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 edges[17]. 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[17]. As this scale increases, decoupling storage from graph query evaluations using distributed cloud storage services like AWS S3[3] may prove to 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 to alleviate the costs related to redundancy. Therefore, it is worthwhile to explore the feasiblity 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[6] described how a relational database could be built on top of AWS S3 while still providing atomicity, isolation, and durability. Snowflake[19] was one of the first database which fully realized the idea of having totally separate storage and compute layers. This idea was adopted by other databases like AWS Aurora[21] in 2017 and more recently by Neon[15]. Despite the existance 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 which enable scaling out using sharding[4], the compute and storage are still tied together. Unlike such architectures, this paper explores an architecture where storage of raw graph 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[6] and can be extended for other systems which follow this general architecture. Therefore, in this paper, we explore how we can reduce the access latency for graph specific operations in the contex of graphs stored on distributed file systems.

In order to study the impact of our techniques on the 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 some of the most commonly used algorithms for performing graph traversals[17]. These traversal algorithms will have a bounded depth and therefore will only access a small part of the graph. On the spectrum of graph algorithms from OLTP to OLAP as described by Besta et al.[4], these traversal algorithms lie more on the OLTP side as the portion of graph explored by a single traversal would be much smaller than the size of the graph.

In order to minimize the latency, we will first propose a graph storage format which would help us 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 of 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 is different 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 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 seek 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 Structure of the thesis

In Chapter-2, we provide the necessary background for this thesis. In Section-2.1 we talk 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 that we use in this thesis. Then, in Section-2.2, we discuss the advantages of database architecutres which separate compute and storage. Finally, in Section-2.3 we discuss the caching and prefetching strategies that we employ in this thesis to lower the latency of graph traversals.

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

In Chapter-4, we present the results of the implementation of the proposed 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.

In Chapter-5, we elucidate the reasoning for various choices made throughout the thesis. In Section-5.1 we discuss other possible architectures for performing traversals on large graphs using S3 and their advantages and disadvantages over the proposed architecture. Then, in Section-5.2,

we discuss 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, in Section-5.3, we consider the threats to the credibility of this work.

Finally, in Chapter-6, we conclude the thesis and suggest 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 filesystem was developed by SUN Microsystems[14]. Their implementation is widely reffered to as the networked filesystem(NFS). Their implementation provided users with the ability to mount a filesystem 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 filesystem is considered to be the Hadoop Distributed Filesystem (HDFS)[18]. This implementation provided distribution transparency and consistency gaurantees on various operations on the files stored in HDFS. This implemnetation was inspired by the Google Filesystem[10] which was developed by Google in 2003 almost seven years before HDFS.

Although filesystems like HDFS were widely used, filesystems like AWS S3[3] provided an additional feature of being multi-tenant. In other words, with filesystems like AWS S3, a user could reap the benefits of a distributed filesystem 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 filesystems in applications like video streaming, database storage[19], 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 filesystems 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 filesystem. 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 redirected

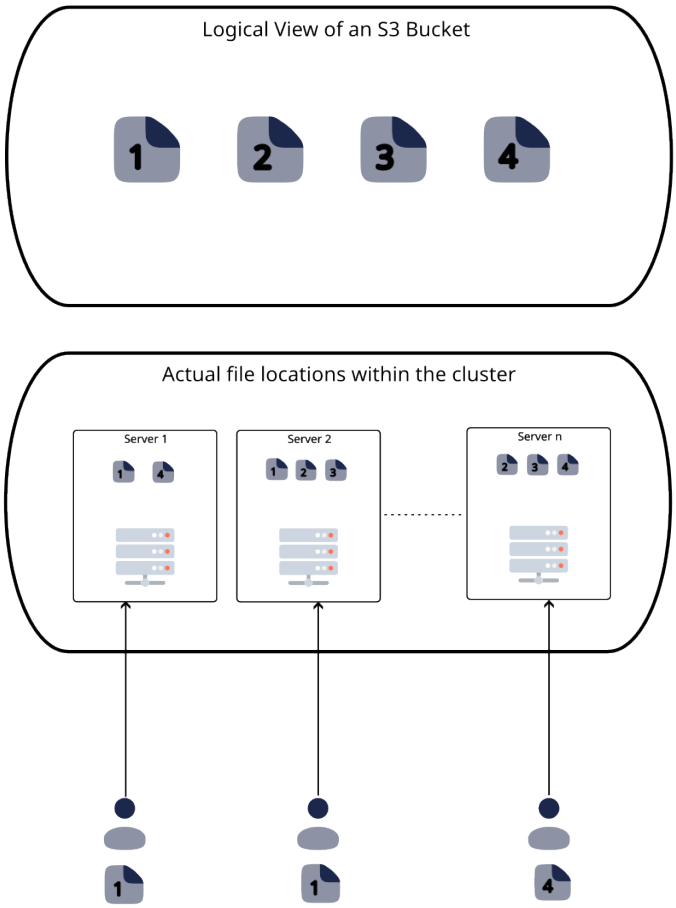

Figure 2.1: Distributed File Store

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 filesystem to argue about the applicability of various techniques.

Apart from the basic understanding of AWS S3 as it related to a distributed filesystem, 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 gaurantees 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 subdirectories**: 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 filesystem, 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[6], 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[21]. 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[7]. 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[15] 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 neccessitate separation between compute and storage and there are indeed databases like CockroachDB[20] 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 upto 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 datacenter, the latency is around 500μs which is still around 10 times higher than access latency for an SSD[11]. This advantage becomes less pronnounced 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 separte 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[12] 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[9] 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 maintainance 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[22], 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[13] 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 weightage 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 cacing 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.[5] where they added a used-cache in addition to a prefetcher cache for speeding up subgraph lookups. This idea helped them outperform CBGA for the task of subgraph 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 explaination 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 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 the users of this interface with 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 do 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[15]. 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.

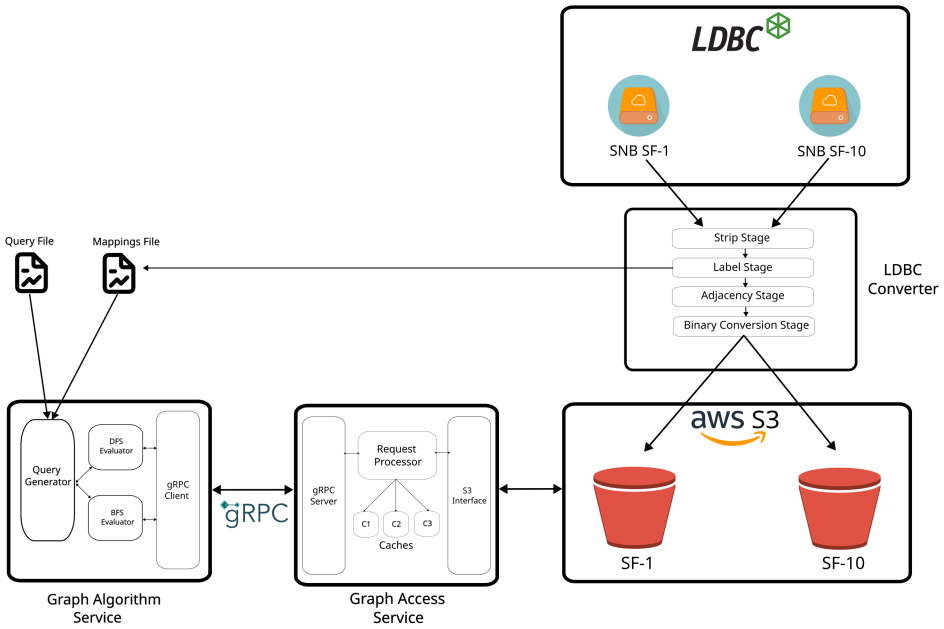

Figure 3.1: High Level System Architecture

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 next 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 which 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 then finally convert it to compressed sparse row (CSR) format[8]. With this partitioned CSR format stored in 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 S3's file names. With this index, we can get the file name for the S3 file which contains a particular node's neighbours. Additionally, this service also contains an LRU cache of some files that were fetched from S3. The size of this cache is limited because each cache entry contains an entire file with adjacency information related to 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 mentioned 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.

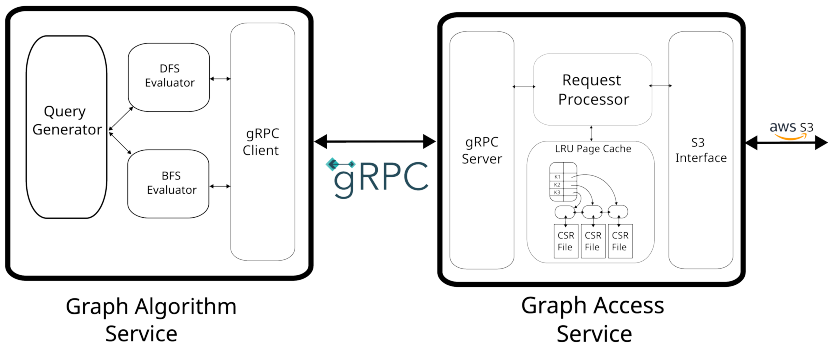

Figure 3.2: Baseline Implementation

3.3 Graph access service

In this section, we first describe a modification of the CSR format in Section-3.3.1 which would help us improve our cache performance. Then, 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 in order 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 have a higher level of granularity for better cache performance. Secondly, this approach makes it harder to utilize the fact that 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 that 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 tells us 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 desitnation. 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 the fact 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 for 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.

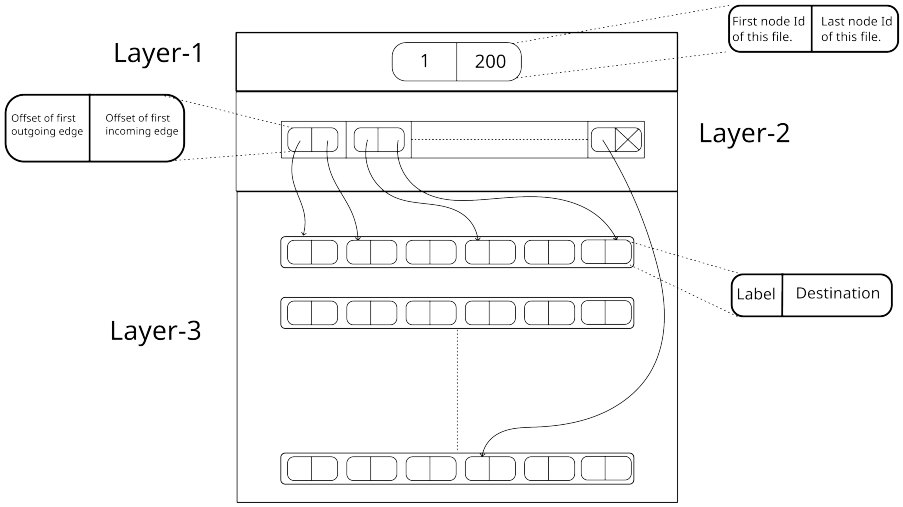

Figure 3.3: Updated CSR Format

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 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 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 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**[13]. 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 weightage 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 as a result of ageing happens at the same rate for every element. This means that only a change in frequency can change 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, a simple LRU cache implementations have an access and insertion time complexity of $\mathcal{O}(1)$, this cache provides us with a good tradeoff 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.[5]. 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 ensure that there is minimal interferance 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, 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 that we are trying to add to the queue. However, in case of DFS, the opposite is true because the node that 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 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 abilities of S3.

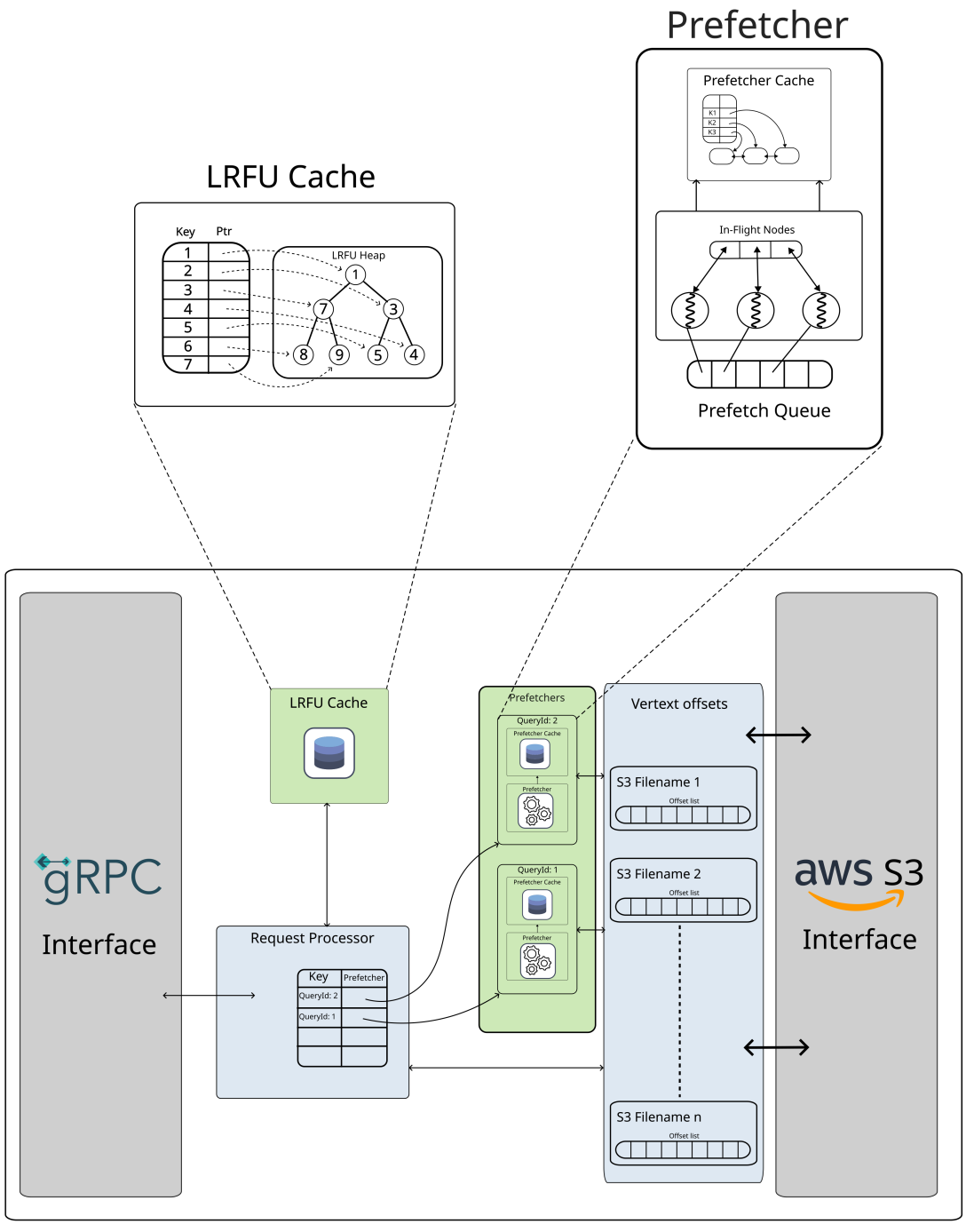

Figure 3.4: Graph Access Service

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 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 which 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 contain 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.

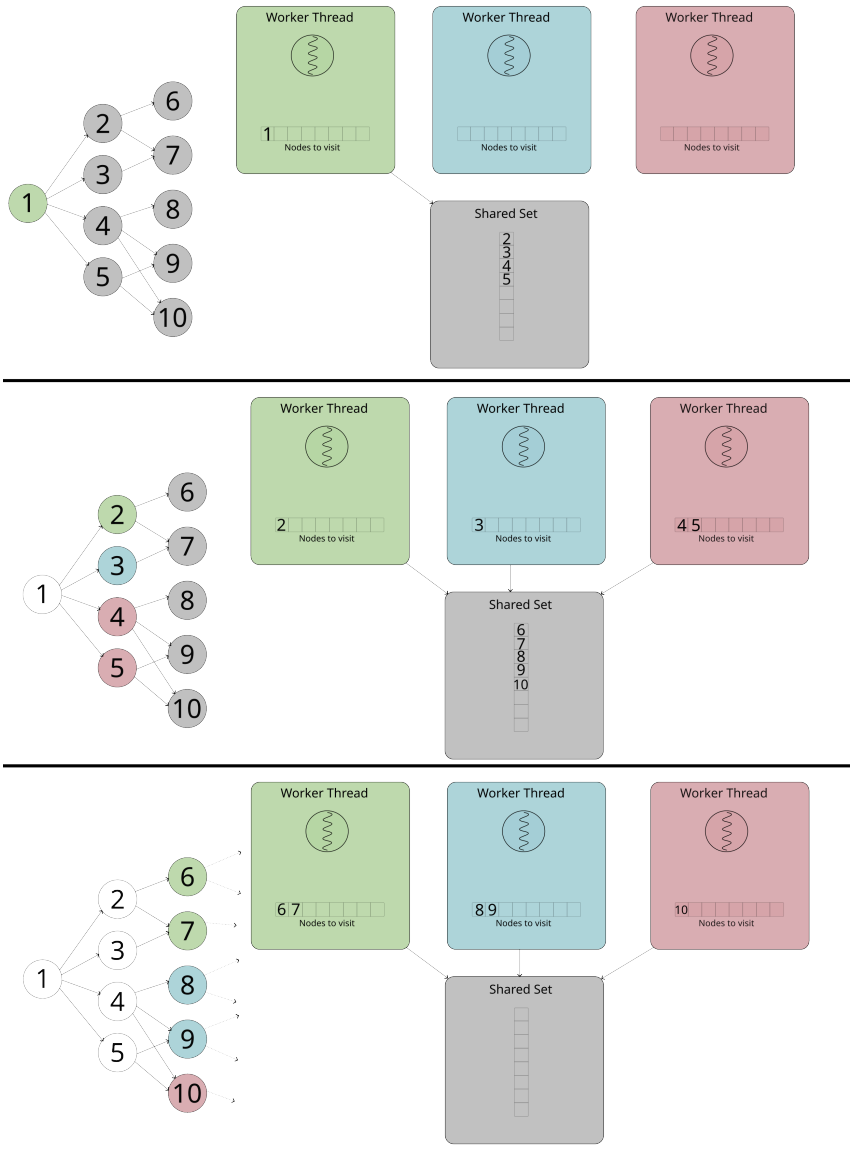

Figure 3.5: Parallel BFS

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 perfroming 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.[16] 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 neighbouring 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.

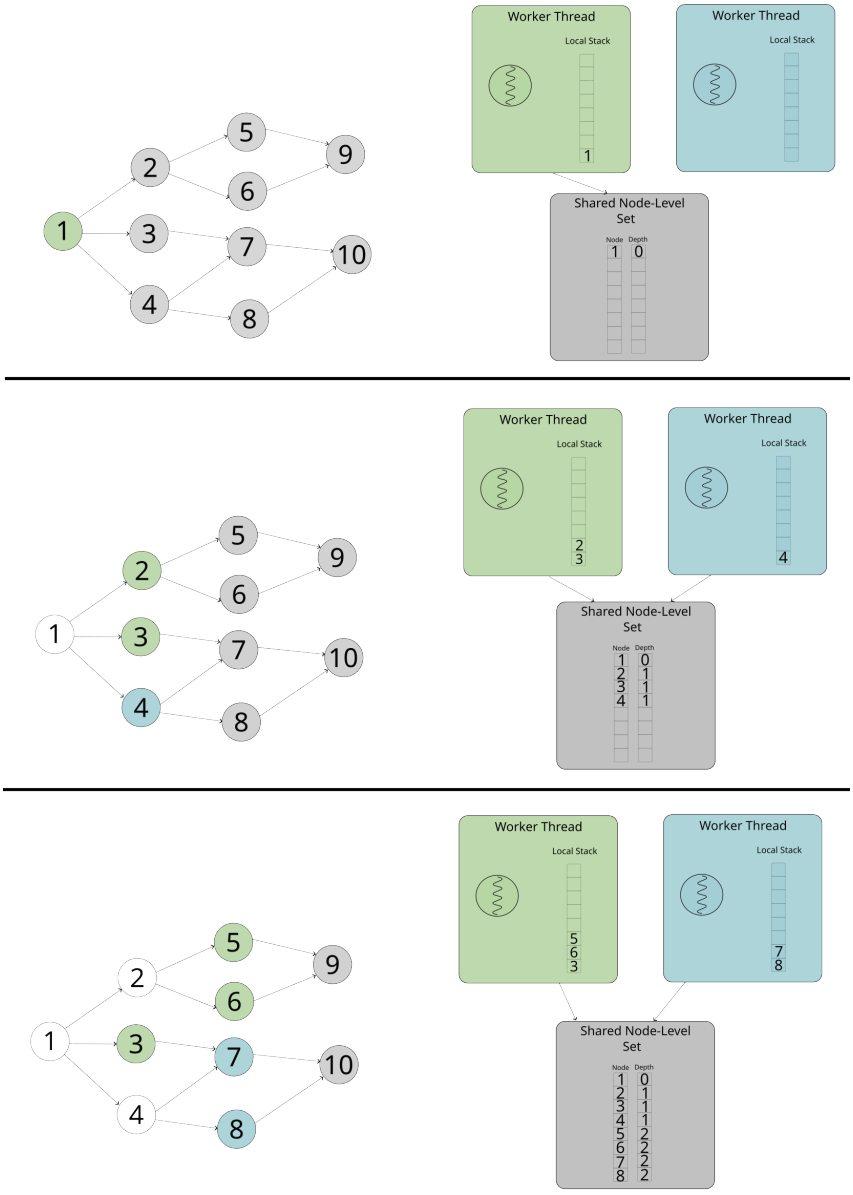

Figure 3.6: Parallel DFS

Chapter 4

Evaluation

4.1 Comparison with baseline

4.1.1 Optimizing partition sizes

4.2 Comparison with other tools

4.2.1 Neo4j

Monolithic Deployment

Distributed Deployment

4.2.2 Apache Flink

Chapter 5

Discussion

5.1 Alternate system architectures

5.2 Use cases for the system

5.3 Threats to credibility of this work

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 s3. `https://aws.amazon.com/s3/`. 1, 4

[4] 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

[5] 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

[6] 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, 6

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

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

[13] 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

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

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

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

[17] 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

[18] 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

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

[20] 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

[21] 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

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

Appendix A

System Details

A.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.