KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Scaling Apache Hudi by boosting query performance with RonDB as a Global Index

Adopting a LATS data store for indexing

RALFS ZANGIS

Stockholm, Sweden, 2022

Scaling Apache Hudi by boosting query performance with RonDB as a Global Index

Adopting a LATS data store for indexing

RALFS ZANGIS

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: May 30, 2022

Supervisor: Jim Dowling
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: Skala Apache Hudi genom att öka frågeprestanda med RonDB som ett globalt index
Swedish subtitle: Antagande av LATS-datalager för indexering

Abstract

The storage and use of voluminous data are perplexing issues, the resolution of which has become more pressing with the exponential growth of information. Lakehouses are relatively new approaches that try to accomplish this while hiding the complexity from the user. They provide similar capabilities to a standard database while operating on top of low-cost storage and open file formats. An example of such a system is Hudi, which internally uses indexing to improve the performance of data management in tabular format.

This study investigates if the execution times could be decreased by introducing a new engine option for indexing in Hudi. Therefore, the thesis proposes the usage of RonDB as a global index, which is expanded upon by further investigating the viability of different connectors that are available for communication.

The research was conducted using both practical experiments and the study of relevant literature. The analysis involved observations made over multiple workloads to document how adequately the solutions can adapt to changes in requirements and types of actions. This thesis recorded the results and visualized them for the convenience of the reader, as well as made them available in a public repository.

The conclusions did not coincide with the author's hypothesis that RonDB would provide the fastest indexing solution for all scenarios. Nonetheless, it was observed to be the most consistent approach, potentially making it the best general-purpose solution. As an example, it was noted, that RonDB is capable of dealing with read and write heavy workloads, whilst consistently providing low query latency independent from the file count.

Keywords

Apache Hudi, Lakehouse, RonDB, Performance, Index, Key-value store

Sammanfattning

Lagring och användning av omfattande data är förbryllande frågor, vars lösning har blivit mer pressande med den exponentiella tillväxten av information. Lakehouses är relativt nya metoder som försöker åstadkomma detta samtidigt som de döljer komplexiteten för användaren. De tillhandahåller liknande funktioner som en standarddatabas samtidigt som de fungerar på toppen av lågkostnadslagring och öppna filformat. Ett exempel på ett sådant system är Hudi, som internt använder indexering för att förbättra prestandan för datahantering i tabellformat.

Denna studie undersöker om exekveringstiderna kan minskas genom att införa ett nytt motoralternativ för indexering i Hudi. Därför föreslår avhandlingen användningen av RonDB som ett globalt index, vilket utökas genom att ytterligare undersöka lönsamheten hos olika kontakter som är tillgängliga för kommunikation.

Forskningen genomfördes med både praktiska experiment och studie av relevant litteratur. Analysen involverade observationer som gjorts över flera arbetsbelastningar för att dokumentera hur adekvat lösningarna kan anpassas till förändringar i krav och typer av åtgärder. Denna avhandling registrerade resultaten och visualiserade dem för att underlätta för läsaren, samt gjorde dem tillgängliga i ett offentligt arkiv.

Slutsatserna sammanföll inte med författarnas hypotes att RonDB skulle tillhandahålla den snabbaste indexeringslösningen för alla scenarier. Icke desto mindre ansågs det vara det mest konsekventa tillvägagångssättet, vilket potentiellt gör det till den bästa generella lösningen. Som ett exempel noterades att RonDB är kapabel att hantera läs- och skrivbelastningar, samtidigt som det konsekvent tillhandahåller låg frågelatens oberoende av filantalet.

Nyckelord

Apache Hudi, Lakehouse, RonDB, Prestanda, Index, Nyckel-värde butik

Acknowledgments

I would like to thank Jim Dowling, Fabio Buso, Mikael Ronström, and the rest of Hopsworks team for their assistance in making of this thesis. Owing to their support much of the work was made possible. Furthermore, i am grateful to my examiner Amir Payberah for his continuous guidance in advancing research.

Stockholm, May 2022
Ralfs Zangis

Contents

List of Figures

List of Tables

List of Listings

List of acronyms and abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
AI	Artificial Intelligence
API	Application Programming Interface
COW	Copy On Write
CPU	Central Processing Unit
DDL	Data Definition Language
ETL	Extract, Transform, Load
I/O	Input/Output
JDBC	Java Database Connectivity
JPA	Java Persistence API
LATS	Low Latency, high Availability, high Throughput, and scalable Storage
LTS	Long-Term Support
ML	Machine Learning
MOR	Merge On Read
MVCC	Multiversion Concurrency Control
NDB	Network Database
NoSQL	Not only Structured Query Language
OCC	Optimistic Concurrency Control
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
SQL	Structured Query Language
UUID	Universal Unique Identifier

Chapter 1

Introduction

Currently, there are three commonly used Big Data storage platforms, consisting of the following: Data warehouse, Data lake, and Lakehouse. Each one offers enticing properties for its usage as a database. However, not all of them are suited to satisfy the ever-changing requirements of a modern company[1].

Therefore, in this thesis, we will examine the most recent such architecture: Lakehouse, explaining how it can offer capabilities that were previously unattainable in a single system. While additionally, it makes its adopters more efficient and scalable, thus gifting a competitive advantage.

Furthermore, for the duration of this research, we are going to attempt to enhance a Lakehouse solution. This would be achieved by improving query performance and reducing the overall amount of Input/Output (I/O), through the introduction of a new approach for indexing. In the end, tests would be performed on the resulting product, and observations documented. Thereafter, providing conclusions on the feasibility of the new indexing approach and making it publicly available for consumption of anyone interested.

1.1 Background

There is a multitude of data administrative tools available for the users dealing with the storage and governance of immense amounts of data. The earliest

[1] A VentureBeat interview with Krishna Subramanian (COO of Komprise) about challenges associated with unstructured data can be found at https://venturebeat.com/2021/07/22/why-unstructured-data-is-the-future-of-data-management/.

such architecture was Data warehouse[2]. It accommodated the data-analysis needs of the organizations for many years. However, Data warehouses could not deal with the changes in requirements posed by a fast-moving industry, compelling companies to be efficient and capable of dealing with data that has high variety, velocity, and volume (The three Vs of Big Data).

Altogether, at this time it is estimated that unstructured data comprises about 80-90% of all information stored within companies, with many of them not benefiting from it [1]. That is why a new system was envisioned and developed. It was called a Data lake[3] and while resolving the aforementioned issues it came at the cost of not offering previously supported management properties (such as: Atomicity, Consistency, Isolation, and Durability (ACID) and schema enforcement). Consequently, without a well-established standardization and primitives, Data lakes thus in time could become messy "Data swamps", which are the bane of any data engineer [2].

The already mentioned aspects of the technologies lead to a trend of using both of them in tandem (often called hybrid/two-tier architecture), introducing inefficiency and complexity throughout the system. Therefore, in recent years a new solution called Lakehouse has emerged, combining the best features of the already discussed platforms [3]. This was done by adding a new layer of abstraction on top of the Data lake. The resulting product makes the approach ideal for companies aiming to support business intelligence, analytics, real-time data applications, data science, and machine learning in one platform. The newly available functionality of the technology thus constitutes an attractive proposition for many up and coming industries. In particular, the ones dealing with Machine Learning [4].

Lakehouses offer a comparative performance to their peers and this in a large part can be attributed to the usage of metadata, in particular indexing [5]. While it is not the only performance-defining factor, it most certainly is one of the most crucial. Accordingly, its continuous evolution is the determining characteristic in the execution time reduction. This led to the thesis goal of introducing a new and improved indexing solution, that could be the next step in the maturation and lead to further advances.

[2]Details regarding the unique aspects of Data warehouse can be found at https://www.oracle.com/database/what-is-a-data-warehouse/

[3]Details regarding the unique aspects of the Data lake can be found at https://cloud.google.com/learn/what-is-a-data-lake

1.2 Problem

The ever-changing business landscape has forced companies to be innovative and agile in their choice of Big Data management tools. This has resulted in the change of expectations from a database. The latest approach tries to combine previously unattainable goals in a singular solution. But in return, it heavily relies on a file system for the core component storage. This makes the approach highly scalable but does so at the cost of efficiency and responsiveness.

The problem arising from using the traditional databases for a component such as index storage is that they are often not capable of being distributed and working in a cloud environment. Moreover, they consume more resources to provide improved performance, resulting in a less easily scalable solution. Therefore, it is often not advantageous for the provider to trade resource consumption for speed.

The various indexing solutions used by Lakehouses offer contrasting benefits to their users. Therefore, identifying and implementing the correct approach is imperative to achieving the vital aspects of the adopter's requirements. However, while the current indexes perform well for their intended tasks, it could be an enticing proposition to have an option fixated on minimizing the response time. This left the author with the following question: ”Can the usage of RonDB as a global index for Hudi result in the lowest latency response times among the available indexing engines?”.

1.3 Purpose

The purpose of this project is the introduction of possibly the fastest-performing indexing solution for Hudi. This would benefit the users through improved query execution times and the host company by showcasing the capabilities of their in-house built data store (RonDB). Thereupon, making Hudi more sustainable as well as an increasingly attractive proposition for the consumer.

1.3.1 Ethics and Sustainability

This project does not encounter any ethical issues. Nonetheless, it helps with the sustainability of Lakehouses and in particular Hudi. The reason for this is that more effective metadata management can lead to notable gains in execution times as well as a reduction in resource consumption (through the cutback on the need for I/O operations) [6].

1.4 Goals

The goal of this project is the introduction of a new indexing solution. This has been divided into the following three sub-goals:

1. Improvement of latency/throughput for Hudi.
2. Introduction of the data science community to the capabilities of RonDB.
3. Comparison of the indexing engines.

As an outcome of this research, the resulting code and its performance analyses would be provided. This is intended to encourage further advances in the field of efficient data management.

1.5 Research question

Can we improve the performance of ACID Data lakes by introducing a global index backed by a scale-out in-memory database?

1.6 Research methodology

The research is conducted using an empirical method. More specifically a combination of practical implementation and performance of exhaustive tests on the resulting product. The evaluation of improvement gained by comparing it to the existing solutions. The reasoning for the approach is that it would provide a quantifiable answer to the previously posed question. This is not obtainable through Conceptual research, making it impractical for the task at hand.

1.7 Delimitations

The focus of this project is placed on improving Lakehouse's performance and documenting results. With a particular interest in response times. Therefore, only a single Lakehouse implementation will be thoroughly investigated. Furthermore, while alternative databases used for indexing would be considered, they would not be implemented due to time constraints. Thus, leaving it as well as other prospects of the approach for future research.

1.8 Structure of the thesis

The research is structured following the traditionally used thesis model. Starting with Chapter 1 providing the basic information about the goals. Chapter 2 presents the relevant background information as well as previous work. Chapter 3 bestows the chosen method for the problem resolution. Chapter 4 expands on the implementation, while Chapter 5 informs of the observations. Finally, Chapter 6 reflects on the results and proposes future work.

Chapter 2

Background

This chapter introduces the concepts used throughout the following report in greater detail. It is done by examining the Lakehouse, its implementations, how they differ, and what is the chosen approach. After that, a more in-depth analysis notifies the reader of the necessary background knowledge. Subsequently, proposing, justifying, and describing the tools for accomplishing the thesis goal. Finally, related work is described.

2.1 Lakehouse

As already briefly introduced in the Section 1.1, Lakehouses are the next logical step in the evolution of data storage. They allow users to share benefits from Data lakes, by having low-cost storage that supports unstructured data, and Data warehouses, by offering management features as well as great performance.

2.1.1 Lakehouse characteristics

Lakehouse solutions through their peculiarity might deviate in their feature set. However, they could generally be characterized as offering the following attributes:

- Simple and reliable data governance: Previously, to obtain the Lakehouse capabilities, the users had to perform several Extract, Transform, Load (ETL) actions between Data lake and Data warehouse, making the system overly complex and error-prone.

- Up to date data: The ETL actions between multiple systems introduce staleness in the data being served.
- Directly accessible data: The Machine Learning (ML) solutions need a lot of data to function, which is most efficiently retrieved using files in open formats.
- ACID capabilities: Provide guarantees previously not obtainable in the Data lake, but expected in the Data warehouse.
- Low cost: All data is stored in a single solution, not requiring unwarranted duplication.
- Data versioning: Users could perform time travel on the data, retrieving information as it was in the past or obtaining changes that have occurred since the specified point-in-time.
- Great performance: Features such as metadata, indexes, z-ordering, and cache allow Lakehouse to benefit from query optimization, which offers comparable performance to the Data warehouse.

2.1.2 How Lakehouses work

The Lakehouses achieve the aforementioned aspects of their approach through their reliance on low-cost object stores and open file formats (Parquet, ORC, and Avro) [7]. This means that they are built on top of the existing Data lake solution, by introducing a new data management layer that enforces the capabilities of the Data warehouse and exposes Application Programming Interface (API) for interacting with data (Figure 2.1). Therefore, often it is a relatively simple process to migrate the existing solution to Lakehouse.

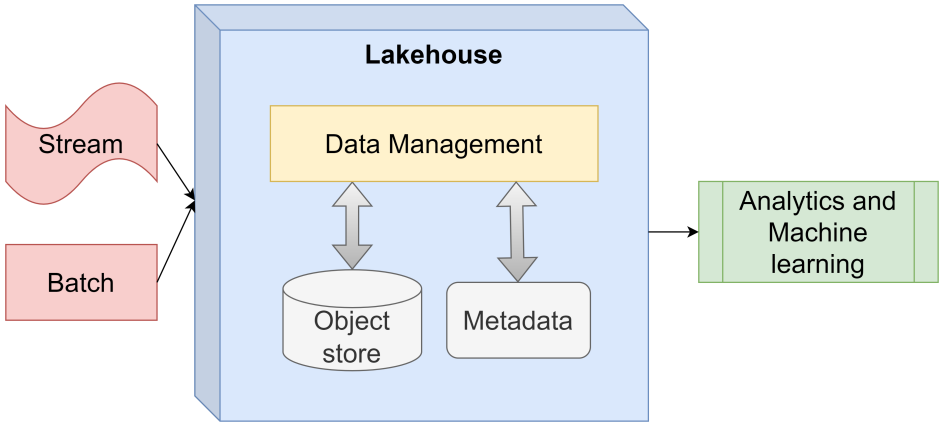


Figure 2.1: Basic Data Lakehouse.

2.1.3 Lakehouse solutions

The most well-known projects offering Lakehouse prospects are Delta Lake, Apache Iceberg, and Apache Hudi. They all have a similar premise (Section 2.1) and were developed nearly simultaneously, with only a slight deviation in their capabilities and implementations [8, 9], depending on the underlying reason for their inception. With possible convergence of their proficiencies, as the technologies continue to evolve [10].

2.1.3.1 Delta Lake

Delta Lake[1] (often referred to as Delta) is an open-source Lakehouse solution allowing for batch and streaming processing [11]. It was developed by Databricks and is designed to integrate with Apache spark. Delta Lake relies on the transaction log (ordered record of every action) as a single source of truth [12]. As a result, using it in tandem with Optimistic Concurrency Control (OCC), Delta can offer ACID properties (The conflict resolution of the overlapping processes in most situations is worked out silently). Altogether, this Lakehouse implementation is well documented and feature-rich. However, its capabilities lack when compared to the commercial version (Delta Engine) [13]. Therefore, it is best suited for clients of Databricks and users of Spark.

[1] Delta Lake GitHub repository can be found at https://github.com/delta-io/delta

An example of the directory structure present in Delta Lake [14, 12] can be seen in Listing 1.

Listing 1 Delta Lake schema example.

```
system
├── _delta_log
│   ├── 000000.json
│   ├── 000001.json
│   ├── ...
│   ├── 000010.checkpoint.parquet
│   └── ...
├── partition_column_name
│   ├── part-00000-(random_UUID)-c000.snappy.parquet
│   └── ...
└── ...
```

2.1.3.2 Iceberg

Iceberg[2] is a solution that was established to rectify the issues (performance, scalability, and manageability) of Apache Hive tables in large Data lakes [15]. The development of it started at Netflix, which subsequently open-sourced it in 2018, and as of 2020, it is a top-level project of Apache Foundation. Iceberg was designed to be engine agnostic and consider everything as a metadata operation [16]. As a result, it boasts excellent reading performance and supports OCC. However, up until recently, it had limited support for modifications such as deletes [17], meaning that it is an actively evolving solution. Apache Iceberg is best suited for huge tables that could be comprised of a large number of partitions or require frequent schema transformation.

An example of the directory structure present in Apache Iceberg [14, 18] can be seen in Listing 2.

[2] Apache Iceberg GitHub repository can be found at https://github.com/apache/iceberg

Listing 2 Apache Iceberg schema example.

```
system
├── data
│   ├── partition_column_name
│   │   ├── 00003-3-(random_UUID)-00001.parquet
│   │   └── ...
│   └── ...
└── metadata
    ├── version-hint.text
    ├── V1.metadata.json
    ├── ...
    ├── snap-1771790970144017284-1-(random_UUID).avro
    ├── ...
    ├── (random_UUID)-m0.avr
    └── ...
```

2.1.3.3 Hudi

Hudi[3] (an acronym for Hadoop Upserts Deletes and Incrementals) was developed in Uber to deal with the glaring inefficiencies and complexity of scaling using at the time available data management architectures. From the very beginning this was done by having data, indexes, and metadata as core storage components [19]. This architecture supports incremental changes over columnar file formats, offering both Multiversion Concurrency Control (MVCC) (using central log and one concurrent writer) and OCC capabilities [20]. Furthermore, the solution awards two types of tables Copy On Write (COW) and Merge On Read (MOR), which allows for it to be flexible in how data is managed for maximum read/write efficiency[4]. Therefore, the best way to describe Apache Hudi is by calling it a "Streaming Data lake Platform" [21]. Currently, Hudi is a top-level project in Apache foundation, offering an ever-increasing selection of features [22]. It is a great option for users who would

[3] Apache Hudi GitHub repository can be found at https://github.com/apache/hudi

[4] Hudi concepts can be found at https://hudi.apache.org/docs/concepts/

like to have support for a multitude of tools out of the box (primarily relying on Apache Spark and Apache Flink for data processing) and place particular interest in stream ingestion workloads.

An example of the directory structure present in Apache Hudi [14] can be seen in Listing 3.

Listing 3 Apache Hudi schema example.

```
system
├── partition_column_name
│   ├── (random_UUID)-0_0-43-62_20220409202006131.parquet
│   ├── ...
│   └── .hoodie_partition_metadata
├── ...
└── .hoodie
    ├── archived
    ├── .aux
    ├── .temp
    ├── hoodie.properties
    ├── 20220320163857541.commit
    ├── 20220320163857541.commit.requested
    └── ...
```

2.1.4 Lakehouse choice

Throughout the following text, the focus will be placed on analyzing Apache Hudi in particular. This decision was made due to the limitations of open-source Delta Lake and the relative lack of documentation for Apache Iceberg, which could stifle the progress. Furthermore, Hudi has an active community[5] and is used by the thesis host institution, granting access to resources and qualified staff.

[5] List of notable Hudi community members can be found at https://hudi.apache.org/powered-by/

2.2 Hudi platform

Hudi is not only a data orchestration apparatus but also a solution that comes with inbuilt services. For example, users have out-of-box support for tools used for ingesting, ETLing, cleaning, and more. Therefore, the best way to call it is a platform [21].

As already mentioned in the previous section, Hudi relies on data, indexes, and metadata for its core functionality. Therefore, knowledge of the structure and the associated terminology is a prerequisite for working within the technology stack and understanding the terms used in subsequent text. The following subsections provide a brief introduction to data management.

2.2.1 Hudi table format

At the very beginning Hudi was designed to work using the Hive table format, which was the de-facto standard for the industry. However, this approach did not bode well for the workloads that Hudi was anticipated to accommodate. Therefore, in time a new approach was developed, solving scaling challenges and bringing in additional functionality.

This was done by creating a new table format. Similar to a file format the table format is used for standardizing contents. Hudi format can be considered as a representation of the table's metadata and it uses different file formats internally to achieve this. Hudi provides a file layout of the table, the table's schema, and metadata tracking changes to the table.

2.2.2 Data structure

Data in Hudi is stored in multiple files, that are partitioned by the underlying information. This is the case since a change/read of a single table entry would not require an operation on the entire data set, thus reducing the amount of I/O.

The directory structure in Hudi:

1. Base file: Contains data after compaction, usually saved in a parquet file format [23].

2. File slice: Includes the base file and any associated delta log files (files containing changes, used when operating using MOR). Each file slice represents a state of the sub-set of records at a certain time.

3. File group: Possesses a pre-defined number of file slices (as determined by the configuration) who can all be identified by the same ”file id”.

4. Partition: Retains all file groups that share some characteristics. A table could contain multiple partitions which are represented as folders saved under the base path. For example, products could find themselves in different partitions based on the country of origin.

Figure 2.2 illustrates the Hudi structure using diagram for better understanding.

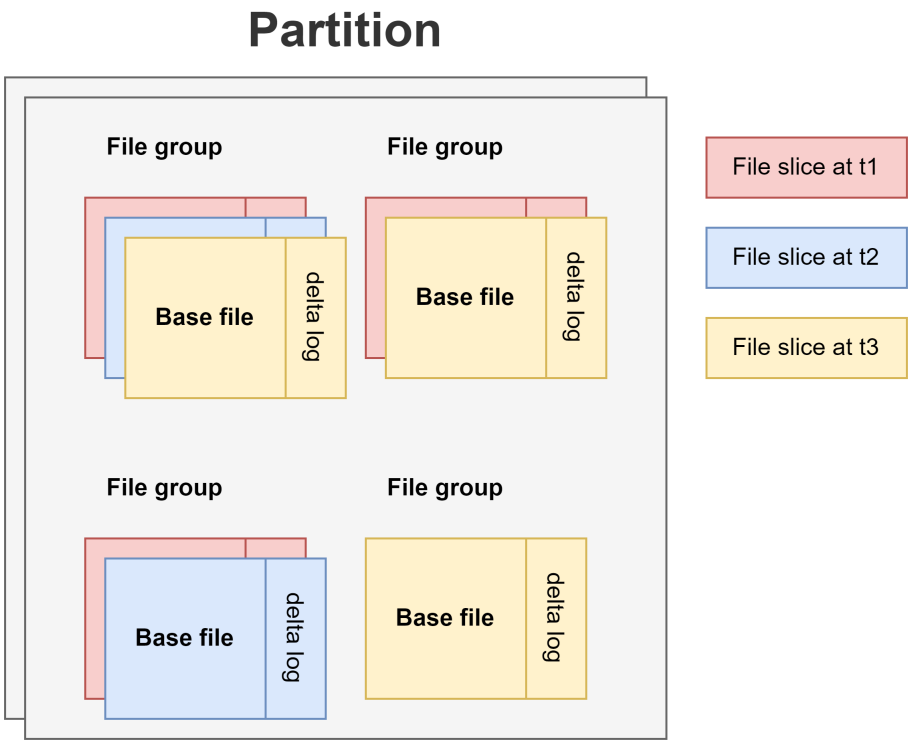


Figure 2.2: Data structure of Hudi.

2.2.2.1 Base file

The contents of a typical Base file in Apache Hudi can be further subdivided into three distinct parts [24]:

1. Record metadata
2. Record data
3. Additional metadata (in file footer)

Parts (1) and (3) make up, what is often known as the "Hudi skeleton". It is essential for supporting Hudi primitives (upserts and incremental pulls). When migrating a massive data set these components can be stored in separate files and referenced for efficiency [24, 25].

Figure 2.3 displays this information with different colours identifying the separation of function (orange = row metadata, red = data, blue = footer).

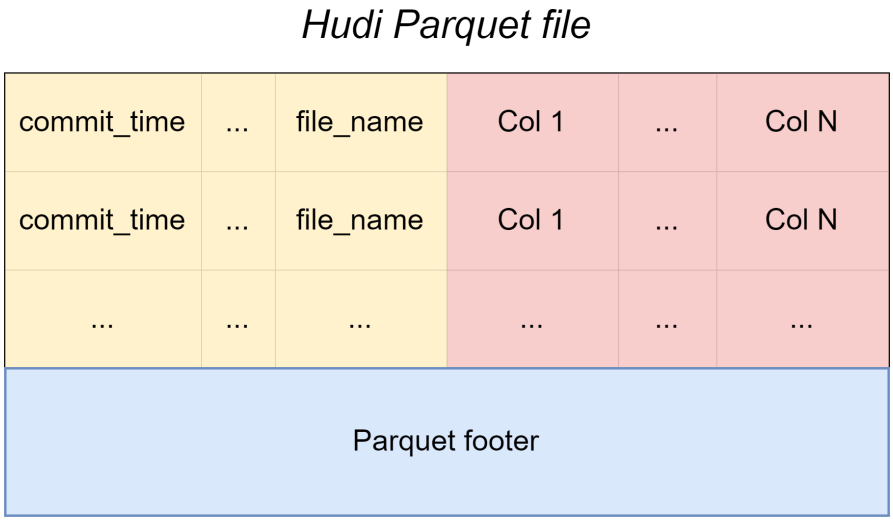


Figure 2.3: Structure of Parquet file in Hudi.

The name of the file itself conveys information as well [14]. Accordingly, the subsequent text lists its parts and provides a template for a parquet file name in Hudi.

1. Arbitrary Universal Unique Identifier (UUID) (used to uniquely identify file group)
2. Number of updates in file group

3. Partition id
4. Spark stage id
5. Task attempt id
6. Commit time

Template parquet base file name in Hudi (replace the number with a corresponding entry in the enumeration provided previously):
`1-2_3-4-5_6.parquet`

2.2.3 Metadata

Hudi achieves the desired properties of the solution by heavily relying on metadata. This information is stored at the record, file, partition, and table level.

Record metadata

As already discussed in Section 2.2.2.1 Hudi maintains per record metadata that is stored in the base file. Such information is used for performing various kinds of record-related operations.

Currently each record has five Hudi metadata fields (with more in planning stage) [25]:

- `_hoodie_commit_time`: Time of the latest mutation
- `_hoodie_commit_seqno`: Number of actions that have impacted entry (necessary in incremental pull)
- `_hoodie_record_key`: Key used for row identification
- `_hoodie_partition_path`: Partition-Path of the record
- `_hoodie_file_name`: Name of the file used for record storage

Base file metadata

This information is stored in the file footers and is used to describe data within a file. It includes schema, statistics, and indexes. The inclusion of these facts allows for actions such as data-skipping to dramatically reduce the amount of data that needs to be processed.

Partition metadata

Each partition maintains basic information that is used to describe it (such as creation time and partition depth). It is stored in the partition folder within a file named `.hoodie_partition_metadata`.

Table metadata

It is stored in a separate folder called `.hoodie`, which contains information about the configuration of the table as well as the timeline.

Configuration Table configuration is maintained in a single `hoodie.properties` file, which specifies the table type, version, and more.

Timeline The timeline is also known as the event log. It functions as the source of truth, regarding the current state of the system [22]. The log is composed of actions that are performed on the table at different instants (time at the server). Figure 2.4 illustrates how it functions using batch upsert action as an example.

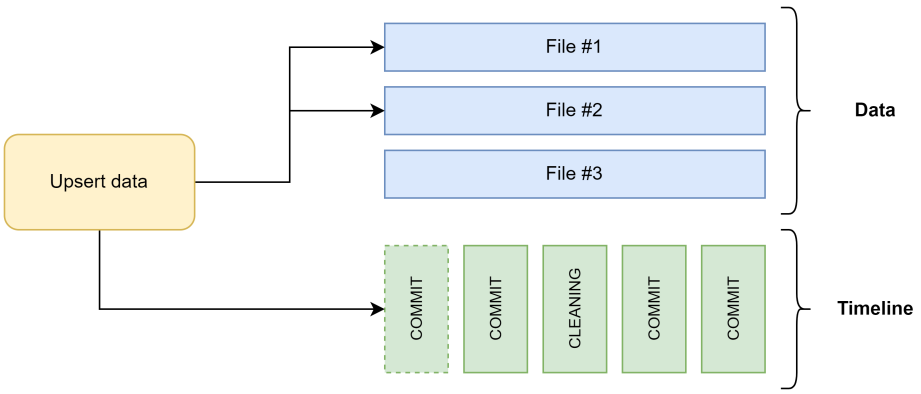


Figure 2.4: Hudi timeline.

Actions in the timeline could be of the following types:

- Commit: Write a batch of records to the table.
- Clean: Remove old versions of files.

- Delta commit: Write a batch of records to the table's delta logs.
- Compaction: Merge delta logs into a base file.
- Rollback: Remove partial changes in case of Commit/Delta commit failure.
- Savepoint: Mark certain file groups to not be cleaned.

Each action changes status during its execution, iterates over the three states:

1. Requested
2. In flight
3. Completed

Log entries are retained for a specified time period or until action is invoked archiving them. Even if information in the timeline is lost, it is often possible to retrieve the past state of the table using only file slices or delta logs (in the case of MOR table) [21]. At the time of writing this document indexed timeline and infinite retention of versions are in the roadmap, but not yet achieved.

Metastore Server

File access in Hudi incurs inefficiencies in form of directory listing (especially in cloud storage systems) and performance limitations of name nodes. That is why additional metadata can help in the management of the platform as a whole.

A metadata table is an example of such a solution. It proactively maintains the list of files and reflects the current table situation. This solution functions similarly to the MOR table, allowing for low write amplification. The information is stored in HFile file format providing indexed lookups of all directory entries [21].

However, the metadata table does not provide a unified view of the system as a whole, rather managing the metadata of a single instance. Therefore, Hudi is working towards a solution that could store all metadata and provide services for using it (in essence taking a step closer to the Iceberg approach, where everything is a metadata operation). This approach is called Hudi Metastore Server and it will be made to be compatible with Hive metastore so that other engines can access it without any changes [26].

2.2.4 Indexing

The indexes enable the management of data at a large volume by offering quick retrieval of records. There are many indexing structures [27] that could be implemented. However, in Hudi they work by mapping the record key with the optional partition path to the file group. Therefore, this allows for only the necessary files to be operated on while avoiding examination of all existing data, making a huge difference when retrieving/working within voluminous tables[6].

2.2.4.1 Index types

Before delving into the index implementations it is worth investigating the different types of indexes supported in Hudi [28]. Currently, there are two such practices, with each one guaranteeing different properties:

- Global- Enforces key uniqueness for all partitions of a table. This means indexes grow proportionally to the table size. Data access occurs using the following relation: **record key -> (partition, file id)**
- Local- Enforces key uniqueness within a single partition of a table. This means indexes grow proportionally to the specific partition. Data access occurs using the following relation: **(partition, record key) -> file id**

Index implementation might not necessarily support both of these types. Therefore, the developer must be aware of the differences and choose the solution accordingly.

2.2.4.2 Indexing options

Hudi provides multiple indexes [28], supporting: Bloom Index, Simple Index, and HBase Index. In the following paragraphs, the most suited use cases for each will be provided. Additionally, summarizes their inner workings.

Simple Index A simple index as the name suggests is the most basic indexing approach. It works by joining the incoming records against the keys extracted from the table in storage. Information about the index is stored in the base files and is maintained by Hudi. It is best suited when having a lot of random modifications to a table [28].

[6]Hudi performance observations can be found at https://hudi.apache.org/docs/performance/

Bloom Index Bloom index is the default indexing approach in Hudi and it works by saving information in the footers of all base files. It is based on the Bloom filter, which is a space-efficient probabilistic data structure that uses hashing to identify the absence of a value [29]. This approach can result in False-positive matches, but it ensures that there would be no False-negatives. The configuration[7] of the False-positive ratio can be altered, with higher precision coming at the cost of increased index size. Its most appropriate use case is when there is a need to deal with the late arrival of data or duplicates [28].

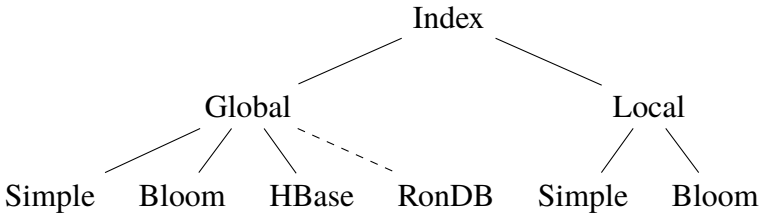
HBase Index This is a straightforward indexing solution, where indexes are stored in an external database [30]. This approach offers the best performance out of all index implementations. However, it is more resource-demanding than storing information in the file system. Moreover, this solution is only capable of being a global index (unlike the aforementioned approaches, who can be also local). This makes HBase the solution of choice for users who are willing/capable of bearing the increased cost of this approach.

2.2.4.3 Indexing conclusions

Previous sections introduced the current indexing capabilities of Hudi (General summary found in Listing 4). Based on this information it can be concluded that there is a growth prospect. This is demonstrated by new and better indexes presently materializing that allow for efficient record level indexing while relying on the file system (not requiring a scan of all files) [31]. For example, the Hudi team envisions a speedier Bloom index, by relying on metadata tables to track Bloom filters [28]. Furthermore, adding a new indexing solution has been encouraged by an architecture design that simplifies the introduction of a new approach. Therefore, the current offerings are in a constant struggle for survival, as a better option could in time replace established approaches.

[7] Hudi configuration properties can be found at https://hudi.apache.org/docs/configurations/

Listing 4 Apache Hudi index options.



2.3 Databases

For the index to function, it has to be stored. At the moment, this is done within a file system or in an external process like a database. The usage of a database improves performance noticeably. Therefore, it is the approach of organizations who believe that the benefits of such a system outweigh its cost [32]. However, many databases could be used for this task, likely differing in performance, resource consumption, and scalability. The subsequent text evaluates and proposes the best solution for offering indexing with the lowest latency.

2.3.1 Database options

Data storage is a massive field, as a result, many databases could be investigated. To reduce the number of considerations we should look into their distinguishing features. In-particular indexing in Lakehouse requires horizontal scaling and low latency (for adding and getting values), with the solution preferably being cloud-native and free to use.

This significantly reduces the number of databases to be examined. However, there are still several solutions fulfilling the role. Therefore, attention will be placed on the most popular. Particular interest is fixated on those that can benefit from mapping parts/whole of the database into volatile memory for faster workloads [33]. The following subsections list the most suitable candidates (including an overview of the current approach- HBase).

HBase

The HBase[8] is the current database choice for indexing in Hudi. It is an open-source, non-relational, distributed column-oriented store, offered by the

[8]Introduction to HBase can be found at https://aws.amazon.com/big-data/what-is-hbase/

Apache foundation. In general, it is a highly scalable and performant solution, that has a low cost of ownership.

Redis

Redis is likely the most commonly used key-value store[9]. It is an in-memory data storage platform[10], whose use cases include: analytics, search, ML, and Artificial Intelligence (AI). Moreover, it supports a variety of programming languages and could be deployed virtually anywhere. However, the enterprise version of the product is superior to the open-source, at least in the case when scalability is the main concern.

MongoDB

Possibly the most well-known Not only Structured Query Language (NoSQL) database is MongoDB. It follows the document store paradigm, which offers an intuitive way to work with data[11]. Additionally, the product is known for its simplicity and consistency in developer experience. However, many capabilities are not available for the community users, requiring financial investment to gain access to an in-memory storage engine, advanced security features, and more.

Memcached

Memcached is an in-memory key-value store, originally intended for caching[12]. It is a fully open-source solution, that is commonly compared to Redis. The most notable differentiation between the two is Memcached's tendency to be able to handle more concurrent operations while being designed for workloads with smaller quantities of data.

---

[9] Ranking of database management systems by popularity can be found at https://db-engines.com/en/ranking

[10] Introduction to Redis can be found at https://aws.amazon.com/redis/

[11] Introduction to MongoDB can be found at https://www.mongodb.com/what-is-mongodb

[12] Introduction to Memcached can be found at https://aws.amazon.com/memcached/

Cassandra

Cassandra is a wide-column store that similarly to HBase is a part of Apache foundation [34], meaning that it is fully open source[13]. What differentiates Cassandra from HBase is that it follows a distinct architecture, which emphasizes Availability over Consistency [35].

RonDB

The only relational database considered for the task is RonDB. It is an open-source scale-out in-memory key-value store based on MySQL Network Database (NDB) Cluster. This data store is optimized for use in modern cloud settings, offering Low Latency, high Availability, high Throughput, and scalable Storage (LATS) capabilities to its users [36]. The design of this data management system was tailor-made for reliability and responsiveness, ensuring the most crucial aspects of any indexing solution[14].

RonDB is powered by an NDB engine, which offers high availability and data persistence while preferring storage of information in memory rather than on disk. For it to function it requires the use of three types of cluster nodes[15] (node in this context is not the same as a computer, rather single computer can have multiple nodes, therefore its more akin to a process):

- Management node: used for administration of all nodes in the cluster (starting/stopping, initiating backups, etc.).
- Data node: responsible for storing information.
- SQL node (API node): responsible for accommodating the interaction from the clients.

2.3.2 Database choice

HBase offers the best response times right now, but it might not be the best option for this role. Other databases with more preferable attributes could in time replace the current technique. While any of the aforementioned data storage platforms could form the basis for an indexing solution. This research

[13] Introduction to the Cassandra can be found at https://cassandra.apache.org/_/cassandra-basics.html

[14] Detailed list of RonDB capabilities can be found at https://docs.rondb.com/rondb_special/

[15] NDB Cluster Core Concepts can be found at https://dev.mysql.com/doc/mysql-cluster-excerpt/8.0/en/mysql-cluster-basics.html

will subsequently focus on the employment of RonDB for this role. The reason for its adoption is that it does not limit the functionality based on the financial involvement of the parties, it offers industry-leading performance, and the host company has extensive experience in the solution. A more detailed analysis of its adoption can be found in Section 2.4 where the previous work was used to derive the comparative performance of the candidate approaches.

2.4 Related work

Researchers have attributed the development of a Lakehouse to the challenges encountered by the users of the preceding solutions [37]. This experience has culminated in products that could achieve ACID Table Storage over Cloud Object Stores for improved efficiency [38]. Therefore, the feasibility of the undertaking was attributed to it offering comparative management functions to Data warehouses while providing the underlying capabilities of Data Lakes [7].

Past work has associated optimization mechanisms (such as indexing, caching, metadata, and entry ordering) to Hudi being able to provide low latency for data access across and within partitions [39]. The index performance relies on fast and efficient storage, with the best response times gained through the usage of an external database [32]. The relatively recent paradigm shift for in-memory databases provides an order of magnitude faster executions [33], making the new technique ideal for satisfying the indexing needs.

A performance evaluation of the most popular in-memory databases proved that key-value stores like Memcached and Redis outperform their rival data storage models in basic operations as well as in the overall memory efficiency [40]. Furthermore, a different study expanded on this by verifying that HBase is incapable of competing with Redis in the performance of workloads that are expected for an indexing solution (loading, updating, and reading) [41]. Therefore, making Redis a plausible candidate. However, it lacks in comparison with RonDB which offers better performance (even on single thread) due in large part to evolution in architecture [42, 43].

Chapter 3

Method

This chapter describes the research method for identifying the feasibility of RonDB as a global index in Hudi. It begins by outlining the research process in Section 3.1. Following that, Section 3.2 specifies the research paradigm. Section 3.3 defines data collection, while the testing environment is described in Section 3.4. The quality measures are portrayed in Section 3.5. Finally, Section 3.6 details the data analysis approach and Section 3.6 describes the documentation of the end product.

3.1 Research Process

The thesis organizes work into steps shown in Figure 3.1. The inquiry begins with the problem identification and its subsequent investigation. Following this, a solution to rectify the issue is proposed, with a particular focus on providing an action plan. Subsequently, the software is developed and iteratively improved until deemed sufficient in its characteristics. Afterwards, the product is exposed to a set of tests that are used to determine the performance, later documenting the implementation details and analyzing results to derive conclusions.

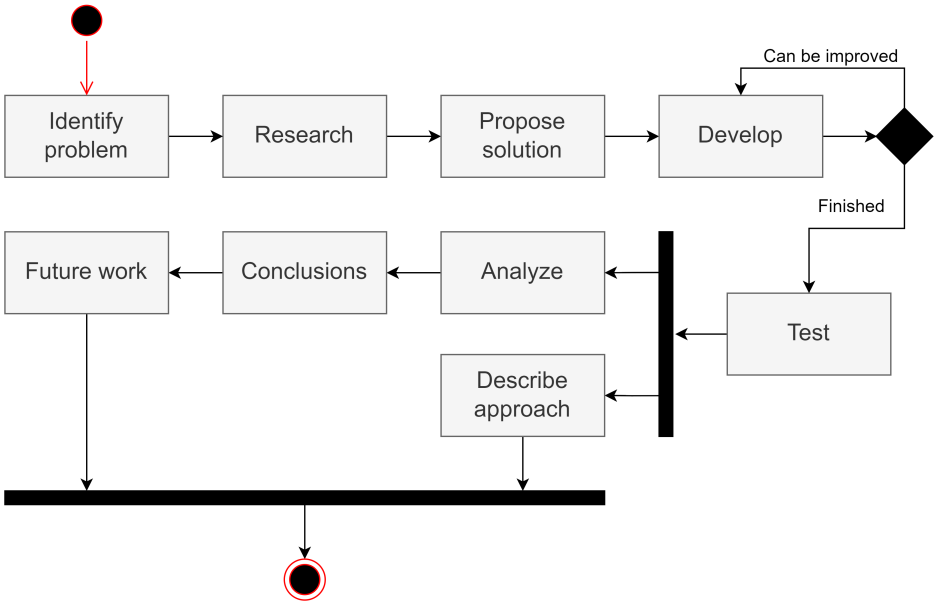


Figure 3.1: Research Process

These steps were chosen due to their capacity of offering a product that is capable of fulfilling the research objectives while simultaneously resulting in a highly performant implementation. Moreover, the iterative improvements allow for a predictable study cycle and the establishment of a more comprehensive collection of future work proposals.

3.2 Research Paradigm

The answer to the study is obtained by following the positivist research philosophy[1]. This quantitative approach is visualized in Figure 3.2.

[1]Explanation of research paradigm can be found at https://www.helpinproject.com/research-paradigm/

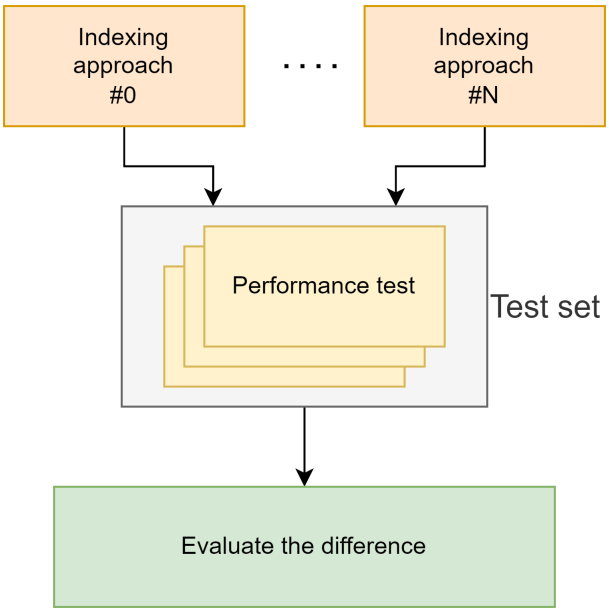


Figure 3.2: Research Paradigm

The diagram shows how observations of reality (gained through testing) are used to conclude the comparative performance through reliance on the existing indexes for the basis of identification.

Contrasting implementations allow for the obtainment of results with actionable information. As observations without the basis of comparison yield outcomes with no point of reference. The subsequent evaluation of the data encompasses various analytical and visualization activities, which in return can help to induce the favorable/ominous properties of each solution. The resulting conclusions, therefore, allow for the recognition of the best-suited application of a technique.

3.3 Data Collection

The collection of data is going to be realized by employing exhaustive testing of current and newly implemented approaches. This is done by surveying response times for use cases mandating a lookup of specific data entries using global indexes.

As already discussed in the previous sections, Hudi is designed to

accommodate petabytes of data. Therefore, it is difficult to create workloads that could adequately represent the solution's usage in any particular scenario, since it theoretically has no upper bound of storage volume. As a result, this examination would identify and work within a preset array of data masses and batch sizes chosen to most sufficiently represent performance variations.

Throughout the inquiry, the tables would be of COW type and the data used for testing (both for the initial table composition and the subsequent sequential changes) will be identical for all solutions on any specific workload. This means that at the start of any particular test the only differentiating factor is the selected indexing implementation.

Additionally, to provide results that are more consistent with reality the tests would use centrality measures implemented on top of multiple executions.

3.4 Test environment

The testing environment consists of a newly installed system with all unwarranted services shut down/disabled during the experimentation to lower the interference. The work is organized using Jupyter notebooks and tests are executed on top of a running solution to avoid the "cold start" problem. Moreover, the external databases are set up using the latest Long-Term Support (LTS) releases (RonDB[2] : 21.04.1 and HBase[3]:2.4.11) and default configuration. Furthermore, the environment includes Spark (version 3.2.1).

Table 3.1 describes the specification of the device used for testing in detail.

3.5 Assessing quality

The quality of the method is assessed by investigating its validity and reliability. Validity ensures that the approach grants answers to the correct questions, while reliability guarantees that results are consistent.

To assure that data used for analysis is valid, the method has to provide answers to the following questions: How long does it take to execute the query? and What impact do different variables play on the performance of any given

[2]The RonDB solution used during testing can be found at https://repo.hops.works/master/rondb-21.04.1-linux-glibc2.17-x86_64.tar.gz

[3]The HBase solution used during testing can be found at https://dlcdn.apache.org/hbase/2.4.11/hbase-2.4.11-src.tar.gz

Table 3.1: Device specification.

Operating System (OS)	Ubuntu (Linux)
Version	20.04.4 LTS
System type	64-bit
Central Processing Unit (CPU)	11th Gen Intel(R) Core(TM) i5-1135G7
Random Access Memory (RAM)	32.0 GB
Capacity	200.0 GB
Storage unit	SSD

index? It is expected that results will produce identifiable groupings, that can uniquely be attributed to the use of a certain indexing approach.

For a solution to be considered reliable its use cases have to produce results with high precision, be reproducible and have a low standard deviation. A slight variation was expected due to factors outside of test conductors' control (such as garbage collection, OS operations, and more). However, to ensure the highest possible reliability the experiments should avoid performing intermittent steps during their execution (like communication over the internet or data generation during test).

3.6 Planned Data Analysis

During the preliminary study, a sample size that is needed to acquire the representative value will be determined. The subsequent data analysis would be conducted by observing the differences between the solutions. This would involve looking at statistics (such as min, max, average, and more) and examining graphs. Results, as well as the derived conclusion, will subsequently be made available for peer review.

3.7 System documentation

The resulting solution will be annotated with all necessary commentary, to make its adoption as easy as possible for prospective users. This includes, but is not limited to, comments provided in the code, short guidelines on how to deploy it (Section 4.4), and initiation on its proper usage (Section 4.5). Furthermore, to manage the user expectations of index resource consumption,

the final NDB size reports are affixed (Appendix A).

Chapter 4

Solution

The solution's development was an iterative process, which involved gaining familiarity with the current software, designing an approach for adopting this knowledge in the new problem area, expanding upon it by optimizing, and finally testing. As a result of this endeavor, two independent solutions were developed, with each one focusing on obtaining different characteristics that are desirable to the users. The details regarding the practical aspects of this undertaking will be granted in the subsequent text.

4.1 Software design

Even though Hudi has been developed to be engine agnostic, not all clients are equal. Subsequently, the focus was placed on Spark, as it offers the most feature-rich interface, owing to it being used since the software's inception [44]. Other, clients were designated as future work due to their presence deemed to be non-crucial for the thesis goal.

The underlying implementation is based on the aforementioned HBase indexing solution, which similarly relies on an external database. To implement a new approach the concept had to extend `HoodieIndex` abstract class, which works as a template. In essence, the indexing exposes three main methods (Table 4.1) and it is managed by the user-provided configuration, using pre-established defaults in the case of the absence of this data.

Table 4.1: The main Hudi index interface elements.

Name	Description
`tagLocation`	Reading database and appending the current location (if present in the database) to the provided HoodieRecord.
`updateLocation`	Modifying the index storage unit by inserting, updating or deleting records. (by default updates are disabled for database based indexes to reduce load on the server)
`rollbackCommit`	Undoing any changes that follow certain timestamp. (by default disabled to reduce load on the server)

4.2 Implementation

As already discussed the solution had to expose certain endpoints so that it could be integrated into the Hudi. However, the choice of how to store and interact with this data was left to the developer. Therefore, two approaches were developed, with design and implementation favoring either efficient storage, adaptability, and security or responsiveness. This choice was made consciously as it is a balancing act, each approach having associated trade-offs, with the goal being the identification of the best practice.

The following subsections list the implementation details of RonDB solutions, with each approach identified by the client used for communication (Java Database Connectivity (JDBC) or ClusterJ).

4.2.1 Solution using JDBC

This solution stores information in a normalized form, meaning that there are multiple tables specializing in maintaining certain aspects of knowledge, while referencing/being referenced by entries in different structures. Therefore, this approach plays on the strengths of the relational database, allowing for efficient storage of data. However, it comes at the cost of execution time, as now operations on a single entry often require the lookup of data in different tables.

Another unique detail of this proposal is its reliance on JDBC API for exchanging information. This makes the code easier to adapt to different Structured Query Language (SQL) databases (only possibly requiring a change of driver).

Overall, this is the solution that likely has the most appealing characteristics for most use cases. Its only shortcoming is a slight overhead, due to the existence of multiple tables and intermediate communication through the MySQL server.

The code related to its implementation can be found under the class named `SparkHoodieRonDBIndex` and it could be used by simply specifying `hoodie.index.type` property value as `RONDB`.

The schema used for storing index-related information is showcased in Figure 4.1.

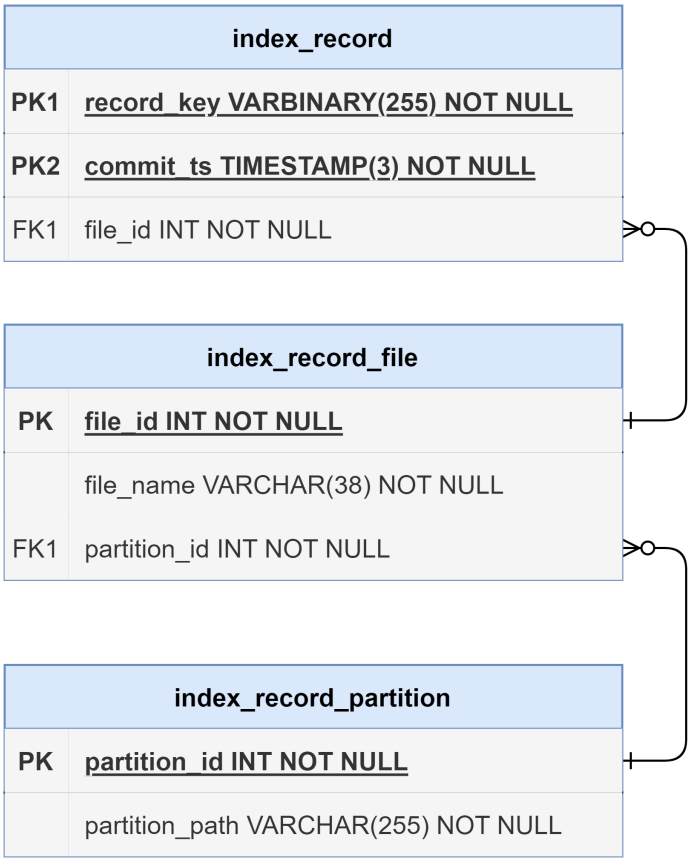


Figure 4.1: Entity relation diagram for RONDB_JDBC

4.2.2 Solution using ClusterJ

This approach, as opposed to the preceding implementation, disregards the storage efficiency clearly favoring performance. It can be seen in Figure 4.2, where all information is stored in a single table, thus allowing for duplicate values for information such as `partition_path` and `file_name`.

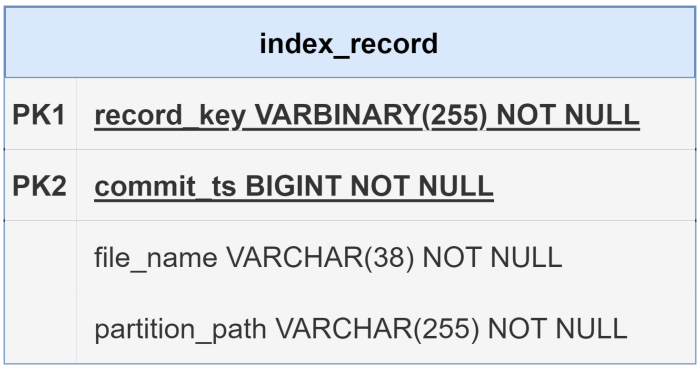


Figure 4.2: Entity relation diagram for RONDB_CLUSTERJ

The improved performance of this approach is not only obtained through having more self-contained entries but also by using more specialized technologies. The tool picked for this task is ClusterJ. It functions on a lower level than JDBC, directly communicating with the NDB cluster, bypassing the MySQL Server entirely [45].

ClusterJ is a relative niche solution that works similarly to other object-relational mapping persistence frameworks (such as Java Persistence API (JPA)) [46]. However, it is limited in its capabilities, offering only the basic interactions with row data, not possessing any functionality to perform Data Definition Language (DDL) actions (such as table creating), lacking security, and relationships between tables[1]. Therefore, in addition to ClusterJ SQL queries are run during initialization using JDBC to ensure that data structure is defined.

This approach is likely unfeasible for many use cases owing to the already listed limitations it incurs. However, its main goal was never explicit real-world usage, but rather obtaining a product that emphasizes speed over

[1]MySQL guide for NDB Cluster API can be found at https://dev.mysql.com/doc/ndbapi/en/

everything else. Working as an additional point of reference used throughout the result analysis.

The code related to this endeavor can be viewed inside the `SparkHoodieRonDBClusterIndex` class and could be used by specifying the `hoodie.index.type` property value as `RONDB_CLUSTERJ`.

4.2.3 Configuration

The management of indexing is performed through configuration. Both JDBC and ClusterJ indexing implementations use a class named `HoodieRonDBIndexConfig` for these purposes. It offers an interface for retrieving user-provided information or offering defaults in case of its absence. Owing to this, the user is not expected to alter code to make adaptations for specific needs. Although, it is impossible to reconfigure the data structure, adjustment of which would encompass revision of code.

4.3 Development

Before delving into the problem resolution, the current approach for handling indexes was investigated. This reduced the number of changes needed in the project as well as keeping the interface uniform.

The development began by establishing the basic functionality for the storage optimized solution, intending to create a product that is able to accommodate the bare minimum of services. Therefore, the first iteration of the solution relied on simple JPA interface, with subsequent updates introducing batch updates/retrievals and directly using SQL queries for reduced overhead.

Thereafter it was realized that it is impossible to further improve the performance without changing some of the underlying technologies or design. Accordingly, steps were taken to advance the data lookup efficiency, which of course is the cornerstone of this solution. This endeavor was accomplished by the already discussed solution using ClusterJ.

Throughout the development, intermediate changes were tested using the Hopsworks platform[2] and conducting version control through GitHub. Furthermore, the expected storage requirements were analyzed using `ndb_size.pl` tool[3].

[2]Introduction to Hopsworks can be found at https://www.hopsworks.ai/

[3]Information on NDBCLUSTER size requirement estimator can be found at https://dev.mysql.com/doc/mysql-cluster-excerpt/5.7/en/mysql-

Altogether, development proceeded smoothly, with only a few hindrances attributed to the author's unfamiliarity with the software.

Code examples

The most important aspect for understanding the newly introduced index and how it functions can be obtained through interpretation of the database schema. Listing 5 shows an example of a table creation script.

Listing 5 Creating Hudi index table for `RONDB_CLUSTERJ`

```
CREATE TABLE IF NOT EXISTS index_cluster_record (
    record_key VARBINARY(255) NOT NULL,
    commit_ts BIGINT NOT NULL,
    partition_path VARCHAR(255) NOT NULL,
    file_name VARCHAR(38) NOT NULL,
    PRIMARY KEY (record_key, commit_ts),
    INDEX idx_record_key (record_key)
) ENGINE=NDBCLUSTER
```

This query is executed upon initialization of RonDB for the ClusterJ approach. Therefore, ensuring that the necessary infrastructure is in place to accommodate indexing. The columns of this table are storing the same information as present in HBase, with only slight alterations for optimization.

In-depth reasoning for the table components:

- `record_key` - Contains the key that is used to pinpoint the file location. This key can contain virtually any information, therefore, the choice was primarily between Varbinary and Varchar types. The final decision was in favor of Varbinary because it has no collation and is not limited to any character set.
- `commit_ts` - Is the timestamp associated with the creation/alteration of the record, provided during the mutation step. It is used to signify the records state.
- `file_name` - Simple string that can be used to locate the file group within a partition.

`cluster-programs-ndb-size-pl.html`

- `partition_path` - Value that identifies the partition containing the desired record (necessary as RonDB implementations are only global indexes).

While, the schema of the RonDB JDBC solution slightly differs it is worth noting that it is storing the exact same information, just in a normalized form. Indexing is extensively used in both approaches to offer fast retrieval of information by fields other than the primary key [47].

The benefits of the normalization can be observed in database sizes after a certain number of inserts have taken place. For example, after inserting one million values generated by Hudi `QuickstartUtils.DataGenerator()` the normalized indexing solution takes up only about 82 MB while de-normalized 146 MB, showcasing the advantageous outcome of this action.

4.4 Deployment

For the solution to be used it and all its dependencies must be accessible by Spark. Therefore, this section provides a step-by-step guide on deploying the newly improved Hudi client.

The following steps assume that the RonDB has already been set up. Guide on how it could be done can be found in RonDB documents[4].

The first step is cloning the provided GitHub repository[5] and building the package. This repository contains the latest available Hudi version (0.10.1) at the time of writing this document. The following commands outline these steps, by listing the Linux commands.

Listing 6 Creating Spark client jar file

```
#!/bin/bash

git clone git@github.com:bubriks/hudi.git
cd hudi
git checkout branch-0.10.0-RonDB
mvn clean package -Pspark3 -DskipTests
```

[4]Documentation on RonDB installation can be found at https://docs.rondb.com/rondb_installation/

[5]Public repository containing the Hudi Spark client with support for RonDB global index can be found at https://github.com/bubriks/hudi/tree/branch-0.10.0-RonDB

After the completion of the aforementioned actions a new jar file is generated (`packaging/hudi-spark-bundle/target/hudi-spark3-bundle_2.12-0.10.0.1.jar`). This file should be made accessible to Spark.

Subsequently, to allow for usage of the RonDB index, libraries related to communication with the database should be provided. The commands for acquiring these dependencies are listed hereafter (Listing 7).

Listing 7 Hudi dependencies

```
#!/bin/bash

# driver for JDBC (needed for both approaches)
wget https://repo1.maven.org/maven2/mysql/mysql-
 ↪  connector-java/8.0.28/mysql-connector-java-
 ↪  8.0.28.jar

# necessary for ClusterJ solution
wget https://archiva.hops.works/repository/Hops/co
 ↪  m/mysql/ndb/clusterj-
 ↪  rondb/{RONDB_VERSION}/clusterj-rondb-
 ↪  {RONDB_VERSION}.jar
```

After this, if using ClusterJ ensure that `libndbclient.so` (found in `mysql-RONDB_VERSION/lib` folder created by RonDB) is provided in the java library path (either update path or copy file to the current location).

Now the solution should be complete and ready for usage. The following sections expand upon how this can be done.

4.5 Usage

Running of the solution in most scenarios would be as simple as just specifying the index type as either `RONDB` or `RONDB_CLUSTERJ` and defining batch size. However, in the case of any kind of deviation of database details a new configuration properties can be easily provided.

An example of basic solutions execution on the Hopsworks platform can be seen in Listing 8. It shows how entries are inserted into a new table, using `hudi_trips_cow` directory as the base path and implementing the `RONDB` approach for indexing.

Listing 8 Creating new Hudi table with 10 entries using RonDB as the global index

```
# pyspark
tableName = "hudi_trips_cow"
basePath = "hdfs:///Projects/my_project/temp_dir/"
 ↪  + tableName

quickstartUtils =
 ↪  sc._jvm.org.apache.hudi.QuickstartUtils
dataGen = quickstartUtils.DataGenerator()
inserts = quickstartUtils.convertToStringList(data
 ↪  Gen.generateInserts(10))
df = spark.read.json(spark.sparkContext.paralleliz
 ↪  e(inserts,2))

hudi_options = {
    'hoodie.table.name': tableName,
    'hoodie.datasource.write.recordkey.field':
     ↪  'uuid',
    'hoodie.datasource.write.partitionpath.field':
     ↪  'partitionpath',
    'hoodie.datasource.write.table.name':
     ↪  tableName,
    'hoodie.datasource.write.operation': 'upsert',
    'hoodie.datasource.write.precombine.field':
     ↪  'ts',
    'hoodie.index.type': 'RONDB',
    'hoodie.index.rondb.batch.size': 100
}

df.write.format("hudi")
        .options(**hudi_options)
        .mode("overwrite")
        .save(basePath)
```

As already mentioned in the case of different requirements users can specify their properties. This can be done by providing value for any of the listed properties. The default values and documentation can be viewed by visiting the class mentioned in Section 4.2.3.

Configuration variables used for managing the RonDB indexing can be

seen in Table 4.2. The property types are used to provide properties for sub-resources (view Listing 9 for example of such usage).

Table 4.2: The RonDB indexing properties.

Name	Type
`hoodie.index.rondb.update.partition.path`	boolean
`hoodie.index.rondb.rollback.sync`	boolean
`hoodie.index.rondb.batch.size`	integer
`hoodie.index.rondb.jdbc.driver`	string
`hoodie.index.rondb.jdbc.url`	string
`hoodie.index.rondb.jdbc`	properties[6]
`hoodie.index.rondb.clusterj`	properties[7]

Listing 9 Example providing value for `hoodie.index.rondb.clusterj` property

```
'hoodie.index.rondb.clusterj': {
    'com.mysql.clusterj.connectstring':
     ↪  '127.0.0.1:1186',
    'com.mysql.clusterj.database': 'hudi'
}
```

4.6 Test setup

During the testing, tables were populated using Hudi quick start utility tool. It generates entries with 10 columns within three unique partitions, meaning that growth in the data will result in the creation of multiple file groups.

Through the preliminary analysis, the sample size of 10 was determined to be of sufficient scope (due to low result variation during testing). Therefore, all test cases are repeated for the same amount of time for consistency.

During testing a local single data node setup was established for external databases, with the rest of the configuration left with default values. The

[6]JDBC configuration properties can be found at https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-reference-configuration-properties.html

[7]ClusterJ configuration properties can be found at https://dev.mysql.com/doc/ndbapi/en/mccj-clusterj-constants.html

communication with databases was conducted using the HBase client (version 1.7.1), MySQL connector (version 8.0.28), and ClusterJ (version 21.04.1).

The observations were documented in auto-generated `csv` [8] files expressing time taken for requests completion (created only after successful execution). These files were subsequently used to extract and visualize information.

All of this endeavor was accomplished by using the Python programming language and its associated libraries, with the most crucial ones listed hereafter:

- os
- happybase
- pymysql
- time
- pandas
- matplotlib
- seaborn

The results recorded execution times for three different types of actions, with each one encompassing different use of an index. These types correspond to values of `hoodie.datasource.write.operation` property. A short explanation of what each one entails:

- insert- creates a new Hudi table entry without examining if the key already exists in the table.
- upsert (update)- updates location of an existing entry, by altering partition value.
- delete- removes record value from Hudi table. Uses soft delete, meaning that the record will be removed from the table, but the index will persist (used since it is the default setup in Hudi).

[8] The observations, as well as configuration files and code used for obtaining/visualizing data, are made available at https://github.com/bubriks/rondb_thesis

Chapter 5

Results and Analysis

In this chapter, different indexing solutions' performance observations are documented. It is subdivided into three main sections that correspond to the introduction of the results, reflection on the information therein, and discussion of findings.

5.1 Results

The observation of index performance includes multiple types of tests, as response times could be impacted by several independent variables. Therefore, each workload can be found in a separate subsection.

5.1.1 Workload A

Figure 5.1 serves as a brief overview of the performance of different indexing solutions when considering distinct types of actions. This is achieved by observing execution times when operating under a low-load environment, as it provides a baseline for the consideration made in the subsequent workloads.

For this evaluation, Hudi uses default values for its properties and the total table size consists of 1000 entries, with the number of changes per assessment equating to 100.

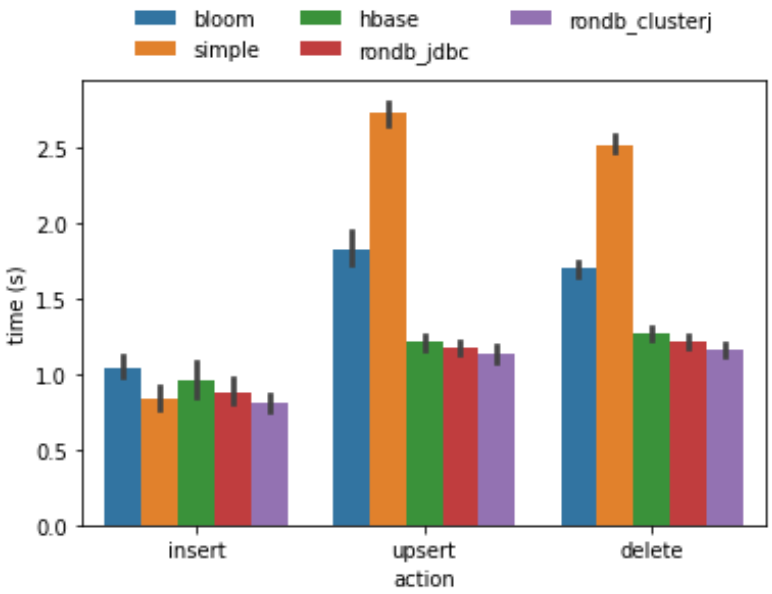


Figure 5.1: Baseline times for index operations under low load conditions

By looking at the bar chart and statistics tables 5.1 to 5.3 it can be noticed that insert is the fastest action for all indexes. Additionally, it can be observed that some solutions perform notably better than others for most tasks, forming a kind of hierarchy based on execution times. The only deviation from this can be observed with simple index uncharacteristically being among the fastest for inserts, while slowest for upserts and deletes.

Table 5.1: Insert execution times (s)

metric	**bloom**	**simple**	**hbase**	**rondb_jdbc**	**rondb_clusterj**
mean	1.046985	0.834320	0.956588	0.876729	0.807058
std	0.131717	0.136526	0.190086	0.134766	0.106279
min	0.919316	0.727889	0.797595	0.784635	0.716516
max	1.293633	1.170720	1.333785	1.221648	1.014967

Table 5.2: Upsert execution times (s)

metric	bloom	simple	hbase	rondb_jdbc	rondb_clusterj
mean	1.829518	2.736187	1.216716	1.176769	1.134291
std	0.192906	0.133183	0.082512	0.073121	0.085568
min	1.600711	2.479992	1.094011	1.082000	1.016648
max	2.258326	2.926483	1.324814	1.299815	1.246474

Table 5.3: Delete execution times (s)

metric	bloom	simple	hbase	rondb_jdbc	rondb_clusterj
mean	1.704032	2.518236	1.274243	1.214424	1.160595
std	0.074139	0.101318	0.075307	0.070587	0.077204
min	1.590105	2.427108	1.156054	1.102276	1.074429
max	1.818256	2.782736	1.370543	1.291193	1.291273

5.1.2 Workload B

As previously discussed Hudi was designed to accommodate voluminous data. Therefore, the indexing should function on practically any scale that the underlying solution could encounter. To observe how scalable the approaches are tests were executed having different Hudi table volumes.

The Hudi load in this scenario is identified by the number of entries rather than table size in the file system (for reference, 1 million entries roughly equals 100MB of information when saved in Hudi without retaining old commits). The number of entries was chosen to most adequately identify tendencies that could be noticed by the end-user, with the stopping point set when trends could be identifiable as subsequent increases would produce no palpable gains.

This workload encompasses the usage of Hudi with a default configuration and 100 mutations per test.

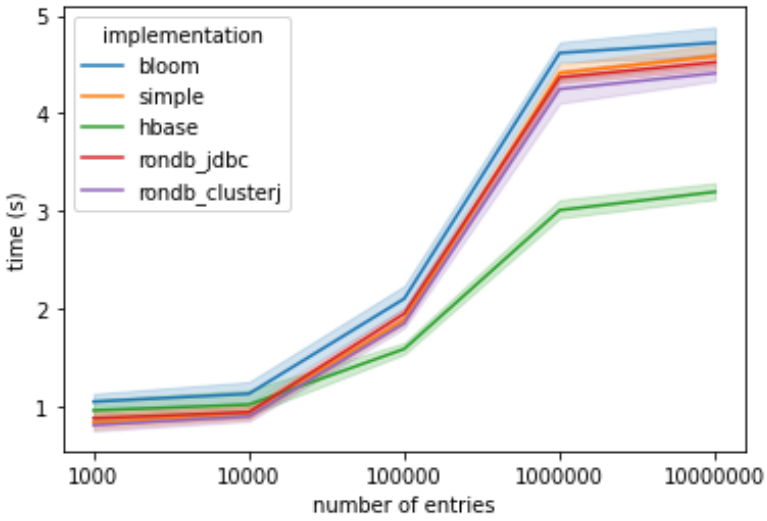


Figure 5.2: Insert times for index engines for increasing Hudi table size

Figure 5.2 shows that insert times for most implementations grow in tandem with one another. The only deviation from this is the `hbase` index, where an increase in table size had a much smaller impact over the course of the test.

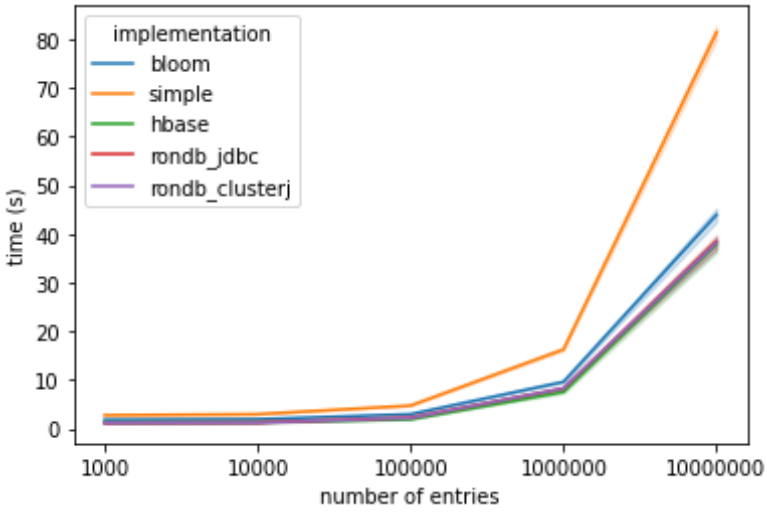


Figure 5.3: Update times for index engines for increasing Hudi table size

The update action (Figure 5.3) presents an inclination that it shares with

delete (Figure 5.4). These line plots show that the worst performing index was `simple`, which by the end of the test case was more than two times slower than the best. With the fastest approach for deletes identified as `rondb_jdbc`, while for updates `hbase`.

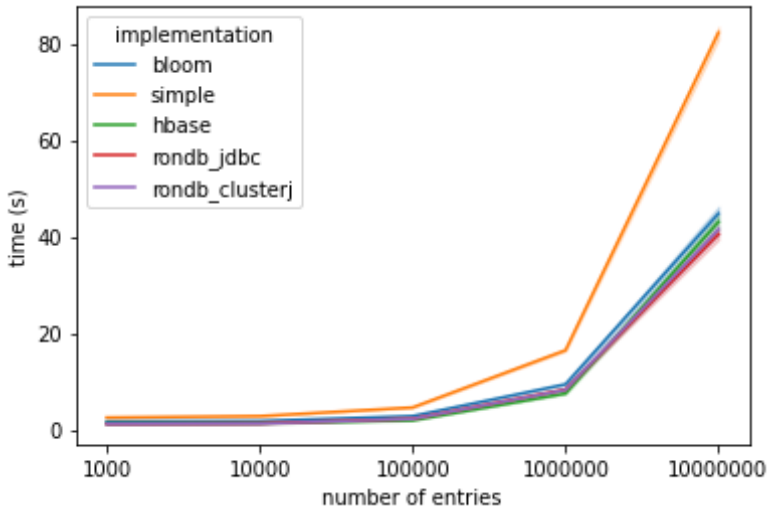


Figure 5.4: Delete times for index engines for increasing Hudi table size

In this workload, it could be observed that all solutions tend to become slower with more data, but they are uniquely impacted. Furthermore, the speed of the Hudi is influenced not only by the indexing implementation but also by the use case it finds itself in.

5.1.3 Workload C

The changes Hudi incurs usually are grouped into data sets with multiple mutations in each action. Therefore, this test case aims to examine how well each index is suited for accommodating changes at different quantities (test sizes). This is crucial as if the approach takes exceeding amounts of time to complete a transformation with multiple table alterations it could be a bottleneck in an otherwise well-operating data structure.

The Hudi table at the commencement of the tests contains 10000000 entries and is configured using standard properties.

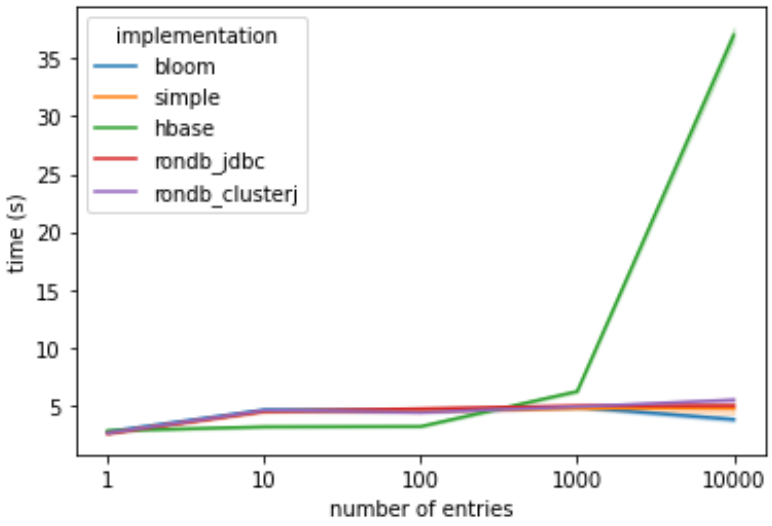


Figure 5.5: Insert times for index engines for increasing number of operations

The results of insert enactment (Figure 5.5) reveal a more noticeable growth in execution time for HBase than for any of its peers (Expanding the number of entries in the test from 1000 to 10000 produced a 30-second longer undertaking, which is an increase of 6 times). The rest of the solutions are notably less impacted when operating under different payloads, some even experiencing slightly faster execution.

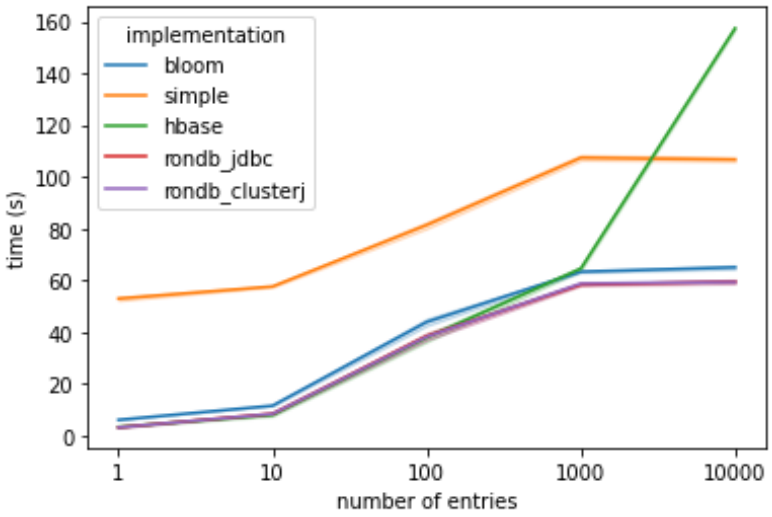


Figure 5.6: Update times for index engines for increasing number of operations

The completion times for updates with a different number of mutations (Figure 5.6) follow an easily identifiable trend. All indexes perform worse for updates than inserts and most solutions grow in tandem with one another. The only outlier is the HBase index, which produces a notably steeper time curve. From the start, the simple index is slower than its counterparts, but throughout the test, it is overtaken by the HBase.

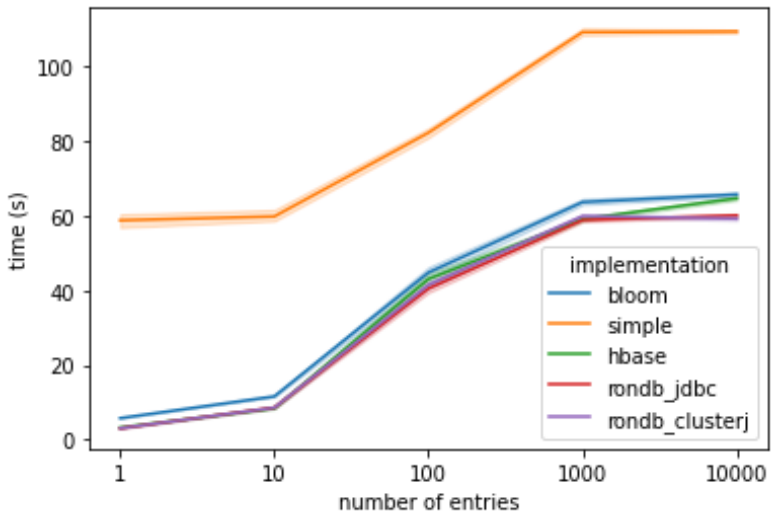


Figure 5.7: Delete times for index engines for increasing number of operations

Figure 5.7 displays delete results. They are akin to upsert action times, but for this scenario, the HBase index does not experience growth in the same proportions.

5.1.4 Workload D

Implementations relying on external databases need to communicate information from the client (Hudi) to the server. This is most efficiently done by grouping changes into batches, rather than executing mutations separately.

Therefore, to identify the best-suited batch size and subsequently conclude how dependent the index is on batch size it is necessary to locate the point where the lowest response times can be observed. This entails performing a constant number of changes in a single transaction.

The trial maintains a table size of 100000 entries and 10000 mutations for each test, with only batch size configuration (for both get and put operations)

altered for Hudi. This workload impacts only HBase and RonDB index implementations, therefore rest of the solutions will be absent from the results.

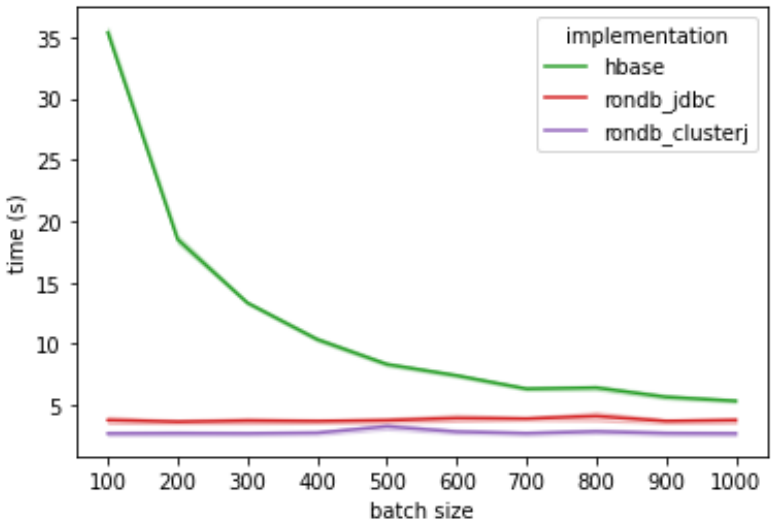


Figure 5.8: Insert times for index engines based on batch size

figs. 5.8 and 5.9 display how changes in batch size clearly benefit HBase, while it does not noticeably impact RonDB indexes in the depicted range. For both actions increase in batch size allows HBase to provide more comparable results to RonDB, but the performance improvements are steadily decreasing never bypassing any of the RonDB solutions. By the end, a plateau can be observed, whereupon any further increase in batch size results in negligible/no advancements.

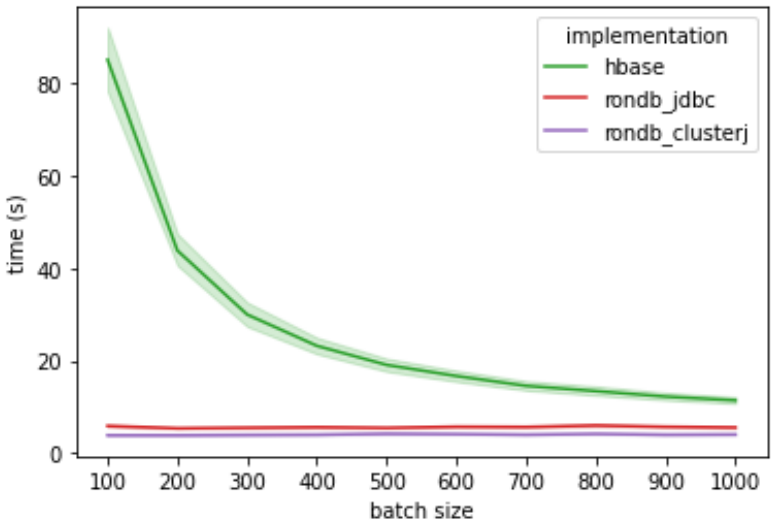


Figure 5.9: Update times for index engines based on batch size

The delete times (Figure 5.10) have similar tendencies to the upsert and insert actions, but here the base value for HBase is much smaller and it manages to outperform `jdbc` index while lagging behind `clusterj`.

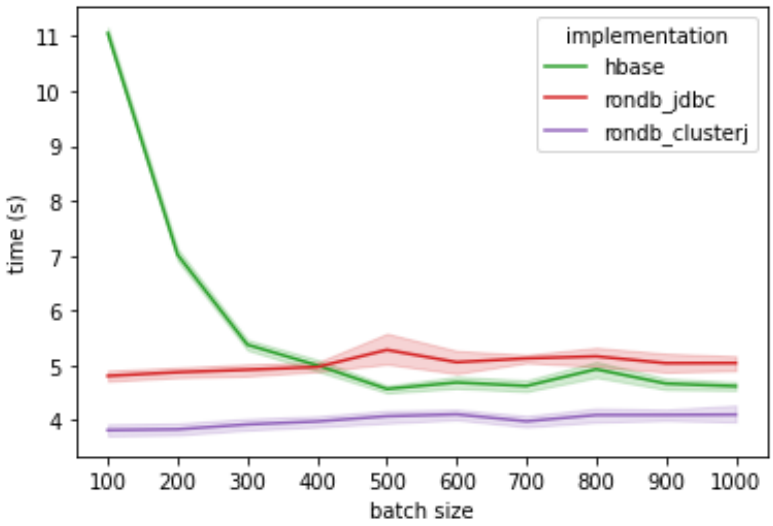


Figure 5.10: Delete times for index engines based on batch size

The previous figures have shown RonDB, producing what could be described as horizontal lines that are parallel to one another.

However, it is not entirely accurate, as batch size has an impact on the RonDB implementations. This just could not be displayed in the same plots due to HBase's tendency to have exponential growth with decreasing batch size. Therefore, figs. 5.11 to 5.13 showcase how batch processing provides notable improvements to RonDB implementations, but does not grant tangible gains once exceeding the size of 100.

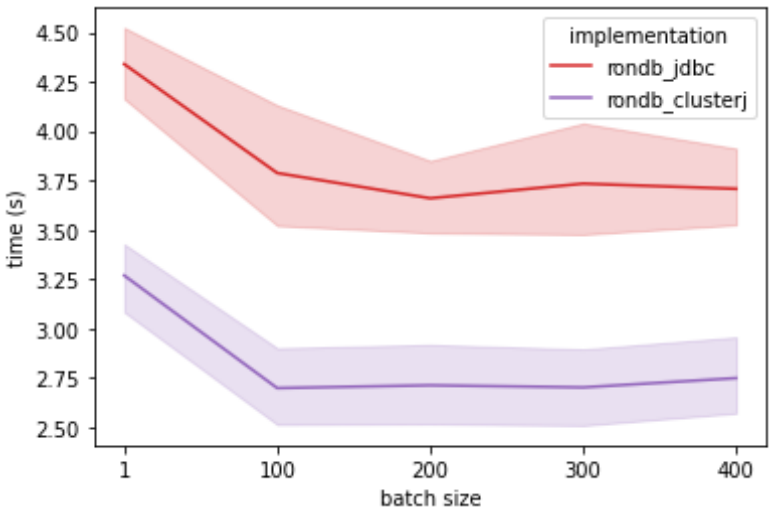


Figure 5.11: Insert times for RonDB index engines based on batch size

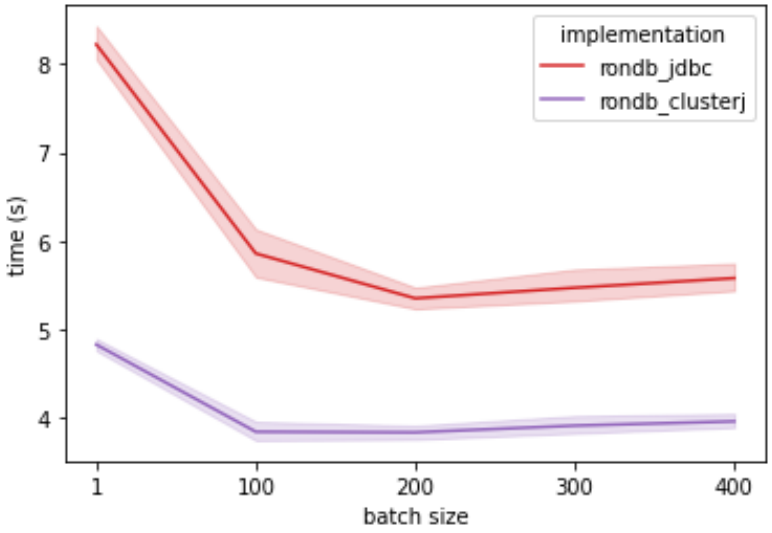


Figure 5.12: Update times for RonDB index engines based on batch size

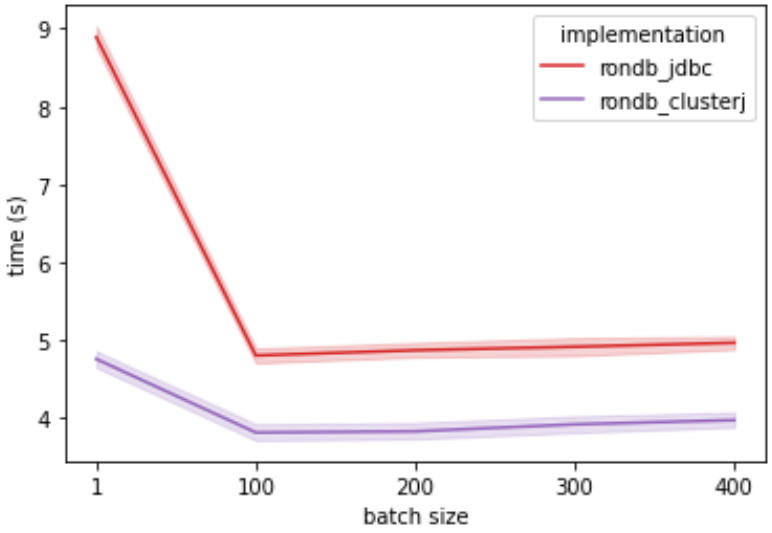


Figure 5.13: Delete times for RonDB index engines based on batch size

5.1.5 Workload E

Some of the indexes use file systems directly while others rely on external data structures for their needs. This results in the potential for different indexing

approaches to have a distinct relation to the number of files present in the Hudi table. However, it is not clear if and by how much the file count/size impacts their performance.

Hudi creates a new filegroup when the preceding one has reached a preset amount of data. By default, this value is set at 120 MB. Throughout this test case, the file size property will be shifted, while the table size is 10000000 entries and observation contains 100 mutations.

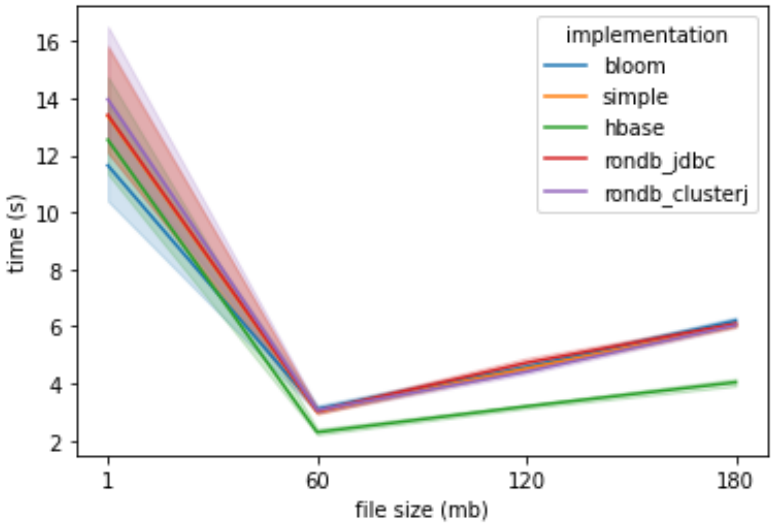


Figure 5.14: Insert times for index engines based on file size

Figure 5.14 shows that 60 MB parquet files provide the optimal performance for inserting. Any change to smaller/bigger file groups results in a worse performing solution, independent of the index engine. However, while most indexes produce results that are similar to their peers, HBase stands as an outlier, being less impacted by the file size once exceeding 60 MB.

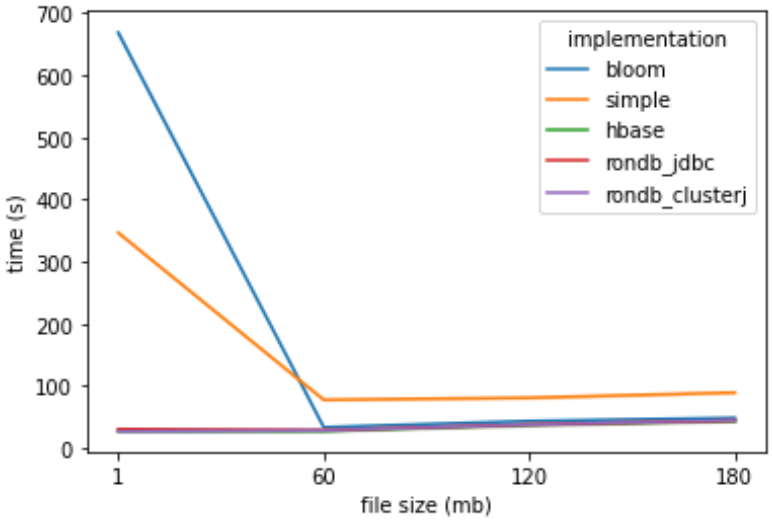


Figure 5.15: Update times for index engines based on file size

The update and delete actions (figs. 5.15 and 5.16) can be observed to have no notable difference between the two. But they both show simple and bloom index lacking in comparison to their peers, especially with small file sizes. This can be more clearly seen in figs. 5.17 and 5.18), where `bloom` index is on average 5 seconds slower than the competing database solutions for any given file size.

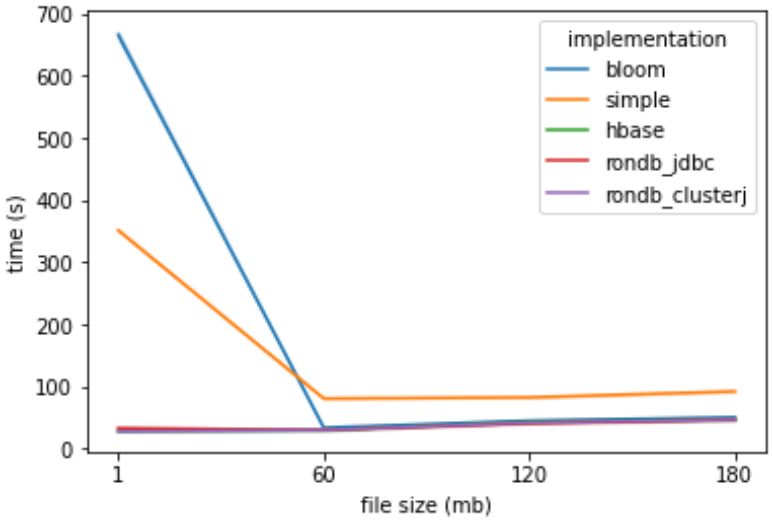


Figure 5.16: Delete times for index engines based on file size

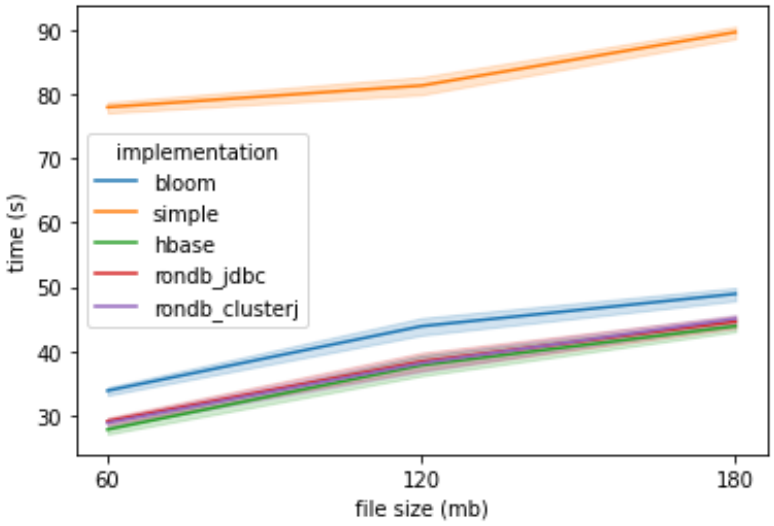


Figure 5.17: Update times for index engines based on file size disregarding 1 MB observations

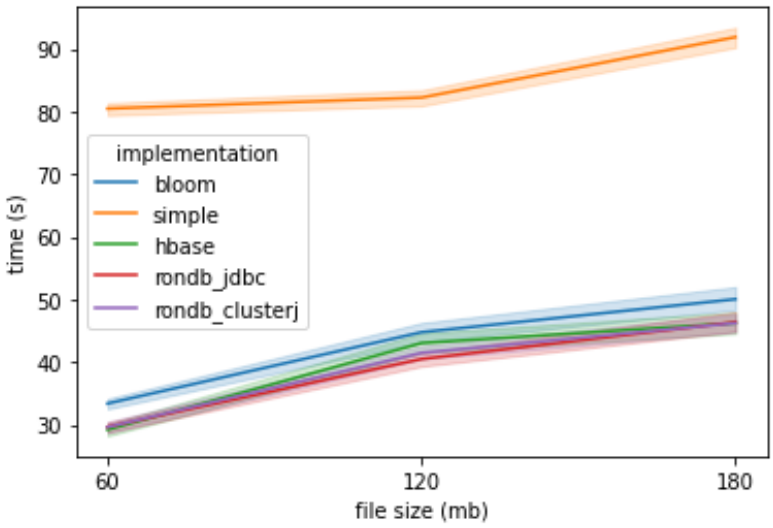


Figure 5.18: Delete times for index engines based on file size disregarding 1 MB observations

5.2 Quality Analysis

The results presented in the previous section display unique tendencies for each approach. This as anticipated allows for clear identification of solutions capabilities concerning its peers and answers the most pressing questions about the performance. Therefore, the author believes that the observations can be considered valid.

Furthermore, reliability has also been achieved, as results are consistent between tests of the same use case. This means that they are reproducible assuming conductors have access to the same environment.

5.3 Discussion

The results of Workload A (5.1.1) display a clear performance advantage for database solutions (at least for actions involving lookup of information). This corresponds to what was anticipated since as already explained usage of HBase, and by extension RonDB, was inspired to provide faster execution times. The reason for all solutions obtaining similar insert times is that the index is not used, only initialized. This is a strong suit of the simple index, as it does not have to perform any additional operations for it to function.

By looking at the observations made during the alteration of Hudi table size (Section 5.1.2) it was noticed that all solutions scale differently. This of course is directly attributed to how each implementation functions internally and not necessarily how it was implemented in Hudi. For reference, observe that difference between the two RonDB implementations is within the margin of error. Additionally, the plots showcased that both of the file-based indexes take the longest time for all use cases (even though bloom was much more analogous to database implementations). The fast insert times for HBase could potentially be attributed to it being a log-structured storage engine [48]. This means that writes are simply an append operation that at first performs changes to the in-memory data structure, which is subsequently flashed to the disk upon reaching some predefined condition/threshold.

Changing the number of mutations that the test incurs (Section 5.1.3) results in easy identification of HBase shortcomings. This is because the HBase index is rate limited to avoid overwhelming the server with requests, which could result in the HBase Region server dying. The reason why the bloom index performance improved by the end of the insert operation can likely be attributed to changes being more self-contained, thus not requiring append to an existing file. The results overall showed both of the RonDB clients coping with the increasing number of changes similarly to file-based indexes, consistently being among the fastest (in particular when changes involved data lookup).

The batch size test case (Section 5.1.4) shows how important it is to bundle information for the different databases. HBase observes notable improvements up until the final tested volume, while RonDB sees enhancement only up to the batch size of 100. The reason for this is the already mentioned HBase solutions rate limiting approach (which uses the java sleep function to avoid a high number of concurrent changes to the database). A more noticeable impact should be anticipated if communication was coordinated over the internet, as the overhead of multiple small messages would increase (negated in this test due to databases being co-located with the execution engine on the same machine). The discrepancy between RonDB clients manifests itself mostly because of the variation in communication medium (but also partially due to the usage of normalization). The lower level API `clusterj` is more efficient as the scale of mutations per test increases, but for this to be appreciable a considerable number of changes have to take place in relatively few files, as a modification to large number of directory entries negate the benefits.

The final test case investigated file size, and by extension file count, impact on the index (Section 5.1.5). Here, as expected, database solutions were

among the fastest, in particular when acquiring existing data with small file sizes. This can be rationalized by recognizing that file-based indexing has to investigate each file, while database solutions avoid these I/O operations. However, an unexpected finding was that HBase performs notably better for inserts, especially with an increase in file sizes. The reason for this is once again a bi-product of HBase being a log-structured storage engine. As the small number of changes are just append operation to an in-memory data structure (following similar reasoning to insert performance in Hudi table size workload).

Chapter 6

Conclusions and Future work

This thesis has investigated how indexing operates in Hudi and why it is needed to provide timely responses. Furthermore, a new indexing approach was introduced and subsequently examined to evaluate its feasibility and determine the best-suited role for each solution.

This chapter concludes the research, reflecting on the work that has been done and proposes ideas for furthering the goals of this thesis.

6.1 Conclusions

This thesis has confirmed that the usage of an external database provides the fastest response times for nearly all use cases. However, there is an additional cost associated with this approach (in terms of storage) that file-based indexing does not incur. Accordingly, there is no single solution that suits the needs of all users.

The results showed that the variation between the RonDB and HBase arises from their design principles (if the batch size is adequately adjusted beforehand). Therefore, the difference between the two in most scenarios can not be easily identified, owing to fast query responses, with the majority of the time taken up by work other than indexing. Likewise, it cannot be conclusively stated that RonDB is better than HBase for indexing in Hudi. Nonetheless, the author believes that the goals of the thesis have been met.

RonDB offers a well-rounded solution capable of suiting most needs. This is because it provides fast operations independent of the workload. Whereas, the bloom index suffers from an increased table size (especially with multiple files) and HBase fails to cope with a large number of mutations (in particular with incorrect batch size). Moreover, the use of Relational Database

Management System (RDBMS) makes this approach more developer-friendly as more users are familiar with SQL syntax and could utilize its full set of features more comfortably. As a result, boosting the potential for the solution to be worked on by the community.

Subsequent work in this area should anticipate the large number of independent variables impacting the performance of any given solution. Therefore, counting on a multitude of configuration changes to provide the best response times. Furthermore, it should be realized that the majority of execution time is spent on I/O operations even when using a database-based index. This means that there is only so much that a faster index can contribute to performance improvements, without reconstructing the rest of the solution.

6.2 Limitations

The limiting factor of this research has been time, the number of configurable properties, the data volume, and the shortage of documentation. Therefore, the research could not have been further expanded to encompass a more substantial set of test cases. However, the author believes that the current results provide an adequate understanding and do not require more observations to derive accurate conclusions.

6.3 Future work

This research helped to conceive a new index implementation and identify its response characteristics concerning peers. But by doing so it also pinpointed aspects that could be subsequently explored in the future.

The most fundamental research could be conducted on the resource consumption of the indexing solutions. As it would help collaborate the findings of this thesis, providing a clear overview of the performance as a whole (not only execution times). Furthermore, the rollback times could be evaluated, as this research considered only successful executions, avoiding investigation into the ”rainy day scenario”. The impact of operating in a cloud environment could also be scrutinized, by looking at the significance of multiple data/worker nodes, potentially operating with more voluminous data in object stores, and observing the effect of concurrent use.

In addition to performance analysis, future researchers could attempt to provide new capabilities for Hudi. For example, using indexing when performing read queries on the Hudi table (At the moment it is utilized only

for mutations of the information). Subsequently, with successful usage of indexing for all operations, the next logical step could be the implementation of a secondary index. Thus blurring the lines between a fully managed database and lake house even further.

6.3.1 Left undone

The implemented solution is capable of providing the same kind of capabilities that can be expected from the HBase approach, but at this time it does not yield advanced features such as batch size auto compute and adaptive key types. This, although negligible, could be a welcome feature. For example, storing integer values instead of binary data would be more efficient. Moreover, the current implementation supports only spark clients, making it logical to expand capabilities for its counterparts as well.

6.3.2 Cost analysis

The expenditure associated with the storage of the index depends on the information that is stored. However, RonDB can hold data both in memory and on disk, while scaling to clusters of up to 1PB in size [42]. Which could equate roughly to 12-13 trillion entries (assuming usage of a normalized solution).

The current solution contains two newly implemented approaches, but as shown by the tests they produce similar outputs in most scenarios, with the difference arising only when performing several changes. Therefore, it is cost-prohibitive and not user-friendly to maintain both. To rectify these issues all subsequent development should be focused on the improvement of a single solution.

The author proposes advancing the `RONDB_JDBC`. The reason for this choice is due to its greater potential for the future (can expand to multiple databases and utilize their full potential through SQL), improved security (one of the building blocks for managed RDBMS), and memory savings (when normalized it was observed that table would store approximately 44% less data).

6.3.3 Security

As already discussed in the previous sections, the JDBC approach uses a MySQL server when communicating with the NDB cluster. Therefore, it comes with all security features expected in a RDBMS. However, when

directly communicating with the NDB nodes there is no authentication/authorization and encryption, meaning users of `RONDB_CLUSTERJ` are prone to a variety of attacks if exposed to the wider public. Accordingly, the usage of ClusterJ implementation should be avoided for most organizations as it could end up as an entry point for a malicious actor [1].

6.3.4 Future prospects

The changes at this point are only available on a fork of Apache Hudi and they are in the prototype phase. It would be a welcome sight to see them be made available for the wider community, allowing for the usage of not only RonDB but also other databases following the relational paradigm (using SQL) to offer index.

6.4 Reflections

The newly introduced indexing approach does not offer exceedingly fast response times when compared to its peers. But it is an adaptable solution, that is capable of providing fast and dependable response times for all use cases, when in fact the performance of the rival is greatly influenced by the situation it finds itself in.

In closure, a summary of the indexing solutions, from the author's point of view, is provided hereafter.

- Simple: the best approach for workloads involving a lot of data insertions, but noticeably pales in comparison to other indexes for lookup times (in a sense fire and forget approach).
- Bloom: great potential, but suffers from the existence of multiple files and results in longer execution times when compared to database solutions.
- RonDB: well-rounded product, capable of accommodating all workloads (being among the fastest for all), but does not notably excel at any.

[1] A brief article describing security and networking issues as well as how to mitigate them in a publicly facing NDB cluster can be found at https://dev.mysql.com/doc/mysql-cluster-excerpt/8.0/en/mysql-cluster-security-networking-issues.html

- HBase: well suited for read-intensive workloads, but performs poorly with an increasing number of mutations (bootstrapping a large table is a particular hurdle).

References

[1] T. Harbert, "Tapping the power of unstructured data," Feb. 2021. [Online]. Available: https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data [Accessed: 2022-03-02] [Page 3.]

[2] J. Phipps, "Data Lake vs. Data Swamp," Aug. 2021. [Online]. Available: https://www.enterprisestorageforum.com/management/data-lake-data-swamp/ [Accessed: 2022-03-30] [Page 3.]

[3] B. Lorica, M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "What Is a Lakehouse?" Jan. 2020. [Online]. Available: https://databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html [Accessed: 2022-01-19] [Page 3.]

[4] D. Bzhalava and J. Dowling, "MLOps Wars: Versioned Feature Data with a Lakehouse," Aug. 2021. [Online]. Available: https://www.logicalclocks.com/blog/mlops-wars-versioned-feature-data-with-a-lakehouse [Accessed: 2022-01-19] [Page 3.]

[5] R. Xin and M. Mokhtar, "Databricks Sets Official Data Warehousing Performance Record," Nov. 2021. [Online]. Available: https://databricks.com/blog/2021/11/02/databricks-sets-official-data-warehousing-performance-record.html [Accessed: 2022-03-30] [Page 3.]

[6] P. Edara and M. Pasumansky, "Big Metadata: When Metadata is Big Data," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3083–3095, Jul. 2021. doi: 10.14778/3476311.3476385 Publisher: VLDB Endowment. [Online]. Available: https://doi.org/10.14778/3476311.3476385 [Page 4.]

[7] M. Zaharia, A. Ghodsi, R. Xin, and m. crossbow, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing

and Advanced Analytics," in *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. [Online]. Available: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf [Accessed: 2022-01-21] [Pages 8 and 24.]

[8] J. Chen, "A Thorough Comparison of Delta Lake, Iceberg and Hudi," Jul. 2020. [Online]. Available: https://databricks.com/session_na20/a-thorough-comparison-of-delta-lake-iceberg-and-hudi [Accessed: 2022-01-19] [Page 9.]

[9] O. Katz, "Hudi, Iceberg and Delta Lake: Data Lake Table Formats Compared," Apr. 2021. [Online]. Available: https://lakefs.io/hudi-iceberg-and-delta-lake-data-lake-table-formats-compared/ [Accessed: 2022-01-27] [Page 9.]

[10] B. Bopanna, "Comparative study of Apache Iceberg, Open Delta, Apache CarbonData and Hudi," Apr. 2021. [Online]. Available: https://brijoobopanna.medium.com/comparative-study-of-apache-iceberg-open-delta-apache-carbondata-and-hudi-c3962e5a0c4a [Accessed: 2022-01-19] [Page 9.]

[11] P. Chockalingam, "Open Sourcing Delta Lake," Apr. 2019. [Online]. Available: https://databricks.com/blog/2019/04/24/open-sourcing-delta-lake.html [Accessed: 2022-01-27] [Page 9.]

[12] B. Yavuz, M. Armbrust, and B. Heintz, "Diving Into Delta Lake: Unpacking The Transaction Log," Aug. 2019. [Online]. Available: https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html [Accessed: 2022-03-02] [Pages 9 and 10.]

[13] A. Conway and J. Minnick, "Introducing Delta Engine," Jun. 2020. [Online]. Available: https://databricks.com/blog/2020/06/24/introducing-delta-engine.html [Accessed: 2022-01-27] [Page 9.]

[14] B. Konieczny, "ACID file formats - file system layout," Apr. 2022. [Online]. Available: https://www.waitingforcode.com/data-engineering/acid-file-formats-file-system-layout/read [Accessed: 2022-04-17] [Pages 10, 12, and 15.]

[15] C. Mathiesen, “A Short Introduction to Apache Iceberg,” Jan. 2021. [Online]. Available: https://medium.com/expedia-group-tech/a-short-introduction-to-apache-iceberg-d34f628b6799 [Accessed: 2022-01-27] [Page 10.]

[16] J. Hughes, “Apache Iceberg: An Architectural Look Under the Covers,” Nov. 2021. [Online]. Available: https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/ [Accessed: 2022-03-02] [Page 10.]

[17] G. P. Kowshik and X. J. C. Mojica, “Taking Query Optimizations to the Next Level with Iceberg,” Jan. 2021. [Online]. Available: https://medium.com/adobetech/taking-query-optimizations-to-the-next-level-with-iceberg-6c968b83cd6f [Accessed: 2022-01-27] [Page 10.]

[18] K. Stokes, “The why and how of partitioning in Apache Iceberg,” Oct. 2020. [Online]. Available: https://developer.ibm.com/articles/the-why-and-how-of-partitioning-in-apache-iceberg/ [Accessed: 2022-04-17] [Page 10.]

[19] P. Rajaperumal and V. Chandar, “Hudi: Uber Engineering’s Incremental Processing Framework on Apache Hadoop,” Mar. 2017. [Online]. Available: https://eng.uber.com/hoodie/ [Accessed: 2022-01-21] [Page 11.]

[20] V. Chandar, “Lakehouse Concurrency Control: Are we too optimistic?” Dec. 2021. [Online]. Available: https://hudi.apache.org/blog/2021/12/16/lakehouse-concurrency-control-are-we-too-optimistic [Accessed: 2022-08-02] [Page 11.]

[21] ——, “Apache Hudi - The Data Lake Platform,” Jul. 2021. [Online]. Available: https://hudi.apache.org/blog/2021/07/21/streaming-data-lake-platform/ [Accessed: 2021-11-07] [Pages 11, 13, and 18.]

[22] N. Agarwal, “Building a Large-scale Transactional Data Lake at Uber Using Apache Hudi,” Jun. 2020. [Online]. Available: https://eng.uber.com/apache-hudi-graduation/ [Accessed: 2021-11-07] [Pages 11 and 17.]

[23] B. Braams, "The Parquet Format and Performance Optimization Opportunities," Oct. 2019. [Online]. Available: https://databricks.com/session_eu19/the-parquet-format-and-performance-optimization-opportunities [Accessed: 2022-05-17] [Page 13.]

[24] B. Varadarajan, "Efficient Migration of Large Parquet Tables to Apache Hudi," Aug. 2020. [Online]. Available: https://hudi.apache.org/blog/2020/08/20/efficient-migration-of-large-parquet-tables/ [Accessed: 2022-01-02] [Pages 14 and 15.]

[25] B. Varadarajan and U. Mehrotra, "RFC - 12 : Efficient Migration of Large Parquet Tables to Apache Hudi," Mar. 2020. [Online]. Available: https://cwiki.apache.org/confluence/display/HUDI/RFC+-+12+%3A+Efficient+Migration+of+Large+Parquet+Tables+to+Apache+Hudi [Accessed: 2022-01-02] [Pages 15 and 16.]

[26] X. Geng, "RFC-36: HUDI Metastore Server," Jan. 2022. [Online]. Available: https://cwiki.apache.org/confluence/display/HUDI/RFC-36%3A+HUDI+Metastore+Server [Accessed: 2022-02-03] [Page 18.]

[27] R. Peterson, "Indexing in DBMS: What is, Types of Indexes with EXAMPLES," Jan. 2022. [Online]. Available: https://www.guru99.com/indexing-in-database.html [Accessed: 2022-01-29] [Page 19.]

[28] V. Chandar, "Employing the right indexes for fast updates, deletes in Apache Hudi," Nov. 2020. [Online]. Available: https://hudi.apache.org/blog/2020/11/11/hudi-indexing-mechanisms/ [Accessed: 2022-01-19] [Pages 19 and 20.]

[29] A. Jain, "Probabilistic Data Structures for Big Data and Streaming Applications," Sep. 2021. [Online]. Available: https://octo.vmware.com/bloom-filter/ [Accessed: 2022-01-27] [Page 20.]

[30] N. Agarwal and K. Devarajaiah, "Consistent Data Partitioning through Global Indexing for Large Apache Hadoop Tables at Uber," Apr. 2019. [Online]. Available: https://eng.uber.com/data-partitioning-global-indexing/ [Accessed: 2022-01-27] [Page 20.]

[31] S. Narayanan, N. Agarwal, and P. Wason, "RFC-08 Record level indexing mechanisms for Hudi datasets," Sep. 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/HUDI/RFC-08+

+Record+level+indexing+mechanisms+for+Hudi+datasets [Accessed: 2022-01-02] [Page 20.]

[32] Z. Guan, "Building an ExaByte-level Data Lake Using Apache Hudi at ByteDance," Sep. 2021. [Online]. Available: https://hudi.apache.org/blog/2021/09/01/building-eb-level-data-lake-using-hudi-at-bytedance/ [Accessed: 2021-11-07] [Pages 21 and 24.]

[33] M. Gupta, V. Verma, and M. Verma, "In-Memory Database Systems - A Paradigm Shift," *ArXiv*, vol. Volume 6, no. Dec 2013, Feb. 2014. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1402/1402.1258.pdf [Accessed: 2022-01-31] [Pages 21 and 24.]

[34] N. Patel, "What is Cassandra and why are big tech companies using it?" May 2020. [Online]. Available: https://ubuntu.com/blog/apache-cassandra-top-benefits [Accessed: 2022-01-31] [Page 23.]

[35] A. Bekker, "Cassandra vs. HBase: twins or just strangers with similar looks?" Jun. 2018. [Online]. Available: https://www.scnsoft.com/blog/cassandra-vs-hbase [Accessed: 2022-01-31] [Page 23.]

[36] M. Ronström and J. Dowling, "Rondb: The world's fastest key-value store is now in the cloud." Feb. 2020. [Online]. Available: https://www.logicalclocks.com/blog/rondb-the-worlds-fastest-key-value-store-is-now-in-the-cloud [Accessed: 2021-11-07] [Page 23.]

[37] D. Orescanin and T. Hlupic, "Data Lakehouse - a Novel Step in Analytics Architecture," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. Opatija, Croatia: IEEE, Sep. 2021. doi: 10.23919/MIPRO52101.2021.9597091. ISBN 978-953-233-101-1 pp. 1242–1246. [Online]. Available: https://ieeexplore.ieee.org/document/9597091/ [Accessed: 2022-01-31] [Page 24.]

[38] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta lake: high-performance ACID table storage over cloud object stores,"

Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 3411–3424, Aug. 2020. doi: 10.14778/3415478.3415560. [Online]. Available: https://dl.acm.org/doi/10.14778/3415478.3415560 [Accessed: 2022-01-31] [Page 24.]

[39] N. Gebretsadkan Kidane, "Hudi on Hops : Incremental Processing and Fast Data Ingestion for Hops," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2019, backup Publisher: KTH, School of Electrical Engineering and Computer Science (EECS) Issue: 2019:809 Series: TRITA-EECS-EX. [Online]. Available: http://kth.diva-portal.org/smash/get/diva2:1413103/FULLTEXT01.pdf [Accessed: 2022-01-21] [Page 24.]

[40] A. T. Kabakus and R. Kara, "A performance evaluation of in-memory databases," *Journal of King Saud University - Computer and Information Sciences*, vol. 29, no. 4, pp. 520–525, Oct. 2017. doi: 10.1016/j.jksuci.2016.06.007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1319157816300453 [Accessed: 2022-01-31] [Page 24.]

[41] P. Martins, M. Abbasi, and F. Sá, "A Study over NoSQL Performance," in *New Knowledge in Information Systems and Technologies*, . Rocha, H. Adeli, L. P. Reis, and S. Costanzo, Eds. Cham: Springer International Publishing, 2019, vol. 930, pp. 603–611. ISBN 978-3-030-16180-4 978-3-030-16181-1 Series Title: Advances in Intelligent Systems and Computing. [Online]. Available: http://link.springer.com/10.1007/978-3-030-16181-1_57 [Accessed: 2022-01-21] [Page 24.]

[42] M. Ronström and J. Dowling, "AI/ML needs a Key-Value store, and Redis is not up to it," Feb. 2021. [Online]. Available: https://www.hopsworks.ai/post/ai-ml-needs-a-key-value-store-and-redis-is-not-up-to-it [Accessed: 2022-01-20] [Pages 24 and 61.]

[43] M. Ronström, "NDB Cluster, the World's Fastest Key-Value Store," Feb. 2020. [Online]. Available: http://mikaelronstrom.blogspot.com/2020/02/ndb-cluster-worlds-fastest-key-value.html [Accessed: 2022-01-20] [Page 24.]

[44] X. Wang, "Apache Hudi meets Apache Flink," Oct. 2020. [Online]. Available: https://hudi.apache.org/blog/2020/10/15/apache-hudi-meets-apache-flink/ [Accessed: 2022-02-20] [Page 31.]

[45] A. Morgan, “Using ClusterJPA (part of MySQL Cluster Connector for Java) – a tutorial,” Mar. 2010. [Online]. Available: http://www.clusterdb.com/mysql-cluster/using-clusterjpa-part-of-mysql-cluster-connector-for-java-a-tutorial [Accessed: 2022-08-03] [Page 34.]

[46] ——, “Using ClusterJ (part of MySQL Cluster Connector for Java) – a tutorial,” Mar. 2010. [Online]. Available: http://www.clusterdb.com/mysql-cluster/using-clusterj-part-of-mysql-cluster-connector-for-java-a-tutorial [Accessed: 2022-06-03] [Page 34.]

[47] B. Barnhill, “Indexing,” Aug. 2021. [Online]. Available: https://dataschool.com/sql-optimization/how-indexing-works/ [Accessed: 2022-04-05] [Page 37.]

[48] P. Tanwar, “Log-Structured Storage Engines,” Sep. 2021. [Online]. Available: https://medium.com/@pnk.tanwar/log-structured-storage-engines-a0c6e78273c [Accessed: 2022-04-29] [Page 57.]

Appendix A

NDB size report of the final solutions

ndb_size.pl report for database: 'hudi' (3 tables)

Connected to: DBI:mysql:host=127.0.0.1;port=3306

Including information for versions: 4.1, 5.0, 5.1

Table List

hudi.index_record

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
commit_ts	timestamp(3)		PRI	4	4	4
file_id	int		MUL	4	4	4
record_key	varbinary(255)		PRI	256	256	256
Fixed Size Columns DM/Row				**264**	**264**	**264**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
record_key	BTREE	N/A	N/A	N/A
file_id	BTREE	N/A	N/A	N/A
Total Index DM/Row		**0**	**0**	**0**

Supporting Tables DataMemory/Row

Table	4.1	5.0	5.1
hudi.index_record_record_key$unique	272	272	280
hudi.index_record_file_id$unique	276	276	284
This DataMemory/Row	**276**	**276**	**280**
Total DM/Row (inludes DM in other tables)	**824**	**824**	**844**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	285	16	16
Indexes IM/Row	**285**	**16**	**16**

Supporting Tables IndexMemory/Row

hudi.index_record_record_key$unique	281	16	16

hudi.index_record_file_id$unique	29	16	16
Total Suppt IM/Row	**310**	**32**	**32**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	16
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	276	276	280

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	1000	1000	1000
Rows/32kb DM Page	118	118	116
Fixedsize DataMemory (KB)	288	288	288
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	28	512	512
IndexMemory (KB)	288	16	16

---

hudi.index_record_file

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
partition_id	int		MUL	4	4	4
file_id	int		PRI	4	4	4
file_name	varchar(38)		UNI	40	40	40
Fixed Size Columns DM/Row				**48**	**48**	**48**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
file_name	BTREE	16	16	16
PRIMARY	BTREE	N/A	N/A	N/A
partition_id	BTREE	N/A	N/A	N/A
Total Index DM/Row		**16**	**16**	**16**

Supporting Tables DataMemory/Row

Table	4.1	5.0	5.1
hudi.index_record_file_partition_id$unique	20	20	28

This DataMemory/Row	**76**	**76**	**80**
Total DM/Row (inludes DM in other tables)	**96**	**96**	**108**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
file_name	N/A	N/A	N/A
PRIMARY	29	16	16
Indexes IM/Row	**29**	**16**	**16**

Supporting Tables IndexMemory/Row

hudi.index_record_file_partition_id$unique	29	16	16
Total Suppt IM/Row	**29**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	16
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	76	76	80

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	3	3	3
Rows/32kb DM Page	429	429	408
Fixedsize DataMemory (KB)	32	32	32
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	282	512	512
IndexMemory (KB)	8	8	8

---

hudi.index_record_file_id$unique

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
file_id	int		MUL	4	4	4
record_key	varbinary(255)		PRI	256	256	256
commit_ts	timestamp(3)		PRI	4	4	4
Fixed Size Columns DM/Row				**264**	**264**	**264**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
Total Index DM/Row		**0**	**0**	**0**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	29	16	16
Indexes IM/Row	**29**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	20
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	276	276	284

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	1000	1000	1000
Rows/32kb DM Page	118	118	114
Fixedsize DataMemory (KB)	288	288	288
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	282	512	512
IndexMemory (KB)	32	16	16

---

hudi.index_record_file_partition_id$unique

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
file_id	int		PRI	4	4	4
partition_id	int		MUL	4	4	4
Fixed Size Columns DM/Row				**8**	**8**	**8**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A

Total Index DM/Row		**0**	**0**	**0**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	29	16	16
Indexes IM/Row	**29**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	20
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	20	20	28

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	3	3	3
Rows/32kb DM Page	1632	1632	1165
Fixedsize DataMemory (KB)	32	32	32
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	282	512	512
IndexMemory (KB)	8	8	8

---

hudi.index_record_partition

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
partition_path	varchar(255)		UNI	256	256	256
partition_id	int		PRI	4	4	4
Fixed Size Columns DM/Row				**260**	**260**	**260**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
partition_path	BTREE	16	16	16
Total Index DM/Row		**16**	**16**	**16**

IndexMemory for Indexes

`RONDB_JDBC` ndb size report.

:

Index Name	4.1	5.0	5.1
PRIMARY	29	16	16
partition_path	N/A	N/A	N/A
Indexes IM/Row	**29**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	16
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	288	288	292

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	3	3	3
Rows/32kb DM Page	113	113	111
Fixedsize DataMemory (KB)	32	32	32
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	282	512	512
IndexMemory (KB)	8	8	8

---

hudi.index_record_record_key$unique

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
record_key	varbinary(255)		PRI	256	256	256
commit_ts	timestamp(3)		PRI	4	4	4
Fixed Size Columns DM/Row				**260**	**260**	**260**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
Total Index DM/Row		**0**	**0**	**0**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	281	16	16
Indexes IM/Row	**281**	**16**	**16**

`RONDB_JDBC` ndb size report.

:

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	20
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	272	272	280

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	1000	1000	1000
Rows/32kb DM Page	120	120	116
Fixedsize DataMemory (KB)	288	288	288
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	29	512	512
IndexMemory (KB)	280	16	16

Parameter Minimum Requirements

* indicates greater than default

Parameter	Default	4.1	5.0	5.1
NoOfTables	128	6	6	6
DataMemory (KB)	81920	960	960	960
NoOfAttributes	1000	15	15	15
NoOfOrderedIndexes	128	8	8	8
NoOfTriggers	768	41	41	41
IndexMemory (KB)	18432	624	72	72
NoOfUniqueHashIndexes	64	3	3	3

ndb_size.pl report for database: 'hudi' (1 tables)

Connected to: DBI:mysql:host=127.0.0.1;port=3306

Including information for versions: 4.1, 5.0, 5.1

Table List

hudi.index_cluster_record

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
record_key	varbinary(255)		PRI	256	256	256
partition_path	varchar(255)	Y		256	256	28*
file_name	varchar(38)	Y		40	40	40*
commit_ts	bigint		PRI	8	8	8
Fixed Size Columns DM/Row				**560**	**560**	**264**
Varsize Columns DM/Row				**0**	**0**	**68**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
idx_record_key	BTREE	N/A	N/A	N/A
PRIMARY	BTREE	N/A	N/A	N/A
Total Index DM/Row		**0**	**0**	**0**

Supporting Tables DataMemory/Row

Table	4.1	5.0	5.1
hudi.index_cluster_record_record_key$unique	276	276	284
This DataMemory/Row	**572**	**572**	**280**
Total DM/Row (inludes DM in other tables)	**848**	**848**	**564**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	289	16	16
Indexes IM/Row	**289**	**16**	**16**

Supporting Tables IndexMemory/Row

hudi.index_cluster_record_record_key$unique	281	16	16
Total Suppt IM/Row	**281**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

`RONDB_CLUSTERJ` ndb size report.

:

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	16
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	572	572	280

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	76

Memory Calculations

No. Rows	1000000	1000000	1000000
Rows/32kb DM Page	57	57	116
Fixedsize DataMemory (KB)	561408	561408	275872
Rows/32kb Varsize DM Page	0	0	429
Varsize DataMemory (KB)	0	0	74624
Rows/8kb IM Page	28	512	512
IndexMemory (KB)	285720	15632	15632

---

hudi.index_cluster_record_record_key$unique

DataMemory for Columns

* means varsized DataMemory

Column Name	Type	Varsized	Key	4.1	5.0	5.1
commit_ts	bigint		PRI	8	8	8
record_key	varbinary(255)		PRI	256	256	256
Fixed Size Columns DM/Row				**264**	**264**	**264**
Varsize Columns DM/Row				**0**	**0**	**0**

DataMemory for Indexes

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
Total Index DM/Row		**0**	**0**	**0**

IndexMemory for Indexes

Index Name	4.1	5.0	5.1
PRIMARY	281	16	16
Indexes IM/Row	**281**	**16**	**16**

Summary (for THIS table)

Fixed Sized Part

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	20
NULL Bytes/Row	0	0	0
DataMemory/Row (incl overhead, bitmap, indexes)	276	276	284

`RONDB_CLUSTERJ` ndb size report.

:

Variable Sized Part

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varsize DM/Row	0	0	0

Memory Calculations

No. Rows	1000000	1000000	1000000
Rows/32kb DM Page	118	118	114
Fixedsize DataMemory (KB)	271200	271200	280704
Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0
Rows/8kb IM Page	29	512	512
IndexMemory (KB)	275864	15632	15632

Parameter Minimum Requirements

* indicates greater than default

Parameter	Default	4.1	5.0	5.1
IndexMemory (KB)	18432	561584*	31264*	31264*
NoOfOrderedIndexes	128	2	2	2
NoOfTables	128	2	2	2
NoOfTriggers	768	13	13	13
DataMemory (KB)	81920	832608*	832608*	631200*
NoOfAttributes	1000	6	6	6
NoOfUniqueHashIndexes	64	1	1	1

TRITA-EECS-EX- 2022:00

For DIVA

{
"Author1": { "Last name": "Zangis",
"First name": "Ralfs",
"Local User Id": "0000000200451376",
"E-mail": "zangis@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Degree1": {"Educational program": "Master's Programme, Software Engineering of Distributed Systems, 120 credits"
,"programcode": "TSEDM"
,"Degree": "Degree of Master of Science in Engineering"
,"subjectArea": "Computer Science and Engineering"
},
"Title": {
"Main title": "Scaling Apache Hudi by boosting query performance with RonDB as a Global Index",
"Subtitle": "Adopting a LATS data store for indexing",
"Language": "eng" },
"Alternative title": {
"Main title": "Skala Apache Hudi genom att öka frågeprestanda med RonDB som ett globalt index",
"Subtitle": "Antagande av LATS-datalager för indexering",
"Language": "swe"
},
"Supervisor1": { "Last name": "Dowling",
"First name": "Jim",
"Local User Id": "0000000294846714",
"E-mail": "jdowling@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Division of Software and Computer Systems" }
},
"Examiner1": { "Last name": "Payberah",
"First name": "Amir H.",
"Local User Id": "0000000227488929",
"E-mail": "payberah@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Division of Software and Computer Systems" }
},
"Cooperation": { "Partner_name": "Hopsworks AB"},
"National Subject Categories": "10201, 10206",
"Other information": {"Year": "2022", "Number of pages": "1,82"},
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": "Karl Axel Pettersson"},
"Presentation": { "Date": "2022-05-25 13:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/2884945301"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€

The storage and use of voluminous data are perplexing issues, the resolution of which has become more pressing with the exponential growth of information. Lakehouses are relatively new approaches that try to accomplish this while hiding the complexity from the user. They provide similar capabilities to a standard database while operating on top of low-cost storage and open file formats. An example of such a system is Hudi, which internally uses indexing to improve the performance of data management in tabular format.

This study investigates if the execution times could be decreased by introducing a new engine option for indexing in Hudi. Therefore, the thesis proposes the usage of RonDB as a global index, which is expanded upon by further investigating the viability of different connectors that are available for communication.

The research was conducted using both practical experiments and the study of relevant literature. The analysis involved observations made over multiple workloads to document how adequately the solutions can adapt to changes in requirements and types of actions. This thesis recorded the results and visualized them for the convenience of the reader, as well as made them available in a public repository.

The conclusions did not coincide with the author's hypothesis that RonDB would provide the fastest indexing solution for all scenarios. Nonetheless, it was observed to be the most consistent approach, potentially making it the best general-purpose solution. As an example, it was noted, that RonDB is capable of dealing with read and write heavy workloads, whilst consistently providing low query latency independent from the file count.

€€€€,
"Keywords[eng]": €€€€
Apache Hudi, Lakehouse, RonDB, Performance, Index, Key-value store €€€€,
"Abstract[swe]": €€€€

Lagring och användning av omfattande data är förbryllande frågor, vars lösning har blivit mer pressande med den exponentiella tillväxten av information. Lakehouses är relativt nya metoder som försöker åstadkomma detta samtidigt som de döljer komplexiteten för användaren. De tillhandahåller liknande funktioner som en standarddatabas samtidigt som de fungerar på toppen av lågkostnadslagring och öppna filformat. Ett exempel på ett sådant system är Hudi, som internt använder indexering för att förbättra prestandan för datahantering i tabellformat.

Denna studie undersöker om exekveringstiderna kan minskas genom att införa ett nytt motoralternativ för indexering i Hudi. Därför föreslår avhandlingen användningen av RonDB som ett globalt index, vilket utökas genom att ytterligare undersöka lönsamheten hos olika kontakter som är tillgängliga för kommunikation.

Forskningen genomfördes med både praktiska experiment och studie av relevant litteratur. Analysen involverade observationer som gjorts över flera arbetsbelastningar för att dokumentera hur adekvat lösningarna kan anpassas till förändringar i krav och typer av åtgärder. Denna avhandling registrerade resultaten och visualiserade dem för att underlätta för läsaren, samt gjorde dem tillgängliga i ett offentligt arkiv.

Slutsatserna sammanföll inte med författarnas hypotes att RonDB skulle tillhandahålla den snabbaste indexeringslösningen för alla scenarier. Icke desto mindre ansågs det vara det mest konsekventa tillvägagångssättet, vilket potentiellt gör det till den bästa generella lösningen. Som ett exempel noterades att RonDB är kapabel att hantera läs- och skrivbelastningar, samtidigt som det konsekvent tillhandahåller låg frågelatens oberoende av filantalet.

€€€€,
"Keywords[swe]": €€€€
Apache Hudi, Lakehouse, RonDB, Prestanda, Index, Nyckel-värde butik €€€€,
}