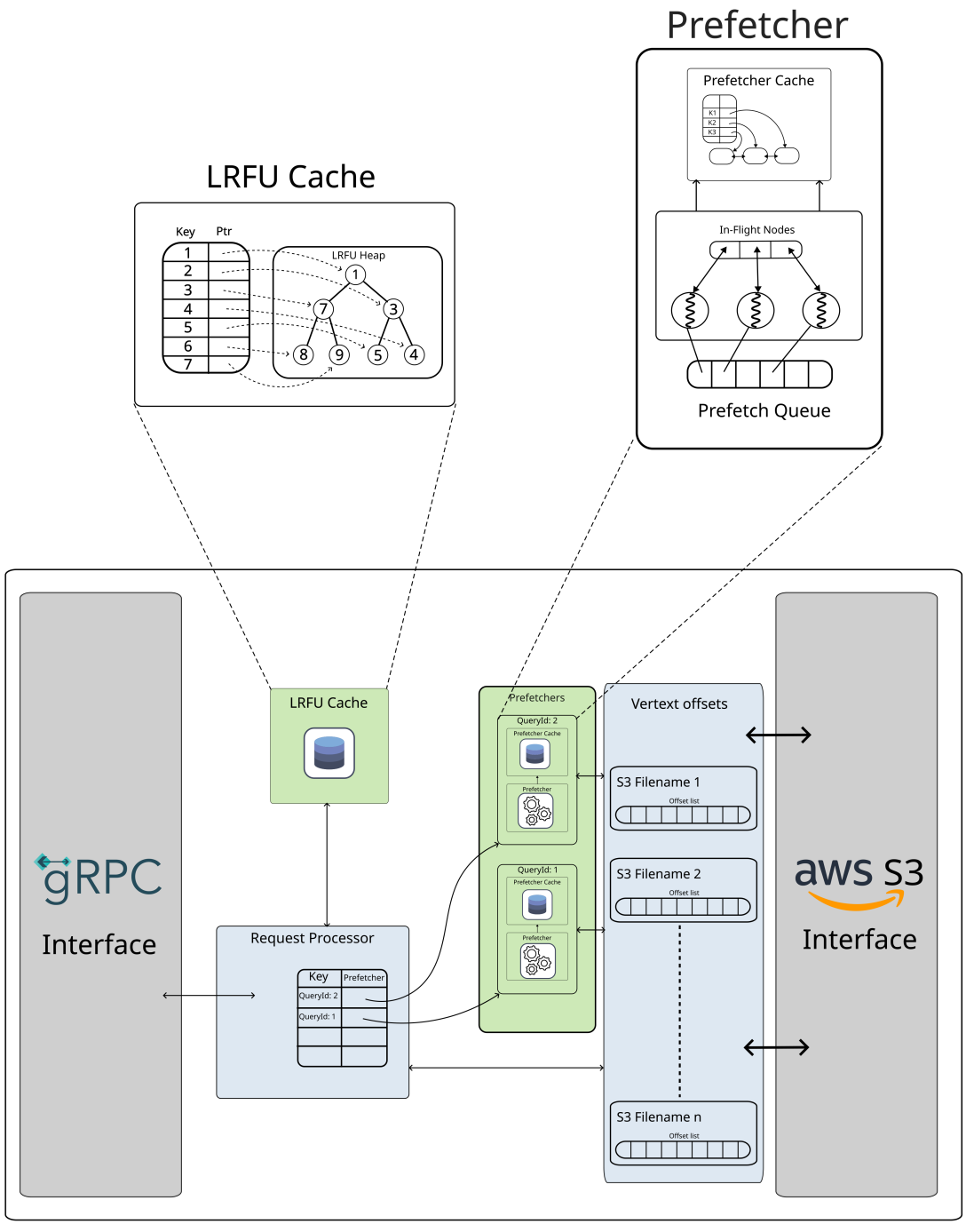


Figure 3.4: Graph Access Service

3.4 Prallelizing traversals

One of the main advantages of using AWS S3 is that it is able to sustain a very high-throughput. The caching mechanisms mentioned in the previous sections reduce the latency for accessing a node's neighbours and mitigate expensive network calls to AWS S3 in the hot-path. These components do help with reducing latency but they do not do anything to ensure that make use of the high-throughput provided by AWS S3. In order to do that, we parallelize our traversal algorithms. In this section, we will present parallel implementations of BFS and DFS that would help us speed up our traversals by utilizing the high-throughput of AWS S3.

3.4.1 Parallel BFS

Parallelizing breadth-first search in relatively straightforward. Listing-3.1 contains the pseudo-code for our implementation of parallel BFS. This implementation contains two queues, one for the current level and the other for the next level of the traversal. Instead of processing the current level within a single thread, we launch a predefined number of workers who take elements from the queue at the current level in a work stealing fashion and add their results to the next level of the traversal. The addition of nodes to the next level as well as to the set of seen nodes needs to be synchronized in accordance with the threading model of the language of implementation.

```
def ParallelBFS(startNode: nodeId, edgesToFollow: list[Edge]) -> list[nodeId] {
    startQueue = Queue[nodeId](startNode)
    for edge in edgesToFollow {
        nextLevelQueue = Queue[nodeId]()
        seen = Set[nodeId]()
        for i = 0; i < MAX_WORKERS; i++ {
            ## Use language provided utility to launch BFSWorker.
            launch_worker(startQueue, nextLevelQueue, seen, edge)
        }
        await_completion()

        startQueue = nextLevelQueue
    }

    return startQueue
}

## Operations on input, output, and seen are synchronized between
## workers.
def BFSWorker(input, output: Queue[nodeId], seen: Set[nodeId], edge: Edge) {
    while (input.hasMoreElements()) {
        toProcess = input.Poll()
        neighbours = fetchNeighbours(toProcess, edge)
        for neighbour in neighbours {
            if neighbour not in seen {
                seen.add(neighbour)
                output.add(neighbour)
            }
        }
    }
}
```

Listing 3.1: Parallel BFS

Figure 3.5 contains an example of how this algorithm would work for a simple tree like graph shown on the left side of the figure. The figure also contains three workers that are colored green, blue, and red. This figure shows three iterations of the algorithm, along with how the distribution of work might take place. We now describe each of these three iterations:

1. **Iteration 1**: In the first iteration, where we only have a single node, this node gets assigned to one of the workers at random. In this case it gets assigned to the green worker. At this stage, the green worker fetches the neighbours of this node and adds them to a shared set,

which will be used as the starting point for the next iteration. In this case, the worker adds nodes 2,3,4, and 5 to this shared set.

2. **Iteration 2**: In this iteration, the nodes from the shared set of the previous iteration are distributed among all three workers. Just like the previous iteration, every worker fetches the neighbours of the nodes assigned to them and adds the result to a shared set. In this iteration, nodes 2 and 3, which are assigned to different workers, try to add the node 7. This is the reason why we need a shared set, otherwise, we may end up processing node 7 twice in the next iteration.

3. **Iteration 3**: Just like the previous iteration, the nodes from the shared set are distributed across workers. Subsequently, these workers add the neighbouring nodes of their assigned nodes to the shared set.

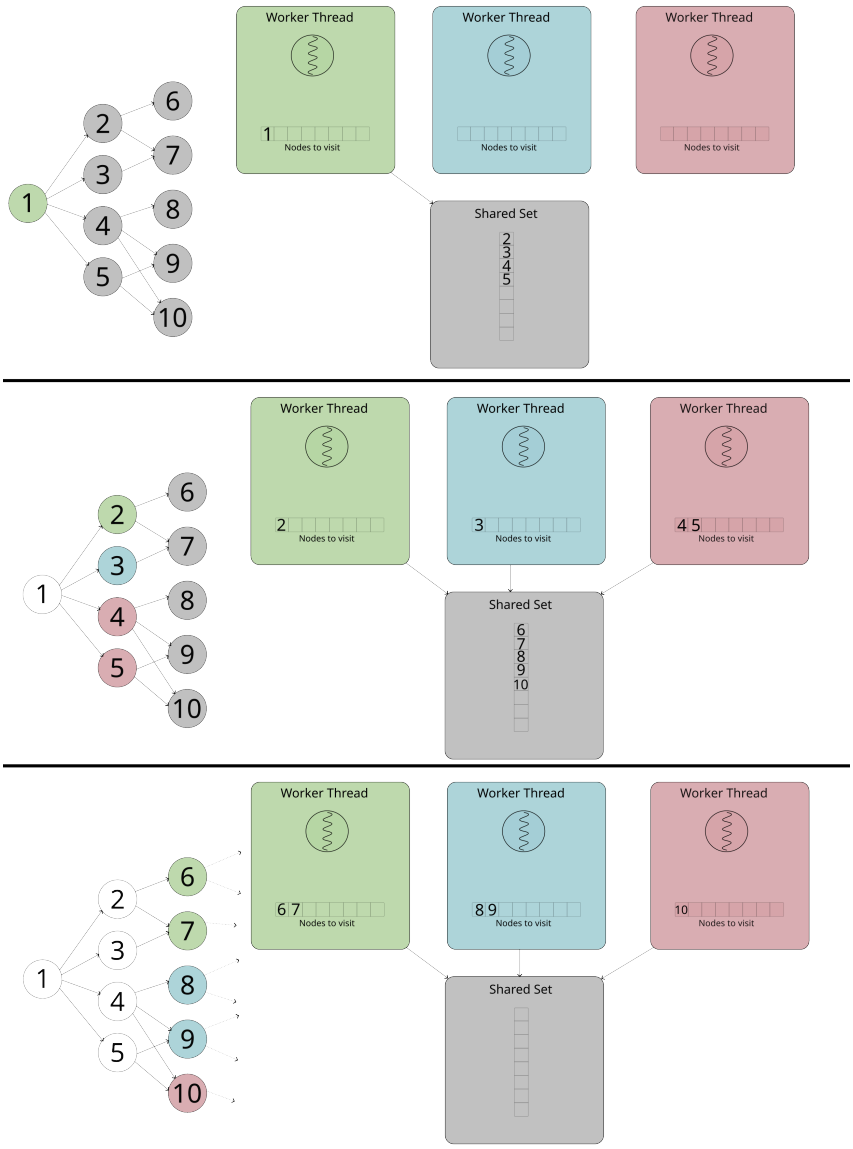


Figure 3.5: Parallel BFS

3.4.2 Parallel DFS

In this section, we describe our approach for parallelizing depth-first search. Before we begin our description, it is important to note why a strategy similar to the one described in the previous section would not result in equal allocation of work. If we simply delegate the task on performing DFS on a node's neighbours to different worker threads, then it is possible that some worker threads might terminate significantly earlier than others. This is because the number of nodes that can be explored from a given node might vary significantly from one node to the other. Let us take the example of traversing a social network graph where we start with a single person and find people connected to this person till a certain depth. If the starting person has two friends 'A' and 'B', let us assume that 'A' has no friends while 'B' is a celebrity and has many friends. In this case, if we assign the work of performing DFS from 'A' to one worker and the work of performing DFS from 'B' to another worker, we would end up with an uneven distribution of work. This is because performing DFS on 'A' would require no further processing while the DFS on 'B' is likely to go on for quite a few iterations. Due to this problem, we need a better way to parallelize depth-first search.

The main idea of our approach is to dynamically rebalance the amount of work being done by each worker in every iteration. One way to perform this dynamic rebalancing was suggested by Rao et al.[21] where they suggest a work stealing algorithm in which a worker 'steals' nodes from other workers' stacks when its own stack becomes empty. Our algorithm, as shown in Listing-3.2 works on a similar principle. The one notable difference is that instead of stealing work, every worker proactively assigns work to different workers after fetching a node's neighbours which ensures that the difference between the amount of work is minimized. Another important point to note about the implementation is the structure of the data structure that stores seen nodes. Instead of simply storing nodeIds, we store nodeId along with the level at which a node was seen. This is done to ensure that for a traversal with max depth 'd', we return all the nodes that can be reached by taking 'd' steps from the source node.

```
def ParallelDFS(startNode: nodeId, edgesToFollow: list[Edge]) -> list[nodeId] {
    stack = Stack[nodeId, level]((startNode, 0))
    seen = Set[nodeId, level]()
    result = Collection[nodeId]

    while stack.isNotEmpty() {
        toProcess := stack.pop()
        ## Termination condition
        if toProcess.level == edgesToFollow.size() {
            result.add(toProcess.nodeId)
            continue
        }
        ## Fetch the neighbours according to the edge that should be followed at
        ## this level.
        nextLevel = toProcess.level+1
        neighbours = fetchNeighbours(toProcess.nodeId, edgesToFollow[toProcess.
            level])
        neighbours = filterSeen(neighbours, seen, nextlevel)

        numWorkersAvailable = getAvailableWorkers()
        ## Equally partition the neighbours into 'numWorkersAvailable+1' lists
        neighbour_partitions = divide_work(neighbours, numWorkersAvailable)

        for i = 0; i < numWorkersAvailable; i++ {
            launch_worker(neighbour_partitions[i], seen, result)
        }
    }
    await_completion()

    return result
}

## Filter out nodes that we have seen at a particular level.
```

```
def filterSeen(nodes: list[nodeId], seen: Set[nodeId, level], level: level) -> list
    [nodeId] {
    res = list[nodeId]()
    for node in nodes {
        if (node, level) not in seen {
            seen.add((node,level))
            res.add((node,level))
        }
    }
    return res
}

## This function mimics the original implementation with the only change being
## that the seen set is shared and that this method has its own stack.
def DFSWorker(nodes: list[(nodeId,level)], seen:Set[nodeId,level], res: list[nodeId
    ]) {
    workerStack = Stack[nodeId, level](nodes)

    while stack.isNotEmpty() {
        ## Processing logic same as the parallelDFS function
    }
}
```

Listing 3.2: Parallel DFS

Figure 3.6 shows an example of how this implementation works. In this figure, we only have two worker threads, colored green and blue. We now describe the three iterations shown in the figure:

1. **Iteration 1**: In the first iteration, the green worker thread processes the node 1. Before the start of this algorithm, the tuple $(1, 0)$ is added to the shared node-level set. This indicates that node 1 was seen at level 0. After fetching the neighbours of the first node, we check if there are any worker threads available to take over the work for some nodes. We find that the blue worker is indeed idle and therefore, some of the neighbours can be added to its local stack. Here we assume that out of the three nodes, two are added to the stack of the green worker and one is added to the stack of the blue worker. During this partitioning, we also add the newly visited nodes to the shared set.

2. **Iteration 2**: In the second iteration, the green worker explores the neighbours of node 2 and the blue worker explores the neighbours of node 4. After exploring the neighboring nodes, both of the workers check if some of the newly explored nodes can be added to another worker's stack. However, since both workers are busy, all the nodes are added to their own stacks. Finally, like in previous iteration, the newly visited nodes are added to the shared set. Notice that there is a race condition between the green worker and the blue worker to add node 7. In order to ensure that this node is not processed twice at the same level, the workers need to check if node 7 has been visited at depth of 2. If it has not been visited, then the worker will add that to the shared set and then add it to its stack. With this mechanism, only one of the two workers will be able to take responsibility for processing this node.

3. **Iteration 3**: During the final iteration, the green worker will process node 5 and the blue worker will process node 7. All the nodes that have already been seen are also added to the shared set. After this stage, the two workers will go on to process nodes 9 and 10. Finally, the green worker will process node 3 but since all its neighbours have already been processed, the algorithm will terminate.

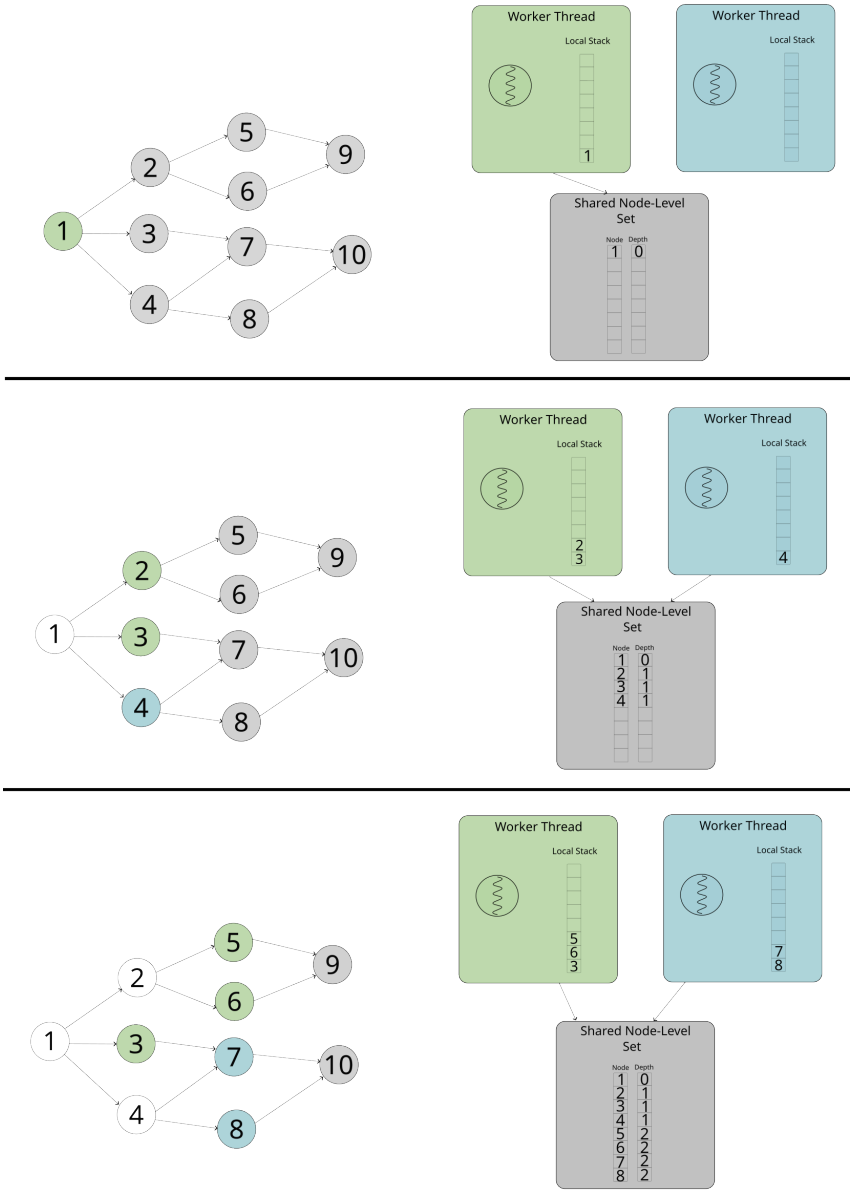


Figure 3.6: Parallel DFS

Chapter 4

Evaluation

Before presenting the results of our comparison with the baseline, we will describe our benchmarking procedure. Within the benchmarking methodology, we highlight the queries we run along with the dataset on which we run the queries. Finally, we describe how the services are deployed and how the results are collected.

Our benchmark contains a total of seven different types of queries. Each of these seven queries starts with a random source vertex and performs the desired traversal according to the traversal specification. A traversal specification specifies the label and direction that needs to be followed for every hop. For example, a traversal specification might instruct the algorithm to start from a node of type 'Person', traverse the outgoing edges with the label 'PersonKnows', and then find all the incoming edges for 'PostCreator'. As a result of running this traversal, we would get the identifiers for all the posts created by a person's friends. The schema containing the specification of the nodes and labels mentioned here can be found in Appendix A. This schema describes a social network consisting of people who can create posts within forums, comment on posts, etc.

The seven traversals that we have chosen can be divided into four parts. The first two traversals are 1-Hop traversals, which means they only access a node's immediate neighbours. Similarly, we have two queries for 2-Hop traversals, and 3-Hop traversals. One of the two queries in each category is chosen in such a manner that the result cardinality is close to one, and the other query has a higher cardinality. However, for 4-Hop traversals, the baseline implementation was so slow that we do not have a high cardinality traversal for 4-Hops. The following list contains the details of the queries that were used for the evaluation. Each of these queries was repeated 20 times with different source nodes chosen at random.

1. **1 Hop**
   (a) **Q-1**: Start with a node of type 'Forum', and traverse outgoing edges with the label 'ForumMember'. This gives us all the forum members for a particular forum.
   (b) **Q-2**: Start with a node of type 'Post', and traverse outgoing edges with the label 'PostContainerForum'. This gives us the container forum for a post.
2. **2 Hop**
   (a) **Q-3**: Start with a node of type 'Person', and traverse the edges with the label 'PersonKnows' twice. This gives us all the people that a person's friends know.
   (b) **Q-4**: Start with a node of type 'Person', and traverse the edge with the label 'PersonWorksAt'. After this traverse the relation 'OrgLocation'. This gives us the locations of the companies that a person has worked for.
3. **3 Hop**

(a) **Q-5**: Start with a node of type 'Person', and traverse the edges with the label 'PersonKnows'. After this, traverse the edge with the label 'PersonInterestTag' to get the interests of all these people. Finally, traverse edges with the label 'TagClass' to find the tags for each of these interests. This query gives us the tags for the interests of a person's friends.

(b) **Q-6**: Start with a node of type 'Post', and traverse the edge with the label 'PostContainerForum' to get the forum which the post is a part of. Then we traverse the 'ForumTag' label followed by the 'TagClass' label. This gives us the tag classes for a post's forum.

4. **4 Hop**

(a) **Q-7**: This query starts with a 'Comment', and looks for this comment's creator by following the 'CommentCreator' label. Then, we look at the tag superclasses of this person's interest tag by following 'PersonInterestTag', 'TagClass', and 'TagClassSuperclass' labels.

As mentioned in the previous paragraph, we run these queries on the LDBC-SNB dataset whose schema is provided in Appendix A. However, LDBC provides multiple datasets with the given schema, all of which are of varying sizes. For this thesis, we use datasets with a scaling factor of 1 and 10 (SF-1 and SF-10). As mentioned in Section 3.1, we convert these datasets to a binary format before processing. During this processing, we remove edge and node-related information since that is not needed during the traversal. After removing the unnecessary information, we partition the adjacency into multiple files, each of which is then uploaded to S3. Table 4.1 contains the specifications of the two datasets that we use.

	SF-1	SF-10
Nodes	3 Million	30 Million
Bidirectional edges	17 Million	170 Million
Number of partitions	37	173
Total size	286 MB	2.8 GB

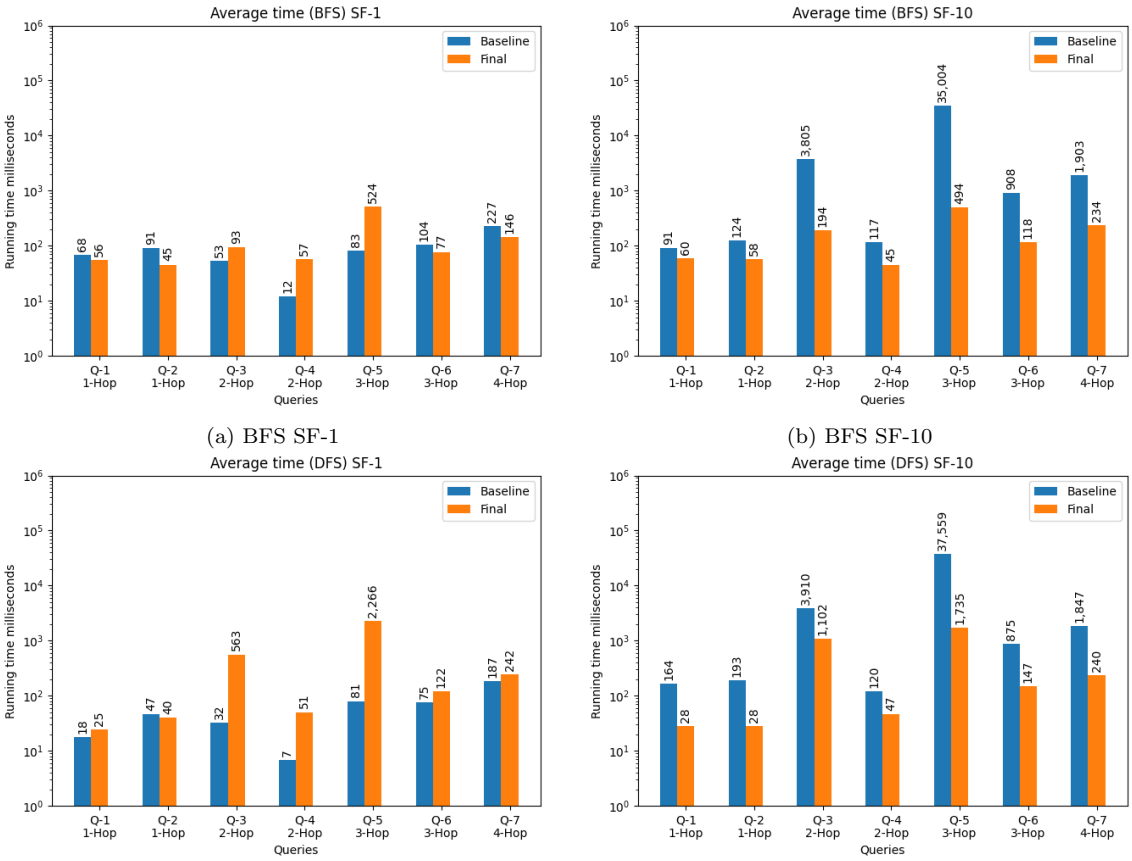
Table 4.1: Dataset specifications

In order to run these queries, we run the graph access service and graph algorithm service on an EC2 instances[3]. AWS provides a variety of compute instances for different workloads. In our case, we need a general purpose compute instance which has high bandwidth. At the time of writing this thesis, instances labeled 'c7gn' fit these specifications the best. The particular instance that we use for most of our benchmarks is labeled 'c7gn.xlarge' which provies a network bandwidth of 50 Gbps, has 8 Gigabytes of RAM, has 4 vCPUs, and costs 0.25 $/hour.

4.1 Comparison with baseline

Figure-4.1 contains the comparison of our final implemnetation, as described in Section 3.3 and Section 3.4, with the baseline implementation described in Section 3.2. Figures 4.1a and 4.1b compare the performance of Breadth first search for scaling factor of 1 and 10. Similarly, Figures 4.1c and 4.1d do the same for Depth first search. Note that the y-axis on these graphs is logarithmic. Looking at these graphs, we notice the following facts:

1. **No benefit for small graphs**: From figures 4.1c and 4.1a, it is clear that our proposed architecture perform worse than the simpler baseline implementation. This is primarily because the number of different files is quite small for SF-1 and therefore most of the files required for a traversal are loaded in the first few runs of that traversal. After that, all the traversals are performed from memory.



(a) BFS SF-1

(b) BFS SF-10

(c) DFS SF-1

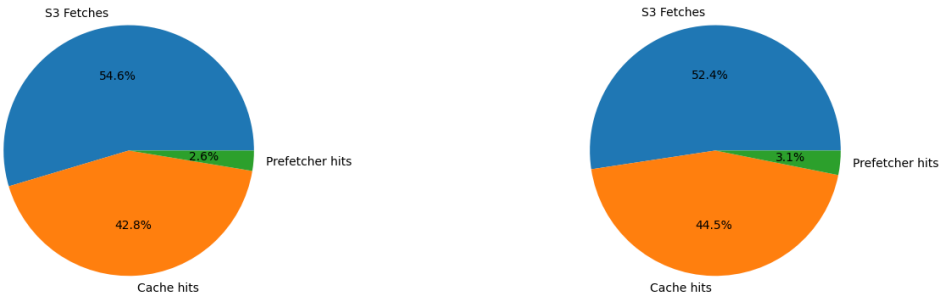
(d) DFS SF-10

Figure 4.1: Baseline comparison

2. **The proposed approach scales better**: If we compare the results of running DFS and BFS on SF-1 and SF-10, we notice that the running times for our final implementation remain relatively unchanged. However, the running times for the baseline implementation degrade by one to two orders of magnitude. Furthermore, the total time for running the entire workload for the baseline implementation on SF-10 with BFS was around 14 minutes whereas the running time for our proposed implementation was just 24 seconds. This is a 35x improvement in the running time of the entire workload.

3. **BFS > DFS**: These results also show that the simpler parallel BFS implementation works much better than the parallel DFS implementation. For SF-10, the entire workload took 68 seconds while running DFS but took only 14 seconds while running BFS. We believe this is primarily due to the fact that the DFS implementation is more complicated, and uses more memory than the implementation for BFS.

Figure-4.2 shows the percentage of requests served by the LRFU cache, the prefetcher, and AWS S3. We can see that the distribution is very similar for both BFS and DFS. We also notice that the LRFU cache is very effective and mitigates almost half of all the network calls to AWS S3. However, we also notice that the prefetcher only serves around 3% of all the requests. We will now investigate why the prefetcher is less effective than we had hoped.



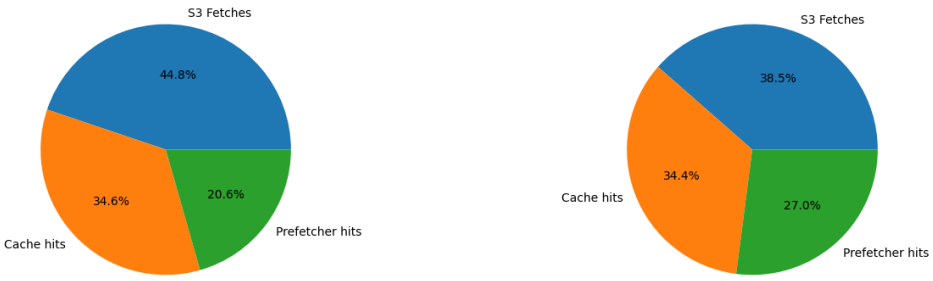
(a) BFS (b) DFS

Figure 4.2: Request Distribution

Is the prefetcher useless?

To answer the question regarding the effectiveness of the prefetcher, we need to remind ourselves that the traversal algorithms being run on the graph algorithm service are parallel. This means that multiple units of concurrency are trying to advance the traversal at the same time. This interferes with the functioning of the prefetcher, which itself is trying to advance the traversal by utilizing concurrency. This means these two mechanisms are competing against each other. Since our implementation has more concurrency units in the parallel traversal algorithm than the prefetcher, the prefetcher is not very beneficial.

This reasoning can be verified by running sequential versions of BFS and DFS. Figure 4.3 shows the request distribution when we run sequential traversals instead of their parallel counterparts. We can see that the prefetcher's share is substantially more. Interestingly, it turns out that the prefetcher is more effective for DFS than for BFS. Since the sequential implementations for BFS and DFS are equally efficient, we found that the workload running time for DFS was 13 % lower than the workload running time for BFS. However, the total running time for the sequential versions was still significantly higher than their parallel counterparts. Parallel BFS was 82 % faster than sequential BFS, and Parallel DFS was 44 % faster than sequential DFS.



(a) BFS (b) DFS

Figure 4.3: Request Distribution Sequential

Given these results, we need to address whether it makes sense to have parallel algorithms

along with the prefetcher. If we want performance without regard for hardware cost, it does indeed make sense to have both. However, in most cases, there is a tradeoff to be made here. If we want faster traversals at the expense of lower throughput, it makes sense to use only parallel algorithms. This is because the traversal algorithm has more precise information, compared to the prefetcher, about the edges that need to be traversed. However, if we want higher throughput by performing multiple traversals in different concurrency units, it makes sense to use a prefetcher. This is because not having the prefetcher would free up more concurrency units for the graph algorithm service, thereby increasing the throughput. For the rest of this thesis, however, we will still use both the prefetcher and the parallel algorithms as they yield the best performance, albeit marginally, when used together.

Exploiting parallelism

As we explained in Section 3.3, one of our motivations for introducing the modified CSR format was to enable more granular access to the graph. With this granular access, we can make more requests for a given bandwidth, this enables us to increase the number of traversals being executed in parallel. In this section, we present the impact of running multiple traversals in parallel.

Figure-4.4 shows the total workload running time for different levels of parallelism. As the figure shows, we see significant improvements in the running time. However, these improvements taper off at higher levels of parallelism. By going from parallelism of 1 to 4, we see an almost 4x improvement, however, going from parallelism of 10 to 20, we hardly see any improvements. This suggests that although increasing the parallelism helps, eventually the hardware becomes the limiting factor for the overall running time.

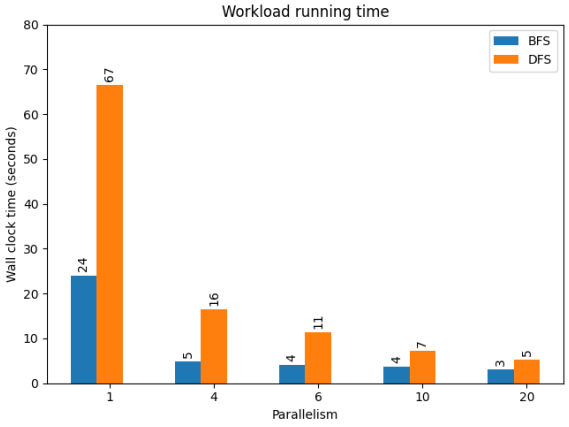


Figure 4.4: Parallelism impact

At this stage, our implementation can process a workload consisting of $7 * 20 = 140$ queries in 3 seconds whereas the baseline implementation processed the same queries in 14 minutes. Even if we increase the parallelism for baseline implementation to 2, the running time is still more than 7 minutes. Increasing parallelism beyond this point is not feasible in the baseline implementation because of its underlying architecture.

Exploring AWS S3 One-Zone

AWS S3 offers different storage classes for objects, and each storage class has different performance characteristics. S3 One-Zone is one such storage class that was introduced by AWS in the past few months, and according to AWS, it has up to ten times lower latency than general-purpose storage. As our main goal for this thesis was to find a way to reduce object access latency, we also performed some of our experiments with S3 One-Zone. Note that all the experiments in the previous sections used S3's general-purpose storage, which is S3's default storage type.

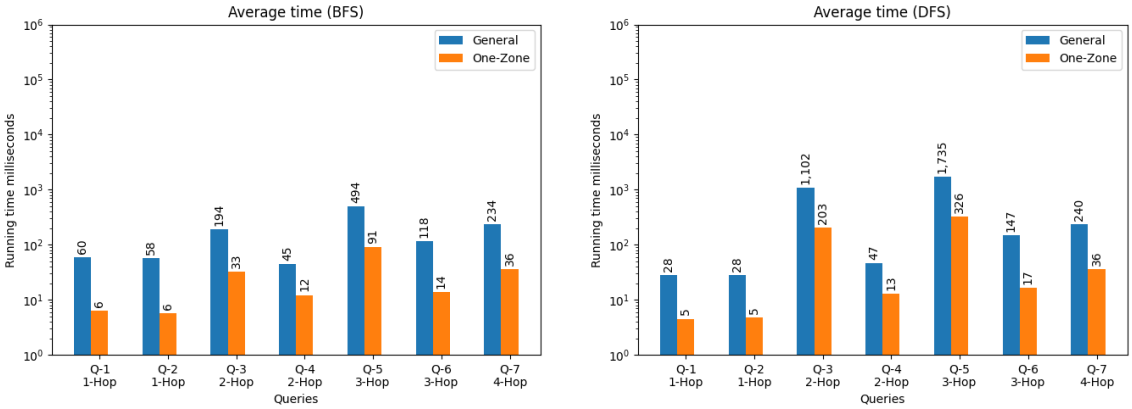
Before we present our experimental evaluation, it is worth discussing how this new storage type achieves ten times lower latency and the tradeoffs associated with its use. AWS S3 is a region-specific service, which means that all its servers along with the underlying data are present in a single AWS region. For redundancy, each AWS region consists of multiple availability zones, and each availability zone consists of multiple data centers. Files stored as general-purpose storage are replicated across multiple availability zones, whereas files stored as One-Zone storage type are replicated only within a single availability zone. This architectural distinction gives rise to the following changes:

1. **Lower latency**: Since the physical distance between the servers is reduced, the network latency is also lower. However, this also implies that the AWS EC2 instances processing the data must be in the same availability zone.

2. **Lower resilience to outages**: Since the general purpose storage is replicated across multiple availability zones, an outage in one of the availability zones is unlikely to have an operational impact on the service. On the other hand, if an object is stored in a single-zone, then it is not resilient to single-zone outages. However, it must be noted that the last time AWS S3 had an outage was in 2017.

3. **Higher storage cost and lower request cost**: At the time of writing this thesis, the cost of storage for S3 One-Zone is 0.16 $/GB/month, whereas the cost for S3 General is just 0.023 $/GB/month. However, the cost of a GET request for S3 One-Zone is lower than that for S3 General. A thousand requests to S3-One-Zone cost 0.0002 $, whereas a thousand requests to S3-General cost 0.0004 $.

With these tradeoffs in mind, we present the performance of S3-One-Zone for our workload. Figure-4.5 shows the average running times of traversals on the SF-10 dataset with inter-traversal parallelism of 1. We observe that the running time is reduced by almost an order of magnitude. The total workload running time was 83 % lower for BFS and 82 % lower for DFS. Therefore, we see consistent improvements across the workload. However, these improvements become less substantial as we increase the level of inter-traversal parallelism because the bottleneck shifts to the data processing capabilities of the underlying instance.



(a) BFS (b) DFS

Figure 4.5: Comparing S3 General with S3 One-Zone

4.1.1 Optimizing partition sizes

In distributed cloud storage systems like AWS S3, a file/object is the entity that gets distributed across the cluster. This distribution enables them to promise (in their 'Service Level Agreement'

(SLA)) that every object can sustain a read throughput of 5,500 requests per second. At this throughput, the SF-10 dataset, which has 173 files (Table 4.1), can sustain a read-throughput of 951,500 requests per second. This read-throughput of almost a million requests per second is not achieved in our benchmarks. However, their SLA does not mention the impact of having more or fewer files on the request latency. In this section, we will investigate whether partition size impacts the request latency.

Figure-4.6 shows the running times for the parallel BFS algorithm at various levels of parallelism and various partition sizes. We do not see any significant difference between the running times with parallelisms of 1 and 20, but we do see a difference for parallelisms of 4, 6, and, 10. We believe that we do not see a difference for parallelism of 1 because the request load is low, however, we do not see any difference for parallelism of 20 because the data processing capabilities of our instance become the bottleneck. For parallelism of 4,6, and 10, we see that there is a difference of around 2.5 seconds between partition size of 2 MB and that of 128 MB. This suggests that smaller partition sizes may yield better latency. We note that even for the largest partition size of 128 MB, the promised throughput(SLA) that AWS S3 can provide is 121,000 requests per second which is still an order of magnitude more than the throughput that we utilize. Therefore, the latency improvements are not because of any throttling. However, it may be the case that with more files we use a larger percentage of AWS S3's cluster which resulted in lower latency.

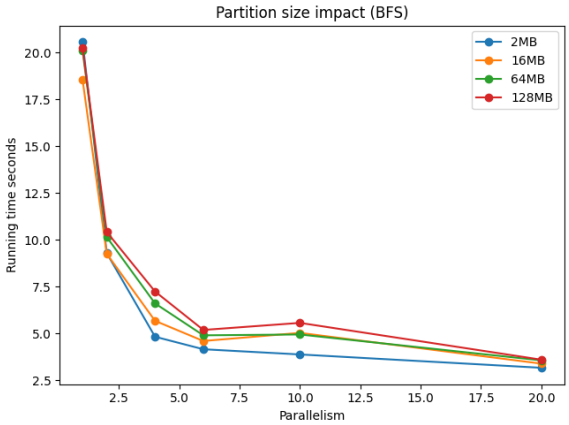


Figure 4.6: Partition Size Impact

Since AWS S3's documentation does not mention any SLAs regarding latency, the results presented in this section might change over time. However, at the time of writing this thesis, we see a noticeable difference in performance by simply changing the partition size of the files. Since AWS S3's pricing model charges us for storage and not the number of files, for a given workload it is worth exploring the implications of partition size on the performance.

4.1.2 Distributed Deployment

In the last few sections, we have noticed that with inter-traversal parallelism of 20, our instance's processing capabilities became a bottleneck. In this section, we use multiple instances for processing. Using this experiment, we argue that the proposed architecture is scalable to multiple instances and that we get the desired speedup by adding more instances. To do this, we deployed the graph algorithm service on a single instance and deployed the graph access service on two separate instances. During this benchmark, we compare the difference in performance when the graph algorithm service utilizes only one instance of graph access service versus when it utilizes both.

Figure 4.7 shows the results of our benchmark. Note that unlike the previous sections, where we ran 20 traversals for each of the 7 traversal specification, here we ran 100 traversals for each

specification. As we can see, the running time with two instances was a third of the running time with a single instance. However, we do see that increasing levels of parallelism after 10 does not have a substantial impact. This indicates that the graph access service is overloaded, as the performance improves when two instances of graph algorithms service are deployed, suggesting that graph algorithm service is not the source of a bottleneck. Interestingly, we see that doubling the amount of compute reduces the running time by a factor of three. This result is surprising because we usually see a speedup of less than twice when we double the amount of resources, but here we see a 3x performance improvement.

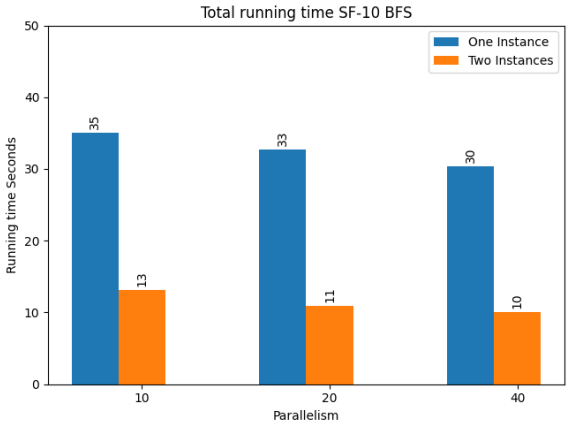


Figure 4.7: Distributed Deployment

Our intuition for this bottleneck is further supported by the fact that running the same workload on a larger instance yields better performance. Figure 4.8 shows the results of running the benchmark with an instance of type c7gn.2xlarge instead of c7gn.xlarge. As the name suggests, this instance has twice the capabilities of the earlier instance. As a result, the total running time was halved for parallelism of 40 compared to Figure 4.7. Furthermore, we continue to see the phenomenon of getting almost 3x performance improvement with twice the resources.

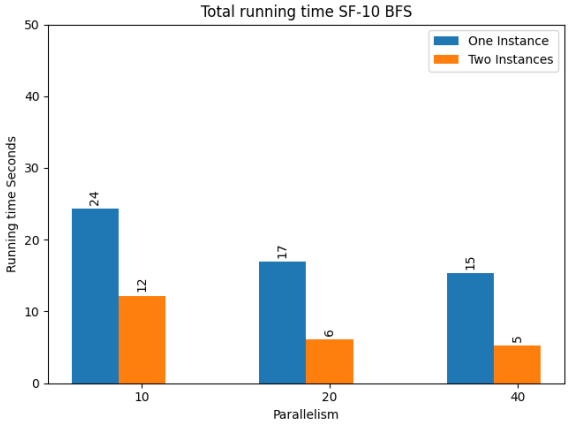


Figure 4.8: Distributed Deployment (Larger instance)

In this section, we verified the feasibility of our architecture for a distributed setting and showed that we can achieve substantial performance improvements by adding more hardware.

4.2 Comparison with other tools

In this section, we will compare the performance of our service with tools that also enable users to perform traversals on graphs. Both tools that we compare our solution with here do not provide a separation between storage and compute, thus they are not direct competitors. However, these tools are widely used for graph processing and therefore, a comparison with these tools enables us to gauge our tool's standing among contemporary graph processing tools.

Figure 4.9 contains four broad categories of frameworks that can be used for performing traversals on large graphs. At the top, we have the big data processing frameworks. These frameworks can distribute data across a large cluster of compute instances and perform a variety of computations on the data. These frameworks can process anything from images in a datalake to real-time information on a message bus. However, due to their generality, these frameworks also have higher latency than databases. Relational databases, depicted below big data frameworks, are useful for a smaller set of use cases but provide lower latency. There is a wide variety of relational databases, some are more suited for transactional workloads while others are better suited for analytic workloads. Next on our list are graph databases which are not as flexible as relational databases but provide lower latency for graph-specific applications like traversals[16]. Finally, we have custom-built technologies that are built to provide the best possible latency for the traversal of a specific type of graph data. For example, Alibaba created a custom distributed system for performing 4-hop traversals over a graph containing 100 billion edges within 4 milliseconds[22].

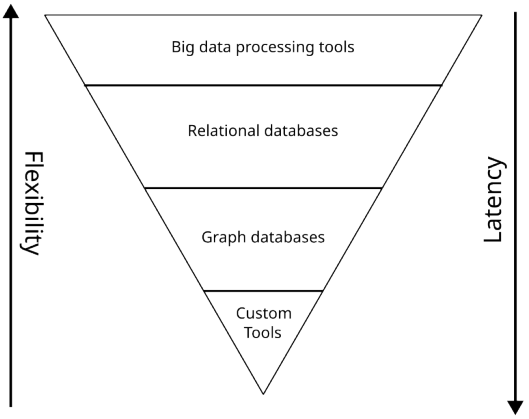


Figure 4.9: Overview of competitors

Within the landscape we have described above, we will compare our tool with Neo4j, which is a graph database, and Apache Flink, which is a big data processing framework. Although it is also possible to compare our tool with relational databases, their wide variety makes it difficult to make a definitive statement about all relational databases based on a comparison with one of them. Furthermore, if a relational database does not separate its storage and compute, then the findings of Section 4.2.1 would apply to that database. However, if it does separate its storage and compute, then it may provide the same benefits but it will still have worse performance because of less efficient joins[16].

4.2.1 Neo4j

Neo4j[19] was one of the first native graph databases and is widely considered the most popular and widely used graph database. The community version of Neo4j is free to use, and its source code is hosted on Github. However, some few features that we use for this evaluation are only available in the paid enterprise edition.

Standard Deployment

In this section, we evaluate the performance of a standard Neo4j deployment. In this deployment, a single node contains all the relevant information to answer any query related to the graph. There are various methods of performing traversal in Neo4j. If we want the highest degree of control over graph accesses, we can use the 'Low-level API'. Then, we have the 'Traversals API', which enables us to provide specifications for a traversal instead of implementing the core traversal algorithm. Lastly, we can also get the desired results by writing a Cypher query to get the desired results. All three APIs provide the same final result but use different underlying mechanisms to get those results.

Figure 4.10 compares the performance of the three APIs mentioned in the previous paragraph with S3 One-Zone. The queries shown in this figure were run on the SF-1 dataset. It is quite apparent from this figure that regardless of the API used, Neo4j performs an order of magnitude better than AWS S3. This result is expected because, unlike our service, Neo4j does not have to make any network calls. However, this approach of keeping the entire graph on a single node can not scale indefinitely. Therefore, Neo4j provides composite databases which enable us to partition the database across various Neo4j nodes.

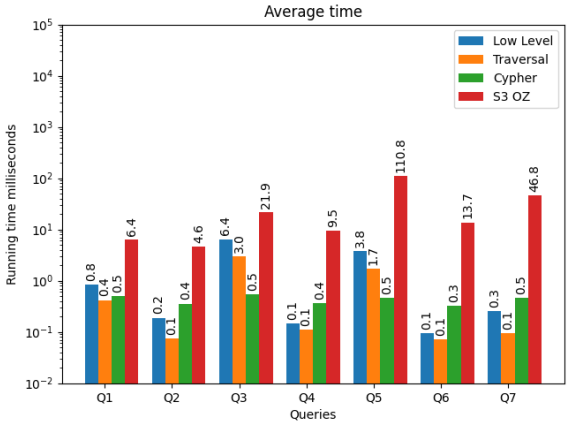


Figure 4.10: Neo4j: Standard Deployment Comparison

Before discussing composite databases, we note that Neo4j also provides a graph data science library(GDS), which provides capabilities to perform traversals. This library has in-built implementations for frequently used traversal algorithms and a Pregel API. However, to use this library, we have to create an in-memory projection of the sub-graph we want to query. Furthermore, this projection can not be mutated like an ordinary graph in Neo4j. During our experiments, we found this library to be less performant than the native Neo4j APIs for our use-case.

Composite Databases

Composite databases in Neo4j consist of multiple standard databases that can be queried from within a single logical database. The main idea behind databases is that one could partition a large graph into subgraphs such that most of the queries can either be served using a single subgraph or a scatter-gather approach. If this is achievable, the query performance would remain comparable to what we saw in the previous section. This approach may work well for some graphs, however, we explore the performance characteristics of this approach when we cannot form such isolated partitions of our graph.

When the partitioning of a graph, as described in the previous paragraph is infeasible, then every step of the traversal might need to be performed on a different shard. For example, if we want to find the work locations of a person's friends, and the Person entity is present in a separate database from the Organization entity, then we would have to wait for results from one

database before we can issue a query to the other database. In this case, we notice that the query performance degrades severely.

Figure 4.11 shows the performance of composite databases where every alternate hop is performed on a separate database, similar to the example we provided in the previous paragraph. In this figure, we see that the query performance becomes equivalent to AWS S3 general and is worse than AWS S3 One-Zone.

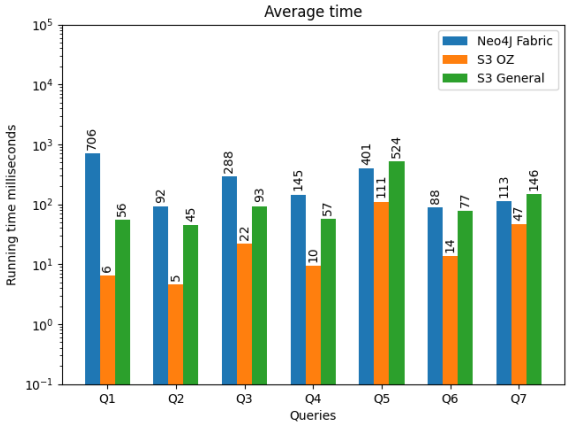


Figure 4.11: Neo4j: Composite Deployment

For some databases, this type of suboptimal partitioning may be avoidable. However, for the general case where we do not know the eventual size of various entities of the graph and the queries that users may perform in the future, this situation may not be avoidable. Furthermore, this approach has the following additional drawbacks:

1. **Distribution transparency**: is one of the primary desirable characteristics of any distributed system. This states that the user of a system should not be aware of how the data and processes are distributed within a distributed system. Neo4j's composite databases violate this principle because the onus of partitioning the data falls on the user.

2. **Special Query language**: To the best of our knowledge, we can not use the traversal or low-level API with composite databases. Furthermore, we need to specify the database that needs to be used for each subquery, which requires a rewrite of the standard Cypher queries. This may become a bigger problem for users who use object-relational mappers (ORMs), since most ORMs generate standard Cypher queries that do not work with composite databases.

3. **Shadow nodes**: If two entities, which are in different databases, have a connection, then that connection requires a shadow node. A shadow node is a placeholder for a node that is not present in the current database but in a remote database. These shadow nodes require additional maintenance overhead.

In conclusion, although Neo4j provides great performance for databases that can be stored on a single machine, this performance may degrade for larger databases. This is true for graphs that can not be partitioned into independent graphs, such that the queries can be answered using a scatter-gather approach or the queries can be answered using a single partition. If this property does not hold, the query performance can degrade by an order of magnitude or more. There are other graph databases, like tigergraph[26], which claim to have distribution transparency for large graphs. However, these claims are challenging to validate independently since, unlike Neo4j, they do not have the option for self-hosting.

4.2.2 Apache Flink

Apache flink[11] is a distributed processing engine for processing streams of both bounded and unbounded data. We chose this framework for our evaluation because, at its core, Flink is built for processing data in a distributed setting. Its programming model is designed to enable users to create pipelines that can be parallelized ad infinitum. However, since Flink is not specifically designed for dealing with graphs; there is a lack of data structures for representing graphs. Therefore, we use a graph processing library named Gelly[12].

This library enables us to run vertex-centric computations on graphs. The idea behind the vertex-centric computation is to 'think like a vertex'. This means that we specify the calculations that a vertex needs to perform based on its neighbors and, at the end of its calculations, it can send some information to its neighbors. Vertex centric computations run for multiple iterations. In each iteration, every vertex runs some calculations based on the input from its neighbors and optionally gives an output to its neighbors. These vertex- centric computations 'converge' after a given number of iterations. At this point, we observe the end state of each vertex to find the result of our algorithm.

In our case, we want to perform 1-hop to 4-hop traversals on a graph. Therefore, we start by labeling our starting node '1' and label the rest of the nodes ∞. Then, during every iteration, we check if the label of the current node is less than ∞, and send an output of $n + 1$ if the label of the current node is n. This way, for each iteration, we label the neighboring nodes of the nodes that we explored in the previous iteration. After the desired number of iterations, we can read the node identifiers of all the nodes whose label is h, where h is the number of hops.

Figure 4.12 compares our solution with Apache Flink for finding n-hop neighbors of a starting vertex in the SF-1 dataset. It is clear that Apache Flink is around three orders of magnitude slower than our solution. This is primarily because a Flink cluster needs to load the data before it can start performing any computations. Furthermore, vertex-centric computations are costly if only a handful of vertices perform meaningful computations. In this case, we have 3 million vertices, whereas there may only be a few hundred nodes that are 3-hop neighbours of a node. This means that most of the computation being performed is useless. Therefore, this result highlights the drawbacks of using vertex-centric computations for cases where only a part of the graph is explored.

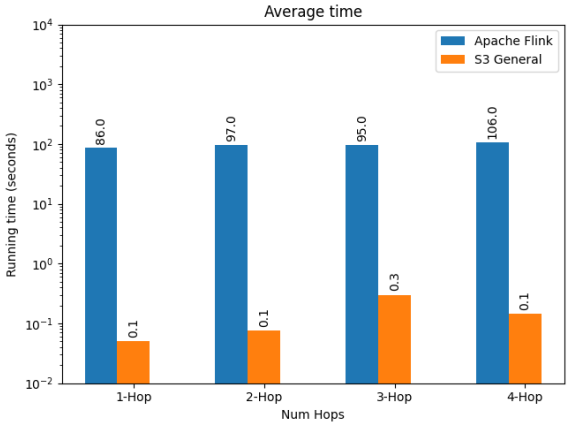


Figure 4.12: Apache Flink

In this section, we examined the performance and architectural constraints of other state-of-the-art systems that may be used to perform traversals on large graphs. With our findings, we can place our tool within the broader landscape regarding its use cases. First, while comparing it to Neo4j, we saw that Neo4j performs better than solution for graphs that can reside on a single machine. However, it gets complicated to maintain and query graphs when the graph needs to be sharded

across various computational units. In such cases, our solution may provide a more straightforward way to query these graphs. On the other hand, Flink inherently relies on the distribution of data among various computational units. Thus, Flink and other big data frameworks are useful when you need to touch the entire graph to get the results for your computation. This might include computations like Page-Rank and other global algorithms. However, the processing model of Flink is not suitable when you only require a part of the graph to answer your query. This is because of the overhead of reading data from storage and the limitations of the vertex-centric computation API. This is the use case where our solution is more efficient than Flink. We provide a more abstract discussion of the cost of using our solution in Section 5.1, however, it is apparent from this section that our solution works well for use cases where users have to deal with large graphs but only need to query a small part of these graphs to answer their queries.

Chapter 5

Discussion

5.1 Cost Model

In the previous section, we gave a subjective description of use cases where using our solution may be more effective. In this section, we provide a more formal comparison between architectures that separates compute and storage, and the architectures that do not. Let us assume that we have a graph database which is defined by two variables: its storage requirements and its computational requirements. The storage requirement can be specified in terms of the number of bytes that the underlying data needs. However, computational requirements depend on various factors like the type of query, and database architecture. Therefore, for simplification, we will consider 'graph access per second' as a proxy for the computational requirements. Graph access per second means the number of times we look up a vertex's neighbours per second. Therefore, we will use A as the graph accesses that need to be performed every second, and S as the storage that the underlying graph requires.

Now, we will consider the cost of running this database with AWS S3 as the storage layer. Let the cost of storing data on AWS S3 be p_{s3} \$/GB/second. Now, we need to take into account the cost of running an EC2 instance for performing the required computations. Let us assume that a single EC2 instance can service a_{s3} graph accesses per second, and that this instance costs p_i \$/second to run. Finally, there is a cost for sending a request to AWS S3 which we will assume to be p_r \$/request. With these variables, the cost of running the workload for one second would be:

$$(S.p_{s3}) + (\left\lceil \frac{A}{a_{s3}} \right\rceil .p_i + p_r.A) \tag{5.1}$$

In equation 5.1, the first component represents the storage cost and the rest of the equation represents the computational component. The first component $S.p_{s3}$ is the cost of storing the underlying data. In the second component, we find the number of instances that we need to spawn based on the required throughput of graph accesses, and then we multiply that with the cost of a single instance. Finally, we have the cost of sending requests to S3 which is $p_r.A$. Although it is possible to group multiple requests to a single object in a request, we consider that each graph access is done with a separate request.

Now, we will formulate the cost for running the workload using a system that keeps the entirety of its data in an SSD. Let us assume that this instance is able to sustain a load of a_{SSD} graph accesses per second and the cost of this instance is p_j \$/second. For this system, these are the only two metrics we need to consider the cost of running the system for one second, which is:

$$\left\lceil \frac{A}{a_{SSD}} \right\rceil .p_j \tag{5.2}$$

Note that equation 5.2 does not consider the storage cost S in the equation. This is because the storage cost is subsumed within p_j because we need an instance which has enough capacity to hold the entire database. Therefore, the p_j depends on the storage as follows:

$$p_j = \min_{i_{\text{cost}}} \left\{ i \in I : i_{\text{storage}} \geq S \right\}$$

In this equation, I is the set of all available instances and we get the price of the cheapest possible instance which has enough storage to store the entire graph.

We see that equation 5.1 has two separate components which scale independently when the storage or the throughput requirements change, whereas equation 5.2 only has a single component which subsumes both the throughput and the storage requirements. By looking at these equations, it would appear that using AWS S3 is more costly because we essentially have a component related to the instance cost in both equations but AWS S3 has additional costs for storage and making request. This would indeed be the case if cloud providers provided all possible permutations of compute and storage so that we would not waste any money while picking p_j. However, this is not the case; usually instances with more compute also have bigger SSDs. This may force us to pay for more compute than we actually need. We will now look at some concrete use cases and elucidate the difference between the cost for both approaches.

5.1.1 Use cases

We will now use equations 5.1 and 5.2 to talk about use cases where separating storage and compute makes more sense than the model of coupled storage and compute. Before we introduce the use cases, we have to make some educated guesses about the accesses per second that AWS S3 and SSDs can serve. To do this, we ran our solution with both backends: AWS S3 and SSDs. We found that a c7gn.xlarge instance can make and process around 5 thousand accesses per second while a instance with SSD can make and process around 50 thousand accesses per second. The exact value of these numbers may vary from one setup to another but it is generally accepted that a single instance would be able to make fewer network calls compared to SSD accesses. With these numbers, we will now consider two use cases to highlight where AWS S3 should be chosen over tools that have coupled storage and compute.

Knowledge Graphs are becoming an increasingly popular use cases for graph databases. Knowledge graph represent various knowledge concepts using a graph structure where related concepts have an edge between them. In such graphs, traversals may be used to find related concepts or to explore a common link between two concepts. Let us assume we have a knowledge graph that is used by a pharmaceutical company to model relationships between studies, populations, diseases, etc. We assume that the size of this graph is approximately 5 TB and that the usage of this database by scientists can be satisfied by 1 thousand graph accesses per second.

5.1.2 Other cost considerations

5.2 Threats to credibility of this work

5.3 Alternate system architectures

Chapter 6

Conclusion and Future Work

6.1 Future Work

Bibliography

[1] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. Graph aware caching policy for distributed graph stores. In *2015 IEEE International Conference on Cloud Engineering*, pages 6–15. IEEE, 2015. 8

[2] AWS. Aws aurora serverless. `https://aws.amazon.com/rds/aurora/serverless/`. 6

[3] AWS. Aws ec2. `https://aws.amazon.com/ec2/`. 21

[4] AWS. Aws s3. `https://aws.amazon.com/s3/`. 1, 4

[5] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys*, 56(2):1–40, 2023. 1

[6] Kyoungsoo Bok, Seunghun Yoo, Dojin Choi, Jongtae Lim, and Jaesoo Yoo. In-memory caching for enhancing subgraph accessibility. *Applied Sciences*, 10(16):5507, 2020. 8, 13

[7] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264, 2008. 1, 2, 6

[8] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016. 6

[9] Iain S Duff. Computer solution of large sparse positive definite systems (alan george and joseph w. liu). *Siam Review*, 26(2):289–291, 1984. 10

[10] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017. 7

[11] Apache Software Foundation. Apache flink. `https://flink.apache.org/`. 31

[12] Apache Software Foundation. Apache flink gelly. `https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/`. 31

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003. 4

[14] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. {FusionRAID}: Achieving consistent low latency for commodity {SSD} arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 355–370, 2021. 7

[15] Douglas J Klein. Centrality measure in graphs. *Journal of mathematical chemistry*, 47(4):1209–1223, 2010. 7

[16] Petri Kotiranta, Marko Junkkari, and Jyrki Nummenmaa. Performance of graph and relational databases in complex queries. *Applied sciences*, 12(13):6490, 2022. 28

[17] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001. 8, 12

[18] SUN Microsystems. Networking on the sun workstation, 1986. 4

[19] Neo4J. Neo4j. `https://neo4j.com`. 28

[20] Neon. Neon. `https://neon.tech/`. 1, 6, 9

[21] V Nageshwara Rao and Vipin Kumar. Parallel depth first search. part i. implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987. 17

[22] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, 2017. 1, 28

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010. 4

[24] Snowflake. Snowflake. `https://www.snowflake.com/en/`. 1, 4

[25] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020. 6

[26] Tigergraph. Tigergraph. `https://www.tigergraph.com/`. 30

[27] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017. 1, 6

[28] George Kingsley Zipf. Relative frequency as a determinant of phonetic change. *Harvard studies in classical philology*, 40:1–95, 1929. 8

Appendix A

SNB Schema

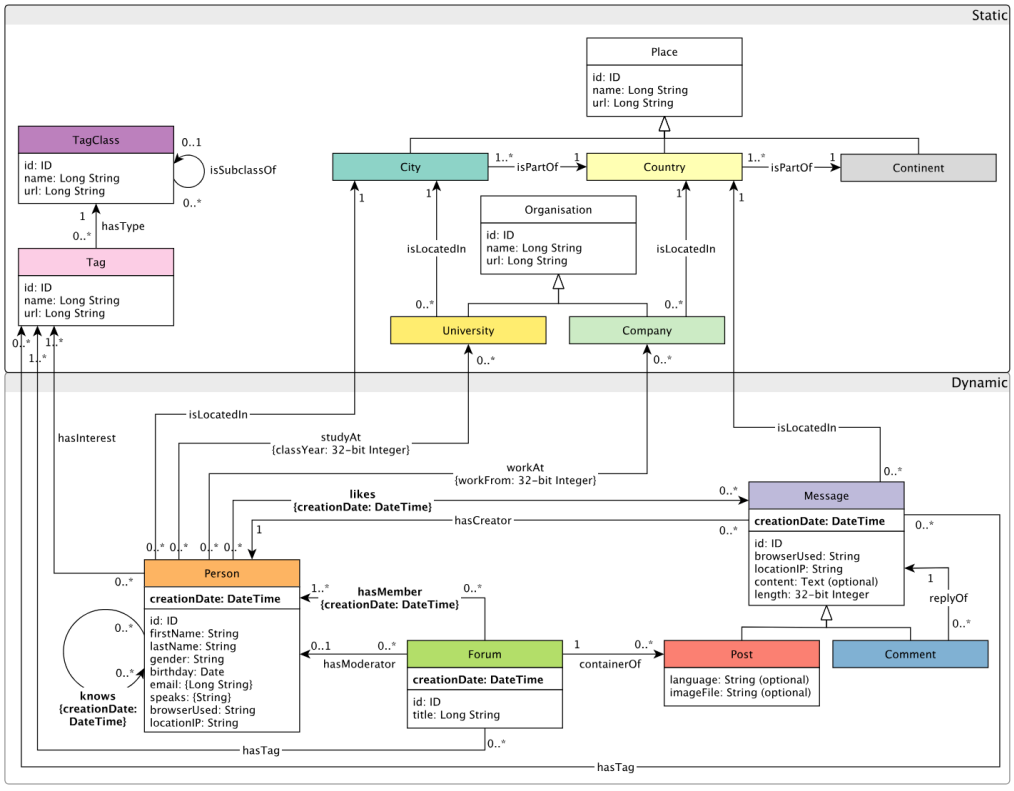


Figure A.1: LDBC SNB Schema

Appendix B

System Details

B.1 Query Evaluation

In this file (appendices/main.tex) you can add appendix chapters, just as you did in the thesis.tex file for the 'normal' chapters. You can also choose to include everything in this single file, whatever you prefer.