Technische Universiteit Eindhoven
University of Technology

Department of Mathematics and Computer Science
Database Group

Efficient Graph Processing 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

In a recent survey titled 'Ubiquity of large graphs'[13], the authors noted that the size of graphs being used in both academia and industry is increasing. Although this survey found that it was common for graphs to have tens of billions of edges, it also found that scalability was the primary concern of all the participants surveyed[13]. In this thesis, we evaluate one way to alleviate this concern of scalability by decoupling storage of base graph from the query evaluation.

Today, all major cloud providers have a distributed cloud storage offering, Google cloud platform offers Google cloud storage[12], Azure offers Azure Blob storage[3], and AWS offers Simple Storage Service or S3[2]. These distributed storage services are already used by Big Data Analytics tools like Snowflake[15] and RDBMSs like Neon[11] for storage of underlying data. The use of these distributed storage services has not yet been explored for storing and querying graph data.

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 almost 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.More precisely, we will focus on the performance of two commonly used graph traversal algorithms: Breadth first search (BFS) and Depth first search (DFS). We use these algorithms on LDBC datasets to provide results for multi-hop traversals.

1.1 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. After that, we describe the traversal queries which will be evaluated by our system in Section-3.2. 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-3.4, 3.5, and 3.6, 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

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[10]. 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)[14]. 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[7] which was developed by Google in 2003 almost seven years before HDFS.

Although filesystems like HDFS were widely used, filesystems like AWS S3[2] 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[15], 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 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

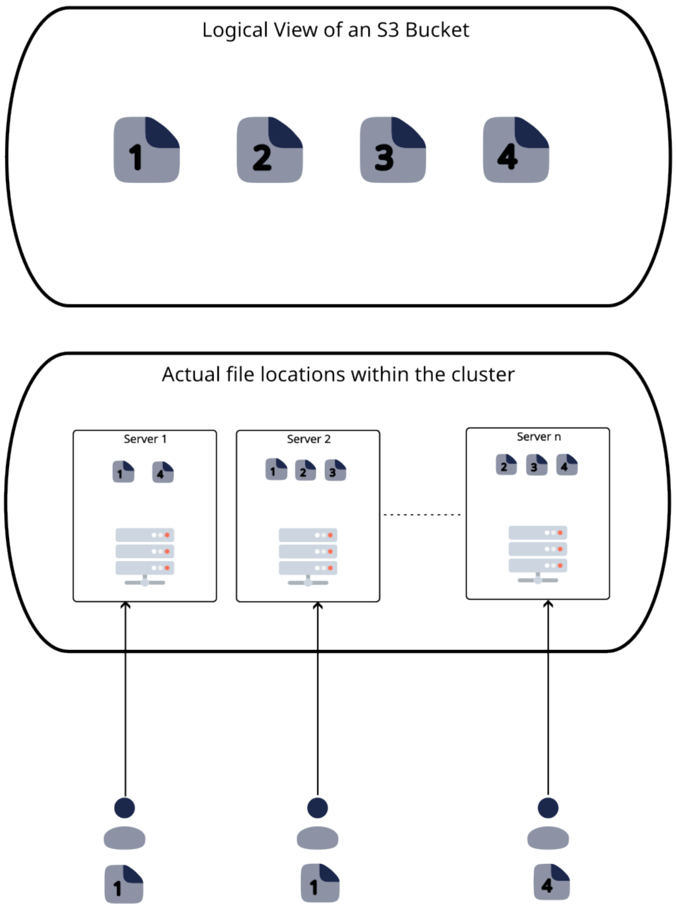
Logical View of an S3 Bucket
1
2
3
4
Actual file locations within the cluster
Server 1
Server 2
Server n

Figure 2.1: Distributed File Store

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[4], 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[17]. 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[5]. 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[11] and AWS Aurora Serverless[1] 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[16] 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[8]. 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[9] 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 for 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[6] 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.

Chapter 3

System Architecture

3.1 Component Overview

3.2 Query Evaluation

3.3 Baseline Implementation

3.4 Graph access service

3.4.1 Modified CSR structure

3.4.2 Caching

3.4.3 Prefetching

3.5 Prallelizing graph algorithm service

3.5.1 Parallel BFS implementation

3.5.2 Parallel DFS implementation

3.6 Optimizing partition sizes

Chapter 4

Evaluation

4.1 Comparison with baseline

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] AWS. Aws aurora serverless. `https://aws.amazon.com/rds/aurora/serverless/`. 5

[2] AWS. Aws s3. `https://aws.amazon.com/s3/`. 1, 3

[3] Azure. Azure blob storage. `https://azure.microsoft.com/en-us/products/storage/blobs`. 1

[4] 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. 5

[5] 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. 5

[6] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017. 6

[7] 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. 3

[8] 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. 6

[9] Douglas J Klein. Centrality measure in graphs. *Journal of mathematical chemistry*, 47(4):1209–1223, 2010. 6

[10] SUN Microsystems. Networking on the sun workstation, 1986. 3

[11] Neon. Neon. `https://neon.tech/`. 1, 5

[12] Google Cloud Platform. Google cloud storage. `https://cloud.google.com/storage`. 1

[13] 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

[14] 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. 3

[15] Snowflake. Snowflake. `https://www.snowflake.com/en/`. 1, 3

[16] 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. 5

[17] 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. 5

Appendix A

My First Appendix

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.