TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Enrico Vompa 221350 IABM

Data Lakehouse Architecture for Big Data with Apache Hudi

Master's thesis

Supervisors: Tauno Treier
MSc

Raido Ivalo
MSc

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Enrico Vompa 221350 IABM

Andmejärvemaja arhitektuur Apache Hudiga suurandmete jaoks

Magistritöö

Juhendajad: Tauno Treier
MSc

Raido Ivalo
MSc

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Enrico Vompa

08.05.2023

Abstract

The Messaging Data Platform (MDP) of Twilio, a cloud communication platform, is the source of truth for all data related to messaging. The MDP platform is designed to facilitate interactive queries and Extract, Transform, and Load (ETL) jobs but has limitations that impede the effective utilization of both the data lake and its associated data warehouse. These challenges include high data latency, missing data, and poor data discovery.

The objective of this thesis is to propose a novel design that addresses the limitations of the existing MDP system and implement a fully functional data lakehouse utilizing Apache Hudi, capable of managing production traffic of hundreds of thousands of updates per second. The proposed data lakehouse design is expected to store, process, and expose data efficiently while adhering to ACID properties, thereby ensuring effective and dependable data management for the organization.

This document presents an in-depth examination of the current design, the proposed design, and the outcomes of a proof of concept (PoC) implementation. The proposed data lakehouse design serves as a reusable and scalable solution for other organizations and companies seeking for an efficient and dependable data management system for large volumes of data.

This thesis is written in English and is 59 pages long, including 5 chapters, 16 figures and 11 tables.

Annotatsioon

Andmejärvemaja arhitektuur Apache Hudiga suurandmete jaoks

Twilio on pilvekommunikatsiooni platvorm, mis võimaldab arendajatel programmaatiliselt teha telefonikõnesid, saata ja vastu võtta tekstisõnumeid ning teha muid suhtlusfunktsioone, kasutades selle veebiteenuse programmiliidest (API).

Sõnumside andmeplatvorm (MDP) on tõeallikas kõigile sõnumsidega seotud andmetele. See platvorm on loodud, et pääseda ligi andmetele läbi erinevate meetodite, keskendudes Hadoopi ökosüsteemile, mis hõlbustab interaktiivseid päringuid ja ETL-töid. Hetkel paljastab MDP teatud disainipiirangud, mis takistavad nii andmejärve kuid ka sellega seotud andmelao tõhusat kasutust.

Mõned peamised väljakutsed, millega kokku puututakse, hõlmavad kõrget andmelatentsust, puuduvaid andmeid, päringute ebaõnnestumist ajutiste probleemide tõttu ning kehva andmete avastust. Need probleemid tõstavad esile vajadust platvormi edasiarenduse järele, et tagada selle vastavus organisatsiooni ja platformi sidusrühmade nõuetele.

Lõputöö eesmärk on välja pakkuda ja implementeerida uus andmejärvemaja disain, mis adresseerib olemasoleva süsteemi piiranguid, kasutades Apache Hudit, mis suudab hallata sadu tuhandeid uuendusi sekundis. Uus disain peab olema võimeline andmeid tõhusalt salvestama, töötlema ja esitama, järgides ACID-i omadusi, tagades seeläbi usaldusväärse andmehalduse organisatsioonile.

Käesolev lõputöö esitab põhjaliku ülevaate praegusest disainist, uuest disainist ja eduka implementatsiooni valideerimisest. Pakutud andmejärvemaja arhitektuur on taaskasutatav ja skaleeruv lahendus ka teistele organisatsioonidele ja ettevõtetele, kes otsivad tõhusat ja usaldusväärset andmehaldussüsteemi suurte andmehulkade jaoks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 59 leheküljel, 5 peatükki, 16 joonist, 11 tabelit.

List of abbreviations and terms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
BE	Business Event
CDC	Change Data Capture
DaaS	Delivery as a Service
DAG	Directed Acyclic Graph
DB	Database
ETL	Extract Transform Load
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
MDP	Messaging Data Platform
MDR	Message Detail Record
MSE	Message State Event
PII	Personal Identifiable Information
PaaS	Platform as a Service
PoC	Proof of Concept
RDBMS	Relational Database Management System
RPS	Requests per Second
SaaS	Software as a Service
SQL	Structured Query Language

Table of Contents

List of figures

List of tables

1 Introduction

Twilio[1] is a cloud communications platform that enables developers to programmatically make and receive phone calls, send and receive text messages, and perform other communication functions using its web service application programming interfaces (API).

The Messaging Data Platform (MDP) serves as the source of truth for all data related to messaging. This platform has been engineered to offer a range of data access methods, with a focus on the Hadoop ecosystem, facilitating interactive queries, and Extract, Transform and Load (ETL) jobs. However, the present implementation of MDP reveals certain design limitations that impede the effective utilization of both the data lake and its associated data warehouse counterpart.

Some of the key challenges encountered include high data latency, missing data, frequent failures of otherwise valid queries due to transient issues, and poor data discovery. These issues highlight the need for further refinement and optimization of the platform to ensure that it aligns with the evolving requirements of the organization and its stakeholders.

The objective of this thesis is to propose a novel design that addresses the limitations of the existing system and implement a fully functional data lakehouse utilizing Apache Hudi[2], capable of managing production traffic of hundreds of thousands of updates per second. The newly designed system is required to exhibit the ability to store, process, and expose data efficiently while adhering to ACID (Atomicity, Consistency, Isolation, Durability) properties, thereby ensuring effective and dependable data management for the organization. This document presents an in-depth examination of the current design, the proposed design, and the outcomes of a proof of concept (PoC) implementation, in which the results are thoroughly validated.

The proposed data lakehouse design serves as a reusable and scalable solution for other organizations and companies seeking for an efficient and dependable data management system for large volumes of data.

The author of this thesis identified the existing challenges and translated them into new functional requirements. Subsequently, the author engaged in the investigation of potential solutions, participated in the development of the new system's design, and implemented a PoC for the proposed solution. Upon completion of the implementation, the author validated the system's functionality, documented the results and observations, and presented the finalized project.

[1] https://www.twilio.com
[2] https://hudi.apache.org

2 Background

MDP serves as a centralized hub for all messaging-related data, effectively acting as the single source of truth for all messaging across the organization. With numerous upstream sources contributing data to MDP, the platform's primary function is to collect, consolidate, and harmonize this information, making it accessible through various systems. A key area of focus for the MDP team is the Hadoop ecosystem, which empowers data scientists, analysts, and engineers to seamlessly run interactive queries and execute ETL jobs to facilitate further data processing and analysis.

To accommodate the diverse analytical requirements of various stakeholders, MDP supports an extensive range of procedures, encompassing data exploration, trend and pattern analysis, ad-hoc analytics, complex analytics, reports, dashboards, and more [20]. In the past, data was predominantly stored in a data lake, with concerted efforts to expose it through a data warehouse system to enable more efficient analytics.

The MDP team's overarching goal is to streamline the process of setting up new integrations and minimize operational burden, thereby enhancing the overall user experience for data scientists, analysts, engineers, and other stakeholders. By emphasizing the centralization of messaging data, the MDP team plays a critical role in promoting data-driven decision-making across the organization. Their efforts contribute to fostering a culture of data literacy and ensuring that insights derived from data analysis are effectively leveraged to drive organizational growth and innovation.

This chapter outlines the existing architecture and main technologies used there, identifies its problems, and highlights the need for a novel architectural approach, detailing the functional requirements the system must fulfil.

2.1 Current technologies

The current technological landscape for data processing and management relies heavily on several key components, which have become integral to modern data-driven applications.

Amazon DynamoDB[1] (DDB) is a fully managed, serverless NoSQL database service from AWS. It supports key-value and document data models and provides automatic scaling, enabling seamless data management across various use cases. [36]

Apache Kafka is a distributed streaming platform used for building real-time data pipelines and streaming applications. It is a durable message broker that enables applications to process, store and forward data streams in a fault-tolerant, highly available and scalable manner. Kafka can be used as a source or for real-time data streaming applications, offering seamless integration between data producers and consumers. [7, 9]

Amazon Simple Storage System (S3)[2] is a scalable, secure object storage service designed for diverse data storage needs, from archiving to big data analytics. It provides high durability and availability, allowing for easy storage and retrieval of data from anywhere on the internet. [31]

Amazon Elastic MapReduce (EMR) is a managed cloud service, designed to streamline the deployment and management of big data processing frameworks, such as Apache Spark. This service allows users to efficiently handle large-scale data processing tasks in a distributed computing environment. [35]

Apache Spark is a powerful open-source data processing engine that enables rapid analysis and processing of vast datasets by distributing the workload across multiple machines. It is widely used for various data-intensive tasks, including data analytics, machine learning, and pattern recognition. [33]

The Spark Java API is a specialized interface that allows developers proficient in the Java programming language to create distributed data processing applications using Apache Spark. By leveraging the capabilities of Amazon EMR, developers can efficiently allocate computing resources and dynamically scale their applications to accommodate the demands of processing large volumes of data. [33, 35]

Apache Presto[3] is a distributed SQL query engine designed to perform interactive analytic queries across various data sources. Its high-performance, in-memory processing capabilities and support for different data formats make it a popular choice for unified big data analytics. [34]

The Hadoop ecosystem is an extensive collection of open-source tools and frameworks aimed at facilitating the storage, processing, and analysis of large-scale, distributed data sets. Key components of the ecosystem include Hadoop Distributed File System (HDFS), which provides reliable and scalable storage, and the MapReduce programming model, which enables efficient distributed data processing. Additionally, the ecosystem integrates with other technologies such as Presto, a high-performance SQL query engine for interactive analytics, and Amazon S3, a scalable and secure object storage solution. [33]

[1] https://aws.amazon.com/dynamodb/
[2] https://aws.amazon.com/s3/
[3] https://prestodb.io/

2.2 Current design

To summarize the functions of these aforementioned technologies, DDB serves as the repository for record storage, while Apache Kafka is responsible for transmitting CDC logs for every record updated in DDB. On Hadoop ecosystem, Amazon EMR executes Spark jobs, which deduplicates CDC logs. Subsequently, the results are stored in Amazon S3, and Apache Presto is utilized to run queries on these records. Current architecture with these technologies is displayed on Figure 1.

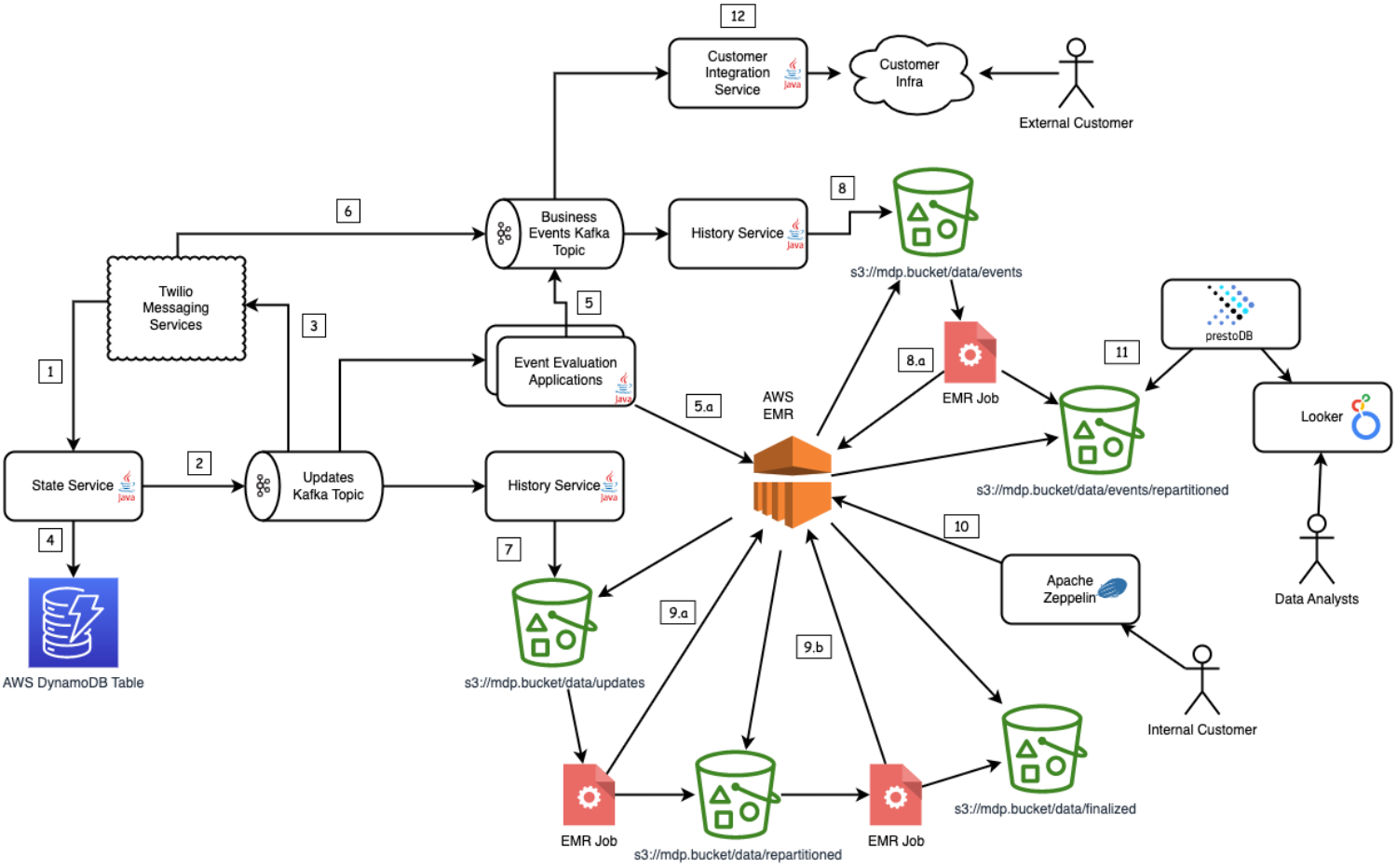


Figure 1. Architecture of existing system

With current design, EMR jobs operate on Spark Java API. The Spark Java API, while powerful and capable of processing large amounts of data, lacks in providing utilities for common tasks such as effective data manipulation through file-level updates. These utilities can be implemented manually, but it is time-consuming and would be equivalent to recreating the functionality for which there is an open-source counterpart already available. If possible, it would be beneficial to utilize existing technologies to save time and reduce development costs.

Since Spark API only supports partition rewrites for data updating, multiple jobs must be chained together, such as interactions 7, 9.a, and 9.b on Figure 1. This design has made the infrastructure more difficult to maintain and has problematic side effects.

Interactions on Figure 1 and problematic side effects that occur are discussed in Table 1.

Table 1. Description and details of Figure 1

#	**Interaction**	**Description and Details**
1	Update state	Upstream service updates message state by issuing a HTTP request to state service. State service is a data access layer service on top of a DynamoDB table for inflight data. State service finds an existing data record in database (DB) and merges upstream changes into it, saving data back to DB. If no record is found, a new record is created. All specifics like hot partitioning, backoffs, retries are handled by this service and auxiliary service, which are out of scope of this thesis. During peak hours, the system experiences a rate of approximately 300,000 updates per second, while maintaining an average rate of 100,000 updates per second.
2	Publish state	State service publishes a successful state update into Kafka topic. For each successful update a message state event (MSE) is evaluated as well.
3	Consume updates	Services consume a Kafka topic with update events and build logic around received events and state changes.
4	Store state	The primary use-case for this system involves simple key-value operations, specifically the creation and updating of records. It employs an optimistic locking mechanism and implements retries in case of conflicts or errors.
5	Event evaluation	Event application translates MSE into a business event (BE) and publishes the BE to a Kafka topic.
5.a	Historical event evaluation	ETL applications designed for use with EMR analyse S3 buckets containing historical data to extract BEs for further processing and analysis.
6	Event publishing	Upstream services function as sole publishers of BEs. They have direct publisher access to a specific Kafka topic.

<table>
<tr><th>#</th><th>Interaction</th><th>Description and Details</th></tr>
<tr><td>7</td><td>History storage</td><td>Message History Service is responsible for storing batched Message Detail Records (MDRs) in S3, which is further elaborated in Table 2.

Table 2. MDR updates data layout properties

<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td>Format</td><td>Apache ORC[1]</td></tr>
<tr><td>Path</td><td><code>data/updates</code></td></tr>
<tr><td>Partitioning</td><td>Each consumed batch is grouped by partitioning function: f: (accepted) → (year, month, day, hour)

Each group is transformed into ORC and stored in S3 with an example path of: <code>s3://data/updates /year=2022/month=7/day=12/hour=4/batch-{random}.orc</code></td></tr>
<tr><td>Notes</td><td>Median file size is less than 100 kilobytes.</td></tr>
<tr><td>Size</td><td>2103 TiB</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Event storage</td><td>Different instance of Message History Service is responsible for storing batched BEs in S3, which is further elaborated in Table 3.

Table 3. BE updates data layout properties

<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td>File format</td><td>Apache Parquet[2]</td></tr>
<tr><td>Path</td><td><code>data/events</code></td></tr>
<tr><td>Partitioning</td><td>Each consumed batch is grouped by partitioning function: f: (accepted) → (year, month, day, hour)

Each group is transformed into Parquet and stored in S3 with an example path of: <code>s3://data/events /year=2022/month=7/day=12/hour=4/batch-{random}.parquet</code></td></tr>
<tr><td>Notes</td><td>Median file size is less than 100 kilobytes.</td></tr>
<tr><td>Size</td><td>135 TiB</td></tr>
</table>
</td></tr>
</table>

[1] https://orc.apache.org/
[2] https://parquet.apache.org/

#	Interaction	Description and Details
8.a	Finalizing BEs	Problem #1. Poor data analysis performance of data/events storage due to a big number of small-sized files in the folder. In order to address this issue, EMR job has been implemented, which repartitions the small files from the "data/events" S3 bucket and stores them in a "data/events/repartitioned" subfolder. Operations done, henceforth referred to as "finalization": ▪ Optimize for bigger file size ▪ Repartition by event type ▪ Deduplicate The rationale behind this approach is to mitigate performance and consistency challenges associated with conducting analysis on the "data/events" bucket. Due to the current implementation of finalization, which rewrites the entire partition when updating finalized records, this process gives rise to a problem #2: the replacement of files in the "data/events/repartitioned" folder disrupts concurrently running queries in upstream systems. A demonstration of the aforementioned issue is provided below: 1. Consider a scenario where we have the following file: "`data/events/repartitioned/eventtype=1/year/month/day/hour/batch-12345.orc`". 2. A query is executed on "`batch-12345.orc`". 3. New data arrives at the location: "`data/events/eventtype=1/year/month/day/batch-12345.orc`". 4. An ETL job is required to replace "`batch-12345.orc`". However, this action disrupts the query on the repartitioned data layout, resulting in a `FileNotFoundException`. Not all BEs necessitate partitioning by year, month, day, and hour, and some events may require different types of optimizations. This gives rise to problem #3: the same partitioning and optimization strategies are applied to all events, which may not be universally effective. In other words, a one-size-fits-all approach is not suitable for addressing the unique requirements of each event. As rewriting the entire table is inefficient, a more targeted approach is employed by focusing on a subset of data. However, this approach leads to Problem #4: High data latency and potentially missing data.
9.a	Optimizing MDR updates	Same Problem #1. To overcome low performance on big number of small-sized files in initial data/updates, EMR runs ETL job to optimize data for analysis, which combines smaller files to larger files.

#	Interaction	Description and Details
9.b	Finalizing updates	For each message, there is an eventual history of its states. Those states can be represented as ordered cumulative change history (MDR1, MDR2, ..., MDRn), where MDRn contains all the information the pervious states contained. `data/finalized` is an S3 subfolder which is a target of ETL job that transform all historical data into final data, skipping all intermediate states. It is essentially a data layout where for each of existing messages only the last state of it is stored. Biggest peculiarity of finalization is that there is no such thing as final state, i.e., there is no state after which there are no guaranteed updates to it. Updates to the message state may occur over an extended period, potentially spanning several months. Because of that, data/finalized is prone to problems #2 and #4.
10	Performing internal analysis	Apache Zeppelin[1] serves as an internal tool for conducting comprehensive analysis on all existing data locations. Zeppelin submits an ETL job to the EMR cluster that has access to all mentioned S3 storage locations. As mentioned above, it has flaws in form of problems #2 and #4 for all the folders where optimization or finalization ETLs are working.
11	Working with external data warehouses	MDP plugs data to Presto and by so, also to Looker.[2] Integration operates on automated scripts; however, this solution is not scalable when applied to more complex schemas containing nested fields, such as those found in MDRs. Consequently, this gives rise to Problem #5: There is no scalable integration with external data warehouses.
12	Publishing to external customers	BEs topics are consumed by consumer integration service that translates BEs into specific customer-defined schemas.

MDR updates table will be referenced as updates layout. MDR finalized table and problematic BE tables will be referenced as finalized layouts.

[1] https://zeppelin.apache.org/
[2] https://www.looker.com/

2.3 Functional requirements

The goal of this thesis is to construct a system that meets the functional requirements, some of which have been recently introduced:

1. Can handle petabyte scale.
2. Is adaptable to different use-cases:
    - Late arriving updates.
    - CDC log deduplication.
3. Customizable to accommodate all use-case related peculiarities.
4. Solution is robust. System handles errors gracefully without side effects.
5. Solution avoids the small file issue, which is detrimental to query effectiveness.
6. Snapshot isolation between writers and readers.
7. Data discovery is simple.
8. Data should be readable with low data and query latency.
9. Support for various analytical procedures, such as exploring the data, trying to understand various trends and patterns, ad-hoc analytics, complex analytics, reports, dashboards, and more [20].

To address the existing problems, new functional requirements were established within the design of the new system, as illustrated in Table 4.

Table 4: New functional requirements to address problems

Problem	**Functional requirement**
Problem #1. Poor data analysis performance due to a big number of small-sized files.	5. Solution avoids the small file issue
Problem #2: replacing files break concurrently running queries in upstream systems.	6. Snapshot isolation between writers and readers.
Problem #3: Same partitioning and optimizations do not fit all use-cases.	3. Customizable to accommodate all use-case related peculiarities.
Problem #4: High data latency and potentially missing data.	8. Data should be readable with low data and query latency.
Problem #5: There is no data warehouse.	7. Data discovery is simple.

The lack of utility in the Spark Java API suggests that an alternative needs to be sought out in order to provide the necessary functionality. It is beneficial to explore alternative technologies that offer similar capabilities to Spark Java API and determine which one is best suited to meet the specific needs of the mentioned use-cases.

2.4 Novelty

The author of the paper was unable to find a comprehensive solution that adequately addressed the problem they were attempting to solve. This highlights the need for further research and innovation in the field to address the challenges associated with managing and analysing complex and diverse datasets.

Historically, there has been a clear differentiation between Data Lakes and Data Warehouses; however, Apache Hudi provides the means of connecting the two into a single entity called Data Lakehouse. [18]

Comparison between Data Warehouse, Data Lake and Data Lakehouse is in Table 5.

Table 5. Data Warehouse, Data Lake and Data Lakehouse comparison [18]

Feature	**Data Warehouse**	**Data Lake**	**Data Lakehouse**
Data	Structured Processed	Structured, semi-structured, unstructured; Raw	Structured, semi-structured and unstructured. Both processed and raw
Processing	Schema-on-write	Schema-on-read	Schema-on-write, Schema-on-read
Storage	Expensive for large data volumes	Designed for low-cost storage	Designed for low-cost storage
Agility	Less agile, fixed configuration	Highly agile, adjustable configuration	Highly agile, adjustable configuration
Security	Mature	Maturing	Both mature and maturing
Users	Business professionals	Data Scientists et. al.	Whole business environment

The system needs to be able to handle the production traffic of hundreds of thousands of records per second, which many companies in general are now requiring. This work provides guidelines on constructing a scalable system to manage an influx of data, as well as identifying problems to avoid.

The author of this work identified the issues that needed to be addressed and converted them into new functional requirements. Subsequently, they assisted in researching alternative solutions, contributed to the design of the new system, and implemented the proposed solution. Following the implementation, they verified the system's functionality, documented the results and findings, and delivered the completed project.

3 Planning

To mitigate the problems with current design, we must look back and align the existing view of modern data solutions and platforms to what has already been built. We need to accommodate our existing product goals and requirements and figure out what could be changed in existing design to reach and implement them. Because the change is fundamental, then every viable alternative solution will be considered. [12]

Proposed design needs to mitigate mentioned problems by explicitly extracting concepts of data lake, data warehouse and data catalogue into dedicated services which can fill the functional requirements.

To formalize this new design, we need to create a detailed architecture that outlines the functional and technical requirements, data flows, and the overall architecture of the new design. This architecture includes diagrams and other helpful visualizations to explain how the different components will interact with each other. Additionally, this architecture includes the steps necessary for implementing the new design.

Twilio adheres to a policy of using as many managed solutions as possible to minimize operational costs. As such, the company has chosen Amazon to host their systems. Amazon's business model revolves around adapting open-source solutions into managed solutions. Nevertheless, this approach does not detract from the adaptability of the solutions for use by other organizations and companies, since the adapted open-source solutions remain open-source and are available for use by anyone via Amazon or not.

3.1 Deployment model

Running an ETL job consists of multiple components:

- A coordinator which communicates with workers and tells them what to do.
- A worker which does the actual work that gets delegated to it.
- Underlying data storage where data gets read from and written to.

Any of the components can be either: [3]

- Completely on-premise, where company hosts their own hardware.
- Cloud instances, such as Amazon EC2[1], through infrastructure as a service (IaaS).
- Cloud service through software, platform or delivery as a service (SaaS, PaaS or DaaS).

However, since our company does not have any on-premise infrastructure and only uses cloud instances or cloud services, on-premise deployment is not an option. Therefore, we must decide whether to implement each component using cloud instances or cloud services.

Our company policy requires that all data storage and computational engines use cloud services. Final option that remains is to choose whether the coordinator uses a cloud instance or a cloud service.

Using a cloud instance provides much more flexibility in terms of customizing the instructions which are sent to the workers making it the preferable option. Additionally, using a cloud instance is also more cost-effective than using a licensed PaaS or SaaS on a petabyte level.[2] Furthermore, cloud services widely rely on most successful open-source projects [28]. Using these open-source projects themselves directly will accomplish the same result.

Considered options for cloud services:

- Google Snowflake[3]
- Amazon Redshift[4]
- Databricks Lakehouse Platform[5]

Given that the company is heavily dependent on Amazon's cloud stack, integrating cloud services from another provider would likely result in significant complications. This could include obtaining the necessary security approvals, setting up the required integrations with existing infrastructure, and addressing compatibility issues. Therefore, with everything considered, in the next section, only Amazon Redshift will be further considered.

[1] https://aws.amazon.com/ec2/
[2] https://askwonder.com/research/data-lake-warehousing-pricing-t1nu01wri
[3] https://www.snowflake.com/en/
[4] https://aws.amazon.com/redshift/
[5] https://www.databricks.com/product/data-lakehouse

3.2 Technologies chosen

Currently, the data layouts are operating on the Hadoop ecosystem[1]; however, it would be beneficial to take a step back and consider other fundamental alternatives:

- Cassandra and other key-value stores lack support for joining with external datasets and performing aggregations, thereby rendering them unsuitable for fulfilling functional requirement number nine. [1]
- Graph databases could potentially be useful, but because MDP data has little to no relations and instead all information is already present in a single record, then it does not make sense to use it even in conjunction with other technologies. [2]
- ElasicSearch[2] is not meant to be used to store petabytes of data. However, they make for a good store for aggregated results of ETL jobs. [24] Example is to run ETL jobs on Hudi data lake and publish results to Kafka as business events, which are currently implemented as Event Evaluation Applications seen on Figure 1. [25]
  The two solutions here complement each other.
- Some query engines may have support for various relational database management systems (RDBMS[3]) [26]. However, the query will be running RDBMS-side which means that it will not be possible to join the dataset to external Hadoop based tables. It will also mean that the server needs to be properly provisioned to handle the influx of read requests and queries will be limited by server resources which results in most solutions becoming unusable.
- Even though Redshift[4] is built on PostgreSQL[5], a RDBMS, it exceeds where others failed. While it performs better and more cost-effectively on smaller workloads as compared to Hadoop, the opposite can be said for large workloads where queries are over billions of rows. For smaller workloads, Hadoop based solution remains to be competitive and not far behind. Redshift also requires reshuffling data which slows down the system and blocks other operations. Hadoop does not require any reshuffling. Most of company's data is already on Hadoop ecosystem, and in order to join Redshifts data to it, Hadoop data needs to be loaded to Redshift. With everything considered, the price for Redshift would be considerably higher in terms of cost and engineering work once solution is up and running. [27, 32]
  The two solutions here also complement each other. For solving the functional requirements described in this paper, Hadoop based solution works best, but use-cases with smaller volume, Redshift solution can be employed alongside with Hadoop.

It is safe to conclude that an alternative solution to Spark Java API on the Hadoop ecosystem should be used. Next, a framework must be chosen which operates on Hadoop and can be utilized by the orchestrator.

[1] https://www.edureka.co/blog/hadoop-ecosystem
[2] https://www.elastic.co/
[3] https://www.oracle.com/database/what-is-a-relational-database/
[4] https://aws.amazon.com/redshift/
[5] https://www.postgresql.org/

There were 3 contenders when choosing the right technology for orchestrator, for which there is comparison between Apache Hudi, Delta Lake and Iceberg is in Table 6.

Table 6. Comparison between Apache Hudi, Delta Lake, and Iceberg [23]

Feature	**Apache Hudi (v0.12.2)**	**Delta Lake (v2.2.0)**	**Iceberg (v1.1.0)**
ACID transactions	+	+	+
File versioning (Copy-on-Write)	+	+	+
Amortized updates (Merge-on-Read)	+	-	+
Concurrency (Optimistic-Concurrency-Control)	+	+	+
Time travel (Point-in-Time queries)	+	+	+
Deduplication	+	+	+
Record level indexes (Efficient updates to data)	+	-	-
Automated file sizing	+	-	-
Compaction/Clustering (Combining files)	+	-	-
Automatic cleaning (Old version deletion)	+	-	-
Schema evolution	+	+	+
Disaster recovery (Savepoints)	+	+	-
Automatic monitoring (Publishes metrics)	+	-	-

The feature support comparison presented in this table is limited and specifically tailored to the needs of the author. It does not encompass all features of the frameworks and only focuses on those that are relevant for the current work.

For orchestrator, Hudi was chosen because of its resiliency, transactionality, customizability, record level indexing feature and in general, it had the potential to fill all the functional requirements. [19]

Finally, a decision must be made on the computational engine on which to run Hudi and the location in which to store the resulting data.

Considered query and analytics engines which operate on Hadoop:

- Apache Tez[1] is partially supported by Hudi. It is possible to read though Tez, but not incrementally load new data. [6] In addition, it has low performance. [22]
- Apache Storm[2] is for streaming workloads where output source benefits from small batches or record level writes, which is not the case with Hudi making it not the correct tool for the job. [13]
- Impala[3], DorisDB[4], StarRocks[5], Presto etc are great for running analytical queries, but they are not made to write data. However, it is possible to query data via these systems through Hudi or Hive connector. [19]
- Pivotal HWAQ[6] is primarily meant for more computational workloads. [22] While current workload is an I/O workload.
- Apache Hama[7] is meant for processing graphs. [16]
- Spark is fully supported by Hudi [19]. Spark also has high performance [23].

For workers, a popular analytics engine, Spark, was chosen, because infrastructure for it was already there and there were no better alternatives for which Hudi or equivalent alternative already had write support for.

Technologies which were considered for storing tables:

- Apache Ignite[8] is effective for small workloads, but since the solution needs to handle petabyte scale, then storing all of it in RAM is not feasible. [17]
- Hadoop Distributed File System (HDFS) is a good option to achieve low latencies, so it is used when spilling shuffle data to disk between Spark job stages, but it's expensive and hard to integrate with for table data storage. [4]
- Amazon S3 is cheap, robust, has excellent availability and is easy to set up, maintain and integrate with. [31]

In terms of storage, Amazon S3 was selected as it was deemed superior to other options available.

Having selected the appropriate technologies, the subsequent step is to identify configurations that will enable the fulfilment of the functional requirements.

[1] https://tez.apache.org/
[2] https://storm.apache.org/
[3] https://impala.apache.org/
[4] https://doris.apache.org/
[5] https://www.starrocks.io/
[6] https://hawq.apache.org/
[7] https://hama.apache.org/
[8] https://ignite.apache.org/

3.3 Hudi configurations

Hudi offers a variety of configurations, which are taken from official Hudi homepage. [19]

Most impactful one being the table type configuration.

3.3.1 Table types

Copy-on-Write (CoW) table stores data exclusively in columnar file formats. Updates are performed by versioning and rewriting the files, with a synchronous merge taking place during write. File slices only contain the base/columnar file, and each commit produces new versions of the base files. This means that only columnar data exists, leading to higher write amplification (number of bytes written for 1 byte of incoming data) and zero read amplification. This is a desirable property for analytical workloads, which are predominantly read-heavy.

The CoW table aims to improve how tables are managed by:

- Providing first-class support for atomically updating data at the file-level, instead of rewriting whole table or partition.
- Offering the ability to incrementally consume changes, as opposed to scanning the whole table or partition.
- Allowing tight control of file sizes to maintain excellent query performance.

Merge-on-Read (MoR) table stores data using a combination of columnar and row-based file formats. Updates are logged to delta files, which are later compacted to produce new versions of the base files, both synchronously and asynchronously.

MoR is a superset of CoW since it still supports read-optimized queries of the table by exposing only the base/columnar files in the latest file slices. Additionally, it stores incoming upserts for each file group in a row-based delta log, to support snapshot queries by applying the delta log onto the latest version of each file ID on-the-fly during query time. Thus, this table type attempts to balance read and write amplification intelligently, to provide near real-time data. The compactor is especially important here, since it needs to carefully choose which delta log files should be compacted onto their columnar base files, to keep query performance in check.

The intention of the MoR table is to enable near real-time processing directly on top of DFS, rather than copying data out to specialized systems which may not be able to handle the data volume. There are also a few secondary benefits such as reduced write amplification by avoiding synchronous merge of data.

Given that enabling near real-time data is not currently a requirement, then all tables will be in CoW table type as it is easier to set up and maintain. Since MoR is a superset of CoW, then table type can always be upgraded from CoW to MoR when it becomes a requirement.

3.3.2 Operation types

Another impactful configuration is the operation type which is used when writing data.

For UPSERT operation, input records are first tagged as inserts or updates by looking up the index. The records are ultimately written after heuristics are run to determine how best to pack them on storage to optimize for things like file sizing. This operation is recommended for use-cases like database change capture where the input almost certainly contains updates. The target table will never show duplicates.

INSERT operation is very similar to upsert in terms of heuristics and file sizing but completely skips the index lookup step. Thus, it can be a lot faster than upserts for use-cases like log de-duplication. This is suitable for use-cases where the table can tolerate duplicates and just needs the transactional writes/incremental pull/storage management capabilities of Hudi.

Both upsert and insert operations keep input records in memory to speed up storage heuristics computations (among other things) and thus can be cumbersome for initial loading and bootstrapping a Hudi table at first. BULK_INSERT operation provides the same semantics as insert, while implementing a sort-based data writing algorithm, which can scale very well for several hundred TBs of initial load. However, this just does a best-effort job at sizing files rather than guaranteeing file sizes like inserts and upserts do.

Finalized layouts which have a functional requirement of only containing the latest version of the record will use UPSERT operation. The updates layouts, which do not have a hard requirement on containing duplicates, will use INSERT operation, which does not guarantee record key uniqueness, but still delivers good results for much lower performance costs. When backfilling the tables, BULK_INSERT operation can be used for fastest results.

3.3.3 Index types

Hudi provides efficient upserts by mapping a given hoodie key, which consists of a record key and a partition path, consistently to a file identifier via an indexing mechanism. This mapping between record key and file group/file id, never changes once the first version of a record has been written to a file. This enables fast upsert and delete operations by avoiding the need to join against the entire dataset to determine which files to rewrite.

Hudi exposes the following index types out of the box.

- **Bloom Index** employs bloom filters built out of the record keys, optionally also pruning candidate files using record key ranges.
- **Simple Index** performs a lean join of the incoming update or delete records against keys extracted from the table on storage.
- **HBase Index** manages the index mapping in an external Apache HBase[1] table, which is a key/value store. This will be prohibitively expensive for big volumes compared to other index types.

Bloom index and Simple index can either be per partition or over the whole table. If the index should span over the whole table, then the global variant of the index should be used.

Existing finalized tables will use Bloom indexing, as it is the most effective option given the nature of the data where there are multiple updates to a single record over the span of weeks. By default, bloom filters are stored in data files, but they can also be stored in metadata table, which make lookups faster.

[1] https://hbase.apache.org

3.4 First iteration

After evaluating various technologies and finalizing the functional requirements with architects, author proceeded with the first iteration. In the first iteration of the solution, key changes are made to address the issues with the existing design and to fulfil the technical functional requirements, including poor analysis performance, high data latency, and missing data. These requirements will be further elaborated in Table 8. The primary aim of the first iteration is to maximize the use of the existing infrastructure while evaluating the feasibility of the proposed technologies. The design for the first iteration is illustrated in Figure 2.

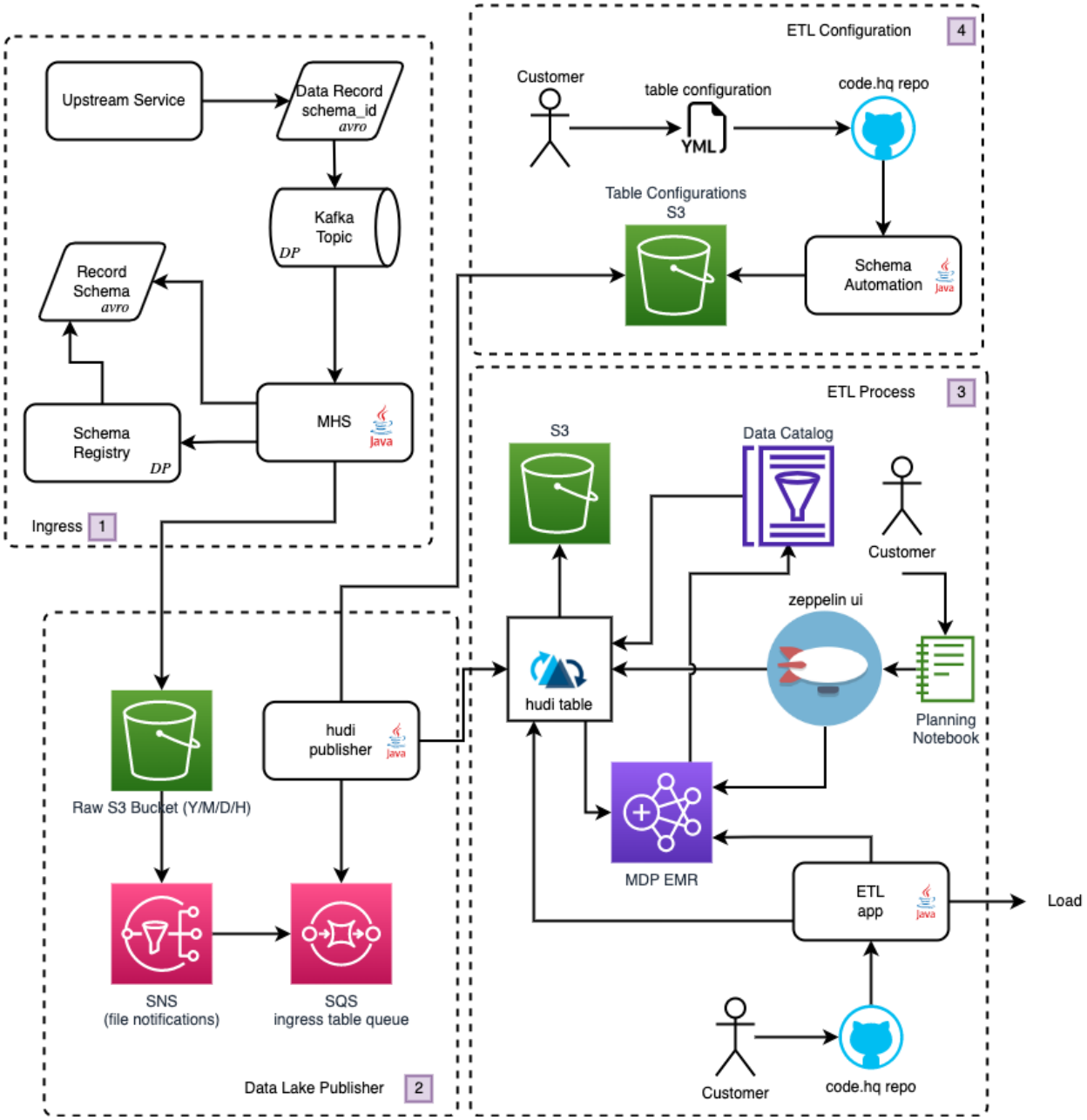


Figure 2. Architecture diagram of first iteration

Domains of the first iteration illustrated in Figure 2 are described in Table 7.

Table 7. Domains and descriptions of first iteration

#	**Domain**	**Description and Details**
1	Ingress	Ingress domain remains the same as described in Table 1.
2	Data Lake Publisher	Raw bucket is partitioned by year, month, day and hour partitions like the pervious updates bucket, but with the partitioning timestamp as current timestamp. This approach prevents late updates from creating small files that slow down Spark. [17] Every time a new file is uploaded to S3, the bucket is configured to send a notification to Simple Notification Service (SNS), which is a messaging service that enables applications, end-users, and devices to send and receive notifications. The notification is then fanned out to Simple Queue Service (SQS), a fully managed message queuing service that enables decoupling and scaling of microservices, distributed systems, and serverless applications. [29] The Hudi publisher periodically polls new files from SQS and then processes them into a Hudi table based on the provided configurations. SQS is a re-playable input source, so if Hudi application runs into an exception, then it will not result in data loss. [5]
3	ETL Process	Data is written to S3 using Hudi writer, which operates on EMR nodes that use Spark. After which the Hudi reader can be used to read that table through various query engines, making it into a fully functioning data lake.
4	ETL Configuration	Raw instructions for processing items in every SQS queue are stored in GitHub[1]. GitHub is a web-based platform for version control and collaboration that allows developers to store and manage their code repositories. Any configuration changes go through a review process and are automatically propagated to every environment by a pipeline after being approved.

The Apache Hudi framework serves as an integral element within the overarching architecture. It constitutes the core technology that facilitates the realization of the design objectives.

[1] https://github.com/

The first iteration addressed majority of the challenges associated with the existing architecture, as explicated in Table 8.

Table 8. First iteration to solve the problems

Problem	**Resolution**
Problem #1. Poor data analysis performance due to a big number of small-sized files.	Hudi supports file clustering[1] and compaction[2], which allow combining existing data and appending new data to existing data respectively, resulting in larger files.
Problem #2: replacing files break currently running queries in upstream systems.	Whenever Hudi writes new data, it creates a new version of the file and later cleans up the old files to reclaim space using the cleaner service[3] once the files are no longer used.
Problem #3: Same partitioning and optimizations do not fit all use-cases.	Here partitioning and general optimization scheme is customizable and is stored in GitHub which eventually makes it to the target Hudi table, allowing each table to be configured differently.
Problem #4: High data latency and potentially missing data.	Because raw bucket has unpartitioned data, then all data gets handled with same priority, and so, lowering worst case data latency. Hudi is also much more performant, because it only updates required files not all files in a partition. And because items from SQS are deleted only after that data made it to Hudi table, then no data will go missing.
Problem #5: There is no data warehouse.	This will be fully resolved in second iteration with the introduction of data catalogues, where Hudi allows updating partition data straight from writer itself after committing changes.[4]

The first iteration of the project has been a success, as it has shown that it is feasible to use existing infrastructure and Hudi framework to build the proposed system. Proof of the resolution will be provided in the results section of this paper.

However, it was necessary to fine-tune the Spark configurations to handle large volumes of data, and the methodology for doing so is discussed in the next section of this paper.

[1] https://hudi.apache.org/docs/clustering
[2] https://hudi.apache.org/docs/compaction
[3] https://hudi.apache.org/docs/hoodie_cleaner
[4] https://hudi.apache.org/docs/syncing_aws_glue_data_catalog

3.5 Spark configurations

Spark operates on a batch processing principle, where batch jobs are organized into directed acyclic graphs (DAGs). This enables Spark to efficiently process data in batches, by breaking down a batch job into separate tasks and optimizing their execution accordingly. Tasks are constructed in a way that all updates for a single file-group are grouped up into a single task.

The Bloom index filter is used to efficiently narrow down a large dataset by discarding files that do not match the given criteria. The remaining files are then sorted and merged using a Sort-Merge Join algorithm to return a single, sorted set of results. For upsert, merging consists of updating the existing row with custom logic. For our use-cases the updates are cumulative for now, so simply replacing the old version with new version is enough. In future, when different merging logic is needed, then it can be implemented. This algorithm is particularly useful when dealing with large datasets, as it can quickly narrow down the search and efficiently return the desired result, which all Hudi tables use when upserting data. [14]

Due to the nature of our data, which involves multiple updates per record across significant number of partitions, we benefit from using large batches. This is because it helps us reduce the overhead associated with multiple overlapping jobs:

- Updates against the same record.
- Updates against the same file group.

When processing tasks, Spark tends to run out of memory, so it is important to ensure that Spark has enough of memory to process the task by increasing the parallelism for different stages of jobs. As mentioned before, Hudi framework itself already partitions incoming data optimally to groups where updates to same file groups are grouped together, so increasing parallelism works out of the box.

Spark executors are individual processes launched for an application on a given worker node, and they are responsible for executing tasks. Executor configurations are computed based on the task size, which is deterministic since input size and parallelism are known [11]. For example, if we have 1 TB of files to process with parallelism of 1000, then each executor will be assigned 1 GB chunk of data to be processed.

Spark reads task input from HDFS and writes task output to HDFS. It is essential to ensure that HDFS is properly configured, which can be a lengthy process [8]. However, since we are using Amazon EMR as our MapReduce cluster, configuring HDFS is drastically simplified. All that is required is to ensure that there is sufficient disk space and IOPS available on the machines, eliminating the need for lengthy HDFS configuration processes.

3.6 Second iteration

In the previous implementation, a significant limitation was the inability to query data from external systems, such as Presto, Athena, and Looker. As a result, the majority of data analysts and scientists within the company could not access the data for their use. The second iteration of the solution does not introduce fundamental changes to the architecture. Instead, it aims to enhance the visibility of the solution throughout the organization and broaden its capabilities to accommodate a more diverse range of use cases. For instance, this iteration allows analysts to execute queries on messaging data via existing Amazon offerings like Athena and data platform team offerings, such as Presto. Consequently, this enables the creation of dashboards, reports, and other metrics, which can inform business decisions. Additionally, it allows for the results to be sent to external systems for further processing and the development of new systems through ETL jobs. The architecture diagram for the second iteration can be found in Figure 3.

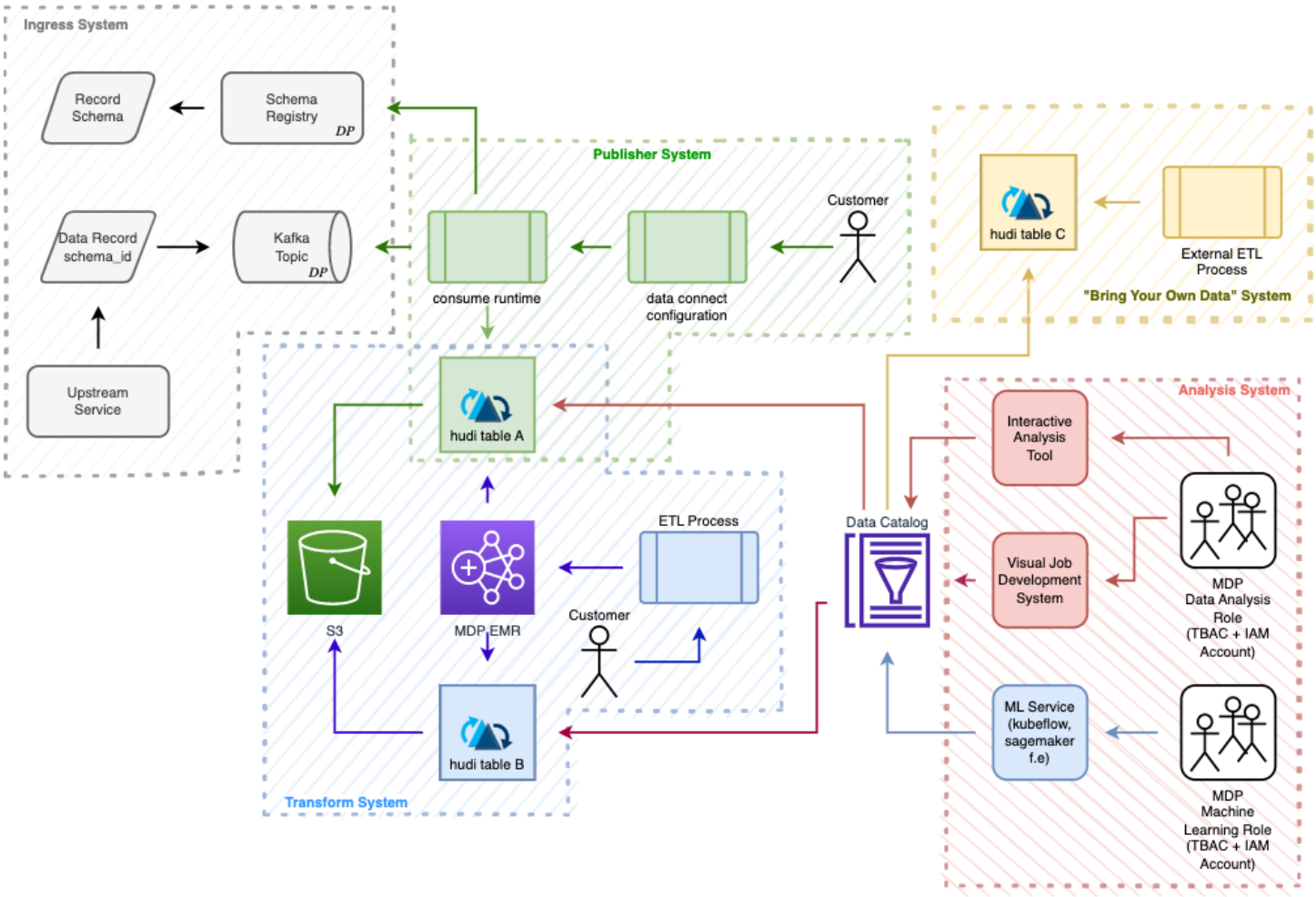


Figure 3: Architecture diagram of second iteration

Domains are further elaborated in Table 9.

Table 9. Domains and descriptions of second iteration

Domain	**Description and Details**
Ingress System	Ingress domain remains as is.
Publisher System	Loading configurations remains as is. Alternative frameworks to orchestrate Hudi publisher application were considered, but not used.
Transform System	A new system where people can run custom ETL jobs to create new datasets via AWS Glue[1], which also has support for incrementally loading new data and updating existing datasets using Hudi framework [21]. Specifics of this is out of scope of this work.
Analysis System	Analysis system now includes an integration with data catalog, for which AWS Glue Data Catalog was chosen. Data catalog is required to integrate with other systems. For analysis various tools can be used, such as Zeppelin, Presto, AWS Glue, which discover datasets from data catalog.
"Bring Your Own Data" System	We provide a way for teams who already have their data and are not interested in migrating to our solution to expose their data to the rest of the company. They can register tables which point to their data source in our data catalog and run ETL jobs via provided tools such as AWS Glue. Specifics of this is out of scope of this work.

Publisher orchestrator and data catalog choice will be explained in their respective chapters.

[1] https://aws.amazon.com/glue/

3.7 Orchestrating Hudi application

To orchestrate the Hudi application, three frameworks are supported:

- Spark Streaming[1]
- Apache Flink[2]
- Native Spark writer

Both Flink and Spark Streaming are open-source stream processing frameworks that can process near-real-time data from Apache Kafka.

Flink uses a streaming-first approach and processes data in a streaming fashion as it arrives, while Spark Streaming uses a batching approach and stores intermediate results on HDFS. This difference affects both the latency and the throughput of the system [15]. However, Flink can also stream data in batches. Because Hudi was initially developed to be run on Spark, it uses batch jobs by design and does not benefit much from the streaming capabilities offered by Flink [30]. Moreover, Spark Streaming is more widely used within the company and thus will be preferred.

In the Spark Streaming approach, the job polls batches of data from Kafka and submits them as Spark jobs. The combination of Kafka's replayable input source and Hudi's transactional batch writes ensures data consistency. Both systems are also configured to withstand hardware and software failures, ensuring high availability and reliability [5]. Migrating from the SQS solution to the Spark Streaming solution would be straightforward, as both solutions utilize Apache Spark as the MapReduce engine.

[1] https://spark.apache.org/docs/latest/streaming-programming-guide.html
[2] https://flink.apache.org/

Comparison between SQS-based solution and Spark streaming is outlined in Table 10.

Table 10: Comparison between SQS-based solution and Spark Streaming

Comparison	**SQS-based solution**	**Spark Streaming**
Extra latency	Up to 10 second extra latency (I/O time of consuming Kafka micro-batches and storing combined batch in S3)	No extra latency
Backup system	Built-in (Raw layout in S3 acts as a long-term backup system)	Another system needs to be used (Kafka cannot cost-effectively store data long term)
Dependencies	MHS, SQS, SNS, Kafka	Kafka
Scheduling	Dynamic scheduling (Next job waits for previous one before scheduling)	Fixed scheduling (Job is scheduled in a fixed interval)
Operational complexity	Low (Batch jobs do not need to finish in certain amount of time)	High (Every batch job needs to finish within set time, so more fine tuning is required)
Autoscaling	Possible (Batch job configuration can be calculated off input batch)	Not possible (Once streaming app is running, there is no configuring batch jobs at runtime)
Backfilling	Easy (Only file names need to be republished to SQS)	Hard (Full record needs to be published to Kafka)
Consistency	Ensured	Ensured
Availability	High	High

After comparing SQS-based solution to Spark Streaming solution, it was decided to keep using SQS-based solution, as Spark Streaming did not provide any required benefits over the current solution.

3.8 AWS Glue Data Catalog

The AWS Glue Data Catalog acts as a Hive metastore and is a metadata repository that stores and organizes information about data sources, their structure, and their properties. It allows data warehouses and data lakes to access and query data stored in different formats without the need to manually manage these data sources or the data itself [21]. This addresses problem #5 as most leading query engines support tables in the Hudi format and Hive metastores [19].

Since the company has an initiative to use cloud services for databases, AWS Glue was chosen as it was found to be easy to integrate with the existing infrastructure and no other alternatives were considered.

Data discovery is improved through the use of indexes, which enable predicate pushdown on the query side. This involves creating an index on a specific data field that can be used to filter out records that do not match the given criteria. Indexes can be used to optimize query execution times by reducing the amount of data that needs to be scanned [10].

For example, the query on Figure 4 filters by indexed fields.

```
%spark.sql

select count(*) from messaging_data_platform.finalized where year=2023 and month=2 and day=26;
```

Figure 4. Example of predicate pushdown

In this query:

- year, month, and day fields are indexed fields.
- messaging_data_platform is the database name in Hive metastore.
- finalized is the table name in database.

The same query can be executed from any query engine that is integrated with our Hive metastore.

4 Results

This section provides proof of the successful implementation of PoC, with detailed evidence demonstrating how the system fulfils each functional requirement.

4.1 Can handle petabyte scale

Figure 5 provides a diagram of the system's functional requirement to manage petabyte-scale datasets. The histogram displays the growth of datasets over time, highlighting the system's expanding capacity to handle large amounts of data. Purple lines represent MDR layouts, while blue lines indicate BE layouts.

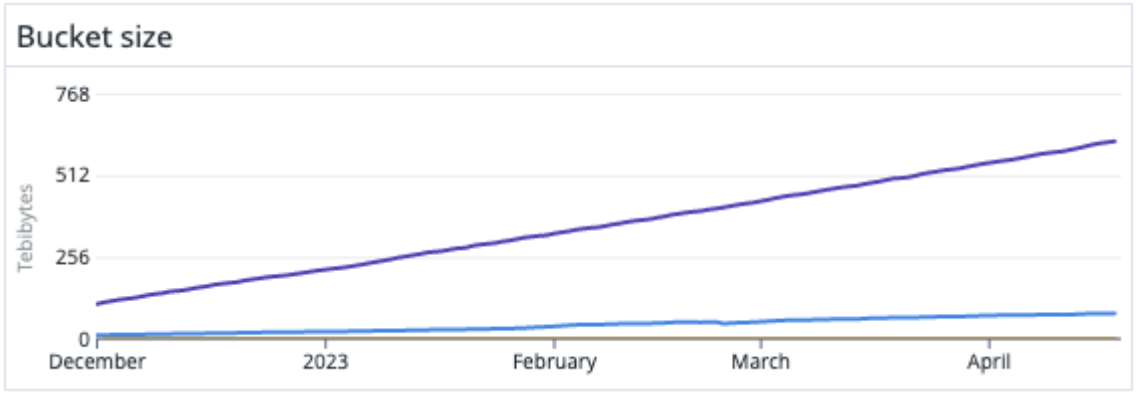


Figure 5. S3 bucket size

The lines' gradual upward movement demonstrates the system's steady progress towards achieving petabyte levels, successfully addressing the functional requirement of handling petabyte-scale datasets.

4.2 Late arriving updates

Figure 6 provides a screenshot of the system's functional requirement to handle late updates, which previously went missing. The diagram presents SQL queries joining input data (raw table) to output data (finalized table) using a left anti join. The left anti join operation effectively signifies the inclusion of all elements found in the raw dataset that are not present in the finalized table. The raw table contains all data partitioned by the current timestamp, generated by the MHS as described in the first iteration. Demonstrating that all data from the raw layout makes it to the finalized table also confirms that all late updates are successfully included. This figure highlights the system's ability to manage late updates and ensure data integrity, fulfilling the functional requirement.

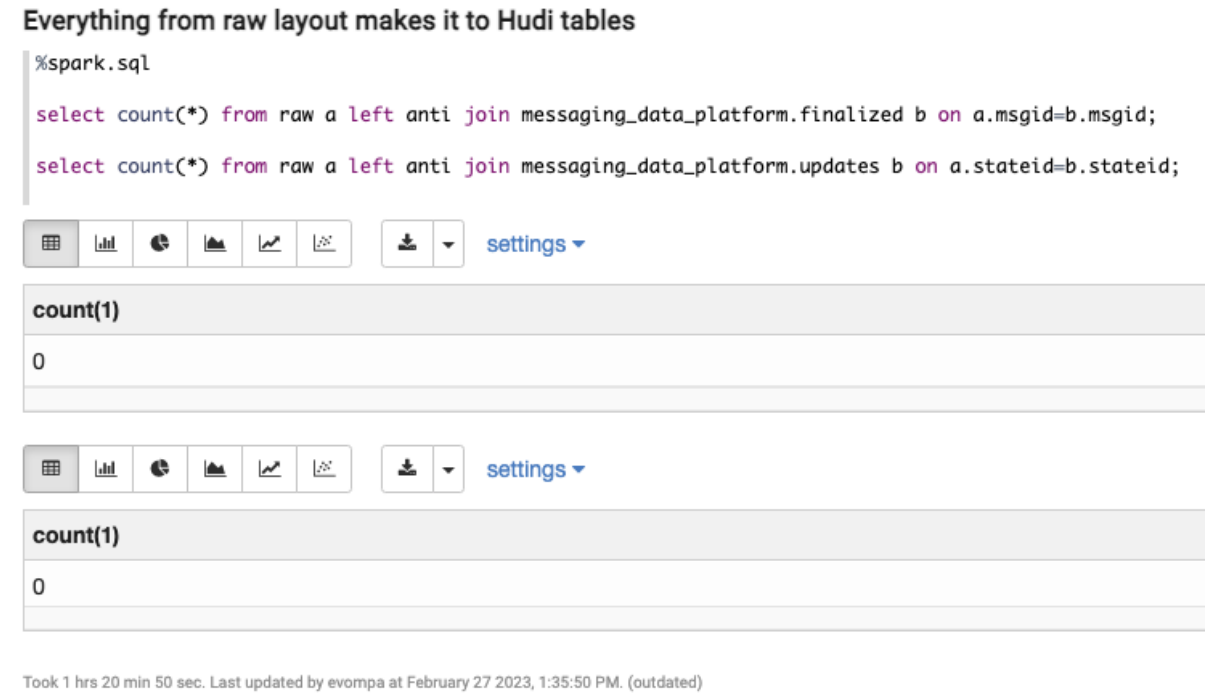


Figure 6. Data consistency SQL query

For the finalized table, a Bloom Index is employed, which uses bloom filter to identify the files requiring updates. Application logs indicate a false-positive rate of approximately 50%, suggesting room for improvement through bloom filter configuration tuning. Increasing the bloom filter size can reduce the false-positive rate while also increasing the filter size on disk. To minimize lookup latency, bloom filters could be stored in a metadata table.

4.3 CDC log deduplication

Figure 7 provides a screenshot of the system's functional requirement to avoid duplicate records, ensuring data accuracy and integrity. The diagram displays an SQL query that compares the total number of records with the total number of distinct records identified by the **msgid** field. The equality of these results signifies that the table is free of duplicates.

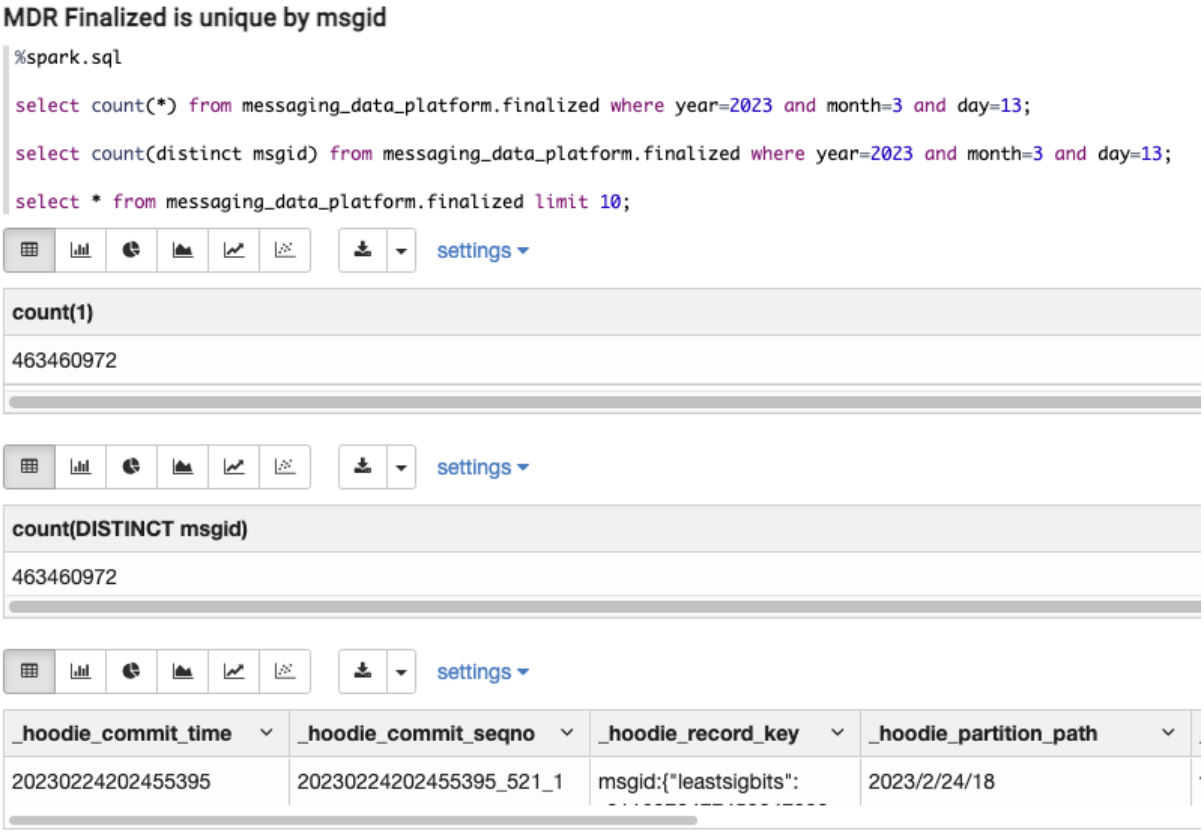


Figure 7. Finalized layout deduplication

The system employs UPSERT when adding new data to the table, which facilitates updating existing records or inserting new ones as necessary. This approach effectively eliminates duplicate entries and contributes to the system's ability to maintain data accuracy.

4.4 All use-case related peculiarities are customizable

Figure 8 presents a table of the system's functional requirement to be adaptable and configurable for different use-cases, ensuring flexibility and versatility. The diagram showcases some of the most impactful configurations for a table, with each table being able to accommodate different values compared to each-other, which was not the case before, as all BEs shared the same configurations.

spark.mdp.hoodieTableName	finalized
spark.mdp.hoodieTableType	COPY_ON_WRITE
spark.mdp.indexType	BLOOM
spark.mdp.partitionFields	year,month,day,hour
spark.mdp.preCombineField	version
spark.mdp.sourceAvroSchemaName	MDR
spark.mdp.sourceAvroSchemaNamespace	MessagingDataInfra
spark.mdp.sourceFormat	ORC

Figure 8. Runtime configuration

In total, there are over 100 configurations that can be modified to suit specific needs. [19]

4.5 Solution is robust

Figure 9 provides a diagram of the system's functional requirement to be robust, demonstrating its ability to handle transient exceptions and recover from them automatically. The diagram features a bar histogram with rollbacks, where app continues to function normally after rolling back changes made during a failed write. Both blue and purple bars are rollbacks which happened to MDR finalized table. The lack of further rollbacks indicates that table was recovered to stable state and continues to function normally.

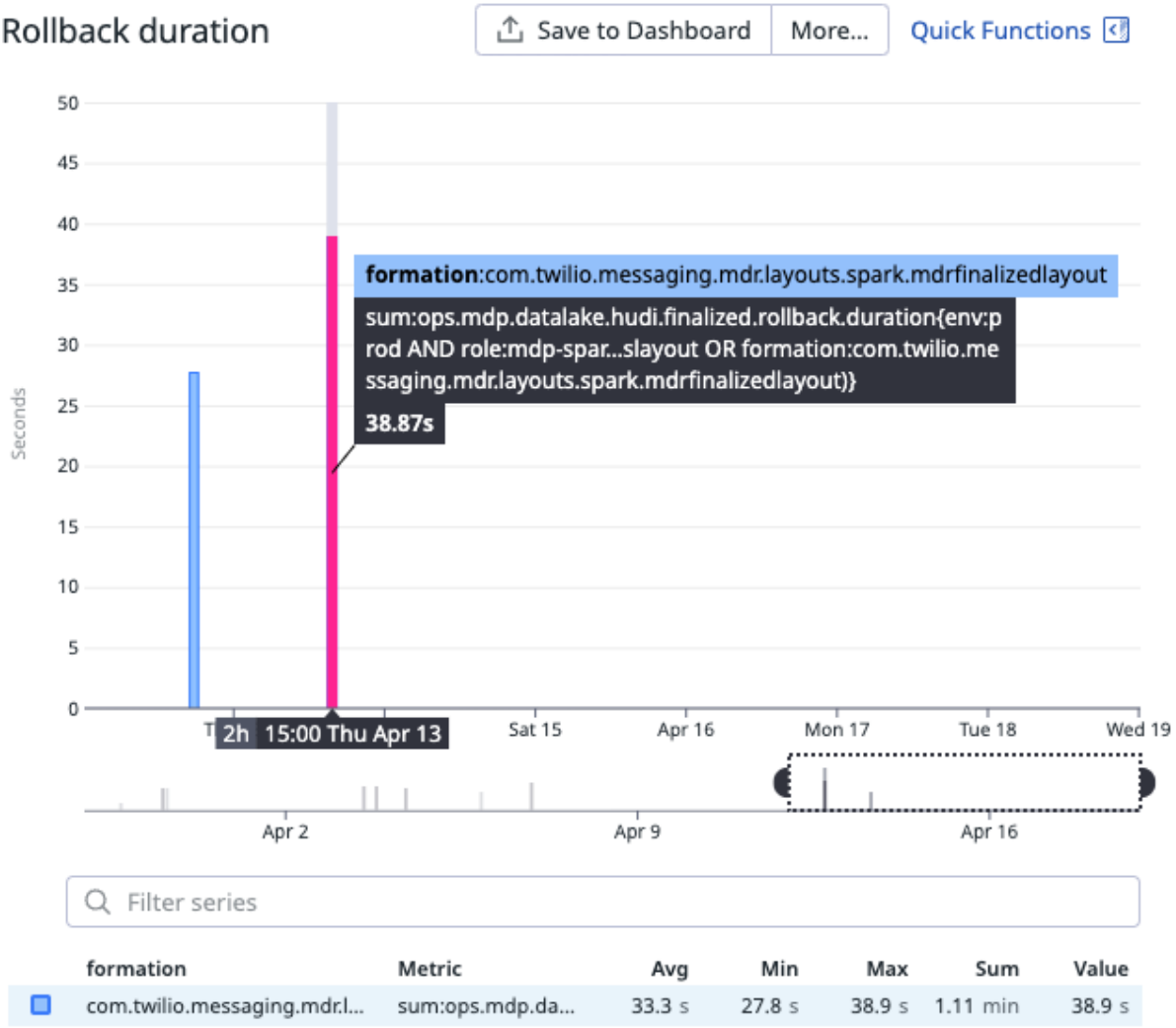


Figure 9. Rolling back changes

Additionally, non-committed changes will not be visible on the query side at any point in time.

4.6 Solution avoids the small file issue

Figure 10 presents a sorted table of the system's functional requirement to circumvent small file issue, ensuring that queries run as efficiently as possible. [17]

Name	Last modified	Size
.hoodie_partition_metadata	February 24, 2023, 00:38:15 (UTC+02:00)	96.0 B
26780973-09bd-4dcb-b724-c33255baaddc-0_2853-18-97947_20230309235033527.orc	March 10, 2023, 02:39:35 (UTC+02:00)	71.9 MB
5c0028c7-4889-475a-bb56-30564eb780c0-0_1821-18-70725_20230308104308330.orc	March 8, 2023, 13:18:30 (UTC+02:00)	116.8 MB
daf38d15-90a3-40df-b0c6-f7c62f707064-0_2473-18-98577_20230308170418945.orc	March 8, 2023, 19:52:29 (UTC+02:00)	117.1 MB
f843a150-851c-4fd4-acf9-3af38e94bcf8-0_2211-18-98533_20230308194820588.orc	March 8, 2023, 22:37:53 (UTC+02:00)	117.1 MB
169d781f-76b9-4b43-bfd8-55fb698c6fa2-0_2466-18-98570_20230308170418945.orc	March 8, 2023, 19:52:30 (UTC+02:00)	117.2 MB
2b5bd96e-11df-4355-9733-570afa50d4ea-0_2147-18-92722_20230308064212985.orc	March 8, 2023, 09:30:37 (UTC+02:00)	117.4 MB
549829a8-6f6e-4cec-a70d-97fd660a6516-0_1627-18-66639_20230308095010926.orc	March 8, 2023, 12:20:14 (UTC+02:00)	117.4 MB
010106c2-9681-411b-b34b-440aefb037c2-0_1734-18-86504_20230309114209443.orc	March 9, 2023, 14:28:01 (UTC+02:00)	122.2 MB
86e8c3a1-6f72-47f1-9e5d-d536a945f0f6-0_2843-18-97086_20230310011247672.orc	March 10, 2023, 04:14:53 (UTC+02:00)	126.2 MB
777f00b2-be96-43ef-a479-576b93370128-0_2589-18-93901_20230310052532669.orc	March 10, 2023, 08:16:19 (UTC+02:00)	126.2 MB

Figure 10. Files in S3

The diagram features a screenshot of the first few data files from a random partition, ordered in increasing order by size. As can be observed from the data presented in Figure 10, all files fall roughly within the 100MB range, with no data files smaller than 1MB.

4.7 Snapshot isolation between writers and readers

Figure 11 provides a representation of the system's functional requirement to maintain isolation between writers and readers, achieved by employing snapshot isolation. The diagram includes the first few rows listing the timestamp when the query finished, and the number of rows queried from selected partitions. Additionally, the stack trace in Figure 11 indicates that the old versions were successfully cleaned after 22 hours of running queries on same metadata. Testing environment was modified to re-use the same table metadata between runs instead of refreshing it.

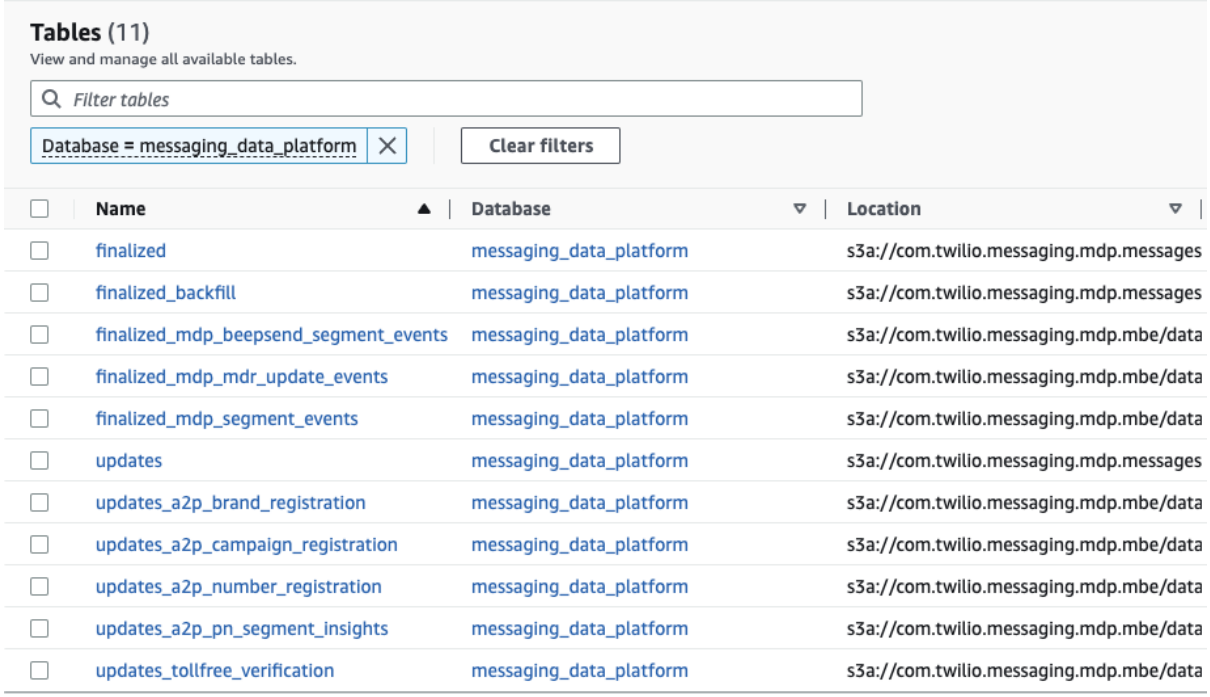
```
List([2022-10-28 12:40:08.370529,924888252])
List([2022-10-28 12:41:49.195745,924888252])
...
List([2022-10-29 11:09:10.241163,924888252])
List([2022-10-29 11:09:58.103512,924888252])

org.apache.spark.SparkException: Job aborted due to stage failure: Task 773 in stage 4417.0 failed 4 times,
most recent failure: Lost task 773.3 in stage 4417.0 (TID 3844510) (ip-10-210-18-206.ec2.internal executor 45):
java.io.FileNotFoundException: No such file or directory:
s3a://com.twilio.messaging.mdp.messages/data/mdr/finalized/2022/10/25/16/c8e4c981-abfa-4094-8add-74c8bb440bb8-0_946-22-
17316_20221028101908033.orc

It is possible the underlying files have been updated. You can explicitly invalidate
the cache in Spark by running 'REFRESH TABLE tableName' command in SQL or by
recreating the Dataset/DataFrame involved.
```

Figure 11. Snapshot isolation logs

Hudi's versioning mechanism enables snapshot isolation by allowing specific versions of the table to be queried as long as they are retained. During the test, which lasted approximately a day, 20 versions were kept before they were cleaned where writer app execution time was around an hour, resulting in the longest supported query time exceeding 20 hours. Each query reads the latest version of the table when it is executed, ensuring that every query can run for at least 20 hours with the given configuration. Moreover, the query returned the same result for all these runs, reinforcing the isolation as the data for the given version remained unchanged.

4.8 Data discovery is simple

Figure 12 displays a screenshot showcasing the current tables constructed using Hudi. The integration of the AWS Glue Data Catalog allows for seamless data discovery across various query engines, enhancing the system's overall usability and accessibility.



Name	Database	Location
finalized	messaging_data_platform	s3a://com.twilio.messaging.mdp.messages
finalized_backfill	messaging_data_platform	s3a://com.twilio.messaging.mdp.messages
finalized_mdp_beepsend_segment_events	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
finalized_mdp_mdr_update_events	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
finalized_mdp_segment_events	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
updates	messaging_data_platform	s3a://com.twilio.messaging.mdp.messages
updates_a2p_brand_registration	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
updates_a2p_campaign_registration	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
updates_a2p_number_registration	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
updates_a2p_pn_segment_insights	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data
updates_tollfree_verification	messaging_data_platform	s3a://com.twilio.messaging.mdp.mbe/data

Figure 12. AWS Glue Data Catalog

Additionally, Figures 14, 15, and 16 present examples of queries from different query engines performed on these tables, demonstrating the versatility and interoperability of the system. By leveraging the AWS Glue Data Catalog and Hudi, the system successfully fulfills the functional requirement of being easily discoverable, thereby facilitating efficient and effective data analysis and management for various use cases.

4.9 Data should be readable with low data and query latency

Figure 13 effectively demonstrates the system's functional requirement for low data latency. In the past, data latency issues were significant, with some data being delayed by several days or even missing indefinitely. However, the current system has successfully addressed these issues, ensuring improved data latency that is now tied to the app execution time, as shown in Figure 13, where MDR finalized execution time is highlighted, which is computationally the most expensive job.

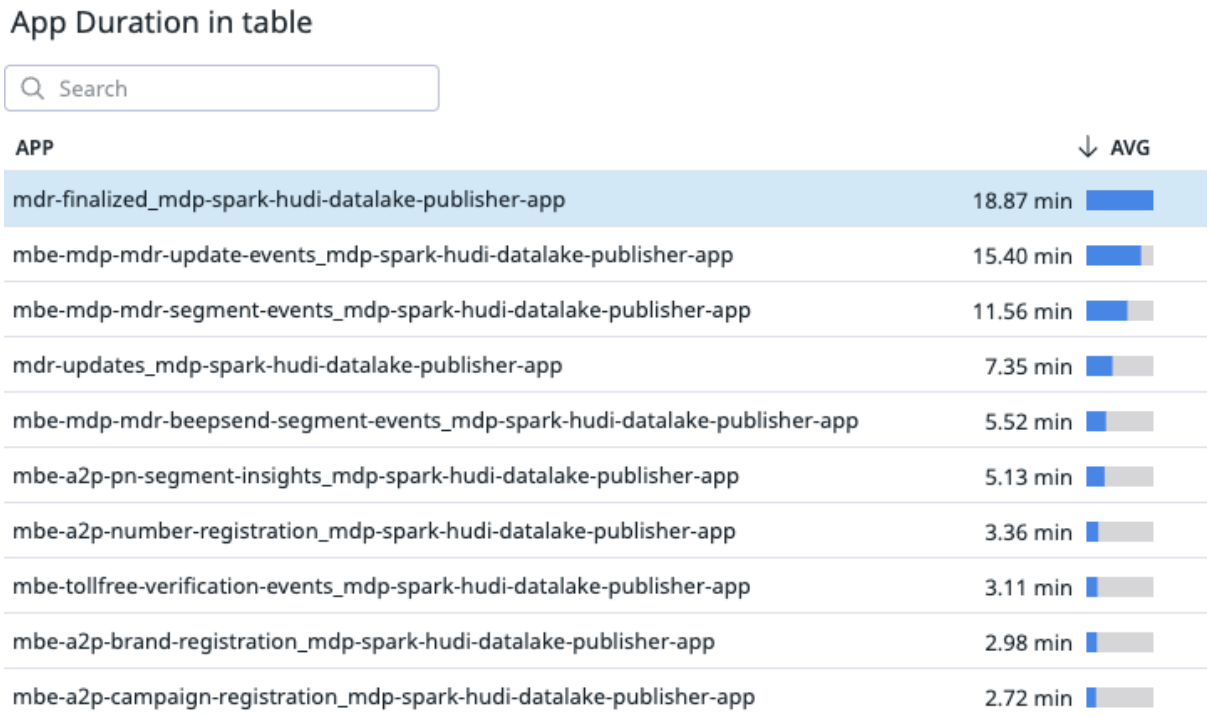


Figure 13. App execution duration

Although data latency can be further reduced by converting the table type to MoR, it is currently not a requirement and thus falls beyond the scope of this work. Nevertheless, the system has significantly improved its data latency, reducing it from multiple days to the duration of the app execution time.

Furthermore, the subsequent Figures 14 and 16 in the following section will showcase the system's low query latency, further solidifying its success in meeting the functional requirement for enhanced efficiency and performance in data processing and analysis.

4.10 Support for various analytical procedures

There are various analytical procedures, such as exploring the data, trying to understand various trends and patterns, ad-hoc analytics, complex analytics, reports, dashboards, and more [20]. In the end, however, they all boil down to SQL queries through various query engines. This section covers example queries from two prominent query engines: Presto and Spark SQL.

Amazon offers a serverless Presto solution called Athena, which enables data querying via SQL. Figure 14 features an example Presto query, which scanned one day's worth of data in just 13 seconds.

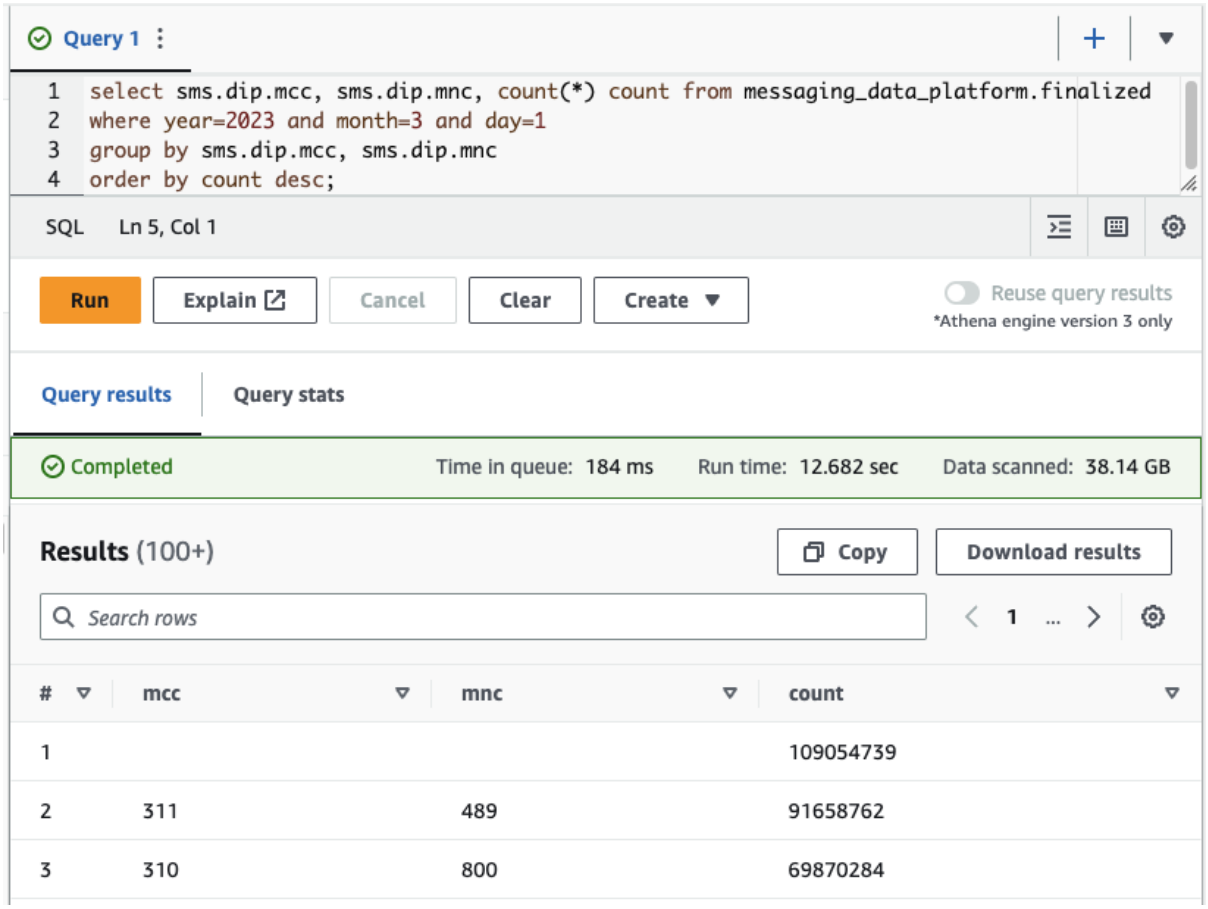


Figure 14. Daily aggregation via Presto

The query performance demonstrated in these examples appears to satisfy the requirements of the relevant stakeholders. It is crucial to note that query execution times should not be compared directly, as the duration depends on the resources allocated for executing the query, which is done automatically by Presto.

SQL syntax for operating on lists and maps, and other more complex analytical procedures differ based on the query engine used, but in general, all data types are supported and can be queried. The following figure shows example Presto queries for nested fields, filtering, aggregation, ordering, and operating on lists and maps as shown on Figure 15.

```
select sms.apiparameters.failoverattemptindex from messaging_data_platform.finalized
where month=11 and year=2022 limit 10;

select sms.dip.mcc, sms.dip.mnc, count(*) from messaging_data_platform.finalized
where year=2023 and month=1 and day=1 and hour=0
group by sms.dip.mcc, sms.dip.mnc
order by sms.dip.mcc, sms.dip.mnc;

select sms.attempts[1] from messaging_data_platform.finalized
where cardinality(sms.attempts) > 0 limit 10;

select message.customertags[''] from messaging_data_platform.finalized
where message.customertags is not null limit 10;
```

Figure 15: Schema peculiarities are supported

Another popular query engine is Spark SQL, which also allows for data querying. Figure 16 includes an example Spark SQL query that scanned one day's worth of data in 30 seconds.

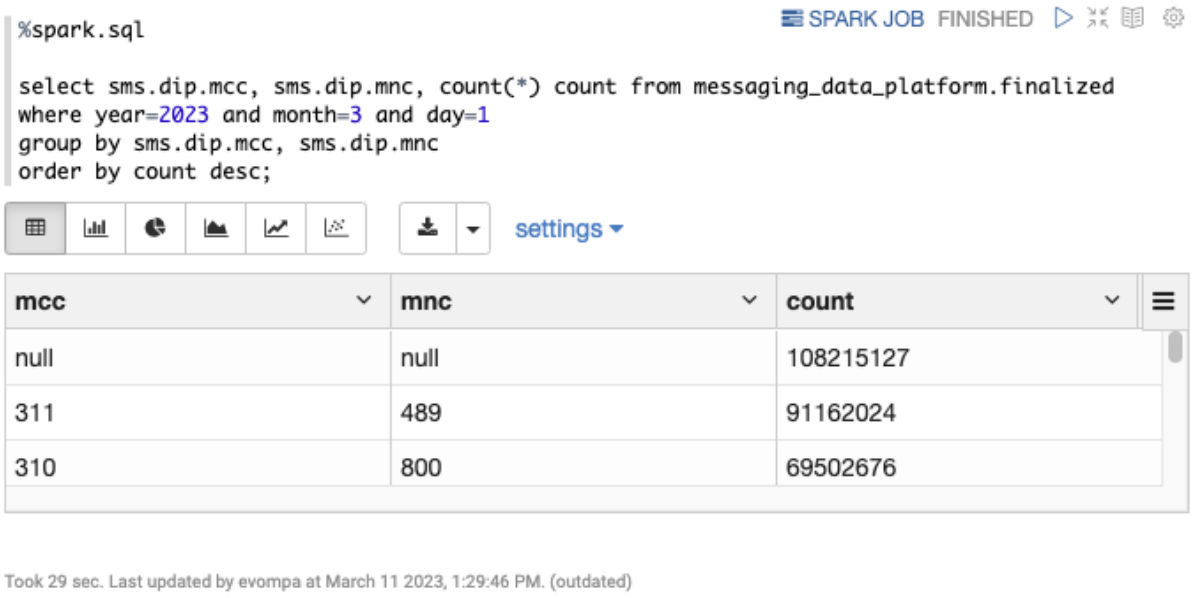


Figure 16. Daily aggregation via Spark SQL

In conclusion, data can be used for various analytical procedures and queries run fast enough.

4.11 Future works

The current solution is satisfactory and fulfils the functional requirements. However, more functional requirements may appear in the future, such as:

- Less than 10-minute data latency, which require the following but not limited to:
  - Converting table to MoR
  - Bloom filter tuning
  - Enabling metadata table
- Support for custom merging logic on table side

For these potential future requirements, there are solutions with the new design.

5 Summary

In summary, Twilio's adoption of the Apache Hudi framework provides a powerful tool for managing and accessing its Data Lakehouse in an efficient manner. With Apache Hudi's features, Twilio can perform incremental updates on its Data Lakehouse, making it easier to maintain a reliable source of messaging data.

The Twilio's MDP team has implemented a modern design that addresses the shortcomings of the previous design, meets the functional requirements needed for an efficient system and better-prepares for more requirements to come down the line. This work has resulted in a fully functioning data lakehouse capable of handling petabyte scale, adaptability to different use-cases, and supports various analytical procedures. This design serves as a reusable asset for other organizations and companies seeking an efficient and scalable data management system.

The Twilio MDP team leveraged the technologies and frameworks mentioned in Table 11 to achieve the desired situation.

Table 11. List of technologies and frameworks used

System	**Technology**
SQL query engines	Amazon Athena (Amazon managed PrestoDB) and Spark SQL via Apache Zeppelin
Analytics engine	Amazon Elastic MapReduce (Amazon managed Apache Spark cluster)
Hive metastore	AWS Glue data catalog (Amazon managed Apache Hive metastore)
Output	Amazon S3
Input	Amazon S3 with file notifications to Amazon SNS which gets fanned out to Amazon SQS or Apache Kafka through Apache Spark Structured Streaming framework
Incremental processing	Apache Hudi

Apache Hudi is the epicentre of all systems, with every other system leveraging or being utilized by it.

References

1. A. Chebotko, A. Kashlev and S. Lu, "A Big Data Modeling Methodology for Apache Cassandra," 2015 IEEE International Congress on Big Data, 2015, pp. 238-245, doi: 10.1109/BigDataCongress.2015.41.
2. H. R. Vyawahare, P. P. Karde and V. M. Thakare, "A Hybrid Database Approach Using Graph and Relational Database," 2018 International Conference on Research in Intelligent and Computing in Engineering (RICE), 2018, pp. 1-4, doi: 10.1109/RICE.2018.8509057.
3. E. Zagan and M. Danubianu, "Cloud DATA LAKE: The new trend of data storage," 2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 2021, pp. 1-4, doi: 10.1109/HORA52670.2021.9461293.
4. A. Jamal, R. Fleiner and E. Kail, "Performance Comparison between S3, HDFS and RDS storage technologies for real-time big-data applications," 2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics (SACI), 2021, pp. 000491-000496, doi: 10.1109/SACI51354.2021.9465594.
5. G. van Dongen and D. V. D. Poel, "A Performance Analysis of Fault Recovery in Stream Processing Frameworks," in IEEE Access, vol. 9, pp. 93745-93763, 2021, doi: 10.1109/ACCESS.2021.3093208.
6. K. Rattanaopas, "A performance comparison of Apache Tez and MapReduce with data compression on Hadoop cluster," 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2017, pp. 1-5, doi: 10.1109/JCSSE.2017.8025950.
7. P. Le Noac'h, A. Costan and L. Bougé, "A performance evaluation of Apache Kafka in support of big data streaming applications," 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 4803-4806, doi: 10.1109/BigData.2017.8258548.
8. D. Wu and A. Gokhale, "A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration," 20th Annual International Conference on High Performance Computing, 2013, pp. 89-98, doi: 10.1109/HiPC.2013.6799133.
9. B. R. Hiraman, C. Viresh M. and K. Abhijeet C., "A Study of Apache Kafka in Big Data Stream Processing," 2018 International Conference on Information , Communication, Engineering and Technology (ICICET), 2018, pp. 1-3, doi: 10.1109/ICICET.2018.8533771.
10. A. Bogatu, A. A. A. Fernandes, N. W. Paton and N. Konstantinou, "Dataset Discovery in Data Lakes," 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 2020, pp. 709-720, doi: 10.1109/ICDE48307.2020.00067.
11. N. Nguyen, M. M. H. Khan, Y. Albayram and K. Wang, "Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 802-807, doi: 10.1109/CLOUD.2017.119.
12. J. Singh, G. Singh and B. S. Bhati, "The Implication of Data Lake in Enterprises: A Deeper Analytics," 2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS), 2022, pp. 530-534, doi: 10.1109/ICACCS54159.2022.9784986.
13. J. Bang and M. -J. Choi, "Docker environment based Apache Storm and Spark Benchmark Test," 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS), Daegu, Korea (South), 2020, pp. 322-325, doi: 10.23919/APNOMS50412.2020.9237049.
14. Y. Tyryshkina, "Understanding join strategies in distributed systems," 2021 International Seminar on Electron Devices Design and Production (SED), 2021, pp. 1-4, doi: 10.1109/SED51197.2021.9444489.
15. Katsifodimos, Asterios & Schelter, Sebastian. (2016). Apache Flink: Stream Analytics at Scale. 193-193. doi: 10.1109/IC2EW.2016.56.

16. K. Siddique, Z. Akhtar, E. J. Yoon, Y. -S. Jeong, D. Dasgupta and Y. Kim, "Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications," in IEEE Access, vol. 4, pp. 8879-8887, 2016, doi: 10.1109/ACCESS.2016.2631549.
17. C. -S. Stan, A. -E. Pandelica, V. -A. Zamfir, R. -G. Stan and C. Negru, "Apache Spark and Apache Ignite Performance Analysis," 2019 22nd International Conference on Control Systems and Computer Science (CSCS), 2019, pp. 726-733, doi: 10.1109/CSCS.2019.00129.
18. D. Oreščanin and T. Hlupić, "Data Lakehouse - a Novel Step in Analytics Architecture," 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), 2021, pp. 1242-1246, doi: 10.23919/MIPRO52101.2021.9597091.
19. Apache Hudi Overview In: Hudi [Internet]. [cited 20 Nov 2022]. Available: https://hudi.apache.org/docs/overview
20. A. Cuzzocrea, "Big Data Lakes: Models, Frameworks, and Techniques," 2021 IEEE International Conference on Big Data and Smart Computing (BigComp), 2021, pp. 1-4, doi: 10.1109/BigComp51126.2021.00010.
21. Data Catalog and crawners in AWS Glue In: AWS Glue [Internet]. [cited 5 Mar 2023]. Available: https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html
22. X. Chen, L. Hu, L. Liu, J. Chang and D. L. Bone, "Breaking Down Hadoop Distributed File Systems Data Analytics Tools: Apache Hive vs. Apache Pig vs. Pivotal HWAQ," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 794-797, doi: 10.1109/CLOUD.2017.117.
23. Apache Hudi vs Delta Lake vs Apache Iceberg - Lakehouse Feature Comparison In: OneHouse [Internet]. [cited 20 Nov 2022]. Available: https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison
24. D. Chen et al., "Real-Time or Near Real-Time Persisting Daily Healthcare Data Into HDFS and ElasticSearch Index Inside a Big Data Platform," in IEEE Transactions on Industrial Informatics, vol. 13, no. 2, pp. 595-606, April 2017, doi: 10.1109/TII.2016.2645606.
25. Incremental Processing on the Data Lake [Internet]. [cited 11 Mar 2023]. Available: https://hudi.apache.org/blog/2020/08/18/hudi-incremental-processing-on-data-lakes/
26. Athena data source connectors [Internet]. [cited 11 Mar 2023]. Available: https://docs.aws.amazon.com/athena/latest/ug/connectors-prebuilt.html
27. Hadoop vs. Redshift [Internet]. [cited 11 Mar 2023]. Available: https://www.integrate.io/blog/hadoop-vs-redshift/
28. N. Yang, I. Ferreira, A. Serebrenik and B. Adams, "Why do projects join the Apache Software Foundation?," 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), Pittsburgh, PA, USA, 2022, pp. 161-171, doi: 10.1145/3510458.3513006.
29. AWS SNS vs SQS: Key differences which to use [Internet]. [cited 17 Mar 2023]. Available: https://ably.com/topic/aws-sns-vs-sqs
30. Apache Hudi meets Apache Flink [Internet]. [cited 17 Mar 2023]. Available: https://hudi.apache.org/blog/2020/10/15/apache-hudi-meets-apache-flink/
31. Amazon S3 [Internet]. [cited 18 Mar 2023]. Available: https://aws.amazon.com/s3/
32. Amazon Redshift vs Hadoop [Internet]. [cited 30 Mar 2023]. Available: https://hevodata.com/blog/amazon-redshift-versus-hadoop/
33. Hadoop ecosystem [Internet]. [cited 30 Mar 2023]. Available: https://www.edureka.co/blog/hadoop-ecosystem
34. Introduction to Presto (PrestpDB). [Internet]. [cited 30 Mar 2023]. Available: https://aws.amazon.com/big-data/what-is-presto/
35. Write A Spark Application. [Internet]. [cited 30 Mar 2023]. Available: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-application.html
36. What is Amazon DynamoDB. [Internet]. [cited 30 Mar 2023]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Appendix 1 – Non-exclusive licence for reproduction and publication of graduation thesis[1]

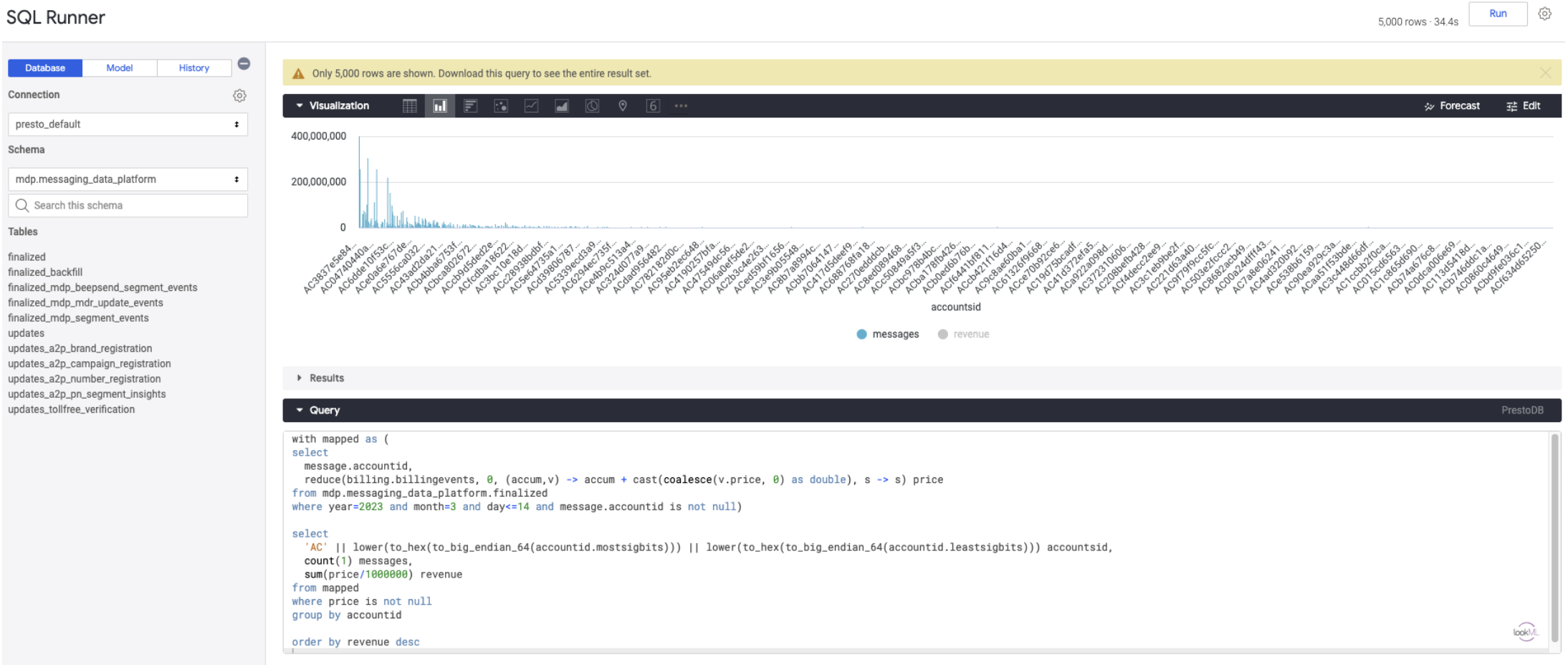
I, Enrico Vompa

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Data Lakehouse Architecture for Big Data with Apache Hudi", supervised by Tauno Treier.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

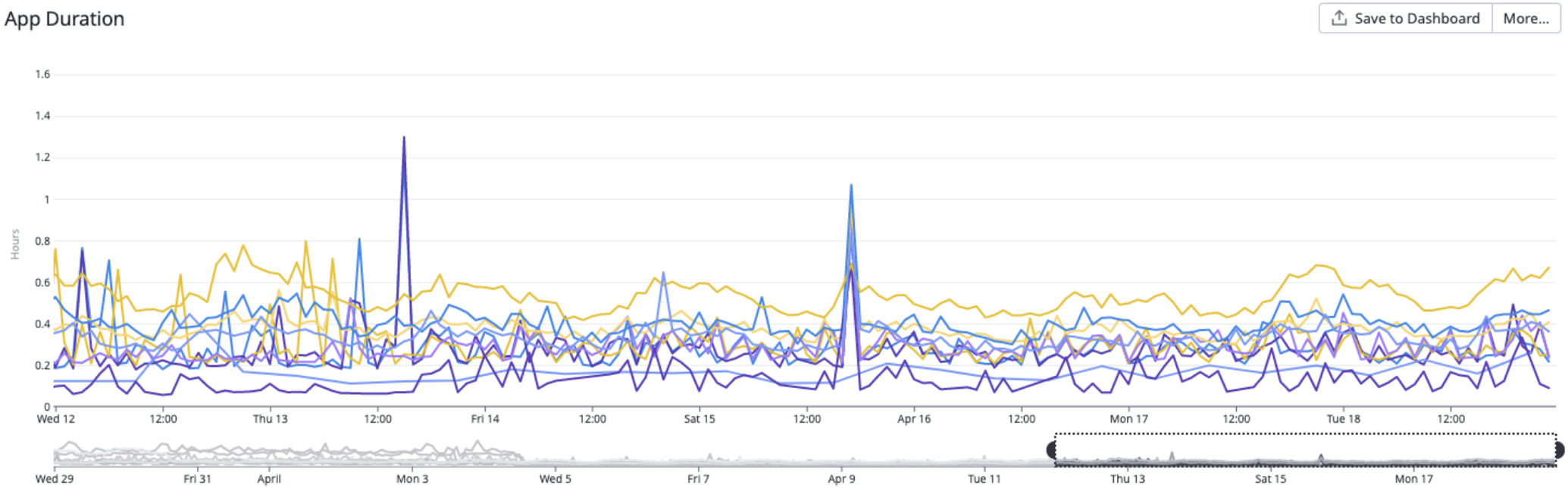
08.05.2023

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Two-week aggregation query from Looker running on Presto in 34 seconds

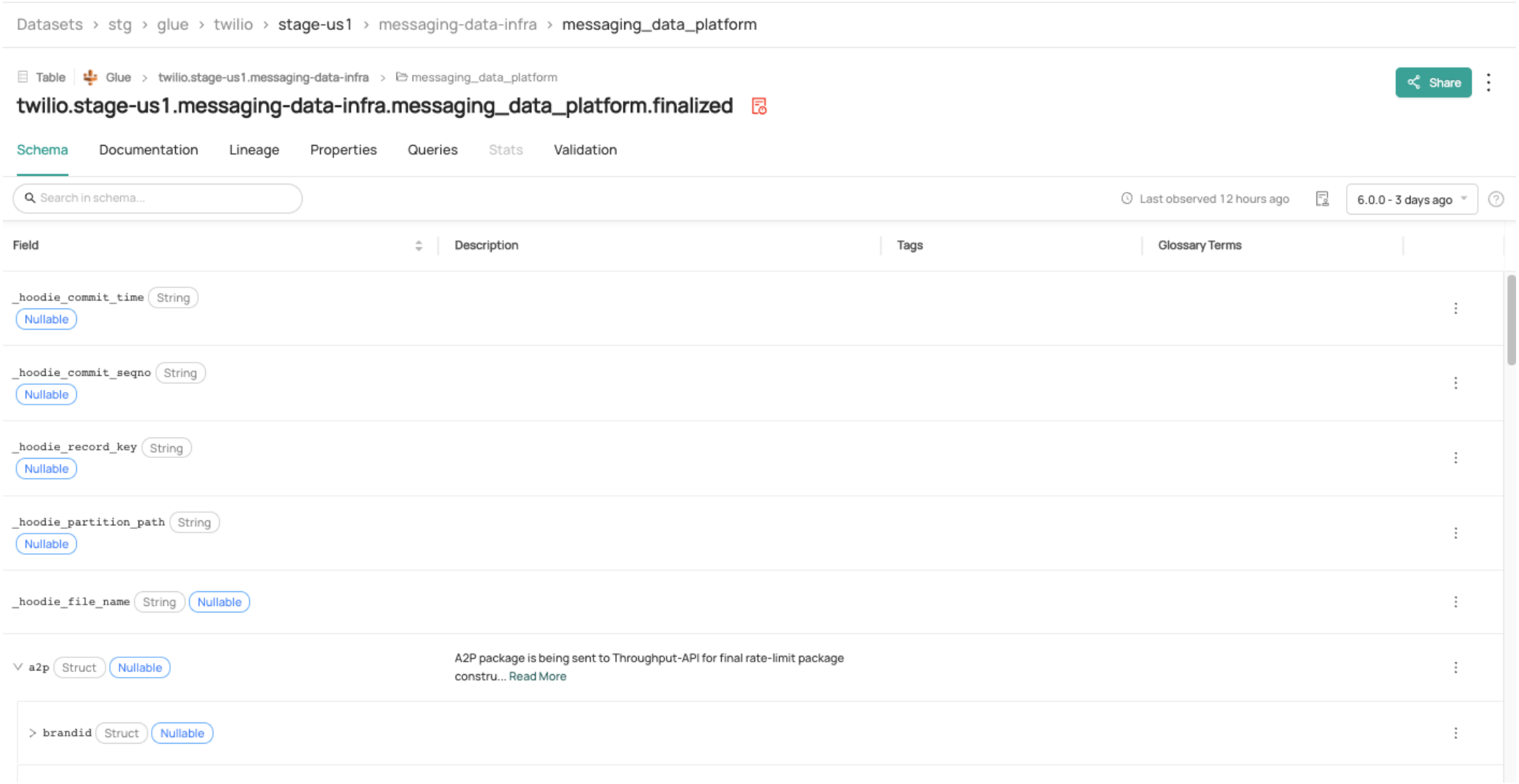


Appendix 3 – Hudi writer app durations

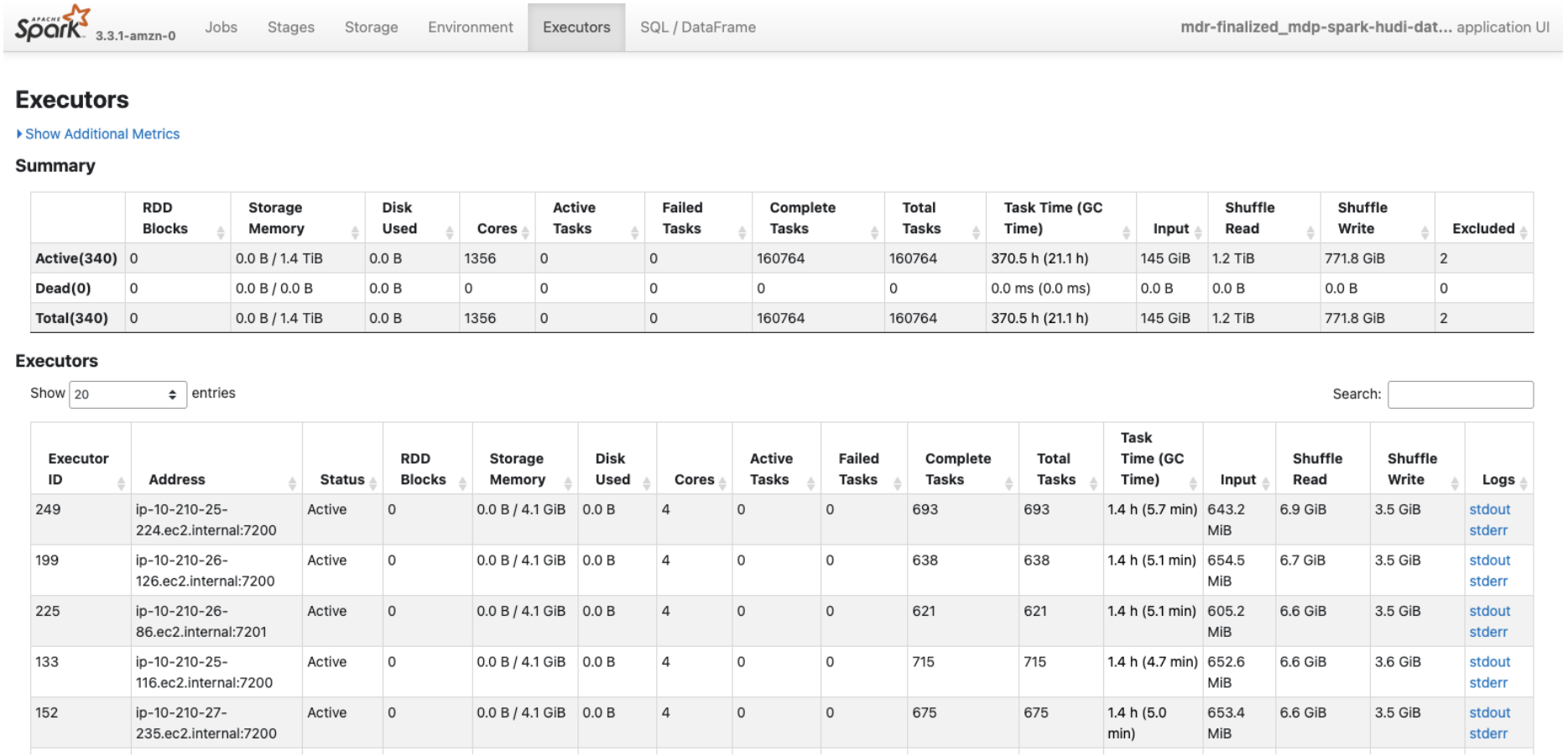


app in max:ops.messaging_system_report_service_v2.apps.app_duration.max{env:prod,role:mdp-spark-hudi-datalake-publisher-app}	Avg	Min	Max	Sum	Value
mbe-a2p-brand-registration_mdp-spark-hudi-datalake-publisher-app	17.2 min	10.8 min	70.7 min	1.99 days	20.4 min
mbe-a2p-campaign-registration_mdp-spark-hudi-datalake-publisher-app	16.6 min	10.7 min	77.9 min	1.93 days	18.7 min
mbe-a2p-number-registration_mdp-spark-hudi-datalake-publisher-app	18.1 min	11.8 min	55.8 min	2.12 days	15.9 min
mbe-a2p-pn-segment-insights_mdp-spark-hudi-datalake-publisher-app	10.2 min	6.6 min	26.7 min	0.20 days	—
mbe-mdp-mdr-beepsend-segment-events_mdp-spark-hudi-datalake-publisher-app	17.3 min	12.5 min	53.3 min	2.02 days	13.1 min
mbe-mdp-mdr-segment-events_mdp-spark-hudi-datalake-publisher-app	22.7 min	18.0 min	58.7 min	2.61 days	25.1 min
mbe-mdp-mdr-update-events_mdp-spark-hudi-datalake-publisher-app	24.3 min	17.9 min	64.0 min	2.83 days	28.9 min
mbe-tollfree-verification-events_mdp-spark-hudi-datalake-publisher-app	7.8 min	3.3 min	41.1 min	0.79 days	4.8 min
mdr-finalized_mdp-spark-hudi-datalake-publisher-app	32.0 min	25.0 min	46.6 min	3.73 days	39.6 min
mdr-updates_mdp-spark-hudi-datalake-publisher-app	20.0 min	16.0 min	50.4 min	2.33 days	26.0 min

Appendix 4 – DataHub view of Glue Metastore



Appendix 5 – MDR Finalized app Executors page in Spark history service



Spark 3.3.1-amzn-0 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | mdr-finalized_mdp-spark-hudi-dat... application UI

Executors

▸ Show Additional Metrics

Summary

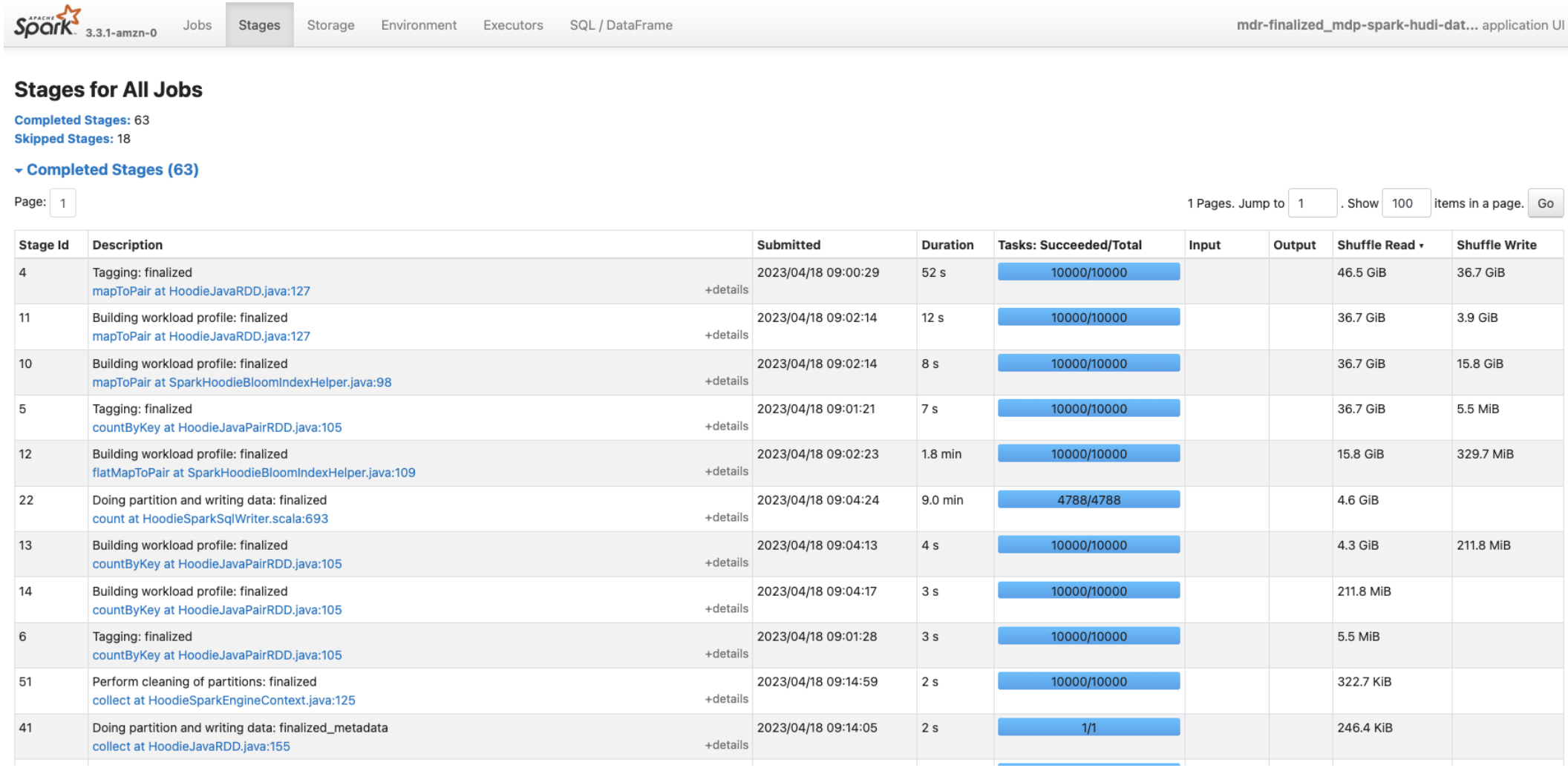
	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(340)	0	0.0 B / 1.4 TiB	0.0 B	1356	0	0	160764	160764	370.5 h (21.1 h)	145 GiB	1.2 TiB	771.8 GiB	2
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(340)	0	0.0 B / 1.4 TiB	0.0 B	1356	0	0	160764	160764	370.5 h (21.1 h)	145 GiB	1.2 TiB	771.8 GiB	2

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
249	ip-10-210-25-224.ec2.internal:7200	Active	0	0.0 B / 4.1 GiB	0.0 B	4	0	0	693	693	1.4 h (5.7 min)	643.2 MiB	6.9 GiB	3.5 GiB	stdout stderr
199	ip-10-210-26-126.ec2.internal:7200	Active	0	0.0 B / 4.1 GiB	0.0 B	4	0	0	638	638	1.4 h (5.1 min)	654.5 MiB	6.7 GiB	3.5 GiB	stdout stderr
225	ip-10-210-26-86.ec2.internal:7201	Active	0	0.0 B / 4.1 GiB	0.0 B	4	0	0	621	621	1.4 h (5.1 min)	605.2 MiB	6.6 GiB	3.5 GiB	stdout stderr
133	ip-10-210-25-116.ec2.internal:7200	Active	0	0.0 B / 4.1 GiB	0.0 B	4	0	0	715	715	1.4 h (4.7 min)	652.6 MiB	6.6 GiB	3.6 GiB	stdout stderr
152	ip-10-210-27-235.ec2.internal:7200	Active	0	0.0 B / 4.1 GiB	0.0 B	4	0	0	675	675	1.4 h (5.0 min)	653.4 MiB	6.6 GiB	3.5 GiB	stdout stderr

Appendix 6 – MDR Finalized app Stages page in Spark history service



Spark 3.3.1-amzn-0 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | mdr-finalized_mdp-spark-hudi-dat... application UI

Stages for All Jobs

Completed Stages: 63
Skipped Stages: 18

Completed Stages (63)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	Tagging: finalized mapToPair at HoodieJavaRDD.java:127 +details	2023/04/18 09:00:29	52 s	10000/10000			46.5 GiB	36.7 GiB
11	Building workload profile: finalized mapToPair at HoodieJavaRDD.java:127 +details	2023/04/18 09:02:14	12 s	10000/10000			36.7 GiB	3.9 GiB
10	Building workload profile: finalized mapToPair at SparkHoodieBloomIndexHelper.java:98 +details	2023/04/18 09:02:14	8 s	10000/10000			36.7 GiB	15.8 GiB
5	Tagging: finalized countByKey at HoodieJavaPairRDD.java:105 +details	2023/04/18 09:01:21	7 s	10000/10000			36.7 GiB	5.5 MiB
12	Building workload profile: finalized flatMapToPair at SparkHoodieBloomIndexHelper.java:109 +details	2023/04/18 09:02:23	1.8 min	10000/10000			15.8 GiB	329.7 MiB
22	Doing partition and writing data: finalized count at HoodieSparkSqlWriter.scala:693 +details	2023/04/18 09:04:24	9.0 min	4788/4788			4.6 GiB	
13	Building workload profile: finalized countByKey at HoodieJavaPairRDD.java:105 +details	2023/04/18 09:04:13	4 s	10000/10000			4.3 GiB	211.8 MiB
14	Building workload profile: finalized countByKey at HoodieJavaPairRDD.java:105 +details	2023/04/18 09:04:17	3 s	10000/10000			211.8 MiB	
6	Tagging: finalized countByKey at HoodieJavaPairRDD.java:105 +details	2023/04/18 09:01:28	3 s	10000/10000			5.5 MiB	
51	Perform cleaning of partitions: finalized collect at HoodieSparkEngineContext.java:125 +details	2023/04/18 09:14:59	2 s	10000/10000			322.7 KiB	
41	Doing partition and writing data: finalized_metadata collect at HoodieJavaRDD.java:155 +details	2023/04/18 09:14:05	2 s	1/1			246.4 KiB	

Appendix 7 – MDR Finalized app Jobs page in Spark history service