HAW HAMBURG

MASTER THESIS
Johannes Kotsch

Possibilities and Limitations of using Cloud Object Stores for a "One Size Fits All" Database

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Johannes Kotsch

Possibilities and Limitations of using Cloud Object Stores for a ”One Size Fits All” Database

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Olaf Zukunft
Supervisor: Prof. Dr. Ulrike Steffens

Submitted on: January 16, 2023

Johannes Kotsch

Title of Thesis

Possibilities and Limitations of using Cloud Object Stores for a ”One Size Fits All” Database

Keywords

Big Data, Object Stores, Distributed Systems, One Size Fits All Database, NoSQL, Data Warehouse, S3, Delta Lake, SAP HANA, OctopusDB, Apache Spark, Apache Kafka, YCSB, Shared Disk

Abstract

Having a single database system serving both OLAP and OLTP use cases is highly desired, yet has never been achieved. Simultaneously, object stores along with metadata storage layers emerged as a new technology, unifying Data Lake and Data Warehousing capabilities. This work examines the feasibility to also unify OLAP and OLTP requirements with this technology. This was done by implementing a novel database system called "deltadb," leveraging object stores. This system was evaluated with a performance benchmark and its results can make us conclude that it is possible to build a system that can serve both OLAP and OLTP workloads. However, the performance on that use cases still lacks those of applications that were particularly designed for a specific use case. Overall, the usage of such a system can only be recommended in very specialized situations.

Contents

List of Figures **vii**

1 Introduction **1**
1.1 Motivation . . . 2
1.1.1 Motivation for "One Size Fits All" . . . 2
1.1.2 Motivation for considering Cloud Object Stores . . . 4
1.2 Research Question . . . 5
1.3 Outline . . . 5

2 Related Work **6**
2.1 Related Work for "One Size Fits All" . . . 6
2.1.1 SAP HANA . . . 7
2.1.2 OctopusDB . . . 8
2.2 Related Work for DBMS functionalities over Object Stores . . . 9
2.2.1 DBMS on top of S3 . . . 9
2.2.2 Delta Lake . . . 11
2.3 Conclusion . . . 11

3 Distributed Databases **12**
3.1 Reasons for Distributed Databases . . . 12
3.1.1 Scale . . . 12
3.1.2 Scalability . . . 13
3.1.3 Resiliency . . . 13
3.1.4 Geographic remote Access . . . 14
3.2 System Architectures . . . 14
3.2.1 Shared Everything . . . 14
3.2.2 Shared Memory . . . 15
3.2.3 Shared Disk . . . 15
3.2.4 Shared Nothing . . . 16

3.3 Tradeoffs in distributed Systems . . . 17
3.3.1 The CAP Theorem . . . 17
3.3.2 Problems with the CAP Theorem . . . 19

4 Prototype Implementation 21
4.1 Role in answering the Research Question . . . 21
4.2 Technical Architecture . . . 21
4.2.1 The C4 Model . . . 22
4.2.2 Context Level . . . 23
4.2.3 Container Level . . . 23
4.2.4 Component Level . . . 26
4.3 Used Technologies . . . 28
4.3.1 LRU Caching . . . 28
4.3.2 Apache Kafka . . . 30
4.3.3 Spark Streaming . . . 34
4.3.4 Amazon S3 . . . 37
4.3.5 Delta Lake . . . 39

5 Benchmarking 44
5.1 Yahoo! Cloud Serving Benchmark (YCSB) . . . 45
5.1.1 Architecture . . . 45
5.1.2 Benchmark Tiers . . . 46
5.2 Conducting a performance Benchmark . . . 47
5.3 Benchmark Setup . . . 50
5.4 Benchmark Results . . . 51
5.5 Result Interpretation . . . 52

6 Evaluation 55
6.1 Comparison with related Work . . . 55
6.1.1 SAP HANA . . . 55
6.1.2 OctopusDB . . . 56
6.1.3 DBMS on top of S3 . . . 57
6.2 Overall Evaluation . . . 58

7 Conclusion 59

Bibliography **61**
Declaration of Autorship 65

List of Figures

2.1 System Architecture of the Database built on S3 [13] 10

3.1 4 different System Architectures for Distributed Databases [27, p.9] 14
3.2 Common depiction of the CAP theorem [23] 18

4.1 C4 Diagram Legend 22
4.2 System Context Diagram 23
4.3 System Container Diagram 24
4.4 System Container Diagram 26
4.5 File structure within an example Delta Table [9] 41

5.1 YCSB Workload generating Client Architecture [17] 46
5.2 YCSB binding directories 47
5.3 Simplified class diagram for YCSB DB Interface Layer 48
5.4 Boxplot of YCSB benchmark results without outliers 51
5.5 Boxplot of YCSB benchmark results with outliers 52

1 Introduction

The database landscape of enterprises nowadays appears scattered. A wide variety of systems are used for serving the same data in different use cases. OLTP, OLAP, text search and stream processing all have different technical requirements toward a database system and there hasn't been overwhelming progress in providing a system unifying them. However, having a system serving all types of cases efficiently is highly desirable, as outlined in **section 1.1**.

While there hasn't been a solution found for a "One Size Fits All" database in recent years, there have been large advances in another technology field in the space of information management. This relatively novel technology has been largely successful in unifying several database use cases. Said technology is the cloud object store.

In 2006, Amazon launched its cloud object store service S3. This marked the beginning of the success story the public cloud has since become. Cloud-based object stores were initially developed to store media files and host pages in the web. Over time, with the rise of the data lake paradigm, they also became the de facto standard solution to store analytical data on. In recent years, solutions that enable ACID transactions over cloud object stores have gained popularity [9]. This led to dropping Data Warehouses from an enterprise's tech stack and implementing their functionality directly on top of a cloud object store [10]. Cloud Object stores have already managed to unify Data Lakes and Data Warehouses, which serve different type of OLAP queries efficiently with a single system serving the super set of requests.

The goal of this thesis is to evaluate the feasibility of performing other use cases like OLTP over cloud object stores.

1.1 Motivation

The motivation for this thesis can be split into two parts. Firstly, I want to make clear why pursuing a ”One Size Fits All” system is worthwhile. Secondly, I want to elaborate on the reasons for considering cloud object stores as a promising solution to this still unresolved problem.

1.1.1 Motivation for ”One Size Fits All”

The incentive for developing a ”One Size Fits All” database architecture is enormous. This has been recognized by many market participants in the space of DBMS providers. Over the years, many attempts for such a system have been made.

Usually, nowadays enterprises maintain a wide variety of database types to serve the same data for different use cases. This causes many pain points which could be eliminated by finding a solution to the ”One Size Fits All” problem.

Firstly, all the different data sources, keeping the same data, need to be ”glued” together because they are not natively integrated with one another. In most cases this means transferring data between the sources via ETL-style data pipelines. This brings multiple downsides.

On the one hand, those data pipelines need to be implemented and, like all pieces of software, maintained. This represents an additional cost factor and additionally increases the time new data needs to be represented in another DBMS. Having a ”One Size Fits All” solution would **reduce engineering costs** as well as **reduce the time it takes to provide new data**.

The implemented data pipelines don’t transfer data instantly as well. Depending on the used data processing technology and source and target data source, processing and moving data can take multiple days. On the target systems, this increases the data staleness. Newly added data just becomes available after a certain period of time. This can disqualify a specific system for use cases, which require new data being available with low latency. Having all the data in one system serving all use cases would **reduce data staleness**.

Additionally, data pipelines, since they are manually implemented, are prone to mistakes, which happen at implementing. In a database context, this could lead to data not

available or incomplete. Another imaginable consequence is that data pipelines could falsify data or make it inconsistent. This would be even worse because it could be the basis for wrong decisions with high costs. Another consequence of unreliable data within a system is that it reduces the user's trust in the system. If a user can suspect stale, incomplete, or false data in an analytical system, they could resort to workflows that were established before decisions could be data-driven. This would dramatically hamper the added value of an analytical data system. Because the database landscape would consist of fewer pieces, having a "One Size Fits All" solution would **minimize the number of sources of error**.

Besides reducing the number of sources of error, reducing the number of components in an enterprise's database landscape brings a number of other advantages.

Most DBMS are associated with **licensing costs** (which aren't negligible) and all of them are associated with **infrastructure** and **maintenance costs**, further increasing the total cost of ownership.

In addition to that, all DBMS use different technologies to guarantee access control and **data protection**. Ideally, a user or an engineer should see all the data points they are allowed to see - not more but also not less. Having a wide variety of database systems with a wide variety of access control methods within an organization adds a huge engineering burden to synchronize all types of access control settings so that every user can have ideal access. At the same time, this engineering burden carries the risk of making mistakes. Either a user could not see all the data they are allowed to see, reducing the created value by a system. Another possibility is that the user could be made capable of accessing data they are not allowed to see, violating the privacy of customers or information control policies of an organization.

It can be said for sure however that having an organization's data in one centralized data store would reduce the total cost of ownership because it eliminates the engineering burden of synchronizing different access control systems. It would simplify the implementation of privacy-related use cases like the right to be forgotten and reduces the risk of unauthorized access to data.

1.1.2 Motivation for considering Cloud Object Stores

As further described in **chapter 2**, there have been multiple approaches to a "One Size Fits All" system. However, there are reasons for considering cloud object stores as particularly promising.

Cloud object stores have some **highly desirable traits** that are almost impossible to achieve using other technologies at reasonable costs. Amazon's object store S3 for example is designed for 99.99% availability and 99.999999999% (11 9's) durability [1]. This means that a request for a data point will be responded to in 99.99% of cases and data loss practically doesn't happen (Databricks mentions not having experienced this in their company history [8]).

Besides, S3 makes use of the known advantages of cloud computing SaaS products: Seamless and de facto infinite scalability, no fixed costs, integrated features like geo-replication, and low/no maintenance effort. Self-hosting a distributed file system like HDFS will lead to inferior characteristics and higher costs [8]. Leveraging the advantages of object stores for a DBMS is of high interest because the advantages of object stores are also relevant in the space of database systems.

A second reason for considering a solution with object stores is that they have proven themselves as highly flexible and capable of providing a solution to multiple different use cases in the past already. When Amazon S3, the first cloud object store, was publicly announced in 2006, its main use was to store and make available media files like images or audio [3]. Over time though, object stores have evolved to support many more different types of use cases. They started getting used for hosting web pages, a task with lower latency requirements than storing media files. By introducing new storage classes, object stores have also become a feasible solution for archiving - A use case that requires saving large amounts of data at a very low cost without the ability to access it in a short period of time [2]. Furthermore, object stores became the go-to solution for building data lake architectures and hence storing a company's analytical data. Object stores thereby practically replaced distributed file systems like HDFS. It didn't stop at just storing analytical data though. Since this data is stored in structured files, it was possible to implement SQL capabilities on top of them. Large chunks of analytical workloads are performed with engines like Presto, Athena, or Trino directly on data which is stored in object stores, rather than a DBMS.

Since object stores represent ”just a bunch of files” there were no mechanisms for DBMS functionalities like ACID transactions. However, this changed as in recent years metadata layers like Delta Lake [9] were introduced. This enables data warehousing capabilities directly on top of object stores and thereby replacing those systems in new architecture paradigms like lakehouse [10].

Despite all this progress, there hasn't been a successful method for enabling use cases like OLTP on top of object stores yet.

1.2 Research Question

The research question of this work is to examine to what extent object stores can be leveraged to achieve ”One Size Fits All” capabilities with a focus on OLTP. I put forward the hypothesis that with the help of object stores, better results than previous attempts can be achieved. A prototype implementation for such a system will be implemented and analyzed regarding its performance for multiple use cases. In the end, it can be concluded to what degree object stores offer new possibilities for a ”One Size Fits All” system and where there are technical limitations to achieving this goal.

1.3 Outline

Since the idea of a 'One Size Fits All” system is nothing new, I will start by recapitulating this journey and give an overview of past and related work in this field in **chapter 2**. Afterward, in **chapter 3**, I will outline some theoretical background of distributed databases by looking at different architecture types. After this, I will switch to the practical part and describe the implemented reference system, deltadb, in **chapter 4**. This will be done by giving an overview of its concept, technical architecture as well as used technologies. The implemented system will then be investigated regarding its performance with a benchmarking tool in **chapter 5**. Afterward, I will evaluate the benchmarking result in **chapter 6** and eventually conclude the whole work in **chapter 7**.

2 Related Work

Just like the motivation of this thesis, also the related work can be split up into two parts. Firstly, I will describe some related work to the ”One Size Fits All” problem. Secondly, I will examine some pieces of work which dealt with implementing DBMS functionalities over object stores.

2.1 Related Work for ”One Size Fits All”

As already described in **section 1.1.1**, there is plenty of motivation for finding a ”One Size Fits All” DBMS. The problems leading to the desire for such a system are nothing new.

In 2005 Stonebraker and Çetintemel already recapitulated a long journey towards a ”One Size Fits All” system [32]. They note that the journey to such a system sums up ”the last 25 years of commercial DBMS development” [32]. They note that there are major technical differences between Database systems for OLTP, OLAP, stream processing, and text search. They conclude that ”The 'one size fits all' theme is unlikely to continue successfully under these circumstances”[32].

A similar conclusion was drawn by French in 1995, when he wrote a publication called ” 'One Size Fits All' Database Architectures Do Not Work For DSS” [20]. Ten years earlier, he described the situation similarly to Stonebraker and Çetintemel and stated that ”all of the leading database manufacturers believe they have the right solution” [20]. He says that databases optimized for OLTP and analytical use cases follow ”Opposing Laws of Database Physics” [20]. For example, databases suited for analytical workloads make heavy use of bitmap indexes while they are essentially not very productive for OLTP systems.

In 2005, much effort was invested in engineering such a system already, yet no promising breakthrough solution was found. Despite this, there have been some remarkable approaches in the past. They leverage interesting techniques in combination with new advances in technology to bridge the gap between the "Opposing Laws of Database Physics" [20].

2.1.1 SAP HANA

The first system which had some success in unifying OLTP and OLAP use cases is considered to be SAP HANA. Nowadays, it is offered as the main product from SAP, and SAP claims that HANA would be a "single database ... for modern applications and analytics across all enterprise data" [7].

The theoretical foundations for HANA were described in 2009 by the co-founder of SAP, Hasso Plattner, in his publication "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database" [28].

Similar to French, Plattner notes that "Both systems, OLTP and OLAP, are based on the relational theory but using different technical approaches" [28]. He sees the main difference in OLTP databases using a row-oriented data model while OLAP stores work column-oriented. A database can only store its data in either way, stopping it from serving both use cases. At the same time, he notes that there have been significant advances in computing technologies over the last years that question previous insights. Namely, the primary computing technology progress has been made through "unprecedented growth of main memory and massive parallelism through blade computing and multi-core CPUs" [28]. This leads to the conclusion that a viable solution for unifying OLTP and OLAP use cases is storing data in-memory in a column-oriented manner. The in-memory storage drastically reduces access times in comparison to disk storage. A column-oriented data model is better suited for in-memory storage than a row-oriented for two reasons:

Columns can be significantly better compressed because, in praxis, many records within a column are similar. In a row store, no reference to other rows can easily be stored. Plattner gives the example of a table with a regular size of 35GB. Compressed in a column-oriented format, this can be reduced to 8GB. Since in-memory storage is still costly compared to disk storage, data compression is very welcome.

The second aspect that, according to Plattner, makes column stores suitable for in-memory storage is their inherent fitness to parallelization. Although massive multi-core processors have gained large popularity in recent years, it is still a big challenge to make full use of them. The power of all processors within a system can be used best when a specific computing task can be split up into multiple (number of processors) procedures of roughly equal size. However, finding such a split is not a trivial task, and some types of computations aren't suited at all for this. A column store, however, makes solving this problem incredibly easy since it consists of multiple parts (columns) independent of each other. A column store can be queried efficiently with multiple processors by simply assigning a column to each processor. OLTP operations like key-value lookups, which would be considered very ineffective in object stores, can now achieve acceptable runtimes since the advantages of parallel processing can be used very effectively.

Since column stores are suited for OLAP operations anyway, Plattner claims that his system design is capable of delivering a unified solution for OLTP and OLAP use cases.

2.1.2 OctopusDB

Another proposed solution for a database that serves multiple use cases was proposed by Dittrich and Jindal. They outline the architecture of their system, which is called "OctopusDB," in their paper "Towards a One Size Fits All Database Architecture." This piece of work was published in 2011 [18]. In contrast to SAP HANA, their proposal has an academic nature and follows a different approach than having an in-memory column store.

The authors explain naming their system OctopusDB by drawing an analogy from an enterprise's database architecture to the animal kingdom. They state that "rather than making the world of data management easier, we have created a zoo of systems" [18]. The octopus is an animal that can adapt its shape and color to its surroundings. Hence, it got taken as a metaphor for the capabilities of the database architecture. OctopusDB is supposed to adjust to new requirements and act on behalf of any type of database architecture.

The core idea of OctopusDB is having an event log store as the single source of truth. This event log store tracks all insert, update and remove operations. Having a log allows the system to keep track of all actions that ever happened to the database, and no information is lost. A log store itself, however, is not well suited to be directly used for OLAP or

OLTP use cases because it lacks the indexes of DBMS systems. That is the reason for OctopusDB building multiple storage views on top of the central event log. Those storage views can then be optimized for fulfilling all different types of requirements.

2.2 Related Work for DBMS functionalities over Object Stores

In order to leverage object storage for "One Size Fits All" functionalities, it is also worthwhile to take a look at previous work of implementing DBMS functionalities over object stores.

2.2.1 DBMS on top of S3

One early piece of work that deals with implementing database functionalities on top of object stores was published in 2008 by Brantner et al. [13]. Their paper "Building a Database on S3" was published during a time when AWS was just getting started and laid the foundation for the powerful cloud services we know today. The only available services within AWS were S3, SQS (messaging queue service), and EC2 (compute). SimpleDB, the first database service, was just getting started, which might be the reason why the authors saw a need to implement a database over S3.

Brantner et al. note the advantages and disadvantages of S3. The points they make are mostly still valid nowadays. They realize that S3 is unmatched in terms of infinite scalability, high availability, durability, and low storage cost. However, they also note that data transfer to and from S3 is quite expensive and has high latency, especially in comparison to traditional OLTP systems. Another concern that the authors had was consistency. Back then, S3 only guaranteed eventual consistency. This means that after any write was performed, it could take an unspecified amount of time until it was available to every client. In 2020 Amazon introduced strong Read-After-Write Consistency to S3 by default [11], eliminating the problem of dealing with eventual consistency.

The authors propose an architecture similar to other OLTP stores. Instead of a disk, they make use of S3 as a persistent storage layer. Because the latency and costs of write and read operations are pretty high, multiple records are aggregated into pages. One page is represented as one object in S3. The application consists of a record manager,

which provides an API on a record level and interacts with a page manager. The page manager provides an API on a page level and interacts with S3 to persist and sync data. The architecture is designed in a distributed way with a shared disk (S3), and can be seen in figure 2.1.

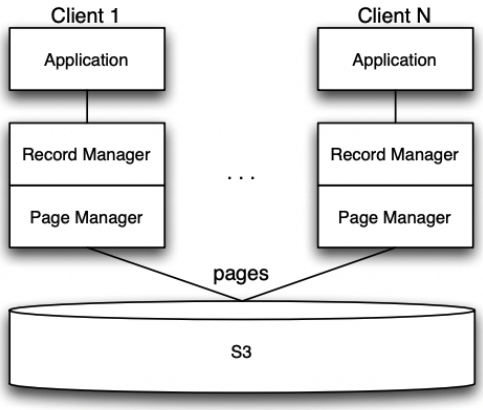

Figure 2.1: System Architecture of the Database built on S3 [13]

Another relevant implementation detail is the way updates are propagated to S3. Because of the high latency within S3, data is stored in the unit of pages that bundle multiple records. For a distributed database, that brings up a new set of problems. When multiple clients update the same record, it is acceptable that the latest write wins; however, the introduction of pages changes that. If clients update a whole page at a record update, they potentially also overwrite an update to a different record on the same page that was previously made by another client. Brantner et al. came up with a set of commit protocols to avoid this issue.

They added a set of SQS queues to the architecture. Similar to S3, SQS is a utility cloud computing service offered by AWS. It allows writing messages to a centralized data structure and retrieving them in order. It also leverages the advantages of seamless, de-facto infinite scalability as well as high availability.

Clients write log entries of their operations to those queues. Later on, the log entries from those queues can be processed asynchronously in checkpointing operations to perform all updates on the persistent S3 storage. This method avoids the previously described problem of conflicting page overwrites.

Having a centralized set of queues and checkpointing operations that happen at a specific interval help with deconflicting overwrites but at the same time increases the latency

of updates to be eventually available to all clients. This latency can be reduced by increasing the checkpointing interval, which at the same time increases the number of write operations and, therefore, costs.

2.2.2 Delta Lake

Another approach to bringing DBMS functionalities to object stores is currently being undertaken by Databricks, a company founded by the original creators of the Apache Spark framework. Their product Delta Lake was described by Armbrust et al. in 2021 [10]. Summing up, they declare a dedicated "folder" within an object store to store log data and hence enable ACID transactions. A detailed overview of the architecture can be found in **section 4.3.5**.

This allows Data Warehousing capabilities over object stores and eliminates the need for proprietary Data Warehousing applications like Redshift. OLAP queries that process large amounts of data and aren't expected to deliver sub-second latency can be served efficiently with Delta Lake. However, this system is not yet suited for OLTP use cases because they are still based on object stores. A read from Delta Lake cannot be performed faster than from the underlying S3 bucket, and each write is just as expensive as a write operation to S3.

2.3 Conclusion

Over the course of the years, there have been some exciting approaches to leveraging cloud object stores as persistent data stores for DBMS applications. The motivation for those approaches was to take advantage of the great upsides of object stores. However, those applications also have to deal with the downsides of the new cloud technology, namely high latency and transaction costs. While Delta Lake seems to have not found a solution for this (because it is not intended for OLTP use cases), Brantner et al. mitigate this issue with the use of additional data structures (SQS queues) and data checkpointing, which at the same time increases latency.

3 Distributed Databases

Deltadb is a distributed database system. A distributed database describes a database that is implemented as a distributed system. Distributed systems can be defined as "a collection of independent computers that appears to its users as a single coherent system" [33, p.18]. In this chapter, I want to investigate some core principles of distributed databases. This will lay the foundation for making conscious design decisions for deltadb.

3.1 Reasons for Distributed Databases

Originally databases, like all computer systems, were operated as single-node instances independently from other systems. Two major changes made distributed systems possible. Firstly, the price of computers began to fall drastically, and therefore their number began increasing. Secondly, computer communication networks that had a high speed and reactively high reliability were established. The combination of those two factors enabled the composition of systems consisting of independent nodes communicating over high-speed computer networks.

One of the first distributed databases, MUFFIN, was developed at Berkeley. Stonebraker discusses the design in his publication "MUFFIN: A distributed Data Base Machine" from 2007 [31]. He sums up the goals of distributed systems over traditional single-node databases. All in all, those arguments are still valid today.

3.1.1 Scale

Stonebraker wanted to achieve "higher transaction rates (by at least a factor of 10)" [31]. Until today processors, as well as networks of single machines, cannot deal with an infinite number of requests. On a single-node system, there simply is a physical limit

of throughput. Much more relevant than the physical limitation is the fact that ever-larger single-node machines lead to **diminishing returns** on further increasing their size. Prices of processors don't scale linearly with their performance. Distributing the application among a large number of cheaper workers, therefore, leads to large economic advantages for operators of large-scale systems.

3.1.2 Scalability

With MUFFIN, Stonebraker wanted to gain "the ability to expand transaction processing capability and/or data base size in a reasonable way" [31]. The aspect of adjusting computing resources to their demand nowadays is more relevant than ever. Today compute instances can be leased from cloud vendors instead of being bought and administered in on-premise data centers. Only renting needed compute instances for peak load times (like black Friday for ECommerce sites) constitutes an immense economic benefit for users. Distributed systems enable much better scalability without sacrificing availability because single instances can simply be added or removed from the compute cluster. Scaling the CPU on a single node is impossible, on the other hand.

3.1.3 Resiliency

Just like system designers nowadays, Stonebraker required "resiliency, i.e. the ability to survive without crashing and without data loss or data unavailability" [31]. Similar to increasing performance in single-node systems, increasing availability and data persistence in single-node systems becomes ever more difficult and leads to **diminishing returns**. In addition to that, there are physical limitations - If a whole data center burns down, backup power generators become useless. Distributed databases introduce a radical change within the debate for resiliency. Instead of maximizing the resiliency of a single node, distributed systems see failures of pieces of infrastructure as something normal that will inevitably happen when running a large number of clusters. Therefore, distributed systems are ideally designed in a **fault-tolerant** manner - The system stays available and doesn't lose data even if single nodes crash. Through a fault-tolerant design, distributed systems can achieve higher availability and durability than single-node systems that were optimized for resiliency at a much lower cost.

3.1.4 Geographic remote Access

A key challenge when providing a service on a global scale is access times at geographically remote locations. For now, this problem is impossible to fix on a single-node system because data can only transmit at the speed of light. By operating multiple clusters in different geographical locations, distributed systems can mitigate the problem of excessive access times.

3.2 System Architectures

Looking at databases from the perspective of distributed systems, they can be categorized into four groups. Those groups are defined by the set of resources that are shared or directly accessible to a CPU of a worker. Since there is memory and storage as a resource layer, we can determine four different system architecture groups, as seen in **figure 3.1**. A system can either share all its resources, memory, and disk, only disk, or nothing. Each architecture type comes with different advantages and is suited for other use cases.

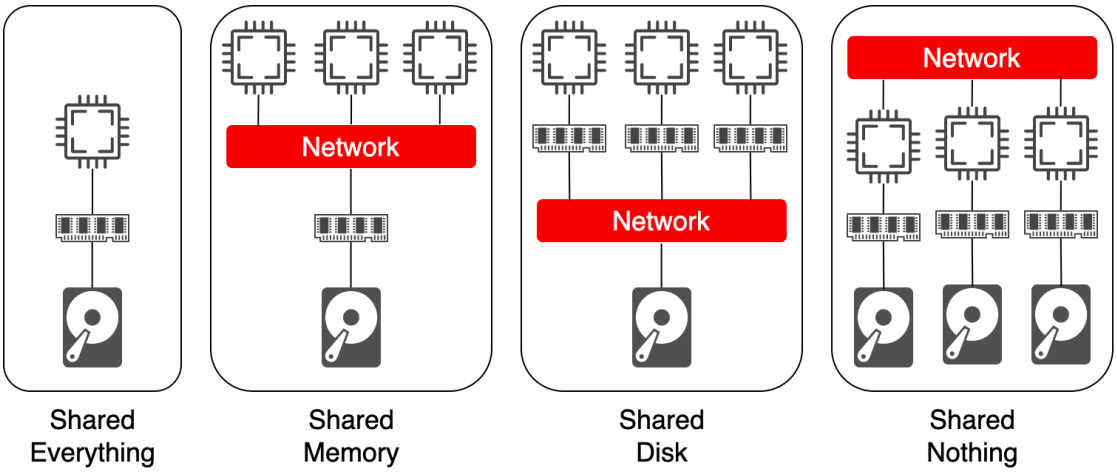

Figure 3.1: 4 different System Architectures for Distributed Databases [27, p.9]

3.2.1 Shared Everything

A system that doesn't make use of distribution and has all its resources in a single node can also be classified as shared everything. All the resources (CPU, Memory, Disk) are

available to all nodes. Because this is only possible in a single node, there is no network involved.

3.2.2 Shared Memory

In the first step towards a distributed system, the memory becomes a shared resource among multiple CPUs. All the processors have a unified view of the address space and auxiliary data structures used within a database application. In such a system, the access time to memory is mainly defined by network speed. Via network, the access to shared memory isn't significantly faster than to shared disk. In database-type applications where the execution time is mainly defined by access times to data stores, this is critical. That is the reason why shared memory systems don't play a prominent role in the DBMS ecosystem. Instead, shared memory systems are primarily used in the use case of high-performance computing. Here the majority of execution time is used for computing, and multiple processors need a unified view of auxiliary data structures for their computations.

3.2.3 Shared Disk

Shared disk system architectures go one level further and leave each worker with its own independent CPU as well as memory. As the name says, the disk part of the system is shared among all workers. In a shared disk environment, workers can have data that is likely to be accessed in their own memory, similar to a cache. Whenever data not contained within the cache is needed, pages can be loaded from the shared disk component.

A major advantage of the shared disk architecture is that the storage layer and compute layer can be scaled independently of each other. For example, a system could, over time, have an increased demand for persistent memory while the demand for computing power stays flat. In a shared disk paradigm, the operators of a system could simply extend the disk layer without affecting all deployed nodes. When the demand for computing grows, the number of nodes could be increased without enhancing storage capabilities. Scaling independently allows the systems to be operated in a much more cost-efficient manner.

Nodes in a shared disk system can be considered stateless. The persistent state of the application is entirely stored on the shared disk layer, and therefore nodes can be added

or removed from the application easily. Adding a node does not require a change in the other nodes since the data layout stays the same. The process of adding a node, therefore, is straightforward in a shared disk system. Being stateless is also advantageous when nodes need to be removed either because of scaling or crashes. Other nodes are not dependent on the state of the removed node and can continue operating without any changes. Because of this, shared disk systems can also be considered effective in terms of fault tolerance and resilience.

Having stateless nodes is also beneficial in the environment of cloud computing. Scaling up and down is much more common here than in on-premise data centers, and a stateless application is even a prerequisite for building applications in the serverless paradigm.

The main challenge within shared disk systems is that each node contains a local cache in memory. The state of a data point will regularly be updated in another node, becoming invalid. The shared disk system must find a strategy to mitigate data becoming outdated in its nodes with communication over the network.

Because systems can be scaled up and down easily and nodes are stateless, to no surprise, many cloud-native databases are implemented as shared disk systems. Furthermore, data warehouse systems that support OLAP use cases are shared disk systems. One reason for this is that compute requirements can vary drastically depending on the analytical query being performed. Concrete examples of shared disk systems include Amazon Redshift, Snowflake, or the query engine Presto [27, p .11].

3.2.4 Shared Nothing

At the end of the spectrum of distributed database architectures lies shared nothing. As the name suggests, in shared nothing systems, each node has its own CPU, memory, and storage independent of the other nodes. There is no shared resource, and nodes only communicate with other nodes via the network.

When done correctly, shared-nothing systems can be the most efficient systems of all. Because there is no centralized storage element, data must be distributed among the different nodes. If a system manages to do this efficiently and a request can be served with data stored within a node, then there is a minimal amount of communication over networks required. Since this communication often is the bottleneck, this is a significant advantage. A shared disk system, for example, must continually update a global state

and communicate the new data point to the shared disk component. In contrast to that, a shared-nothing system can ideally perform the update solely within the node and thereby be much more efficient.

However, leveraging the advantages of shared-nothing is not trivial. Shared nothing systems face considerable challenges in scaling. When a node is removed from a shared-nothing system, for example, the data stored on it must be distributed on the other nodes since the nodes are not stateless like in shared-disk. The same problems arise when adding a node. Doing this in a transactionally safe manner poses additional challenges.

Because shared-nothing systems can be made the most efficient, especially databases that support OLTP use cases are implemented with this architecture style. Notably, the databases that gained popularity in the NoSQL hype of the 2010's use shared nothing. Prominent examples include Apache Cassandra, Redis, and MongoDB [27, p.13].

3.3 Tradeoffs in distributed Systems

Whenever designing but also choosing a database system, the question of tradeoffs is tremendously important. A tradeoff describes a situation where multiple desired qualities stand in conflict with another, and increasing one quality will automatically decrease the other quality. There is a multitude of reasons for the existence of tradeoffs. It is important to formalize them so that a conscious decision about the qualities of a deltadb can be made.

Deltadb was designed as a distributed system to enable horizontal scaling (scaling out). The reason for this is that systems that require vertical scaling quickly reach their physical limits, and deltadb is supposed to support near-infinite scale. While physical reasons exist in the space of engineering, distributed systems have their own "laws of nature."

3.3.1 The CAP Theorem

The by far most prominent framework for decision-making for distributed database systems is the so-called CAP theorem. It is taught in undergrad courses for databases as well as in preparation courses for systems design interviews at hyperscaler companies like Amazon [23].

The CAP theorem was first formulated by Eric Brewer in the year 2000 in an invited talk. He noted that distributed web services have a major flaw and cannot provide the three expected and desired guarantees of consistency, availability, and partition tolerance at the same time. This conjecture was documented and proven by Gilbert and Lynch [22].

The three meant guarantees can be defined as follows:

- **Consistency:** Gilbert and Lynch define consistency within the CAP theorem as linearizability. This means that all operations performed on the distributed system must be put into a total order. Hence, the system must behave similarly to all operations being performed on a single instance.
- **Availability:** In general, availability means that all requests return a result in an acceptable time range. Acceptability is hard to be defined, though. Gilbert and Lynch, therefore, define availability as "any algorithm used by the service must eventually terminate" [22]. While this definition alone allows for unbounded computation time, together with the guarantee of partition tolerance, it becomes a strong definition.
- **Partition Tolerance:** In contrast to intuition, partition tolerance doesn't measure how well a system can be partitioned, i.e., be distributed. Partition tolerance describes the ability of a system to handle network partitions. "A network partition is a particular kind of communication fault that splits the network into subsets of nodes such that nodes in one subset cannot communicate with nodes in another" [25].

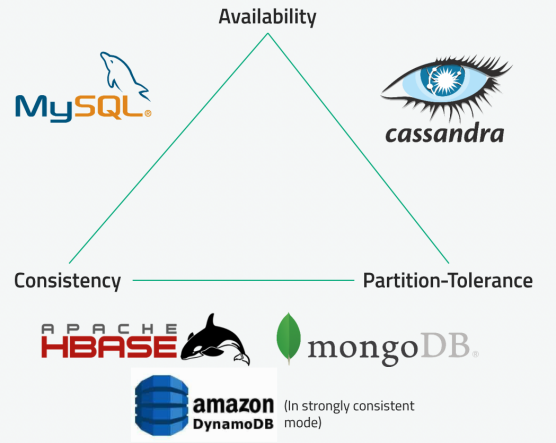

Figure 3.2: Common depiction of the CAP theorem [23]

The CAP theorem has gained large popularity and, since then, has been presented as a ”two out of three” framework for choosing database systems. A visualization of this can be seen in **figure 3.2**. The general wisdom that is often drawn from the CAP theorem is that consistency needs to be sacrificed to gain availability. Large-scale systems with high availability (like cloud object stores), therefore, only support eventual consistency.

3.3.2 Problems with the CAP Theorem

Despite the high popularity of the CAP theorem, its use as a ”two out of three” decision-making framework is highly flawed. Martin Kleppmann, the author of the standard work for designing big data applications [24], recognized this and published ”A Critique of the CAP Theorem” [25] in 2015. He criticizes the theorem, its proof as well as its use on multiple levels.

Kleppmann sees the main problem with Gilbert and Lynch's proof in their definition of availability. They define availability as terminative in an unspecified amount of time. For obvious reasons, this definition does not conform to reality. A database system that requires 100 years to process a request would be considered available according to Gilbert and Lynch but wouldn't be classified as such by any user. Although ”Gilbert and Lynch's formalization can be proved correct” [25], Kleppmann concludes that ”although the formal model may be true, it is not useful” [25].

Another main flaw lies in the formulation of the CAP theorem as a ”two out of three” selection. This kind of selection logically requires three binary options to choose from. While Gilbert and Lynch manage to define consistency availability and partition tolerance as binary characteristics, this is also not applicable to reality. In distributed systems, consistency, as well as availability are both characteristics that are continuous rather than binary. For availability, this is obvious since availability SLAs are defined in numeric values (99.99% for Amazon S3). But also, for consistency, there are multiple gradations. While Gilbert and Lynch define the strictest consistency form, linearizability, as consistent, there are also weaker consistency models such as sequential consistency, causal consistency, or pipelined RAM consistency. Because of those graduations, the usage of the CAP theorem as a ”two out of three” selection is logically flawed.

The CAP theorem is often used to argue that systems with high availability cannot fulfill strong consistency requirements. Therefore, they must settle for eventual consistency,

which is considered to be the weakest form of consistency still being useful to an application. Kleppmann shows that drawing this conclusion from the CAP theorem is logically wrong. He does this by further looking into the type of links within a distributed system. In many cases, we can assume *fair-loss links*. This describes a type of link where messages can be lost but will eventually arrive at the recipient, e.g., through retry mechanisms. Kleppmann argues that fair-loss links are a realistic depiction of systems used in reality. Having a link not fulfill the fair-loss requirements would imply partitions of infinite duration within the system (no retry would ever be successful). Partitions with an infinite duration, on the other hand, would contradict eventual consistency since the partition remains in place indefinitely and doesn't disappear eventually. Having fair-loss links, on the other hand, wouldn't contradict either of Gilbert and Lynch's definitions of availability and consistency (because availability is only defined as terminative). Gilbert and Lynch's proof require partitions of indefinite length. With those partitions, however, eventual consistency cannot be guaranteed because they don't eventually resolve. Therefore the CAP theorem is no suitable framework to argue for the need for eventual consistency in order to achieve high availability.

The problems with the use CAP theorem were also described by its original author, Eric Brewer, in 2012 [14]. Brewer reflects on the use of the CAP theorem and assesses that the CAP theorem was misused, e.g., from the NoSQL movement to argue against traditional databases. Brewer clarifies that "CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare" [14]. Consistency and availability are only mutually exclusive in the environment of a partitioned system. "The '2 of 3' view is misleading on several fronts". In an unpartitioned system, it is not necessary to sacrifice consistency to gain availability" [14]. In fact, partitions are rare, and "there is an incredible range of flexibility for handling partitions and recovering from them" [14]. Furthermore, we must consider that partitions are only a small part of possible failures within a distributed system. Other failures, like individual node failures, are more common and meaningful. The CAP theorem, therefore, only describes a framework for design decisions in an uncommon edge case.

From this, I can conclude that the CAP theorem is unsuited to argue about tradeoffs and design decisions within a distributed database system. It can only be applied to design a narrow part of the behavior in the rare case of network partitions. In my opinion, the CAP theorem has gained popularity for the wrong reasons and was wrongfully used in a simplified way to underline the argumentation for higher availability and lower consistency models.

4 Prototype Implementation

4.1 Role in answering the Research Question

A prototype implementation was chosen to answer the research question in a qualitative and quantitative matter.

Quantitative it was needed because there is no ”One Size Fits All” DBMS implementation on top of cloud object stores which is known to me. In order to evaluate the feasibility, however, it is necessary to measure the performance in different scenarios through a performance benchmark.

Qualitative it was needed to prove that the theoretical concept does not oversee any fact that would make an implementation unfeasible. In addition to that, the concrete implementation can be seen as a basis for future discussions and future work regarding this topic.

4.2 Technical Architecture

In this section, I want to depict the concrete implementation of deltadb with a technical documentation of its architecture. I use the C4 model to visualize the software architecture, which breaks down a software system into four levels. Firstly, I will give some foundational information about the C4 model and then provide three of the four C4 levels.

4.2.1 The C4 Model

The C4 model was introduced by Simon Brown in 2018 [15]. It divides a software system into four hierarchy levels to focus on the composition of the system from different viewpoints. The four hierarchy levels in C4 are:

- **Context:** The context level takes a birds-eye view of the documented software system and sees it as a single unit. The context level focuses on capturing the connections to other software systems or persons using the documented software system.
- **Container:** The container level diagram starts splitting up the documented software system into containers. A single container can is a relatively high-level component like a microservice, datastore, or application.
- **Component:** On the component level, C4 zooms into a specific container and depicts its composition of components. Thereby a single component can be something like a group of code, e.g., a package.
- **Code:** The code level is the lowest level represented in the C4 model. As the name suggests, it zooms into the code level and visualizes the relationships of entities within the code (like classes and interfaces). Since this level tends to be tedious to make and is also prone to the most changes, it is common practice not to manually create the Code level diagram. Instead, modern IDEs can generate class diagrams on demand.

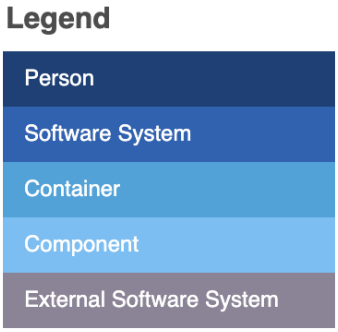

Figure 4.1: C4 Diagram Legend

4.2.2 Context Level

The context level diagram can be seen in **figure 4.2**. It can be seen that two user groups, as well as one software system, interact with deltadb.

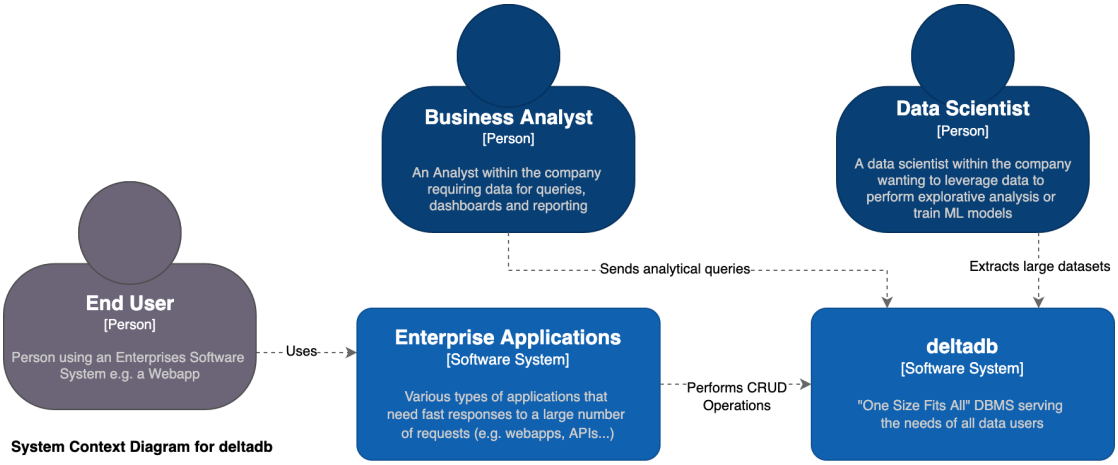

Figure 4.2: System Context Diagram

Within the enterprise, business analysts, as well as data scientists, want to have access to data. Business analysts would mainly perform OLAP queries against deltadb to support their business hypothesis. Data scientists will mainly extract data directly in a file format to get large datasets for their exploratory analyses and model training.

The software system using deltadb is, in fact, a generalization of multiple possible software systems which could use deltadb as an application database. Those systems could include web apps or APIs which use deltadb with a large number of transactions and expect a quick response time. The end user indirectly uses deltadb by using enterprise applications. They mainly expect short response times from the used applications.

Overall the context level diagram visualizes the diversity of requirements that different stakeholders have for a ”One Size Fits All” database system.

4.2.3 Container Level

The container level diagram is depicted in **figure 4.3**. It shows that the deltadb software system itself is divided into six containers which can be put into a logical order considering the flow of data coming in from enterprise applications.

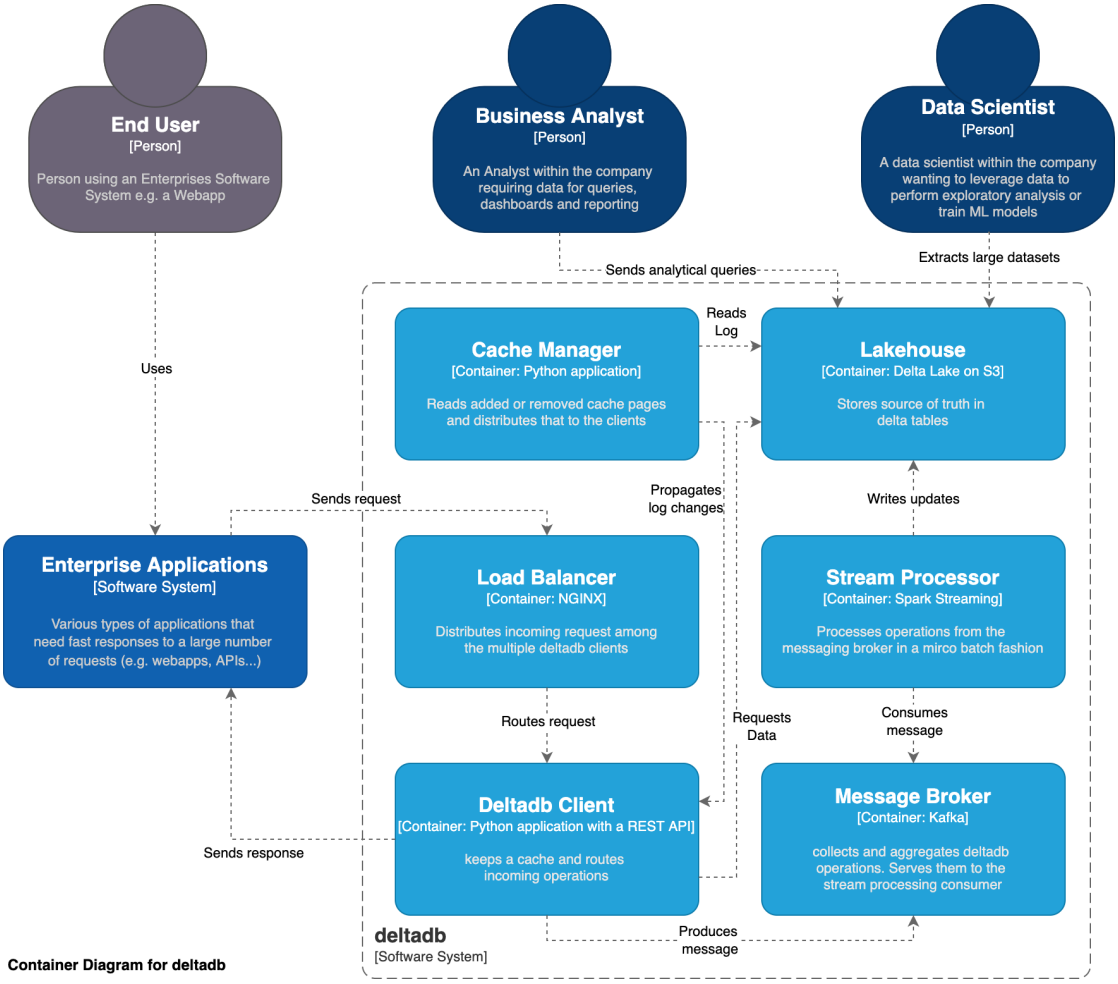

Figure 4.3: System Container Diagram

The first container which receives a request is the **load balancer**. Deltadb was designed to scale horizontally with multiple nodes of its client. A load balancer is used to offer the functionalities of the different clients under a single endpoint. The incoming request can be distributed in a way that each node gets an equal amount of requests, but there are also other strategies. The overall goal of the load balancer is that the node can overall perform optimally without being either overwhelmed by traffic or sitting idle. NGINX was chosen as a load balancer since it is a widely used, battle-tested, lightweight solution, and no advanced load-balancing capabilities were needed.

The load balancer routes the incoming requests to the **deltadb client**. The client itself is a web application written with Python and FastAPI and provides a REST interface for

CRUD operations. The application itself keeps a cache of database entries in its memory and tries to serve read requests with priority from that cache. If a read request cannot be served from the cache or the cache becomes invalid, the client directly requests the required data points from the lakehouse. All non-read operations must eventually be propagated to the lakehouse as the single source of truth for deltadb. The deltadb client does this in an asynchronous manner by publishing a message for each operation to the message broker.

The Kafka **message broker** itself collects the messages from all deltadb clients, aggregates them into a single topic, and serves them for the stream processor consumer with low latency.

The messages provided by the message broker are consumed by the **stream processor**. Spark streaming was used as a technology for stream processing. By default, Spark streaming supports micro-batch processing instead of ”real” stream processing. This proves to be no drawback but rather advantageous since the lakehouse can only support a far lower amount of transactions than deltadb has to support operations. Within each batch, the operations ,therefore, have to be aggregated, deduplicated, and written into the lakehouse as a single transaction.

The eventual goal of the data stored within deltadb is the **lakehouse**. Each table within deltadb is stored as a delta table in the lakehouse. Technically the delta table is a ”folder” within S3, and the items are stored in parquet files. Delta lake is used as a metadata layer on top of those files to enable transactions. Business analysts and data scientists can directly perform their use cases on the lakehouse, and the deltadb client can query items that are not in its cache from the lakehouse.

The goal is to have an efficient cache within the client that has new entries available and outdated entries invalidated as soon as possible. Since the lakehouse constantly adds and removes files from the current delta table, this update must also be reflected in the client's cache. If a client has a specific page in their cache and the lakehouse removes it from the delta table, this page needs to be invalidated. On the other hand, newly added pages to the lakehouse should also be added to the cache. The task of communicating those changes is performed by the **cache manager**. It constantly queries the delta log if a new version of the table is available. If so, it propagates the information of added and removed pages to multiple deltadb clients.

4.2.4 Component Level

The component level diagram is depicted in **figure 4.4**. The diagram further breaks down the deltadb client and lakehouse containers into three components each.

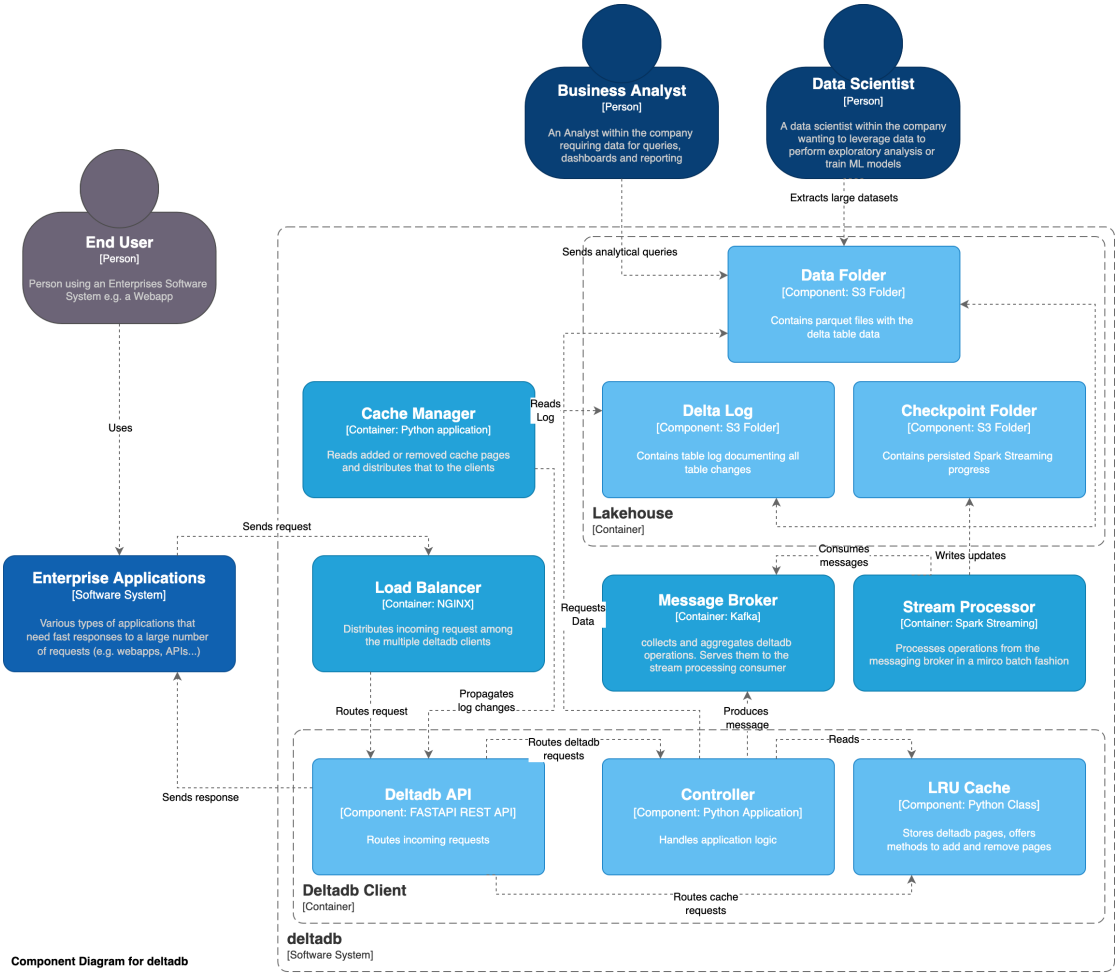

Figure 4.4: System Container Diagram

The deltadb client is split up into its API, a controller, and an LRU cache.

The **deltadb API** uses the REST paradigm and offers two sets of methods as an entry point to deltadb. The first set consists of the methods that are exposed to the users of deltadb. They allow getting, putting, and deleting items into a deltadb table. The second set of methods is destined for internal use - concretely for the cache manager. Two endpoints are provided that enable putting and deleting specific pages from the LRU

cache. Specifically, the REST API was implemented with the relatively new Python framework *FastAPI*. It was chosen because it is more lightweight and performant compared to extensive web frameworks like Django. Resource efficiency and performance are more important for deltadb than advanced web app functionalities.

The **controller** provides the application logic within an instance of the deltadb client. For example, it is responsible for checking whether an item is available in the cache. If not, it must get it directly from the lakehouse and transfer it to the cache. Furthermore, the controller produces the messages for the message broker.

The third and last component in the client is the **LRU cache**. It stores parquet files from the data folder in memory to enable fast access from the controller. Because memory is a finite resource, it evicts pages with the LRU strategy as soon as all memory is used. Since items are not accessed from the controller by the parquet filename in which they are stored, the LRU cache must also implement an index that maps item ids to parquet file names. To the outside, the LRU cache exposes methods to add and remove cache pages since Delta Lake continuously adds and invalidates specific files.

The lakehouse itself can also be split up into three components: the data folder, delta log, and checkpoint folder.

The **data folder** can be seen as the lakehouse centerpiece. For a specific table, all content is stored in the data folder. This content consisting of multiple rows and columns, is partitioned into multiple files of the Apache Parquet format. A table is prone to continuous change, meaning that rows get updated, added, and removed. Adding rows can be performed quite simply by only using a single folder - files with the new rows can simply be added. Updates and deletes require that old rows, stored in parquet files, need to be changed or removed. This is done by merely invalidating the old files and writing new content to a new file. The old file does not get deleted from the data folder though, because this would disturb operations on old versions of data.

In order to make specific versions of a table available, the **delta log** is needed. It is a folder consisting of a sequence of JSON files that have numerically incrementing file names. Each JSON file documents the changes to a table that lead to the creation of a new version of a table. Those operations include the addition and removal of files in the data folder. A client can read files from the log up to a specific version number to construct a version of the table. Because this could be a very extensive process, there are checkpoint files created for every 10th version of a table. The checkpoint files sum

up the computation of a table version so that it does not have to be repeated by the client.

The third and last component in the lakehouse is the **checkpoint folder**. It has nothing to do with the table itself but stores information for the stream processor. It could be located at any other location that offers persistent storage but using the target destination of a streaming process is more practical than creating a separate container. Because a streaming application is supposed to be capable of recovering from failures, it must persist its state in some way. Without having a persisted state, a restarted streaming application would have to reprocess every source data entry available in the streaming source. The checkpoint folder allows the streaming application to set a bookmark and start over where it left off at a restart of the application.

4.3 Used Technologies

In this section, I want to elaborate on the technologies used for my prototype implementation. For each technology, this will be done by firstly giving some foundational knowledge about the technology, secondly discussing the role the technology takes in the prototype implementation, and lastly arguing what led to the choice of the respective technology.

4.3.1 LRU Caching

Caching is used in systems where we have two different groups (levels) of memory available, and ”the first level is small and has short access time, while the second level is very large but slow” [16]. In such a system, the smaller memory level with the shorter access time is referred to as the cache. The deltadb client clearly deals with a caching situation since it has access to its RAM memory which has a fast access time but has a limited size, and on the other hand, to the lakehouse on S3, which has long access times but a nearly unlimited size.

Since the size of the cache is limited, the essential element defining the cache is the strategy by which elements get evicted from it. Formally the problem for finding the optimal eviction strategy via an algorithm is defined as the **paging problem**. The paging problem denotes the elements as pages and assigns a size of k to the cache. ”A

request to a page p can result either in a hit, when p is in the cache, or a fault, when it is not. In response to a fault we need to bring p into the cache. If the cache is already full, some other page q currently in the cache needs to be evicted. The goal of a paging algorithm is to choose the evicted page for each fault so as to minimize the cost, defined as the number of faults." [16]

A paging strategy that has proven to be optimal by Belady [12] is *longest forward distance* (LFD). It preferably ejects pages never used again from the cache. In case there is no such element, the page that is used the furthest in the future gets evicted.

LDF is defined as an offline algorithm. This means that the whole input data is required from the beginning for the algorithm to make its decisions. In a real-world scenario where a DBMS is used, it is not the case that the future sequence of requested elements is known. Situations like these require online algorithms - algorithms that make decisions without knowing their future input. The key metric to describe the effectiveness of an online algorithm is called the competitive ratio. It describes the maximum ratio between the performance of an online algorithm compared to the performance of an optimal offline algorithm. The competitive ratio can be determined by analyzing the worst-case performance of the online algorithm. If the maximum ratio between the performance of an online algorithm compared to an optimal offline algorithm is n, the algorithm is referred to as *n-competitive*.

For the paging problem with a cache size of k, there are two popular online algorithms that have been proven to be k-competitive to LFD [30].

- **FIFO:** If a page is requested that is not in the cache, make space by evicting the page which has entered the cache the earliest.
- **LRU:** If a page is requested that is not in the cache, make space by evicting the page which has been accessed least recently.

Although FIFO and LRU have the same competitive ratio in theory, LRU performs much better in practice [16]. One intuitive reason for this is that most applications have a small number of elements that have a major share in requests. With FIFO, the pages containing those elements would be repeatedly evicted because their position in the eviction order continuously increases independent of how often the pages are accessed. With LRU, on the other hand, their position in the eviction order is set to the last one with every access. Instead of items that are accessed with a high frequency, items with a low access frequency get evicted.

Role of LRU Caching

LRU caching is used to implement the cache inside the deltadb client. It stores pages (parquet files from Delta Lake) in memory. If a read request is made to the deltadb API, it is first checked whether the requested item is inside a page of the LRU cache. If so, it is served directly from the cache. If not, the page containing the item is first retrieved from the object store and loaded into the cache.

Reasons for LRU Caching

It is useful to make use of caching for deltadb because the access times for files from object stores are quite long for the use case of retrieving an item from a DBMS. To shorten access times, it is ideal to read items from an in-memory cache instead of an object store. Caching is exceedingly useful if the majority of requests are performed on a small number of items. Applications using a DBMS for transactional processing tend to fulfill this characteristic in contrast to OLAP applications. A web shop, for example, will always have best sellers that account for a large chunk of traffic, while it also sells items that get rarely looked at all. The same pattern can be detected in many other types of applications as well.

LRU was chosen as an eviction strategy because it is not known upfront which future items will be requested. Hence an online algorithm was needed. Although FIFO and LRU are both k-competitive to LFD, experience shows that LRU is way more effective in real-world scenarios.

4.3.2 Apache Kafka

Apache Kafka is a distributed messaging system that was originally developed at LinkedIn and is now maintained by the Apache Software Foundation. In 2011 Kreps et al. outlined the technical design of Kafka in their publication "Kafka: a Distributed Messaging System for Log Processing" [26].

As the name suggests, Kafka was originally designed for the use case of log processing. In modern software landscapes, log files represent a continuous stream of large amounts of data. Before Kafka existed, hyperscaler companies like Facebook developed their own frameworks, specialized for log processing and aggregation, like scribe [5] or Flume [4].

Those frameworks, however, were only capable of delivering the data to file systems like HDFS where they could be further processed by MapReduce or Spark. LinkedIn developed Kafka with the intention of supporting log processing use cases within seconds. Because of the static data sinks like HDFS, this was not feasible with the old log processing engines.

Other types of systems, already available at the time, were messaging queues like RabbitMQ [6]. Those systems could fulfill LinkedIn's latency requirements but lacked capabilities in other areas. Their main problem was that they could not be scaled to process the required throughput. Furthermore, the number of messages which could be stored within those systems was limited, limiting their capabilities for analyses on larger sets of data.

Kafka was designed to combine the strengths of those two types of systems for log processing. It should have a low latency like messaging queues but should be scalable in terms of throughput and amount of stored data like log aggregators.

To achieve this, Kafka was built similarly to a messaging queue, but some design changes were introduced, which allowed for higher throughput and message storage.

Firstly, Kafka allows batches of multiple messages to be produced and consumed. In traditional messaging systems, every single message had to be sent and acknowledged, which significantly lowered the maximum throughput of those systems.

Secondly, Kafka has horizontal distribution embedded in its core design. This is perhaps the most important design change. Like traditional SQL databases, messaging queues could not be partitioned very well. This makes distribution among multiple clusters of machines an incredibly difficult task. Other systems like NoSQL databases, HDFS, or Spark have shown that scaling out instead of scaling up is the key to storing and processing nearly unlimited amounts of data.

As usual in distributed systems, Kafka lowered the guarantees for its data. For example, Kafka originally only guaranteed at-least-once delivery of its messages. In opposition to an exactly-once delivery guarantee, it can happen that an acknowledged message appears twice in the log (but it will never be lost). Within the distributed messaging system, this at-least-once guarantee can be given if the producer just tries to resend a message to any broker which failed to acknowledge a write. This can lead to duplicated messages (for example, if only the acknowledgment failed but the write was successful) but eliminates the need for distributed transactions. Since its inception, Kafka has made significant progress. Confluent, the company founded by the original Kafka authors, fo-

cussing on commercial versions of Kafka, claims that exactly-once delivery is possible in Kafka without significant impairments to performance [34].

The last design decision which makes Kafka more suitable for log processing than other messaging queues is a relatively large amount of accumulated messages which can be stored within Kafka. Theoretically, a queue is specified in a way that an element is dropped after it has been consumed (dequeued). Messaging queues follow this design specification and only keep a short backlog of messages. Knowing when messages have been consumed by all subscribed consumers and then deleting them proves to be quite a difficult task in distributed systems. In addition to that, deleting a message provides no inherent value. This is why Kafka decided to introduce retention periods for its messages. Messages are kept within Kafka for a specified amount of time (by default, one week) and then deleted without having to track the consumption of the message. The responsibility of tracking the already consumed messages is handed to the consumers. Having a relatively large backlog of messages available also enables the consumers to ”replay” data that they had already consumed. This proves very useful, for example, in cases where there has been a failure of a consumer after a message consumption. To enable such a large backlog, Kafka is implemented in a way that enables constant performance, independent of the number of accumulated messages. Other messaging queues tend to decrease in performance as the amount of accumulated messages increases.

Specifically, a Kafka cluster can consist of multiple **topics** that can be seen as distinct log stores. Topics store a set of **messages** which can represent any information in a byte format. Multiple **consumers** and **producers** can produce and consume messages on a topic independently of each other. A topic itself is sharded into multiple **partitions**. Within a partition, it is guaranteed that events will be read in order. Across partitions, this guarantee is not given. Furthermore, messages can (but must not) have a message key. It is also guaranteed that messages with the same key will be written to the same partition, which is enabled through a MurmurHash hashing algorithm. One can conclude from this that Kafka also guarantees that messages with the same key will be read in the same order as they were produced. Physically a Kafka Cluster consists of multiple **brokers** that store the data. A single broker can store multiple partitions of multiple topics. In general, the partitions are replicated across multiple brokers (often across three different brokers). For each partition, there is a lead broker determined. Reads and writes are primarily performed to the lead broker. As long as the broker does not go down, this means that Kafka can guarantee strong and not just eventual consistency like other distributed data stores. The partitions themselves consist of multiple segment

files that simply append incoming messages. There are no auxiliary data structures like B-Trees which have to be maintained by Kafka. Maintaining those data structures is the reason for the decreasing performance of other messaging systems in relation to a larger amount of accumulated messages. A consumer can address a message by its message offset, a consecutive number that is assigned by Kafka. Kafka only maintains an index that maps index ranges to a specific segment file.

Role of Kafka

Within the prototype implementation of deltadb, Kafka assumes the role of collecting and aggregating the performed operations on the application nodes, which act as data producers. Each operation is passed into Kafka as a message which is consumed by Spark Streaming afterward.

Reasons for Kafka

Because of the nature of object stores, Delta Lake, the final sink for data within deltadb, only has a highly limited transaction rate. A distributed database system designed for scale should support a very high amount of operations. Because of this, it is not feasible for the clients to write data to the delta table by themselves. It is definitely necessary to aggregate multiple operations on deltadb into a single transaction to Delta Lake. It would be possible to implement this kind of aggregation and retry mechanism within the clients, but since the number of clients is also not limited in theory, this could also exceed the supported transaction rate of Delta Lake and lead to very long backlogs. This problem was solved by using Kafka as a centralized aggregator for database operations, similar to its original use case as an aggregator for log entries.

Kafka was used because of its two core characteristics: High, scalable throughput and low latency. Because deltadb is designed for large-scale workloads, a messaging system with a limited throughput would act as a bottleneck, impairing the performance of the whole system. Additionally, it is highly desirable for an eventual consistent DBMS to have inconsistent timespans as short as possible. A system with high latency, therefore, would not be feasible.

In addition to that, each operation on deltadb is related to an item that can be identified by a key. This key can also act as the message key within Kafka. It follows that operations

on the same objects will be written to the same partition within Kafka, where a correct read order is guaranteed. Since an operation on one item does not affect any other items (and transactions are not supported), this means that deltadb can guarantee monotonic reads (a later read can never deliver a state prior to an earlier read) [35].

4.3.3 Spark Streaming

The basis for the Spark engine was described by Zaharia et al. in their publication "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" [36]. This was published in 2012 when cluster computing frameworks like MapReduce were already popular. These frameworks gave users access to distributed computing power, but not memory. This made those frameworks very ineffective for use cases where multiple computing steps depend on the memory of other steps. A clear example is an ETL job consisting of multiple steps, performed with MapReduce. In practice, the ETL job would have to persist the state to an external memory system like HDFS and then have the next step read it again. Those I/O operations are fairly costly and slow down those applications enormously.

As its centerpiece, Spark introduces a distributed memory representation of a dataset - the Resilient Distributed Dataset (RDD). The core requirement for such a representation in a distributed environment is efficient fault tolerance. Previous approaches for distributed memory enabled fault tolerance by distributing the elements (rows and cells) of a dataset among multiple nodes in a cluster. This turns out to be quite inefficient because this leads to the whole dataset being sent across the network of clusters. With datasets of multiple Terabytes, this proves to be unfeasible. The RDD solved the problem of enabling efficient fault tolerance by representing a dataset based on a source and a set of coarse-grained transformations (transformations that affect the whole dataset rather than just a single row or cell). This representation is in order of magnitudes smaller than previous approaches and can therefore be distributed efficiently. In case of a node failure, the missing partition of data can simply be recomputed by another node applying the transformation steps again. In order to enable the recomputeability of an RDD, they have to be immutable and read-only. A change to data can only be applied by a coarse-grained transformation.

The memory concept of an RDD enabled Spark to undertake further optimizations in terms of computing. The available coarse-grained transformations can be split up into

2 groups: narrow and wide transformations. Narrow transformations lead to a narrow dependency between the input and output partition - the output partition can be completely computed by the input and data from other nodes is not required. Examples of narrow transformations are the map or filter operation. Wide transformations, like a join, for example, on the other hand, require data from multiple partitions being brought together, similar to MapReduce. The distinction between those two types of operations allows Spark to create so-called stages in its computing. Stages describe a chain of narrow transformations that can be executed in a distributed manner without having to transfer data. The stages themselves are then connected via wide transformations. Those connections allow Spark to create a DAG (Directed Acyclic Graph) of stages. Having this DAG lets the Spark scheduler use parallelism between its worker to the maximum amount possible, which is a key factor for the high performance of the Spark compute engine.

On top of RDDs, there was a whole ecosystem for big data processing built within Spark. For example, the Spark SQL API was introduced on top of RDDs. While RDDs stored data without a schema in a row-based manner, Spark-SQL enforces a schema on top of an RDD, making up a so-called dataframe that can be seen as equivalent to a table in SQL. Enabling plain SQL statements against these dataframes makes the usage highly attractive since SQL knowledge is very widespread.

Another for which RDDs proved useful is streaming. Streaming requires the continuous processing of data in near real-time. Zaharia et al. described their streaming implementation on top of RDDs in the paper "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters" which was published in 2012 [37]. They note that streaming engines popular at that time used a relatively low-level programming model. Consistency and fault-tolerance had to be implemented by the user. They introduced Discretized Streams (D-Streams) that aim to provide a high-level programming model with built-in consistency and fault tolerance. They implemented it within the Spark ecosystem as Spark Streaming and used the same interface which Spark used for batch processing and interactive analytical queries.

Previous record-at-a-time systems ran into complications in terms of fault tolerance and consistency. Fault tolerance could be achieved by either replicating the computational steps among two or more nodes or buffering sent data. Replication proved to be fairly expensive (increases the infrastructure cost by at least two times), and buffering led to very long recovery times. Consistency is hard because some nodes could be lagging

behind others, corrupting the global state of the data stream. Another problem was that a processing engine working on a level of single entries was hard to unify with batch processing use cases.

To solve those problems, some differentiated design decisions were made for D-Streams. The main idea is to split up a stream into small intervals and process each interval as a batch computation. Since, in this model, each record can exactly be assigned to a specific interval, consistency is way easier to achieve. To achieve fault tolerance, the advantages of RDDs were put to use. Using an RDD, it became much more lightweight and ,therefore, feasible to distribute the state of a stream among multiple nodes. Failure recovery was made efficient in terms of cost and performance by that. The Spark compute engine was used to execute the streams consisting of micro-batches. With the help of that, a sub-second latency could be achieved, which fulfills the requirements of most streaming use cases.

The operations which can be applied on a stream can be split up into two groups: stateless and stateful. Stateless operations can be performed on a single micro batch without knowing the state of previous batches. The available transformations are identical to those available in the Spark batch processing API (map, filter, reduce...). Stateful operations, on the other hand, perform operations on multiple micro-batches. They consist of windowed and naïve aggregations where windowed aggregations get recomputed at each batch for a specific amount of windows, and naïve aggregations simply work on the output state of the previous naïve aggregation. D-Streams support two output modes: save and foreach. Save persists the RDD of a stream while foreach performs a Spark batch operation on the incoming interval of data.

Role of Spark Streaming

Within deltadb Spark Streaming is used to implement the stream processor. It collects a stream of messages, representing deltadb operations, from the Message Broker, deduplicates them, and applies the updates to the lakehouse in near real-time.

Reasons for Spark Streaming

Spark streaming was used over other ways of updating the lakehouse state for multiple reasons. Firstly, Spark has a solid and battle-tested integration with Delta Lake since

both are heavily maintained by the same company, Databricks. Other connectors, like a direct connector directly with Kafka, are not as mature. Secondly, Spark Streaming supports micro-batching instead of "real" record-for-record streaming by default. This is necessary for the use case of deltadb because the transaction rate supported by Delta Lake on top of S3 is limited to a small number of transactions per second. Since a deltadb is supposed to support a significantly higher transaction rate, it is necessary to aggregate multiple deltadb operations into larger transactions. This can easily be done by the foreach output within Spark Streaming. Using a "real" stream processing engine like Apache Flink would introduce the need to add an additional step (like a Spark job) before loading data into Delta Lake [19].

4.3.4 Amazon S3

As already outlined, there have been multiple computing frameworks that enabled large-scale data processing by distributing their workload across a cluster of machines and being fault-tolerant to failures of individual machines. MapReduce and Spark for computing and RDDs for memory. In addition to that, there has also been a solution to provide fault-tolerant persistent storage on top of a cluster of machines.

Google pioneered in this field and developed the Google File System (GFS) that was described by Ghemawat et al. in 2003 [21]. The Hadoop File System (HDFS) is an open-source implementation similar to GFS. For a long time, it has been the go-to system for storing large amounts of unstructured data.

With the release of S3 as the first service within AWS, Amazon sparked the revolution of object stores. To achieve desirable characteristics, some guarantees that file systems and also HDFS provided were reduced.

A file system has a folder structure that represents a tree data structure. Operations on non-leaf nodes (e.g., metadata operations like renaming a folder) can therefore be performed efficiently with an asymptotic runtime complexity of O(1). Object stores like S3 discard the concept of organizing files in a tree-like manner. Each file is stored as an individual object which must be addressed with its full identifier. Folders are only augmented by listing multiple objects with the same prefix. This fact leads to significant inefficiency in performing metadata operations. The renaming of a folder, for example, can only be executed with an asymptotic runtime complexity of O(N), whereas N can stand for the number of files within a folder or, in the case of S3, for the amount of data

stored within the folder.

Traditional POSIX file system operations like ”open,” ”close,” ”read,” or ”write” are not available within object stores. An object stored in an object store can simply be read or written (PUT and GET operations). Once put into the object store and object is seen as immutable. This removes capabilities for continuously writing to a single file or using locks for file operations. On the other hand, the removal of I/O blocking drastically improves the scalability of an object store since multiple clients won't block each other's file access.

For object stores, consistency requirements have also been drastically reduced in comparison to file systems or HDFS. Just like POSIX systems, HDFS supports *one-copy-update semantics* as its consistency model. This means that multiple clients accessing a specific file will see the same content of a file, similar to only one copy of the existing file. Traditionally object stores like S3 only support eventual consistency. This means that an update can take an indefinite time till it is propagated to all nodes and will therefore be read by every client. It is only guaranteed that the update will be visible eventually. With reference to the CAP theorem, it is only possible for a system to achieve two of the three desired attributes consistency, availability, and partition tolerance [22]. Reducing consistency to eventual consistency is, therefore, commonly used to either increase partition tolerance or availability that is considered more important for object stores. In 2020, 14 years after the original release of S3, Amazon announced strong consistency for S3 [11], thereby eliminating the tradeoff between consistency and availability.

Object stores made all those tradeoffs to optimize for other desired characteristics. The availability and durability that can be achieved with systems like S3 are unmatched by any other application. According to its SLAs, S3 achieves 11 9's (99.999999999%) of durability and 99.99% of availability. Furthermore, considering the total cost of ownership, its pricing is hard to match with another form of storage. The ability for S3 to scale in an elastic way and independently of computing resources brings even more benefits. All of those factors have led to cloud-object stores becoming the de-facto standard for storing large amounts of unstructured data.

Role of Amazon S3

Amazon S3 is used as infrastructure to store the lakehouse data. Thereby it functions as the single, persistent source of truth for data stored in a deltadb table. The stream

processor writes data into S3 in the form of files, and the deltadb client can access data stored in S3 with GET requests.

Reasons for Amazon S3

Amazon S3 was used as a technology because of its supreme characteristics compared to a traditional file system or a distributed file system like HDFS. Other cloud object stores like Azure Blob Storage or Google Cloud Storage could have been used as well.

The most important quality of object stores is their very high availability in comparison to other storage systems. Deltadb was designed to be highly available, so the storage system with the best availability was chosen. In addition to that, deltadb was also designed to scale nearly infinite in terms of throughput and storage, another benefit of object stores. The seamless scaling and ability to separate storage and compute significantly simplified the way deltadb could be implemented. The other advantages of object stores, like low costs or high durability, come on top of that.

4.3.5 Delta Lake

As described in **section 4.3.4**, there is a multitude of advantages to using cloud object stores. While they were originally used to store unstructured data like image or video files, over the course of the years, the number of use cases has expanded. Because of their high availability, durability, and infinite scale, enterprises increasingly resorted to object stores to implement their analytical data infrastructure, mostly following the data lake paradigm. Against the nature of object stores mostly structured data was stored in them using file formats like CSV or parquet.

Although object stores come with a great set of advantages, their usage in the data lake context still lacks some of the advanced capabilities that data warehouse systems offer. Those lacking capabilities include support for ACID transactions, mutability of stored data, and high performance over large datasets. Because of this, virtually all enterprises decided to implement an architecture that would combine the advantages of object stores and data warehouses. They did this by using both systems in combination, firstly loading data into a data lake built on top of an object store and then processing and transferring data via a pipeline to a data warehouse. This so-called two-tier architecture was born and became the de-facto standard for analytical data processing in enterprises.

Although having both systems connected via a pipeline, on the one hand, offers the joined advantages of both systems, on the other hand, it introduces a new set of downsides that pose challenges to enterprises. Those downsides include increased cost, data staleness, effort for implementation and maintenance, and decreased capabilities for data protection. Delta Lake was implemented to power a shift in analytical data architectures. This shift is described in a new architecture style that can combine the advantages of data lakes and data warehouses in a single system, hence eliminating the downsides of a two-tier architecture. This architecture paradigm, the lakehouse, was first described by Armbrust et al. in 2021 [10].

The technical implementation of Delta Lake was described by Armbrust et al. in 2020 [9]. At its core, Delta Lake represents an "open source ACID table storage layer over cloud object stores" [10]. Delta Lake achieves ACID table storage over cloud object stores by declaring a specific "folder" inside an object store as a table - the delta table. Within this table, there are multiple parquet files, each one storing a set of table rows. In addition to that, Delta Lake defines a separate log folder within the delta table folder. This folder contains a sequence of JSON files with increasing IDs containing the log events. There is a multitude of events that can be logged within the JSON files, but the most common ones are *adding* or *removing*. After a specified amount of log files (by default, 10), the log events get compressed to a bookmark file in parquet format. This bookmark file contains the computed last stand of the table after all log events have been applied. The log file is useful for clients that want to compute a specific version of the table. Without the bookmarks, they would need to read each JSON file and apply their events which would take a large number of resources, also because some log events become redundant over time. A visualization of the Delta Lake file structure can be found in **figure 4.5**.

By leveraging a log folder in combination with a traditional collection of parquet files, Delta Lake gains some remarkable characteristics.

The core capability Delta Lake was aimed to support is **ACID transactions**. In a data lake environment, all files within a folder are considered a part of a data table. To remove their data points from the table, they must be removed. This makes it nearly impossible for data lakes to perform transactional actions in an atomic way. As soon as a transaction writes more than one file into the folder, the data lake will have a corrupted state because data of an unfinished transaction is considered part of a table. In contrast to that, Delta Lake does not consider all files within its folder as part of a delta table. Only after a file has been described as added within the log files its data points are added to the table. Because one transactional step (one new JSON file) can perform multiple

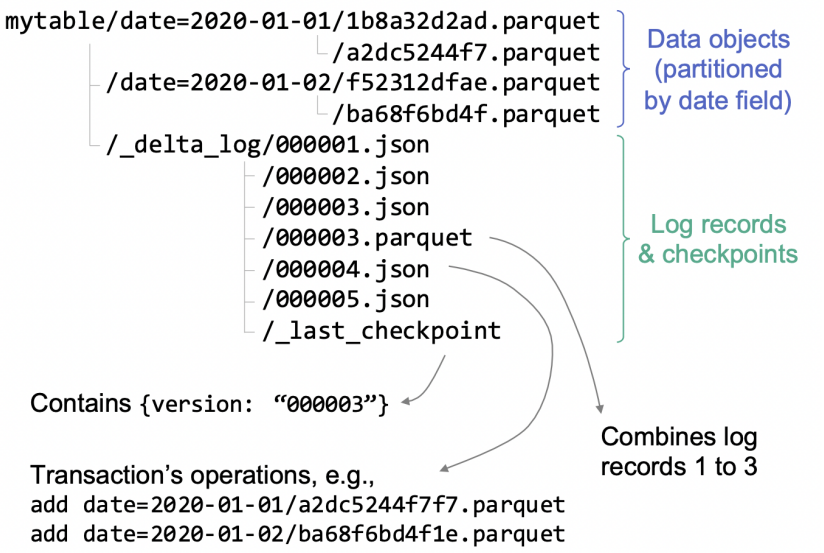

Figure 4.5: File structure within an example Delta Table [9]

add operations, Delta Lake can perform atomic transactions. This is largely important for having a reliable source of data. Actions performed on data from a data lake that is corrupted because a transaction is still going on can have bad consequences because the decision could be made on the basis of incorrect data.

In addition to that, Delta Lake enables the functionality of **time travel**. A table can be rolled back to any point in time in the past and be investigated. Classical data lake systems are not capable of this. The feature of time travel is crucial when investigating reasons a decision was made at a specific point in time in the past or to have a consistent dataset for data science use cases.

In contrast to data lakes, which are append-only data stores, data stored within lakehouses can be considered **mutable**. Data stored within a file inside a data lake cannot be manipulated because this would require deletion and re-writing of the file at the same time. Therefore, typically a data point just gets an increased version number and gets added to the data lake whilst the old entry stays persisted in the system. Delta Lake will add a transaction that invalidates the old file a specific data point was written into and add an updated version instead.

After a specified period of time or according to request, the old files can be completely removed. This is required for specific use cases in the context of **data protection**. This became especially relevant after 2018 when the General Data Protection Regulation (GDPR) of the European Union was put into place. With Delta Lake functionalities like

SLAs for deletion of data after a specific amount of time (vacuum function) or just the general ability to have a single copy of a data point, GDPR requirements like the **right to be forgotten** can be fulfilled.

Despite all the advantages, the use of Delta Lake also comes with a number of challenges.

The main one is that the supported **transaction rate** is limited to the supported transaction rate of the underlying object store. In contrast to OLTP database systems, object stores haven't been designed for a high transaction rate and thus only support a small number of transactions per second. For use cases like web applications, this is clearly insufficient.

Another performance metric limited to the underlying object store is the **latency** for queries since data cannot be extracted faster because of a metadata layer. Object stores also don't perform well when it comes to latency. Fulfilling a GET request takes lays in the 100's milliseconds range. This is also not sufficient for databases that need to support transactions for web applications. The transaction log, however, does enable better caching because it is clearly tracked what data gets added and invalidated from a table.

Independently from the performance limitations of object stores, Delta Lake can only support transactions over a single table. The transaction log is only stored in a single object store "folder", and there is no consistency guarantee across multiple folders. In order to support transactions among multiple tables, we would need to store the transaction log in another database system. In many classical SQL databases, transactions among multiple tables, however, are a key element of their capabilities and are not possible with Delta Lake.

Role of Delta Lake

Delta Lake is used as a metadata storage layer on top of the data stored in S3. It enables Data to be written to the shared disk layer with atomic transactions. This gives deltadb the capability to enable read-after-write consistency for its clients because no data point can overwrite an entry that happened at a later point in time.

Reasons for Delta Lake

The main reason Delta Lake was used is that it enables ACID transactions over object stores. Object stores were the system of choice for the OLAP site of deltadb. Their

advantages of low cost, seamless, infinite scale, and unmatched durability, as well as availability, are just too valuable to ignore. In order to build an OLTP service that uses the OLAP storage as a shared disk layer, it was necessary, however, that atomic transactions were supported within the shared disk. Without transactions, a data point that was originally written at an earlier point in time could overwrite a later entry. This would destroy the capability of read-after-write consistency, limiting a central consistency guarantee.

Another aspect that cannot be ignored is that Delta Lake can significantly improve the **performance of analytical queries on large amounts of data** through internal optimizations like Z-ordering. Also, for analytical queries, it is crucial to minimize response time. Furthermore, a longer response time is proportional to higher infrastructure costs because more data needs to be scanned from object store services like S3.

Delta Lake does all this while at the same time using an open-source file storage format. Data independence, as it exists in many other database systems, does not exist in deltadb. Systems accessing data must not do this via the API for operational data (comparable to JDBC in SQL databases). Instead, various consumers can read data directly from the object store. This becomes especially useful for applications that scan entire tables because they need to extract a whole large dataset. Important examples of these applications are mainly data science use cases that need a whole dataset to perform tasks like machine learning or statistical exploration.

5 Benchmarking

A benchmark describes a standardized test that can be applied to multiple instances of an object. This test can measure results of multiple dimensions in a quantitative way in order to deliver a decision basis to choose a specific instance. Benchmarking is prevalent in many fields (e.g., business or computer hardware) but can certainly also be applied to databases. For databases, mostly performance metrics like read or write latency are the subject of benchmarks.

In the early days of traditional SQL databases, each vendor produced and published their own method for performance benchmarking of databases. Of course, these vendors had little incentive to design a benchmarking method that would not have their systems come out on top, highly limiting the value of those benchmarks. It is fairly simple to design a benchmark that tests the exact "sweet spot" workload of a specific system (for example, the benchmarking test would only consider read-heavy workloads, totally ignoring write-heavy use cases). This would limit the objective understanding of tradeoffs that have to be made at making a decision for a specific database system. Because there was a need for objective comparison, after a while, standardized benchmarking methods were established. In 2017, when NoSQL databases had already risen in popularity, Reniers et al. conducted a systematic survey on benchmarking for NoSQL benchmarks and published it with the title "On the State of NoSQL Benchmarks"[29]. They note that originally the situation for NoSQL systems was similar, with each vendor publishing their own benchmarking method. After a while, however, the "Yahoo! Cloud Serving Benchmark" (YCSB) emerged as a standard.

Reniers et al. also note that YCSB lacks capabilities for benchmarking use cases that exceed key-value style operations. For example, there is no concept for more sophisticated access patterns that are served by NoSQL databases, like graph or document stores.

Because deltadb only supports rudimentary functions on the transactional side, YCSB can be seen as sufficient for this use case and will therefore be used for performance benchmarking.

5.1 Yahoo! Cloud Serving Benchmark (YCSB)

YCSB was formally described by Cooper et al. in their publication "Benchmarking Cloud Serving Systems with YCSB," published in 2010 [17]. The authors note that a new type of use case, which they call "Cloud OLTP," had emerged in prior years. This use case neither falls into the category of traditional SQL databases that support ACID transactions like MySQL nor into the bucket of large-scale cloud data processing systems like Hadoop or MapReduce. Examples of databases supporting this new type of use cases include NoSQL systems like Cassandra, CouchDB, or SimpleDB (de facto the predecessor of DynamoDB). Yahoo classifies this family of NoSQL databases that aim to provide infinite scale, elasticity, and high availability as "Cloud Serving Systems" that, as a tradeoff, sacrifice the complex query capabilities as well as the robust and sophisticated transaction models found in traditional systems" [17]. This group of databases had previously not been recognized very well among performance comparison frameworks. Methods like TPC-C only consider transaction-heavy workloads. Yahoo introduced YCSB in order to close this gap in frameworks but also in order to compare their own NoSQL database PNUTS to other systems.

5.1.1 Architecture

The YCSB framework consists of two main components

- Workload generating client
- Package of predefined standard workloads (read-heavy, write-heavy, or scan workloads)

The architecture of the actual executing, workload-generating client can be seen in **figure 5.1**. We can see that the system takes two inputs: The workload file and the command line properties, defining the type of operations the client has to perform as well as how it should connect to a database that will be tested. The client itself is split up into four components.

- **Workload Executor** - Coordinates multiple Client Threads to generate actual operations according to the specified workload file.
- **Client Threads** - Enable the workload executor to generate its operations in a multi-threaded way

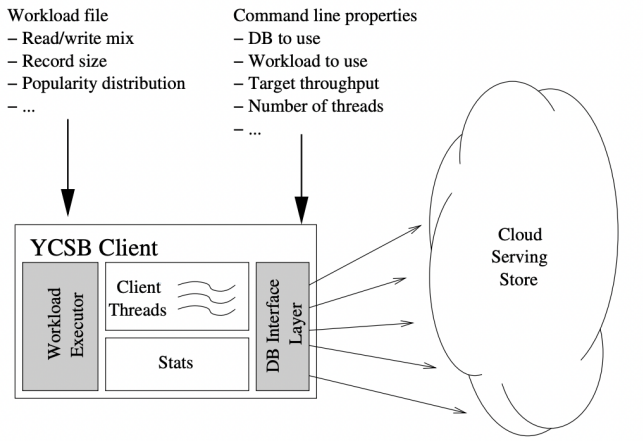

Figure 5.1: YCSB Workload generating Client Architecture [17]

- **Stats** - Collects metrics like latency about the operations from the client threads and eventually aggregates them to produce meaningful metrics like average or 99th percentile latency.
- **DB Interface Layer** - Translates internal operations from YCSB (Insert, Update, Read, Scan) into API requests the specific database can process (for example, REST).

Looking at **figure 5.1**, it can be noted that 2 of the four component boxes are colored grey. This is done to display that those components are designed to be adapted. In general, a core principle of YCSB is adaptability to be able to account for different, new databases with different interfaces and different types of workloads. The Workload Executor can be adapted for generating non-standard workloads, while the DB Interface Layer can be adapted to benchmark a non-standard database. Especially adapting the DB Interface Layer makes YCSB very useful for benchmarking deltadb since it is a new implementation and does not serve a standard API.

5.1.2 Benchmark Tiers

Cooper et al. propose two benchmark tiers, meaning dimensions on which the performance of a system can be quantified.

The first one is **performance**. Typically, on the same hardware, there is a tradeoff between latency and throughput. This is the case because random access I/O can either

be used to populate auxiliary data structures (low latency, low throughput) or to look up data because there are no auxiliary data structures (high latency, high throughput).

An example of this is log files stored in S3. They can have an immensely high throughput because data is simply appended to files. When a specific log entry is requested though, this results in very high loading times. Data structures that maintain indexes, like DynamoDB, on the other hand, have a low throughput and low latency.

The performance benchmark tier aims to characterize how a specific database system handles the tradeoff between performance and latency. This is done by measuring the latency while increasing the throughput until the benchmarked database system is saturated and does not process more throughput.

The second benchmark tier is **scaling**. Since horizontal scaling capabilities are a key selling point of "Cloud OLTP" systems, it is of high interest how a system responds to workloads in relation to the number of nodes it is run with. Ideally, adding additional nodes does not result in diminishing returns.

There are two modes for testing the scaling benchmark tier - static and elastic. Static scaling performs a specific workload, increases the number of nodes, and then performs another workload to compare the results. Elastic scaling, in contrast, increases the number of nodes while testing performance. The goal here is to additionally measure the time needed to get improved results from a higher number of nodes. This is highly relevant for use cases with burst-like workloads.

5.2 Conducting a performance Benchmark

The YCSB framework contains a large set of binding directories (see **figure 5.2**), most ones implementing the DB Interface Layer for a commonly used database.

```
johanneskotsch@Johanness-MBP YCSB % ls -a
.                   README.md           checkstyle.xml      elasticsearch5      jdbc                rados               tarantool
..                  accumulo1.9         cloudspanner        foundationdb        kudu                redis               target
.editorconfig       aerospike           core                geode               maprdb              rest                voltdb
.git                arangodb            couchbase           googlebigtable      maprjsondb          riak                workloads
.gitignore          asynchbase          couchbase2          googledatastore     memcached           rocksdb             ycsb-0.17.0.tar.gz
.idea               azurecosmos         crail               griddb              mongodb             s3                  zookeeper
.travis.yml         azuretablestorage   distribution        hbase1              nosqldb             scylla
CONTRIBUTING.md     bin                 doc                 hbase2              orientdb            seaweedfs
LICENSE.txt         binding-parent      dynamodb            ignite              pom.xml             solr7
NOTICE.txt          cassandra           elasticsearch       infinispan          postgrenosql        tablestore
```

Figure 5.2: YCSB binding directories

The DB interface layer is implemented by extending the abstract Java class ”DB” that defines a standard set of methods for interacting with a database system. A simplified UML class diagram of the DB class with its children is depicted in **figure 5.3**. The

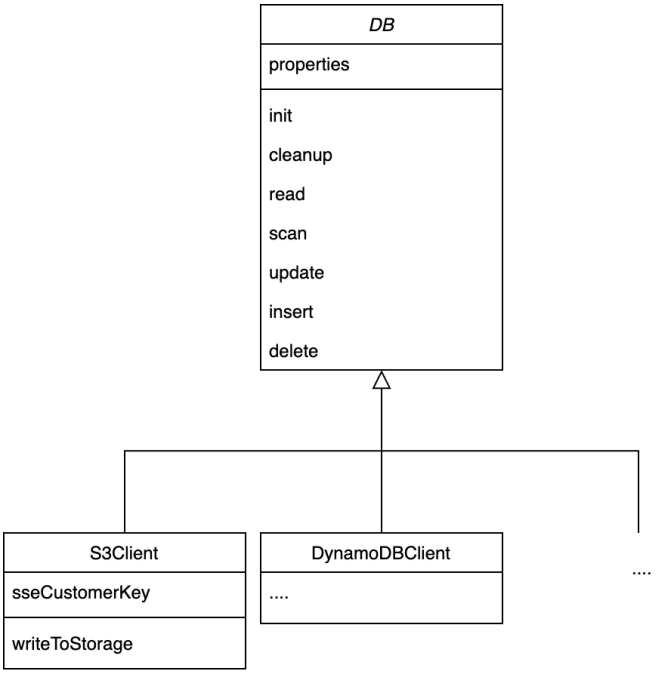

Figure 5.3: Simplified class diagram for YCSB DB Interface Layer

abstract DB class requires the ability to load **properties** from a configuration file and set up a database for usage as well as cleaning up afterward (with the methods **init** and **cleanup**). On top of configuring and building a connection, the standard methods for interacting with a database **read**, **scan**, **update**, **insert** and **delete** are required.

Classes inheriting from ”DB” can implement those functionalities in different ways under the hood. The client for S3, for example, stores the access key details in the variable **sseCustomerKey** and uses the single method **writeToStorage** to implement both insert and update.

In order to conduct a performance benchmark for deltadb, a custom client had to be implemented and registered with YCSB. A code extract of this can be seen in **listing 5.1**. It can be seen that the **insert**, as well as the **update** method, is implemented by delegating to the private **put** method, similar to the previously described S3 client. Since deltadb exposes a rest interface, the put method itself creates an HTTP put request by encoding the new data in a JSON format. The **okhttp3** library by Square is used to

create HTTP calls with java in a relatively easy manner. It expects the status code 201 to return an OK status that can further be processed by YCSB. In case there is another return code, it returns an ERROR status.

Listing 5.1: Implementation of the upate method in DeltaDBCLient.java

```
package site.ycsb.db;

import okhttp3.*;
...

@Override
public Status insert(String bucket, String key,
                     Map<String, ByteIterator> values) {
    return this.put(bucket, key, values);
}
...

@Override
public Status update(String table, String key,
                   Map<String, ByteIterator> values) {
    return this.put(table, key, values);
}

private Status put(String table, String key,
                 Map<String, ByteIterator> values) {
    String url = "http://localhost:80/" + table + "/" + key;

    JSONObject jsonObject = new JSONObject();
    try {
      for (Map.Entry<String, ByteIterator> val : values.entrySet()) {
        jsonObject.put(val.getKey(), val.getValue().toString());
      }

      MediaType JSON = MediaType.parse("application/json; charset=utf-8");
      RequestBody body = RequestBody.create(JSON, jsonObject.toString());
```

```
        Request putRequest = new Request.Builder()
            .url(url)
            .put(body)
            .build();
        try (Response response = client.newCall(putRequest).execute()) {
          if (response.code() == 201) {
            return Status.OK;
          } else {
            return Status.ERROR;
          }
        }

      } catch (Exception e) {
        e.printStackTrace();
        return Status.ERROR;
      }
}
```

5.3 Benchmark Setup

In order to properly place deltadb in the existing database landscape, I will compare its performance against established commercially available data storage systems. I chose to compare it with S3 and DynamoDB because both are popular, widely used data stores that cover completely different use cases. While DynamoDB is a good choice for applications that require fast access to small amounts of indexed data, S3, on the other hand, can store large amounts of data without indexing and high latency.

The benchmark was conducted with the workload ”workloada” which loads 1000 objects and then performs 1000 read operations. The workload uses the Zipfian distribution for picking the items.

Table 5.1: Statistics of the benchmarking results for different data stores and operations.

	S3 write	S3 read	Dynamo write	Dynamo read	Deltadb write	Deltadb read
mean	313.131429	39.515616	30.246343	29.214943	26.525285	93.489632
std	1932.862763	26.334209	21.950595	32.079455	6.068288	372.096437
min	121.423000	27.493000	23.128000	21.845000	21.592000	19.970000
25%	142.587000	32.803750	27.146000	25.108500	23.926500	22.071000
50%	155.162500	35.551500	28.475000	26.516500	25.135500	23.324000
75%	176.508500	38.812250	30.413500	28.676500	26.866250	24.754250
max	56585.618000	655.355000	699.604000	702.438000	114.589000	3549.872000

5.4 Benchmark Results

The statistics of the benchmark results are displayed in **table 5.1**. The mean value, standard deviation, minimum, maximum as well as the quartiles can be seen. There are two visualizations for this. **Figure 5.4** displays the values without outliers. In **figure 5.5**, the outliers can be seen.

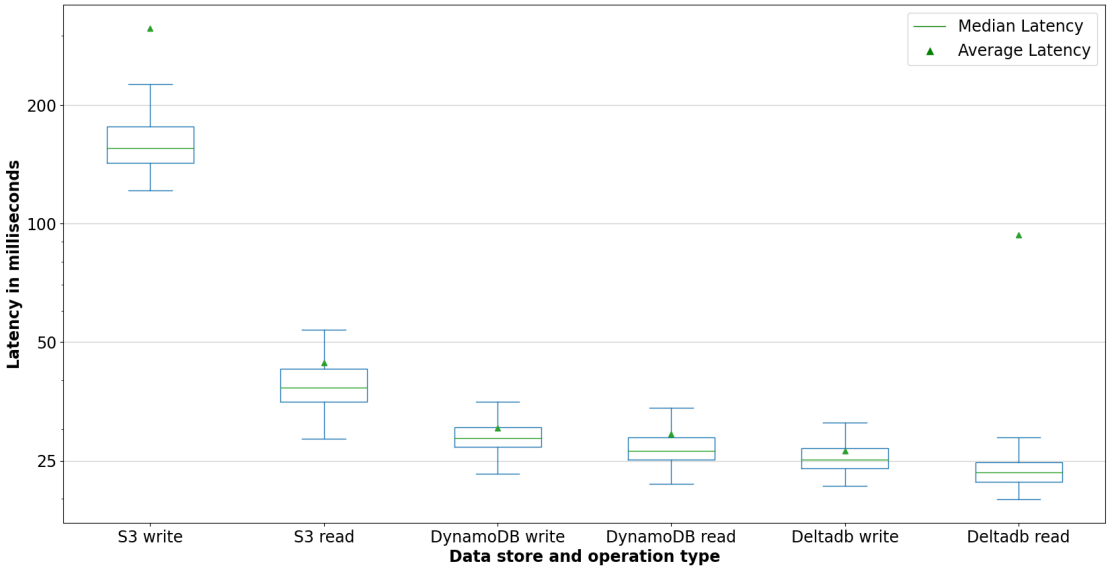

Figure 5.4: Boxplot of YCSB benchmark results without outliers

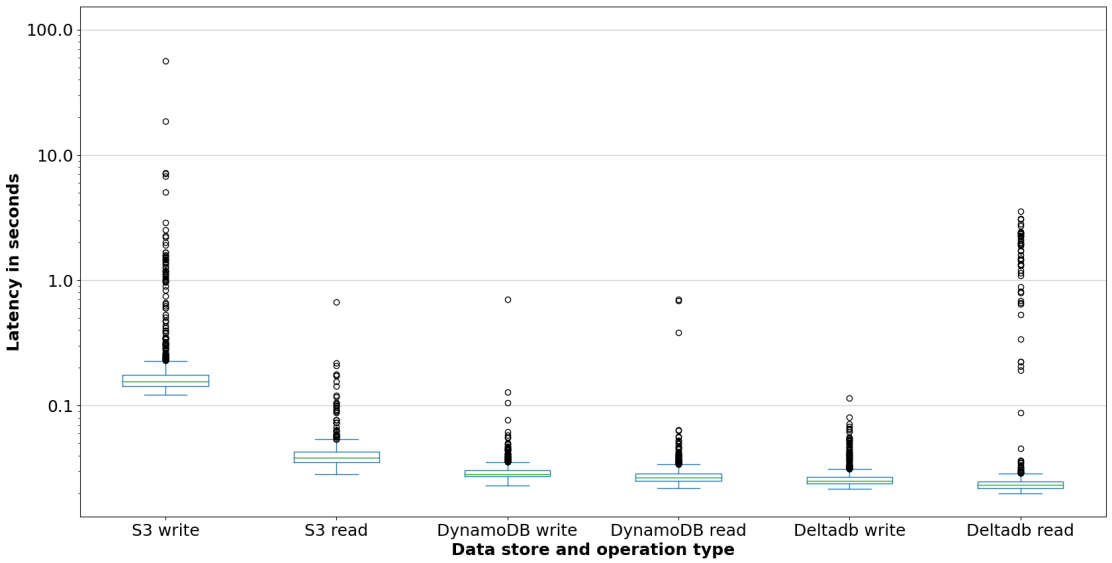

Figure 5.5: Boxplot of YCSB benchmark results with outliers

5.5 Result Interpretation

Comparing the results of the two of-the-shelf data stores, it can clearly be seen that they were developed for different purposes. Looking at the read performance for example, it can be seen that DynamoDB is, on average, a quarter faster, taking only 40ms instead of 30ms on average. While this is significant, the difference becomes even clearer when considering the writing performance. Firstly, the median writing operation takes more than six times as long to perform on S3 (155ms vs. 28ms). In addition to that, S3 is by far more inconsistent with write times. There is a large number of outliers that lie above the standard range of values, with the longest operation taking almost a minute.

Overall, when comparing those two data stores, it can clearly be seen that DynamoDB is better suited for operational use cases where low latency is desired.

Comparing deltadb against those data stores leads to multiple insights.

Looking at the write performance of deltadb, we can note that the system seems to perform very well. Looking at the median and average latency as well the quartiles, it can be seen that it has a better performance than DynamoDB of about 10% across the board (median write latency of 28ms in contrast to 25ms).

However, this seemingly superior performance can be explained by the looser constraints to the consistency that deltadb enforces in comparison to DynamoDB. Deltadb

acknowledges a write as successful as soon as the written value is added to the cache and the message containing it is added to the Kafka message broker. After the message was written to the broker however, it takes quite some time in addition for the written value to be added to the persistent storage layer. If a client hits another node than the note a value was written to, it can take relatively long until the new value is visible there. DynamoDB, on the other hand, can be configured for different modes of consistency. By default (as used for the performance benchmark), the database replicates a value to 3 storage nodes and acknowledges a write as soon as it is performed on 2. In most cases, the third write is performed directly after. With DynamoDB, a client will therefore hit the latest value with a higher chance and with a smaller duration to wait. In practice, DynamoDB, although eventually consistent, feels like a consistent data store, while the eventual consistency of deltadb can clearly be noted when using the API.

To sum it up, deltadb achieves slightly better read performance than DynamoDB because it significantly reduces consistency constraints.

The read performance gives insights about the pros and cons of deltadb as well. Only considering the median case reading performance, deltadb performs comparably to writing. The latency is about 10% lower than DynamoDB (23ms vs. 25ms) as well. Looking at the boxplot in **figure 5.4**, however, the "Deltadb read" column looks distinguished compared to the other ones. While all other datastore operation combinations have an average value somewhat close to the median, this is not the case for deltadb reading at all. On average, a deltadb read operation takes four times as long as the median read (93ms vs. 23ms), while there is only a 10% difference for DynamoDB, for example (29ms vs. 27ms). This can be explained by a high amount of outliers for deltadb read operations that mostly take longer than one second. Looking at the raw data, there are 44 out of 1000 operations with a latency greater than 100ms. Such a large group of significant outliers does not exist in any other column.

Considering the inner architecture of deltadb, a result like this is not surprising. The database has two ways of retrieving a value: from its cache or from the persistent store located in S3. That there seem to be two "groups" the data points come from can easily be explained. The data points with lower latency, such as the median, come from the read operations where the caller could be served from the cache, while the slower requests required access to the persistent S3 layer to be fulfilled. The performed workload performed 1000 read operations with a Zipfian value distribution. Therefore, a very high amount (95%) could be served from the cache because many values were requested repeatedly and only a few for the first time. With another value distribution, like the

uniform one, deltadb would perform significantly worse.

It is also notable that many of the deltadb reads that needed to request data from S3 took over a second, while the average S3 read operation time was roughly 40ms. This is the case because, for the S3 performance benchmark, each data point was stored in a separate file. The data could, therefore, be accessed directly via filename. Having such a large number of files, however, would severely reduce the performance of workloads that need to extract or perform queries on large amounts of data. Deltadb, therefore, groups multiple data points into larger files. On the negative side, this leads to a remarkably slower read performance when using the primary key because multiple files need to be read and processed.

Summing up the performance benchmark of deltadb, on the positive side, we can note that the API is quite fast and achieves roughly 10% faster performance than DynamoDB. Since the benchmark was performed on an EC2 instance, thus comparable Amazon hardware, a possible explanation is that the used API framework, FastAPI performs superior. For write operations, deltadb delivers astonishingly good results as well, but only by significantly sacrificing consistency. On the read side, deltadb performs great for values that are already cached and very poorly for data points that need to be retrieved from the S3 layer.

Overall, deltadb is inferior for ”Cloud OLTP” use cases but can be used if the requirements for consistency are very lax and partially very slow reads can be accepted.

6 Evaluation

After having implemented deltadb as an attempt for a ”One Size fits All” database and performing empirical testing on it, multiple evaluations can be drawn.

6.1 Comparison with related Work

As shown in **chapter 2** there have been multiple approaches for ”One Size fits All” databases as well as database functionalities. Deltadb does not propose a radically new idea for this set of tasks. The main contribution of deltadb is that it combines the used methods of all previous approaches into a single system. This section will describe the transferred methods for each one of the previous pieces of work.

6.1.1 SAP HANA

HANA tried achieving a unification of OLTP and OLAP by compressing the table data into a column-based format. The column-based format significantly reduced the size of database tables. HANA made use of this and stored all tables in memory and served OLAP as well as OLTP use cases from this memory. Column-based formats lead to significantly larger access times for single data points, but the fast memory access times counteract this and therefore enable OLTP use cases.

Deltadb discards the idea of a database kept completely in memory. The source of truth resides in S3 in the shared disk layer. One reason for this is that large amounts of memory are extremely costly compared to disk storage. Because databases contain lots of data that rarely need to be available, having this data permanently in memory can be considered wasteful.

Deltadb does incorporate some techniques of HANA in its cache, though. Within Delta Lake, the different data pages are stored as Apache Parquet files. Parquet also is a

column-oriented file format that compresses its data and therefore reduces storage resources. Deltadb leverages the parquet compression by storing the data in its cache as parquet binary files instead of decompressed data. With the same amount of resources, a lot more data points can be stored in the cache. Accessing specific rows stored in parquet binaries is less efficient than having them stored decompressed, but this fact is counteracted by the superior performance of memory in comparison to disk storage. Deltadb leverages parquet compression on its data pages, and although this is not as efficient as having the whole table stored in a column-oriented format, it still can significantly increase caching capabilities and, therefore, performance.

6.1.2 OctopusDB

OctopusDB was based on the idea of having a log store as a single source of truth for all use cases. Multiple views optimized for their specific use cases were then implemented on top of the central log store.

To some extent, deltadb uses a similar approach. The single source of truth is the lakehouse consisting of multiple parquet files stored in S3, having a Delta Lake metadata layer on top of them. The Delta Lake component can be viewed as a log store as well. There are multiple parquet files containing a set of rows within an S3 folder. On top of this ”raw” data resides a metadata folder that tracks changes in the overall table. The combination of raw data and metadata layer leads to an outcome similar to a log store. All operations on the table are documented, and they can be rolled back to a previous state. Using OctopusDB terminology, two different ”views” can be identified in deltadb. The first view is the OLAP view that accesses data from S3 with a Delta Lake client, for example, via Spark. The second view lies inside the distributed nodes that cache Delta Lake data inside their memory and use this to power OLTP use cases efficiently.

By using Delta Lake as a ”log store,” deltadb gains significant advantages over OctopusDB though. The set of files that make up a delta table can easily be computed and then directly read from storage without having to do any sort of data transformations. That makes it much easier to transform a log store into a table that supports use cases like OLAP.

6.1.3 DBMS on top of S3

The ”DBMS on top of S3” implementation described by Brantner et al. [13] also uses techniques that have found their use in deltadb. The S3 database uses the cloud object store as a persistent storage for pages within the DBMS application. The multiple nodes of the application then interact with S3 to load pages into their own memory and serve OLTP-like requests.

Deltadb also stores data in S3 and then loads those files into the memory of distributed workers. The multiple parquet files within the delta lake can, in this context, be seen as equivalent to pages in the S3 database. Both systems implement a shared disk architecture and use S3 as their shared disk component. Caches of individual workers are used to enhance performance.

The database of Brantner et al. uses traditional storage formats based on B-Trees within their persistent storage layer. Those types of data structures can be reformatted and reparationed at any time from the database engine without having to notify the outside world. Only the defined application interfaces via the distributed clients are guaranteed to be consistent. This means that external applications cannot directly access the data stored in S3 but need to take a detour over the client nodes. Especially in use cases where large amounts of data need to be processed, this is suboptimal since object stores offer more efficient interfaces for reading and writing large amounts of data.

In contrast to the S3 database, deltadb drops the concept of data independence. The shared disk component technically uses the format of Delta Lake. Delta Lake uses open file formats, like Apache Parquet, that can be processed by most engines. Because of this, external applications can directly access stored database data residing in S3. Since the detour over application nodes is skipped, this leads to a much better performance for queries that process large amounts of data. In combination with efficient use of the delta log, deltadb can enable an external application to execute OLAP queries or extract large amounts of data. Because of data independence, this is not possible with the proposal of Brantner et al.

6.2 Overall Evaluation

Summing up, deltadb managed to unify the approaches of previous attempts for "One Size Fits All" systems. It also keeps data in memory in a compressed format like SAP HANA and thereby achieves acceptable performance for operational and analytical use cases alike. Additionally, it uses log data as a source of truth, from which multiple views can be compiled. This approach is similar to OctopusDB. However, the added capabilities of Delta Lake enable precomputing of logs and increase the performance of this approach notably. Deltadb also loads ”pages” from an object store into memory, similar to the database of Brantner et al. Because Delta Lake enables logical processing on those pages, the feature of processing large amounts of data from the object store is added on top.

Overall, it can be concluded that the combination of storing data in open format files in an object store in combination with a metadata layer like Delta Lake is a very powerful combination. Being able to perform ACID transactions makes object stores applicable to a substantial amount of use cases they could not be considered for before. This combination holds significant potential for new data architectures built on top of it that significantly alter the system landscape we know today. Since analytical data processing has gained importance in the modern enterprise world, I see this as likely to happen.

7 Conclusion

After having implemented and evaluated deltadb, we get many insights about the original research question about the possibilities and limitations of using cloud object stores for a ”One Size Fits All” Database.

The usage of cloud object stores can be seen as positive since it solved many problems that would have occurred if a traditional storage technique had been used. The database has instant access to a de-facto infinite pool of resources that scale seamlessly. For anyone who managed a traditional SQL database in production, it is clear that this capability cannot be underestimated.

Especially the use of a metadata layer on top of object stores that allows ACID transactions, enabled object stores to become an efficient source of truth for a database system. Its importance cannot be underestimated.

On the other hand, the limitations of cloud object stores can also clearly be seen. Those systems still have quite a high latency which disqualifies them for their use in operational use cases. Although the impact of this downside can notably be reduced with the use of caching, this comes at the cost of reduced consistency and does only apply to cached items.

When evaluating the performance of the system, it can be concluded that it is possible to design a system that is capable of serving a wide range of use cases. However, as expected, such a system does not perform as well as another system that was specifically designed for a single use case. Deltadb performs worse for ”Cloud OLTP” use cases than DynamoDB, for example.

The question is whether it is useful to invest in developing ”One Size Fits All” systems and for an enterprise to make use of them becomes an economic one. The costs of maintaining an infrastructure that integrates multiple data sources need to be balanced with the costs of reduced performance for specific use cases a ”One Size Fits All” data store has. It is pretty clear that a hyperscaler company like Amazon would decide to maintain multiple data stores. To them, the engineering costs of maintaining such a

”Zoo” are vanishingly small compared to the impact on millions of users.

A possible use case where a ”One Size Fits All” database would make sense is a startup that has requirements for both operational and analytical use cases. For a company like that, the engineering costs of maintaining multiple data stores could outweigh the reduced performance for specific use cases since engineering resources are sparse and user impact is negligible in such a company.

Bibliography

[1] *Amazon S3 FAQs*. https://aws.amazon.com/s3/faqs/. – Accessed on May 10 2022

[2] *Amazon S3 Glacier storage classes*. https://aws.amazon.com/s3/storage-classes/glacier/. – Accessed on May 10 2022

[3] *Amazon Web Services Launces - Press Release - S3 Provides Application Programming Interface for Highly Scalable Reliable, Low-Latency Storage at Very Low Costs*. https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-launches-amazon-s3-simple-storage-service. – Accessed on May 10 2022

[4] *Apache Flume*. https://flume.apache.org/. – Accessed on June 9 2022

[5] *Github – Facebook Archive: Scribe*. https://github.com/facebookarchive/scribe. – Accessed on June 9 2022

[6] *RabbitMQ*. https://www.rabbitmq.com/. – Accessed on June 9 2022

[7] *SAP HANA Cloud*. https://www.sap.com/products/hana.html. – Accessed on May 11 2022

[8] *Top 5 Reasons for Choosing S3 over HDFS*. https://databricks.com/de/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html. – Accessed on May 10 2022

[9] Armbrust, Michael ; Das, Tathagata ; Sun, Liwen ; Yavuz, Burak ; Zhu, Shixiong ; Murthy, Mukul ; Torres, Joseph ; Hovell, Herman van ; Ionescu, Adrian ; Łuszczak, Alicja u. a.: Delta lake: high-performance ACID table storage over cloud object stores. In: *Proceedings of the VLDB Endowment* 13 (2020), Nr. 12, S. 3411–3424

[10] Armbrust, Michael ; Ghodsi, Ali ; Xin, Reynold ; Zaharia, Matei: Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In: *Proceedings of CIDR*, 2021

[11] Barr, Jeff: *Amazon S3 Update – Strong Read-After-Write Consistency*. https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/. – Accessed on May 11 2022

[12] Belady, Laszlo A.: A study of replacement algorithms for a virtual-storage computer. In: *IBM Systems journal* 5 (1966), Nr. 2, S. 78–101

[13] Brantner, Matthias ; Florescu, Daniela ; Graf, David ; Kossmann, Donald ; Kraska, Tim: Building a database on S3. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, S. 251–264

[14] Brewer, Eric: CAP twelve years later: How the" rules" have changed. In: *Computer* 45 (2012), Nr. 2, S. 23–29

[15] Brown, Simon: The C4 Model for Software Architecture. (2018). – URL https://www.infoq.com/articles/C4-architecture-model/

[16] Chrobak, Marek ; Noga, John: LRU is better than FIFO. In: *Algorithmica* 23 (1999), Nr. 2, S. 180–185

[17] Cooper, Brian F. ; Silberstein, Adam ; Tam, Erwin ; Ramakrishnan, Raghu ; Sears, Russell: Benchmarking cloud serving systems with YCSB. In: *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, S. 143–154

[18] Dittrich, Jens ; Jindal, Alekh: Towards a One Size Fits All Database Architecture. In: *CIDR* Citeseer (Veranst.), 2011, S. 195–198

[19] Fisher Max, Gessner D. ; Vini, Jaiswal: Using Apache Flink With Delta Lake. (2022). – URL https://databricks.com/de/blog/2022/02/10/using-apache-flink-with-delta-lake.html. – Accessed on June 16 2022

[20] French, Clark D.: "One size fits all" database architectures do not work for DSS. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, 1995, S. 449–450

[21] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google file system. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, S. 29–43

[22] Gilbert, Seth ; Lynch, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *Acm Sigact News* 33 (2002), Nr. 2, S. 51–59

[23] Kane, Frank: *Sundog Eductation - Mastering the System Design Interview*. http://media.sundog-soft.com/SystemDesign/SystemDesign.pdf. – Accessed on July 12 2022

[24] Kleppmann, M.: *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017. – ISBN 9781491903117

[25] Kleppmann, Martin: A Critique of the CAP Theorem. In: *arXiv preprint arXiv:1509.05393* (2015)

[26] Kreps, Jay ; Narkhede, Neha ; Rao, Jun u. a.: Kafka: A distributed messaging system for log processing. In: *Proceedings of the NetDB* Bd. 11, 2011, S. 1–7

[27] Pavlo, Andy: *Intro to Database Systems - Introduction to Distributed Databases*. https://15445.courses.cs.cmu.edu/fall2019/notes/22-distributed.pdf. Fall 2019. – Accessed on July 21 2022

[28] Plattner, Hasso: A common database approach for OLTP and OLAP using an in-memory column database. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, S. 1–2

[29] Reniers, Vincent ; Van Landuyt, Dimitri ; Rafique, Ansar ; Joosen, Wouter: On the state of nosql benchmarks. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, S. 107–112

[30] Shedler, Gerald S. ; Tung, C: Locality in page reference strings. In: *SIAM Journal on Computing* 1 (1972), Nr. 3, S. 218–241

[31] Stonebraker, Michael: MUFFIN: A Distributed Data Base Machine. / CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB. 1979. – Forschungsbericht

[32] Stonebraker, Michael ; Çetintemel, Uğur: " One size fits all" an idea whose time has come and gone. In: *Proceedings of the 21st International Conference on Data Engineering (ICDE 05), Tokyo, Japan*, 2005, S. 5–8

[33] Tanenbaum, A.S. ; Steen, M. van: *Distributed Systems: Principles and Paradigms*. Pearson Education, 2016. – ISBN 9781530281756

[34] Narkhede, Neha: *Exactly-Once Semantics Are Possible: Here's How Kafka Does It*. https://www.confluent.io/de-de/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/. 2017. – Accessed on June 10 2022

[35] Viotti, Paolo ; Vukolić, Marko: Consistency in non-transactional distributed storage systems. In: *ACM Computing Surveys (CSUR)* 49 (2016), Nr. 1, S. 1–34

[36] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauly, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, S. 15–28

[37] Zaharia, Matei ; Das, Tathagata ; Li, Haoyuan ; Shenker, Scott ; Stoica, Ion: Discretized Streams: An Efficient and {Fault-Tolerant} Model for Stream Processing on Large Clusters. In: *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original