



DEGREE PROJECT IN INFORMATION AND COMMUNICATION
TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Hudi on Hops

Incremental Processing and Fast Data Ingestion
for Hops

NETSANET GEBRETSADKAN KIDANE



Hudi on Hops

Incremental Processing and Fast Data Ingestion for Hops

Netsanet Gebretsadkan Kidane

Master of Science Thesis

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

22 October 2019

Examiner: Professor Seif Haridi
Supervisor: Associate Professor Jim Dowling
Company Supervisor: Theofilos Kakantousis

TRITA Number: TRITA-EECS-EX-2019:809

Abstract

In the era of big data, data is flooding from numerous data sources and many companies have been utilizing different types of tools to load and process data from various sources in a data lake. The major challenges where different companies are facing these days are how to update data into an existing dataset without having to read the entire dataset and overwriting it to accommodate the changes which have a negative impact on the performance. Besides this, finding a way to capture and track changed data in a big data lake as the system gets complex with large amounts of data to maintain and query is another challenge. Web platforms such as Hopsworks are also facing these problems without having an efficient mechanism to modify an existing processed results and pull out only changed data which could be useful to meet the processing needs of an organization.

The challenge of accommodating row level changes in an efficient and effective manner is solved by integrating Hudi with Hops. This takes advantage of Hudi's upsert mechanism which uses Bloom indexing to significantly speed up the ability of looking up records across partitions. Hudi indexing maps a record key into the file id without scanning over every record in the dataset. In addition, each successful data ingestion is stored in Apache Hudi format stamped with commit timeline. This commit timeline is needed for the incremental processing mainly to pull updated rows since a specified instant of time and obtain change logs from a dataset. Hence, incremental pulls are realized through the monotonically increasing commit time line. Similarly, incremental updates are realized over a time column (key expression) that allows Hudi to update rows based on this time column.

HoodieDeltaStreamer utility and DataSource API are used for the integration of Hudi with Hops and Feature store. As a result, this provided a fabulous way of ingesting and extracting row level updates where its performance can further be enhanced by the configurations of the shuffle parallelism and other spark parameter configurations since Hudi is a spark based library.

Keywords: Hudi, Hadoop, Hops, Upsert, SQL, Spark, Kafka

Sammanfattning

I dag är stora data mängder vanligt förekommande bland företag. Typiskt så flödar datan från många datakällor och det är populärt bland företag att använda olika typer av verktyg för att läsa och bearbeta data i en data lake. En av de stora utmaningarna som företag står inför idag är att kunna uppdatera stora mängder data på ett effektivt sätt. Tidigare lösningar för att uppdatera stora mängder data är baserat på att skriva över datan, vilket är en ineffektiv metod. En ytterligare utmaning med stora data mängder är problemet av att bokföra ändringar till datan på ett effektivt sätt. Hopsworks är en webbplattform som lagrar och bearbetar stora mängder data och står således inför dessa utmaningar.

I denna avhandling så presenteras en lösning på problemet med att uppdatera stora datamängder på ett effektivt sätt. Lösningen är baserad på att integrera Hudi med Hops. Genom att integrera Hudi med Hops så kan Hops utnyttja Hudis mekanism för effektiv uppdatering av data. Mekanismen som används i Hudi för att uppdatera stora mängder data är baserad på Bloom-indexering samt logg-baserad lagring av data. Hudi indexerar datan för att snabba upp uppdateringsoperationer. Dessutom så stödjer Hudi att varje uppdatering bokförs till en loggfil, vilket i praktiken fungerar som en tidslinje över datan. Genom att använda tidslinjen så stödjer Hudi att läsa stora datamängder inkrementellt, samt att inspektera datan vid specifika tidpunkter.

I denna avhandling så beskrivs hur HoodieDeltaStreamer-verktyget samt Hudis DataSource API används för integrera Hudi med Hops Feature Store. Tack vare denna integration så möjliggörs en mer effektiv och användbar uppdatering av stora datamängder i Hops.

Nyckelord: Hudi, Hadoop, Hops, Upsert, SQL, Spark, Kafka

Acknowledgements

I would like to thank Theofilos Kakantousis, my supervisor Associate Professor Jim Dowling and my examiner Professor Seif Haridi for their continuous support and insights during my thesis period. My gratitude also goes to Fabio, Kim and the remaining Hops team members for their substantial assistance and precious help in solving technical problems to get the goals of the thesis achieved.

Contents

1	Introduction	1
1.1	Problem	3
1.2	Purpose	4
1.3	Goals	4
1.4	Research Methodology	5
1.5	Research Sustainability and Ethical Aspects	6
1.5.1	Research Sustainability	6
1.5.2	Ethical Aspects	6
1.6	Delimitation	6
1.7	Outline	6
2	Background	9
2.1	Apache Hudi	11
2.1.1	Storage Types: Copy on write and Merge on read	11
2.1.2	Copy on Write	12
2.1.3	Merge on Read	14
2.1.4	Indexing	15
2.1.5	Hudi Tools	16
2.1.6	SQL Queries	17
2.2	Feature Store	18
2.3	Related work	19
2.3.1	Delta Lake	19
2.3.2	Iceberg	20
3	Research Methodology	23
3.1	Procedures, Measurements and Experimental Setups	23
3.2	Implementations	24
3.2.1	DFS as a Source Integration of HUDI with Hops	24
3.2.2	Kafka as a Source Integration of HUDI with Hops	26
3.2.3	HUDI and HiveSyncTool	28
3.2.4	Hudi Integration with FeatureStore	30

4	Result and Analysis	33
4.1	Row Level Updates with Hudi Upsert	33
4.2	Incremental Pulling and Time Travel	36
4.3	Partition and Schema Management	39
4.4	Performance Comparison	39
4.4.1	Hudi Upsert versus Bulk Loading	39
4.4.2	Feature Store versus Hudi Integrated Feature Store	42
4.5	Qualitative Comparison of Delta Lake, Iceberg and Hudi	43
5	Conclusion and Future Work	45
5.1	Conclusions and Recommendations	45
5.2	Future Work	45
	Bibliography	47

List of Figures

2.1	Copy on Write [1]	13
2.2	Merge on Read [1]	14
2.3	Delta Lake [2]	20
3.1	Partition log and offsets of Kafka [3]	26
3.2	Hudi DataStreamer and Other Utilities	29
3.3	Implementation System Flow of Hudi and Hudi Integrated Feature Store	30
3.4	Mechanism Of Integrating Hudi with Feature Store	31
4.1	Sample Commit Data of Hudi Upsert	34
4.2	Row-level Updates Skew over Input Batch Size	35
4.3	Effects of Hudi Upsert Operation to Number of Parquet Files	37
4.4	Hudi DataSourceRead API for Incremental Pulling	38
4.5	Performance Comparison of Hudi Upsert with Bulk Loading	41
4.6	Qualitative Comparison of Hudi, Delta Lake and Iceberg	44

List of Listings

2.1	Addition of Aux Jars Path to HiveServer2	17
3.1	DFS Source HUDI Configuration Properties	24
3.2	Hudi Spark Job Argument Specification for DFS Source	25
3.3	Kafka Source HUDI Configuration Properties	26
3.4	Hive Sync DDL for Creating Table	29
3.5	HiveSyncTool sample to Sync Partition	30
4.1	Hudi Cleaning Policy Setting	37
4.2	HiveIncremental Puller For Incremental Pull	38
4.3	Feature Store SQL Definition	42
4.4	Hudi Integrated Feature Store SQL Definition	43

List of Acronyms and Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
DAG	Directed Acyclic Graph
DDL	Data Definition Language
DFS	Distributed File System
DML	Data Manipulation Language
HDFS	Hadoop Distributed File System
Hops	Hadoop Open Platform-as-a-Service
Hudi	Hadoop Upserts and Incrementals
JSON	JavaScript Object Notation
ORC	Optimized Row Columnar
RDD	Resilient Distributed Datasets
SQL	Structured Query Language

Chapter 1

Introduction

In the era of big data, the data size of big companies can range from few terabytes to many petabytes of data. As an example, companies can generate 6,000 data samples every 40 milliseconds. Thus, big data analytics is vital foundation for forecasting manufacturing, maintenance, automation of data processing, and other technological capabilities [4].

The effective use of big data can create productive growth and transform economies through creating new competitors that have the critical skills on big data [5]. The big data analytics focuses on data management systems and hybrid service platforms to turn big data into decision-making so that enterprise have additional context and insight to enable better decision making [6]. For example, extracting valuable information regarding the inefficiency of machines by analyzing patterns can lead to effective maintenance decisions that can avoid unexpected downtime. Having highly distributed data sources brings enormous challenges in data access, integration, and sharing due to the fact that those different sources are defined using different representation methods. Fortunately, companies like Facebook and Google, have been creating new tools for storing and analyzing their large datasets with their desire to analyze streams, web logs, and social interactions. Google created GFS and MapReduce models to cope with the challenges brought by the massive data generated by users, sensors, ubiquitous data sources, and data management and analysis at the large scale [7].

Apache Hadoop [8] is a well-established software platform that provides large-scale data management and analytics through its partitioning mechanism which distributes its workload across different machines. It is designed for batch processing and consists of the Hadoop kernel, Map/Reduce as its programming model for execution, and Hadoop distributed file system (HDFS) as its storage [9]. Map/Reduce have been facing some problematic issues as the stream data gets

high in volume, velocity and data type complexity. Therefore, real-time stream processing frameworks, such as Apache Storm [10], Apache Flink [11], and StreamCloud [12], are important specially for real-time stream data analytics.

Data ingestion tools are crucial in big data as they are responsible for integrating variety of big data sources into the big data platform after formatting the data to have a uniform format. Distributed File systems such as Apache Hadoop distributed filesystem (HDFS) and other storage systems are used as a long-term analytical storage for thousands of organizations. Analytical datasets can be built by using two different paradigms which differ according to the processing timeline requirement [4]:

- **Batch Processing:** In batch processing, computation happens in blocks of data that have been stored over a period of time through reading data from a source databases, transforming the data, writing results back to storage, and making it available for analytical queries. The dominant batch processing model is MapReduce with its core idea to first divide data into small chunks and process these chunks in parallel and in a distributed manner for driving the final result from the aggregations of all intermediate results. This style of processing incurs large delays in making data available to queries as well as lot of complexity in carefully partitioning datasets [13].
- **Streaming processing:** Real-time stream processing allows as to process data in real time as they arrive and quickly generate fresher and faster analytics results within minor period from the point of receiving the data. Well known open source stream processing systems include Spark, Storm, S4, and Kafka. Even though this approach yields low latency, the amount of data managed in such real-time is typically limited and the overall data architecture is more complex with duplicated data.

Different types of storage systems, distributed file systems and data warehouses support different kinds of frameworks for real time queries and historical data. Hudi [1] is a good example of these frameworks, which is an open source framework developed by Uber and supports a unified framework for real-time queries and historical data, in columnar storage. Hudi manages storage of large analytical datasets on Hadoop Distributed File system. It takes a hybrid approach of both batch and stream processing through adding the streaming primitives (upserts & incremental consumption) onto existing batch processing technologies to provide an increased efficiency, greatly simplifying the overall architecture in the process [14].

1.1 Problem

Many different systems can be used for processing data on top of HDFS and coexist nicely with the trending technologies. Huge volumes of data are being generated related with the evolution of social medial platforms and Internet of Things (IoT) which can demand real time data ingestion [15]. For example, autonomous car have many on-board, position (GPS) and fuel consumption sensors which can generate large volumes of data through imagining this at a large scale with millions of these cars in a cities [16].

One of the biggest challenges when working with data lakes is updating data in an existing datasets and pulling out only the changes since the latest ingestion [17]. Hopsworks is a web platform which allows users to store and process big data projects [18]. The platform is built with collaboration in mind, allowing multiple users to collaborate on a project by creating datasets, uploading data and writing data processing software using the supported frameworks. Datasets can be shared among the different projects and they can be made available to the whole Internet. Hopsworks integrates data processing frameworks such as Apache Spark [19], Apache Flink [11] and Tensorflow [20] and message brokering system such as Apache Kafka [3] and data management layer such as Feature Store [21] that allows teams to share, discover, and use a highly curated set of features for their machine learning problems. However, it is facing challenges with regard to incremental pulling that is tracking and getting the updated dataset only since a given instant of time. In addition to this, there is no way of supporting time travel in the dataset which enables us to access the data according to a time series and generate useful reports which can help in facilitating decision making.

Hudi [1] is an incremental processing framework with an open-source storage format (Apache Hudi Format) that brings ACID transactions to Apache Spark. Hudi allows incremental updates over key expression, and update data into an existing dataset using upsert primitive instead of reading the entire latest dataset and overwrite it as the classical file based data lake architecture performs its update operations. How can we integrate the capabilities supported by Hudi in to big-data ecosystems specifically Hops? The main motive behind this thesis project is to integrate and extend Apache Hudi with Hopsworks and Feature store so that the platform will have a completely fresh look that enables it to perform the data integration, analytics, cleaning and transformation of a time series data in near real time. In addition to this, to provide support for incremental pulling and have a materialized views for different purposes through tracking changes and increase the importance of big data ecosystems such as Hive, as data volumes and performance increase with the near real time processing capabilities of Hudi.

1.2 Purpose

The purpose of this thesis is to support update to an existing dataset without reading the entire dataset and overwriting it and enable a way of tracking changes to a dataset given a point in time, which can be summarized as integrating Hudi with Hops specifically feature store and Hive on Hops. Hive on Hops is a data warehouse management and analytics system that is built for Hadoop and uses SQL like scripting language called HiveQL that can convert queries to MapReduce and Spark jobs to facilitate reading, writing, and managing large datasets residing in distributed storage using SQL [22]. The integration will give support for taking advantage of Hudi's analytical, scan-optimized data storage abstraction which enables it to apply mutations to data in HDFS on the order of few minutes and chaining of incremental processing. This gives us support to perform complex joins with near real time data processing techniques that minimize latency's. Furthermore, to be able to handle a materialized view since it allows incremental updates over key expression and provide an insight into the coexistence of distributed systems with each other.

1.3 Goals

The main objectives are shaped towards bringing near real time analytics on petabytes of data via support for upserts and incremental pulling, which are the main features of Hudi to enable incremental processing on Hive and Feature Store. Hudi is designed to work with an underlying hadoop compatible filesystem and ecosystems like Presto, Spark by relaying on their storage since it does not have its own fleet of storage servers.

Currently, Feature store is using Hive managed table to store its datasets in a columnar format which is specifically ORC and this wealth of feature data is used to operate machine learning systems and to train their models. Feature Store allows teams to define, publish, manage, store, and discover features for use in machine learning projects, which in turn facilitates the discovery and feature reuse across these projects. Enabling Feature store with a way to track updates to its dataset could be useful for analytics, monitoring changes in data, time traveling in the feature store and training the model according to the needs of the machine learning algorithms.

1.4 Research Methodology

The project will use the empirical research approach [23], by first performing a set of observations to describe the state and properties of real-world incremental processing systems like Hudi and its integration with Hive on Hops. After analyzing the collected data to improve our understanding and knowledge on the co-existence of big data ecosystems, a qualitative Comparison of Delta Lake [2], Iceberg [24] and Hudi with regard to keeping track of data versioning and management of files based datasets on object stores and incremental pulls.

My task is to first investigate how Apache Hive, Feature Store and Hudi operate with further search on the coexistence of Hadoop ecosystems like apache spark and Hudi to grasp some concepts on how to integrate these systems with each other. More advanced tasks revolve around the concept of incremental processing, which is the main feature of Hudi and needs to be integrated with Hive on Hops and Feature store for effectively speeding up typical batch pipelines and bring a near real time analytics on petabytes of data. Supporting incremental processing needs inclusion of the following 2 primitives:

- Upserts: Applying changes to a dataset instead of rewriting the entire partition, which is inefficient and ends up writing much more than the amount of incoming changes.
- Incremental consumption or pulls: Mechanism to more efficiently obtain the records that have changed since the last time the partition was consumed instead of scanning the entire partition/table and recomputing everything, which can take a lot of time and resources [1].

Hudi is related with both batch and stream processing due to its hybrid architecture through the integration's of copy-on-write and merge-on-read storage's for enabling batch and stream processing respectively and storing the computed results in Hadoop. Hence, in case of Non-Spark processing systems like Hive, the processing can be done in the respective systems with the use of a Kafka topic or HDFS intermediate file for sending intermediate result to a Hoodie table. This leads to usage of Kafka as a messaging queue. After all the integration's, the system will be tested to check if its performing accordingly using the IPUMS USA dataset [25]. Furthermore, a test will be performed on the coexistence of the Hadoop ecosystem specifically Apache Hive with Hudi.

1.5 Research Sustainability and Ethical Aspects

We anticipate that the impact of our research might have the following implications towards research sustainability and ethical aspects.

1.5.1 Research Sustainability

We tried to look at 2 key aspects of the project towards contributing its share for the sustainability of a research. First, contributions towards minimizing data ingestion latency and enabling of time travel through the use of Hudi to do incremental pulling for retrieving the necessary data according to the commit instant meta data and secondly, validating the performance of Hudi streaming primitives, scalability and availability in comparison with the previous data manipulation languages to allow users know why integrating Hudi with their systems is best decision for sustaining economic growth.

1.5.2 Ethical Aspects

We performed extensive measurements and detailed analysis on the system mainly on Hudi streaming primitives and its integration with feature store with regard to performance, scalability and attempt to be very explicit in stating the setup and tools used in each measurement with a aim to facilitate reproducibility of our results.

1.6 Delimitation

This work depends on many other previous works. It mainly uses the Apache Hudi project and Hops. From the previous Hudi implementation it changes the implementation of consumption from kafka source due to the depreciated code of kafka cluster from kafka 0.8 and enhance it by providing usage of kafka consumer from kafka 0.10 which changes the way of consumption, creation of RDD, and way of retrieving offsets from kafka. This thesis work will enable a way to do near real time data analytics through the usage of Hudi's incremental pulling and a mechanism to create a fast feature store by using Hudi as an ingestion system to store a dataset in modeled hoodie table with synchronized external hive table.

1.7 Outline

As introduced above, this paper discusses on how to integrate Hudi into Hopworks and Feature Store. Chapter 2 goes through a deeper discussion of the Apache Hudi

and Feature Store with a brief description of Hudi's main concepts relevant to the incremental consumption and time traversal followed by related previous works attempted to provide fast data analytics, time traversal and schema management. Chapter 3 discusses the methodology used, the experimental setups and the proposed solution of this project. The solution with its result and analysis is analyzed thoroughly in chapter 4 that covers the detailed thesis work. Chapter 5 goes through the best discussion points that can be raised in the project and explanation. Finally, chapter 6 concludes the paper by providing conclusions and future works.

Chapter 2

Background

In computing, a file system that allows access to files from multiple hosts through a computer network can be referred as a distributed file system (DFS). This allows multiple users on multiple machines to share files and storage resources. These Distributed storage systems currently constitute a significant part of storage systems due to their important property which is scalability and provide users a system in a way that they have a unified view of stored data on different file systems and computers [26]. However, there are some issues such as reliability and consistency. Hence, it is important to know about the designs, architectures, and some trade-offs among the storage systems for choosing one of them according to the needs of users. Distributed file systems have difference on their performance, way of handling lose of nodes and concurrent writes, mutability of their content, and their storing policy.

Hopsworks [18] is a web platform which allows users to store and process big data projects. The platform is build around the concept of projects and datasets, multiple users can collaborate on a project by creating datasets, uploading data and writing data processing software using the supported frameworks. Datasets can be shared among the different projects and they can be made available to the whole Internet. Hopsworks integrates data processing frameworks such as Apache Spark [19], Apache Flink [11] and Tensorflow [20] and message brokering system such as Apache Kafka [3] and data management layer such as Feature Store [21] that allows teams to share, discover, and use set of features for their machine learning problems. Users can write, execute and collaborate on data processing pipelines using the interactive notebooks provided by Apache Zeppelin. The platform provides peripheral security being the only point of access to operate on the whole Hops ecosystem. Hopsworks provides REST APIs to allow external services to integrate with the platform.

On the platform files and activities are organized in projects and datasets.

Projects contains zero or more datasets, a collection of Jobs and Kafka topics and Apache Zeppelin notebooks. A project is mapped on the HopsFS as a sub-directory in the /Projects directory. In Hopsworks, a dataset represents the unit that can be shared across different projects and they can be made publicly available to all the projects and to the public internet. Datasets are stored as sub-directories of the Project directory under which they are created. Metadata can be attached to a dataset, which is indexed by Elasticsearch and searchable as free-text. Users can also use the free-text search to find directories and files in a dataset. When a new project is created, the platform automatically creates 3 datasets: Resources, Logs and Notebooks. The Resources dataset can be used by data scientists to upload jobs. Logs is used to write the aggregated Yarn logs and finally, in the Notebooks dataset are stored the Zeppelin interactive notebooks.

Hopsworks uses HopsFS as a hierarchical distributed file system to achieve higher scalability and throughput by storing metadata in a database called MySQL-Cluster. HopsFS is forked from Apache Hadoop Filesystem (HDFS) and it achieves better scalability in terms of throughput and space. Hopsworks also uses Hive as a data warehouse software solution built on top of the Hadoop ecosystem that facilitates reading, writing, and managing large datasets including querying big data residing in distributed storage using SQL. Hive's architecture involves storing data in the Hadoop Filesystem (HDFS) and the metadata in a separate database such as MySQL [27]. It has an eventual consistency model for its database that is if the schema is changed eventually the data in the HDFS should reflect the change. Data is stored in a distributed filesystem like HDFS or HopsFS and organized in databases, tables and partitions. A relational schema is attached to the data allowing users to write SQL-like queries to analyze the data. Apache Hive stores the metadata describing the structure of the data and schema attached to it in a relational database. Queries are then translated into MapReduce jobs and executed by Hadoop MapReduce framework.

Databases, tables and partitions are represented as directories on HDFS, using the following organization: Hive stores all the databases into a warehouse directory, each child subdirectory represents a database and inside each database there are 0 or more subdirectories each representing a table. There are two types of tables: managed and external. Managed tables are managed by Hive, data is stored into the warehouse and when the table is dropped from Hive the data is deleted. On the other hand, external tables describe the layout of external files not managed by Hive itself. External tables can be stored anywhere in the file system and can be manipulated by other processes than Hive. Hive supports different formats when writing files on the file system such as TEXTFILE, ORC, PARQUET and AVRO. There are three main components in the Hive architecture: Hive Server 2, the Hive Driver and Hive Metastore. Hive server

which allows remote users to execute queries and retrieve results. Hive driver is the Hive internal component that manages the whole lifecycle of a query which have sub component's such as driver, compiler and executor. Hive Metastore is the component responsible for managing the metadata and implements a Thrift interface which allows clients to invoke operations like create table, drop table, and alter table on the metadata.

Objective of this thesis is to support updating to an existing datasets and enable way of consuming only the changed datasets in Hops ecosystem and Feature Store in particular in the Hopworks platform.

2.1 Apache Hudi

Hudi [1] is a spark library and an open source framework developed by uber for supporting incremental processing. It stands for Hadoop Upserts and Incrementals and manages storage of large analytical datasets on HDFS and via use of two kinds of tables:

- Read Optimized Table: Provides excellent query performance via purely columnar storage which is Parquet.
- Near-Real time Table: Provides queries on real-time data, using a combination of columnar and row based storage which are Parquet and Avro respectively.

Hudi is able to strengthen rich data ecosystem where external sources can be ingested into Hadoop in near real-time through carefully managing how data is laid out in storage and how it's exposed to queries. The ingested data is then available for interactive SQL Engines like Presto and Spark, while at the same time capable of being consumed incrementally from processing/ETL frameworks like Hive and Spark to build derived Hudi datasets. The broad spark library contained in Hudi is used to build datasets and access these datasets via its integration with existing query engines.

2.1.1 Storage Types: Copy on write and Merge on read

Hudi is implemented as a Spark library, which makes it easy to integrate into existing data pipelines or ingestion libraries referred as Hudi clients. Hudi Clients prepare an RDD[HudiRecord] that contains the data to be upserted and Hudi upsert/insert is merely a Spark DAG, that can be broken into two big pieces:

- **Indexing:** Hudi's efficiency comes from indexing that maps record keys to the file ids they belong to. This index also helps the HoodieWriteClient separate upserted records into inserts and updates, so they can be treated differently. HoodieReadClient supports operations such as filters and an efficient batch read api, that can read out the records corresponding to the keys using the index much quickly, than a typical scan via a query. The index is atomically updated at each commit, and is also rolled back when commits are rolled back.
- **Storage :** The storage part of the DAG is responsible for taking an RDD[HoodieRecord], that has been tagged as an insert or update via index lookup, and writing it out efficiently onto storage.

Hudi organizes a datasets into a directory structure under the provided base path, very similar to Hive tables. Dataset is splitted into partitions, which are folders containing files for that partition. Each partition is uniquely identified by its partition path, that is relative to the base path. Inside each of these partitions, there are multiple files with distributed records. Each file is identified by an unique file id and the commit that produced the file. Multiple files can share the same file id but written at different commits, in case of updates. A record key is used to uniquely identify each record and map it to a file id forever. This mapping between record key and file id, never changes once the first version of a record has been written to a file. In short, the file id identifies a group of files, that contain all versions of a group of records.

Hudi storage types capture how data is indexed and laid out on the filesystem, and how the upsert and incremental pull primitives including timeline activities are implemented which states how data is written in general [1]. While the Read Optimized and Near-Real time tables or views specify a different notion , which is ways of exposing the underlying data to the queries which can be summarized as how data is read. Hudi supports 2 kinds of storage types which are Copy on Write and Merge on Read.

2.1.2 Copy on Write

Copy on Write is a heavily read optimized storage type, that creates new versions of files corresponding to the records that changed. Each commit on Copy On Write storage, produces new versions of files which implies that every commit is compacted so that only columnar data exists. As a result, this type of storage is desired by a system like hadoop since number of bytes written for incoming data is much higher than read which signifies the read heavy property of copy on write

storage type. The purpose of copy on write storage is to fundamentally improve the management of datasets through:

- First class support for atomically updating data at file-level, instead of rewriting whole tables/partitions.
- Ability to incrementally consume changes, as opposed to wasteful scans.
- Tight control of file sizes to keep query performance excellent as small files hurt query performance.

Figure 2.1 illustrates how copy on write storage type works conceptually with specific details on how data is written into copy-on-write storage and how to run queries on top of it.

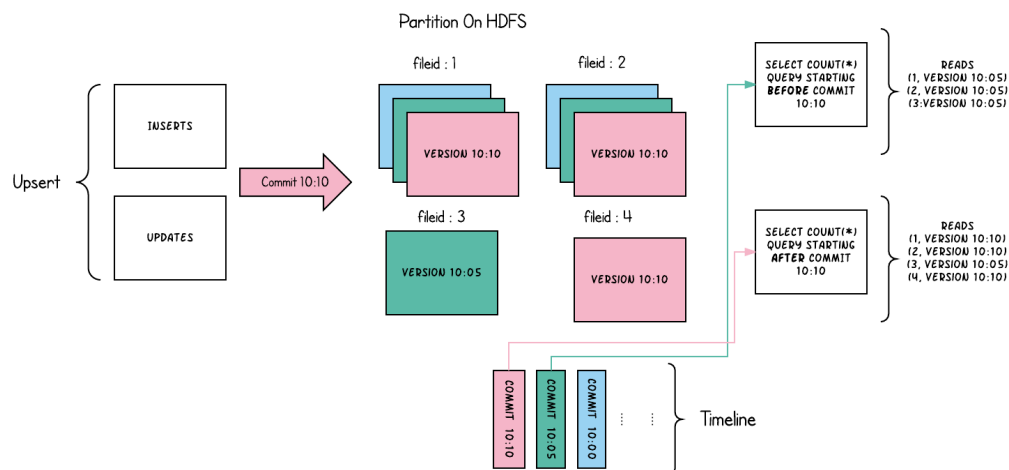


Figure 2.1: Copy on Write [1]

Snapshots is a copy of set of files and directories at a point in time. There could be a read only or read write snapshots usually done by the file system itself. Copy-on-write is a simple and fast way to create snapshots. The snapshot is done on the current data and a request to write to a file creates a new copy. As data gets written, updates to existing file ids, produce a new version for that file id stamped with the commit and inserts allocate a new file id and write its first version for that file id. The different colors in Figure 2.1 show file versions and their commits. Normal SQL queries such as counting the total number of records in a partition, first check the timeline for latest commit and filters all but latest versions of each file id. As shown above, an old query does not see the current inflight commit files colored in light pink. While a new query starting after the commit can obtain the

new data which strengthens the concept that queries only run on committed data and they are immune to any write failures.

2.1.3 Merge on Read

In a similar way to Copy on Write storage, Merge on Read storage also provides a read optimized view of the dataset via its Read Optimized table. But, additionally stores incoming upserts for each file id onto a row based append log and adds the append log with the latest version of each file id on-the-fly for enabling near real time data output during query time. Hence, this storage type provides near real-time queries by balancing the read and write operations. A significant change is made on to the compactor so that it carefully chooses the append logs that need to be compacted onto their columnar base data, in order to keep the query performance under control as larger append logs would incur longer merge times with merge data on query side. Merge on Read is currently under development with the intention to enable a near-real processing directly on top of Hadoop by shifting some of the write cost to the reads and merging incoming and on-disk data on-the-fly, against the idea of copying data out to specialized systems which is not feasible as the data volume increases [1].

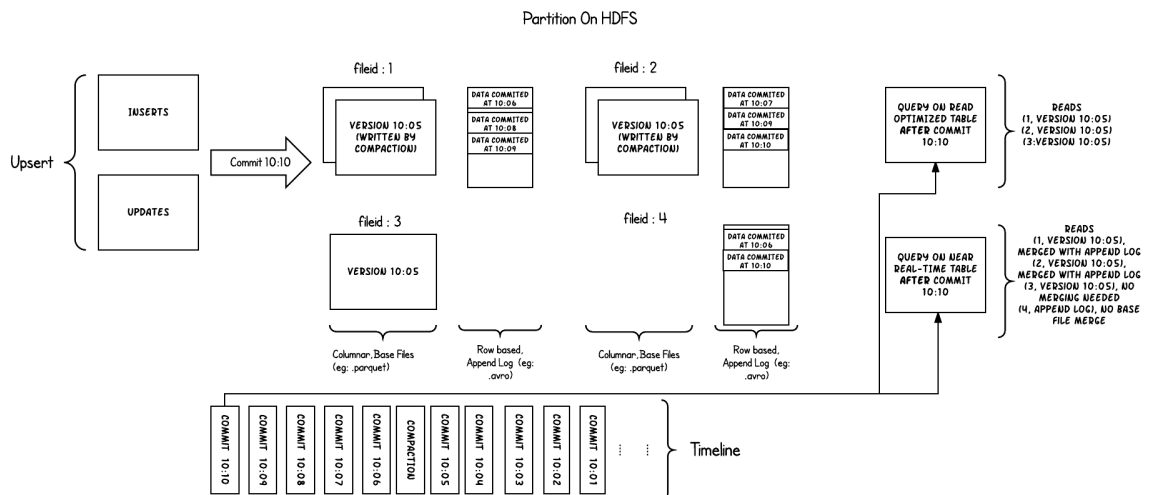


Figure 2.2: Merge on Read [1]

Figure 2.2 illustrates how the merge on read storage works, and shows near-real time table and read optimized table queries as an example. As shown in the figure,

commits can happen in every 1 minute and now merge on read have the following additional features:

- **Append log:** There is an append log with in each file id group. Append log holds incoming updates to records in the base columnar files. Both Merge on Read and Copy on Write storage types have the base columnar files which are the versioned parquet files with the commit. Thus, this append log is the one that's making the difference.
- A periodic compaction process reconciles these changes from the append log and produces a new version of base file.
- The underlying storage can be queried using the ReadOptimized (RO) Table or Near-Realtime (RT) table, depending on our choice of achieving query performance and freshness of data respectively.

2.1.4 Indexing

Hudi provides two choices for indexes: BloomIndex and HBaseIndex to map a record key into the file id to which it belongs to. This enables us to speed up upserts significantly, without scanning over every record in the dataset. This classification is based on their ability to lookup records across partition. A global index does not need partition information for finding the file-id for a record key but a non-global does.

- **HBase Index (global):** A global index does not need partition information for finding the file-id for a record key and its a straightforward way to store the mapping. The challenge with using HBase or any external key-value store is performing rollback of a commit and handling partial index updates. Since the HBase table is indexed by record key and not commit time, we would have to scan all the entries which will be prohibitively expensive.
- **Bloom Index (non-global):** This index needs partition information for finding the file id for a record key and is built by adding bloom filters with a very high false positive tolerance (e.g: $1/10^9$), to the parquet file footers. The advantage of this index over HBase is the obvious removal of a big external dependency, and also nicer handling of rollbacks and partial updates since the index is part of the data file itself. At runtime, checking the Bloom Index for a given set of record keys effectively amounts to checking all the bloom filters within a given partition, against the incoming records, using a Spark join. Much of the engineering effort towards the Bloom index has gone into scaling this join by caching the incoming

RDD[HoodieRecord] and dynamically tuning join parallelism, to avoid hitting Spark limitations like 2GB maximum for partition size. As a result, Bloom Index implementation has been able to handle single upserts up to 5TB, in a reliable manner [1].

2.1.5 Hudi Tools

Hudi consists of different tools for fast ingesting data from different data source to HDFS as a Hudi modeled table and further sync up with Hive metastore. The tools include:

- **DeltaStreamer:** Ingesting tool that supports DFS and kafka as data sources and json and avro as data formats.
- **Datasource API of the Hoodie-Spark:** The read write via the datasource happens using the hoodie-spark through choosing table names, output paths and one of the storage types.
- **HiveSyncTool:** For synchronizing the Hudi modeled table on HDFS with Hive metastore as external hive table. It register hive tables to access Hudi views (Read optimized and Real Time table view) and in addition it handles schema evolution.
- **HiveIncremental puller:** Incrementally consume data from hive via HQL.

As the Hudi Documentation shows, different parameters can be set for running a Hudi delta streamer according to the choice of clients source for the ingestion and their expectations of the output after a successful job run [28]. The most necessary settings options include:

- Storage type and Target base path : choice of Copy-on-write or Merge-on-read storage types and base path for storing the target Hudi dataset.
- Properties: path to properties file on file system, with configurations of Hudi client, schema provider, key generator, data source and hive configurations if hive sync is enabled.
- Source and Transformer class: source class is used for reading from source and provide the appropriate transformations according to the source which could be Json or Avro source from DFS or Kafka. While a user defined transformer class is used to transform raw source dataset to a target dataset conforming to target schema before writing.
- Schema provider class: to attach schemas to input and target table data.

- Operation type: UPSERT, INSERT and BULK_INSERT as possible values of operation type where INSERT is used when input is purely new data with aim to gain speed. While UPSERT is used for inserting data records that change frequently.

2.1.6 SQL Queries

Currently, Hudi is a single writer and multiple reader system which supports writing only parquet columnar formats. Commit timeline and indexes are maintained by Hudi for managing the dataset. The commit timelines are useful for understanding the current state of the dataset as well as the actions that were happening on a dataset before its current state. While indexes are used for maintaining a record key to file id mapping with the aim to efficiently locate a record. Hudi registers the dataset into the Hive metastore backed by HoodieInputFormat which makes the data accessible to Hive, Spark, and Presto automatically.

For Hive to recognize the Hudi tables and query them correctly, the hudi-hadoop library needs to be installed into the auxiliary jars path so that HiveServer2 can have access to it.

Listing 2.1: Addition of Aux Jars Path to HiveServer2

```
<property>
  <name>hive.aux.jars.path</name>
  <value>../hoodie-hadoop-mr-bundle-0.4.5.jar</value>
  <description>jar path </description>
</property>
```

Hudi works by defining three important keys which are the Key, Time and Partition key columns where the key column is used to distinguish each input row. Based on the Time key column, Hudi updates rows with newer values while the partition key is used to determine how to partition the rows in a Hudi modeled table [17]. Apache Hudi format is an open-source storage format that brings ACID transactions to Apache Spark and provide a way to capture and track the changed data through allowing incremental updates over key expression, which could be a primary key of the table.

Cleaning policy needs to be set to determine how to clean up older file versions which include old parquet and log files after a Hudi upsert operation is performed. There are 2 cleaning policies which are used for limiting or bounding the storage growth:

- `KEEP_LATEST_COMMITS`: Retains and does not delete any file that was touched in the last `X` commits which enables to pull the incremental changes worth up to `X` commits.
- `KEEP_LATEST_FILE_VERSIONS` : Used when there is no any interest in incremental pull and choose to just retain only the latest `X` files per file group that share same prefix instead.

2.2 Feature Store

Data Engineering is the hardest problem in machine Learning with a focus of getting the right data. There are many different feature engineering steps which are used as mechanisms of extracting features from raw data. The most common ones include converting categorical data into numeric data, normalizing data, one hot encoding, converting continuous features into discrete and extracting features using clustering, embeddings, and generative models.

Feature Store [21] is a data management platform for machine learning with an Api for reading, searching and adding features. Models are trained using sets of features and can be cached for reuse, reducing model training time and infrastructure costs. A feature store is used as a data transformation service besides its simple data storage service, as it supports feature engineering. Feature engineering transforms raw data into a format that is understandable by predictive models reducing the cost of developing new predictive models that understands the data provided. Feature is single versioned data column documented in the feature store. While Feature group is a versioned group of features stored as a Hive table and they can be used as a raw data in a specific Spark, Numpy, and Pandas job to output the computed features.

The storage layer for the feature data is built on top of Hive/HopsFS with additional abstractions for modeling feature data. Because Hive in combination with columnar storage formats such as ORC or Parquet, supports data modeling of the features in a relational manner with the use of tables that can be queried using SQL and work with datasets in scale larger than terabyte-scale.

Feature store has two basic writing and reading interfaces (API's) where one interface is used for writing features to the feature store and other interface for discovering and reading features from the feature store to be used for training.

2.3 Related work

Hudi processes data on top of HDFS and co-exists nicely with other technologies. Contrasting Hudi with other related systems could be useful in understanding how Hudi fits into the current big data ecosystem, and analyze the different trade-offs that arise from the system design choices.

2.3.1 Delta Lake

Delta Lake [2] is an open source storage layer that brings reliability to data lakes by providing ACID transactions, scalable metadata handling, and unified streaming and batch data processing. Delta Lake runs on top of existing data lake and is fully compatible with Apache Spark APIs. It provides built in data versioning for easy rollbacks and reproducing reports. Delta lake key functionalities [29]:

- ACID transactions: Serial order for writes recorded in a transaction log which tracks writes at file level and reads from latest snapshots.
- Schema Management: Schema validation and ability to update the schema automatically.
- Scalable metadata handling: By storing metadata in in transaction log instead of metastore.
- Data versioning and time travel : create newer versions of files and preserves older versions, important to reproduce experiments and reports and revert a table to its older versions. Delta Lake provides snapshots of data enabling developers to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments. Previous snapshots of Delta Lake table can be queried by using a feature called Time Travel which allows access to the overwritten data using the versionAsOf option.
- Open Format: All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet.
- Unified batch and streaming sink: Near real time analytics.
- Record update and deletion along with data expectations will be implemented in the future.

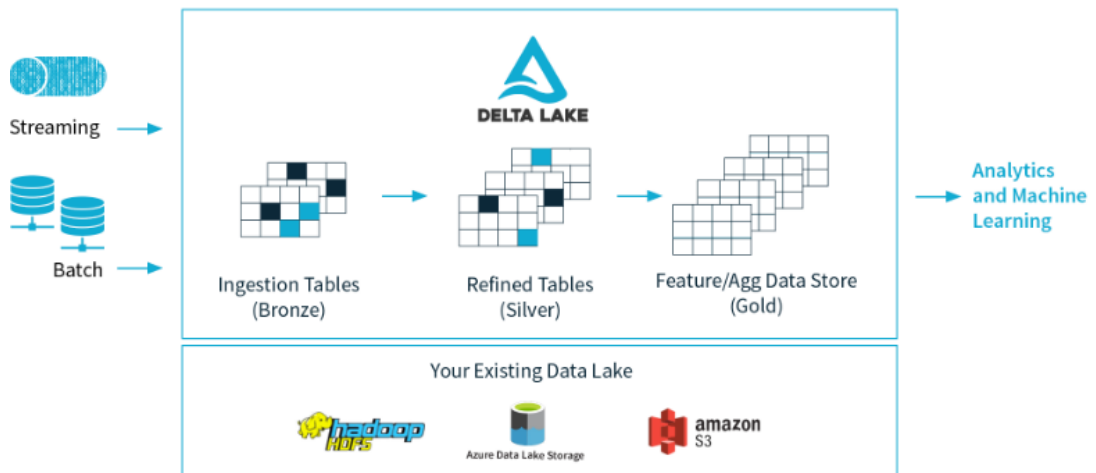


Figure 2.3: Delta Lake [2]

Time travel of Delta Lake enables you to query an older snapshot of its table and it can be used for recreating analyses, reports, or outputs, writing complex temporal queries, fixing mistakes in the data and providing snapshot isolation for a set of queries for fast changing tables.

2.3.2 Iceberg

Apache Iceberg [24] is a new table format for large tabular data designed to improve the defacto standard table layout built into Hive, Presto, and Spark. It tracks individual data files in a table instead of directories and allows writers to create data files and add files to the table only in an explicit commit. Table state is maintained in metadata files through replacing old meta data with atomic operation and creating new metadata file when the table state changes. This metadata file keeps track of partition configurations, schema of the table, snapshots of the table contents and other properties. Snapshot isolation is provided when transitioning from one table metadata file to the next which enables reading the latest snapshot or table state for the currently loaded metadata file until the new meta data location is provided.

Manifest files store data files in snapshots containing row for each data file in the table, partition data, and metrics. Snapshot is a combination of all files in its manifests and its listed in the metadata file. For avoiding rewriting of slowly changing metadata , manifest files are shared between snapshots. Design benefits of Iceberg:

- Version history and rollback: Table snapshots are kept as history and tables

can roll back if a job produces bad data.

- Safe file-level operations: Compacting of small files and appending of late arriving data is enabled due to Icebergs support for atomic changes.
- Schema evolution: Columns are tracked by ID to fully support add, drop, and rename across all columns and nested fields.

Chapter 3

Research Methodology

The chapter describes about the methodology used to integrate Hudi with Hops and feature store including the software used for the implementation and a clear specifications of the procedures and measurements used for the performance benchmarking of Hudi primitives, that provide solutions to the already identified problem.

3.1 Procedures, Measurements and Experimental Setups

For integrating and extending Hudi to work with Hops and Feature store, the compatibility of the different system versions such as spark-kafka library, need to be considered and modified to make the system with suitable classes easily integrate according to our need. In addition to this, the current implementation of Feature store needs to be modified and extended to support a way of tracking the changes with incremental pull primitives that could be useful for analytics, monitoring changes in data, search index, measuring data quality, and triggering event based actions in your ecosystem. The thesis used different Apis of Hudi and hops for the integration with Ubuntu 18.04 LTS being the operating system. Besides this, a set of measurements and observations were performed to describe the state and properties of real-world incremental processing system that is Hudi's upserts and incremental pull primitives for improving our understanding. The performance benchmark focuses on the latency of ingesting data and updates to previously stored data by measuring latency of read write operations as we increase amount of data being ingested.

3.2 Implementations

This section covers ways of ingesting new changes from external sources to the data lake using DeltaStreamer tool, mechanisms of tracking changes and speeding up large Spark jobs via upserts of Hudi.

The Hudi DeltaStreamer utility as described in section 2.1.5 provides ways to ingest data from different sources such as DFS or Kafka, with the following capabilities:

- Support json, avro or a custom record types for the incoming data.
- Exactly once ingestion of new events from Kafka, DFS, or incremental imports from output of HiveIncrementalPuller.
- Manage checkpoints, rollback and recovery
- Leverage Avro schemas from DFS or Confluent schema registry.
- Support for plugging in transformations.

3.2.1 DFS as a Source Integration of HUDI with Hops

Hudi is implemented as a Spark library, which makes it easy to integrate into existing data pipelines or ingestion libraries. Hudi Clients prepare the RDD of HudiRecord that contains the data to be upserted as a Spark DAG. Hudi can ingest data from different sources to HopsFS and hive. The source for the data with its specific properties is set in the properties file where the hoodie-utilities package gives it support to fetch the input data in a AVRO row format through the use of SourceFormatAdapter with ability to fetch new data in a row format.

The data source configuration file should be placed in the dfs.source.properties file having every description about the source and other info for the hive sync since the configuration files are taken from that.

Listing 3.1: DFS Source HUDI Configuration Properties

```
include=base.properties
# Key generator properties
hoodie.datasource.write.operation="upsert"
hoodie.datasource.write.recordkey.field=key
hoodie.datasource.write.partitionpath.field=date
hoodie.datasource.write.precombine.field=ts
# Schema provider properties with absolute path
```

```

hoodie.deltastreamer.schemaprovider.source.schema.file=
  hdfs:///source.avsc
hoodie.deltastreamer.schemaprovider.target.schema.file=
  hdfs:///target.avsc
hoodie.insert.shuffle.parallellism="2"
hoodie.upsert.shuffle.parallellism="2"
# DFS Source
hoodie.deltastreamer.source.dfs.root=
  hdfs:///Projects/Hudi/DataSource

```

Listing 3.2: Hudi Spark Job Argument Specification for DFS Source

```

--target-table Stock
--storage-type COPY_ON_WRITE
--target-base-path
  hdfs://10.0.2.15:8020/Projects/Hudi/Output/Stock
--props
  hdfs:///Projects/Hudi/Resources/dfs-source-sync.properties
--source-ordering-field ts
--schemaprovider-class
  com.uber.hoodie.utilities.schema.FilebasedSchemaProvider
--payload-class
  com.uber.hoodie.OverwriteWithLatestAvroPayload

```

The above two listings describe Hudi's configuration properties. Further configurations which are related to the Deltastreamer, Hive sync, and metric can also be set in the `dfs.source.properties` file. Different classes can be used for setting such configurations. For example:

- **HoodieWriteConfig:** Sets the name of the Hudi table to which the data is going to be stored in the hdfs path.
- **HoodieStorageConfig:** For setting the file size of parquet files.
- **RecordPayload Config:** Defines how to produce new values to upsert based on incoming new records. Hudi provides default implementations such as `OverwriteWithLatestAvroPayload` which simply update storage with the latest/last-written record. This can be overridden to a custom class extending `HoodieRecordPayload` class.
- **HoodieDeltaStreamer Metric:** For detailed metric configuration settings.

3.2.2 Kafka as a Source Integration of HUDI with Hops

Kafka is a streaming platform with ability of processing streams of records as they occur with publish and subscribe operations to streams of records. It is used in building real time streaming data pipelines that reliably get data between systems or applications and run as a cluster on one or more servers spanning multiple data centers. Kafka cluster stores streams of records in a categories called topics in fault tolerant durable way with each record consisting of key, value and timestamp [3]. The four core API's of kafka are producer, consumer, stream, and connector. The producer and consumer APIs allow an application to publish and consume stream of records to and from kafka topics respectively. Stream API transforms input streams to output streams by consuming input streams and producing output stream to one or more output topics. While the connector API, allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

Records are published into topics in kafka and topics can have zero or many subscribers to consume the data written to it. The kafka cluster maintains a partitioned log for each topic and each partition is an immutable ordered sequence of records where each record is uniquely identified by its sequential id number which is the offset.

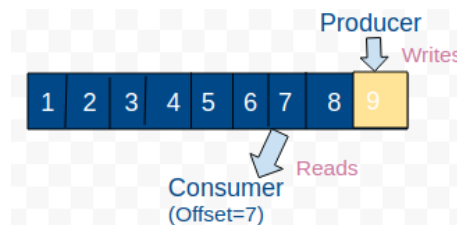


Figure 3.1: Partition log and offsets of Kafka [3]

A high level kafka gives guarantees that message appending to a particular topic partition will happen in the order of sent by the producer and records are seen by a consumer in the order they are stored in the log. The only metadata retained on a per consumer basics is the offset of the consumer and this offset is controlled by the consumer through advancing its offset linearly as it reads records or consuming records in any other it likes as the position is controlled by the consumer. For instance, a consumer can reset to an older or to the most recent offset for processing data from the past and present respectively.

Listing 3.3: Kafka Source HUDI Configuration Properties

```
include=base.properties
```

```
# schema provider configs
hoodie.deltastreamer.schemaprovider.source.schema.file=
  hdfs:///source.avsc
hoodie.upsert.shuffle.parallellism="2"
# Kafka Source
hoodie.deltastreamer.source.kafka.topic=netsi
#Kafka properties
bootstrap.servers=10.0.2.15:9091
auto.offset.reset=earliest
group.id=test-consumer-group
enable.auto.commit =false
key.deserializer=
  org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=
  org.apache.kafka.common.serialization.StringDeserializer
security.protocol=SSL
ssl.keystore.location=k_certificate
ssl.truststore.location=t_certificate
ssl.truststore.password=password
ssl.keystore.password=password
ssl.key.password=password
# Key fields
hoodie.datasource.write.operation="upsert"
hoodie.datasource.write.recordkey.field=key
hoodie.datasource.write.partitionpath.field=date
hoodie.datasource.write.precombine.field=ts
```

I modified Hudi to be able to work with kafka as a main source. Because the current implementation of Hudi uses the spark-kafka 0.8 version with a deprecated kafka cluster class while hopsworks is using the 0.10 version. The old spark-kafka cluster class uses old classes for offset generation and managing it with 'TopicAndPartition', 'KafkaCluster' and 'LeaderOffset' classes which are not part of spark-kafka-010. Hence, publishing a json file to kafka using Python in Jupyter Notebook and trying to run a Hudi as spark job with kafka as a main source would result with being not able to connect to the kafka cluster and its available topics in the partition.

The previous kafka cluster class was being used for interacting with Kafka cluster which is not compatible with the hops spark-kafka version. It had methods such as getPartitions, getLatestLeaderOffsets, and getEarlistLeaderOffset for returning the partitions and offsets off the newest and oldest kafka read. Kafka offset strategies with Largest and Smallest. After spark 2.2.1 it seems that the spark.streaming.kafka package with some helper classes wrapped around Kafka

API was dropped Which makes it possible for Hudi to work directly with Kafka instead of using the Spark-Kafka helpers. The current kafka cluster have Latest and earliest offset Strategies and uses the kafka consumer class to obtain the meta data of different topics subscribed including partition Infos instead of using the kafka cluster class to obtain the current meta data for a topic. Offset consumption is done by seeking to the beginning and end positions of the topicPartitions of a consumer for earliest and latest offset strategies respectively. These offsets will enable us to calculate the set of offset ranges to read from a kafka topic.

KafkaConsumer $\langle K, V \rangle$ is used instead of kafka cluster. Its instantiated by providing a set of key value pairs as configuration. Kafka maintains a unique numerical identifier, offset, for each record in a partition which denotes the position of the consumer in a partition. For instance, a consumer at position 5 is waiting to receive the record with offset 5 after consuming records with offsets 0 to 4. Manual offset control have direct control over when a record is considered to consumed against automatic offsets [30].

The kafka utils class provides java constructor for a batch oriented interface for consuming from kafka. A key value pair called ConsumerRecord $\langle k, v \rangle$ is to be received from kafka. This consists of topic name and partition number of the the record being received, offset number pointing to the record in a kafka partition, and timestamp referring to the mark of the corresponding producer. It requires the kafka configuration parameters, offset ranges, and location strategy to be passed as parameters to consume from a specified topic partition.

3.2.3 HUDI and HiveSyncTool

As described in section 2.1.6, Hudi store data physically once on HDFS, while providing 3 different kinds of logical views on top of it and syncs the dataset to the hive metastore using the HiveSyncTool while providing external Hive tables backed by Hudi's custom input formats that can be queried by popular query engines like Hive and spark after providing the proper Hudi bundle. Figure 3.2 shows all components and utilities of Hudi to perform its full functionality.

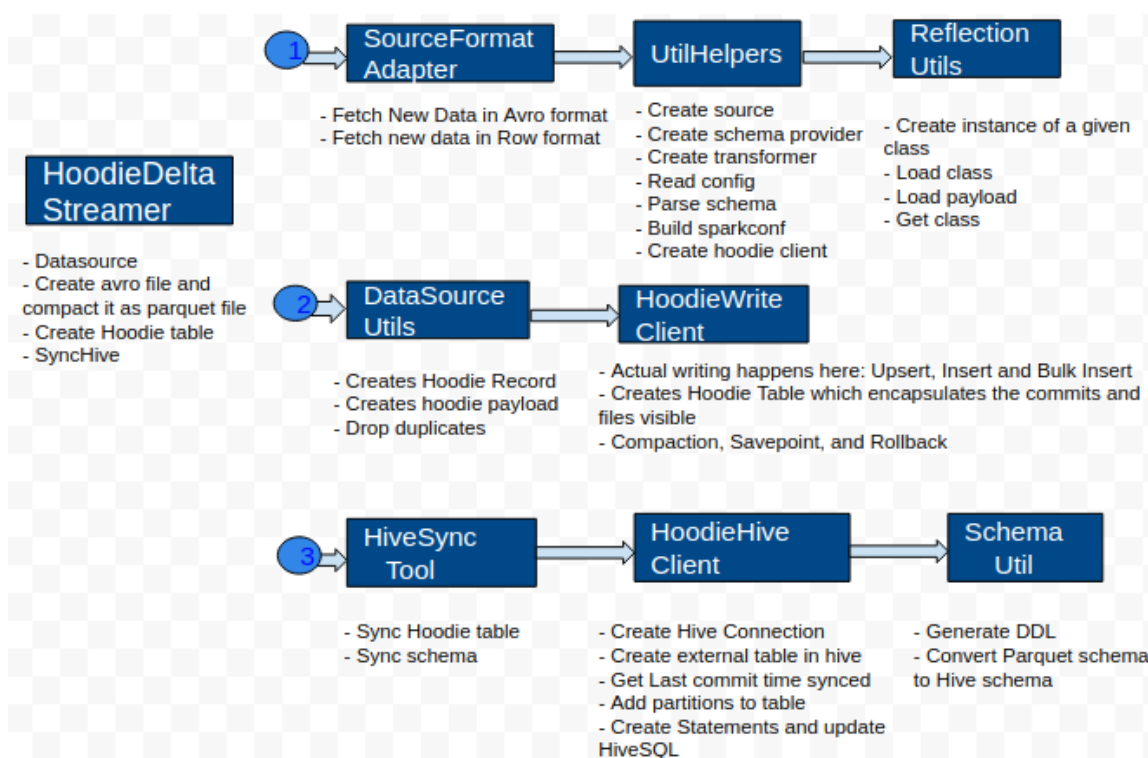


Figure 3.2: Hudi DataStreamer and Other Utilities

The HiveSyncTool can be called every time from DeltaStreamer and spark source by enabling the hive sync parameter setting or manually calling it after every commit to synchronize the Hudi modeled table on HDFS with hive table. The tool checks the commit time and updates and syncs the schema and partitions incrementally with out dropping the table every time the hive sync tool is called. This functionality is partially described in listing 3.4 and 3.5.

Listing 3.4: Hive Sync DDL for Creating Table

```
CREATE EXTERNAL TABLE IF NOT EXISTS Hudi.hoodie_trial(
  `_hoodie_commit_time` string, `_hoodie_commit_seqno`
  string, `_hoodie_record_key` string,
  `_hoodie_partition_path` string, `_hoodie_file_name`
  string, `ts` string, `symbol` string, `year` int,
  `month` string, `key` string)
PARTITIONED BY (`date` string)
ROW FORMAT SERDE
'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
'com.uber.hoodie.hadoop.HoodieInputFormat'
```

```

OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION 'hdfs://10.0.2.15:8020/Projects/Hudi//Model'

```

Listing 3.5: HiveSyncTool sample to Sync Partition

```

ALTER TABLE Hudi.hoodie_trial ADD IF NOT EXISTS PARTITION
(`date`='2018-08-31') LOCATION
'hdfs://10.0.2.15:8020/Projects/Hudi/Model/2018/08/31'

```

3.2.4 Hudi Integration with FeatureStore

Currently, Feature store is using Hive managed table to store its datasets on a columnar based format which is specifically ORC and this wealth of feature data is used to operate machine learning systems and to train their models. Feature Store allows teams to manage, store, and discover features for use in machine learning projects. It gives teams the ability to define and publish features to this unified store, which in turn facilitates the discovery and feature reuse across machine learning projects [31].

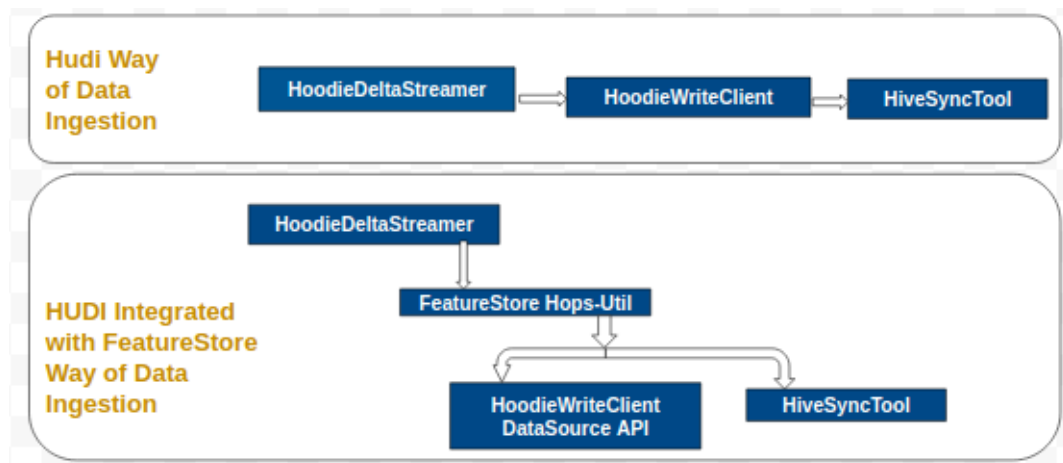


Figure 3.3: Implementation System Flow of Hudi and Hudi Integrated Feature Store

The current implementation of Feature store doesn't support a way of tracking the database and have a materialized views for different purposes. Hence, the challenge here is to track database changes and have materialized views that could be useful for analytics, monitoring changes in data, search index, and taking actions using the changed data. On the other hand, Hudi provides a way to solve

one of the biggest challenges that is updating data into an existing dataset while working with data lakes and uses Avro format before compaction since avro has a static type of schema that can evolve.

As described in section 2.1.6, Hudi storage format is able to capture, track and handle the changed data, because it allows incremental updates over key expression, which could be a primary key of the table. For integrating Feature store with Hudi, the DataSource API is used to quickly start reading or writing Hudi datasets with the use of DataSourceReadOptions and DataSourceWrite options respectively. This API is used to enable storing of a feature store as a Hudi modeled table on top of HDFS and synchronize the Hudi modeled table stored on HDFS with an external Hive table with the use of sync tool after checking the schema and partition management.

In addition to this, the schema util API of Hudi was used to generate the DDL for the external table that refers to the Hudi modeled table stored on HDFS. This DDL is generated by getting the parquet schema from the stored dataset by looking at the latest commit. After the featurestore is created and synced with external hive table , we are able to query the latest dataset, and expose data in our data lake such as Hive metastore. This architecture helps in maintaining up to date fully monitored datasets into the data lake and consume changes to have the latest view.

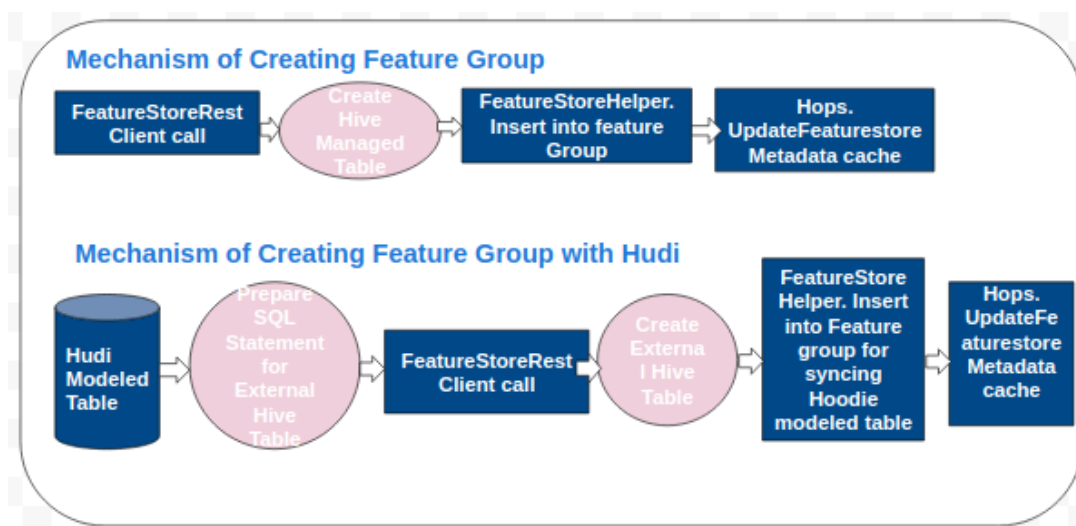


Figure 3.4: Mechanism Of Integrating Hudi with Feature Store

Chapter 4

Result and Analysis

This chapter explains the results of the integration and analysis done on the thesis based on the results of our implementation. It mostly covers performance analysis that compares hudi's upsert primitive with bulk loading and other kinds of performance comparisons that focus on the level of capturing updates to existing data. In addition, it discusses the test configurations used for testing the performance followed by a qualitative comparison of the 3 similar systems with a high focus on the features and difference's of Hudi, Delta Lake, and Iceberg.

4.1 Row Level Updates with Hudi Upsert

Hudi groups inserts per partition by assigning a new file Id and appending its corresponding log file until it reaches the limited HDFS block size and continues to create another file id and repeat this process for all the remaining inserts in that partition. A time-limited compaction process is started by a scheduler every few minutes to generate ordered list of compactions which compacts all the avro files for a file Id with the current parquet file for creating the next version of that parquet file.

Compaction is a background activity which runs asynchronously to reconcile differential data structures through moving updates from row based log files to columnar formats within Hudi [32]. Compaction locks down specific log versions which are being compacted and writes the new updates to that file Id into a new log version. For a successful ingestion job, an ordered commit with details about partitions and the file Id versions is created and recorded in the Hudi meta timeline through renaming an inflight file to a commit file.

The upsert operation of Hudi became feasible due to the bloom indexing mechanism of mapping records to a file id. In addition, the monotonically

increasing metadata of the commit time line gives the ability of tracking which data was upserted at what instance of time and if a failure occurs to which savepoint to rollback. Figure 4.1 shows the commit metadata of an upsert operation with 197 number of writes and 100 number of update writes in its next ingest operation. The checkpoint in the extra metadata section is used as a savepoint. For a kafka as a source, the topic name with its beginning and end offsets read are used as a checkpoint preparing it for its next ingestion.

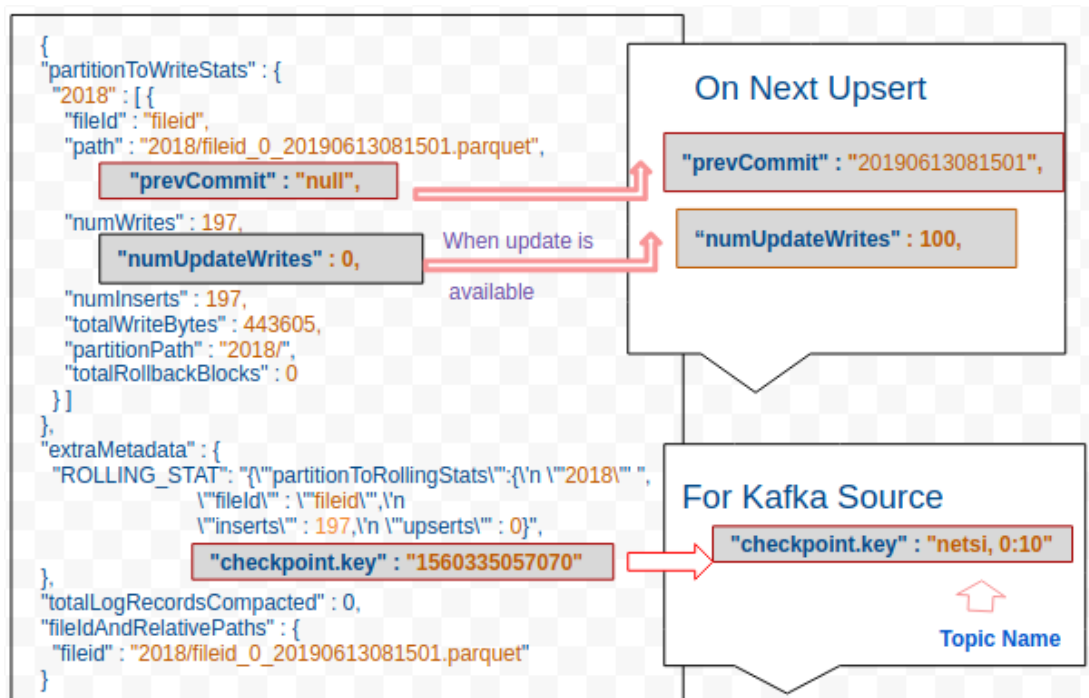


Figure 4.1: Sample Commit Data of Hudi Upsert

How good is Hudi for row-level updates? Row-level updates in this case refers to just a single row update read from Kafka or any DFS source, where that row is already persisted in the Hudi table. Instead of the typical case where a lot of rows are read in a batch from Kafka and inserted into Hudi.

When a Hudi ingestion operation with row-level updates is running, Hudi loads the bloom filter index of all parquet files in the involved partitions of the previous ingestion and marks the record as either an update or insert through mapping the incoming record keys to existing files. The corresponding log file of a file id is appended with the updates or a new log file is created if it doesn't exist. Compaction prioritizes large amounts of logs. Thus, files with the largest amount

of logs are compacted first and other compactions follow based on the size of their log data referring to small log files being compacted last. The performance of the row level updates or join could differ according to the input batch size, partition span, or number of files in a partition. Hence, a range partitioning on a joined key and sub-partitioning is done to automatically avoid and handle the 2GB limit for a remote shuffle block in Spark.

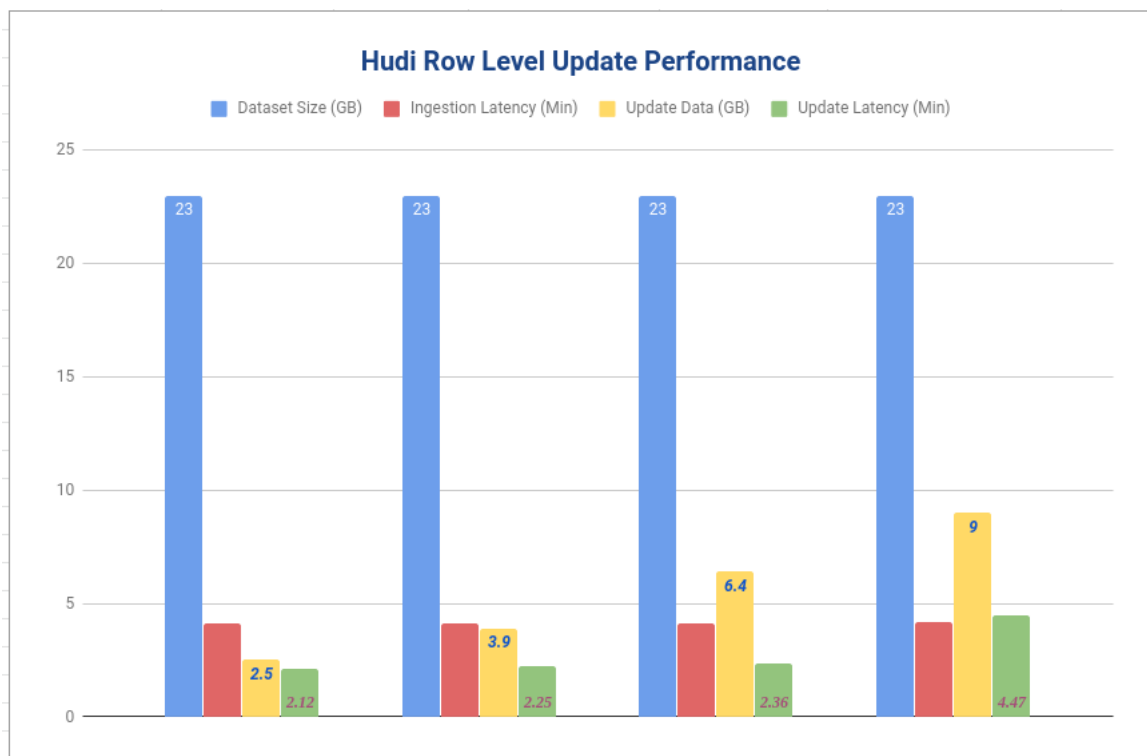


Figure 4.2: Row-level Updates Skew over Input Batch Size

Figure 4.2 shows, the update performance of Hudi to an existing data stored in a data lake. The data lake consists of a 23 GB data size and different sizes of updates are ingested in each round. This indicates that the upsert mechanism is a more efficient approach if ingestion is needed to keep up with the typically high update volumes. The main reason is due to incremental update which separates its updates to the data lake by providing a batch of the updates instead of loading all the data including the updates at once which will cause high ingestion latency.

4.2 Incremental Pulling and Time Travel

What is the content of the different versions of a parquet file inside a partition after a the completion of Hudi upsert? This question raises a main concept on how Hudi's upsert mechanism works, how many files are created for a single spark job with Hudi upsert operation and the difference among the different parquet files inside a partition. As data gets written, updates to existing file groups produce a new slice for that file group stamped with the commit instant time, while inserts allocate a new file group.

This concept was verified by having 2 json files as an input source. The first input source have 197 rows as a json file while the second data source have 99 json rows as a source. After running the Hudi upsert operation with the first data source, a single parquet file was created having all the first time insert along with a commit meta data. Then, running the same operation with the second data sources that have 99 updates to the previous upsert operation created a single parquet file that have the latest dataset with a new commit metadata file.

If we try using these json source files by exchanging their order that is first using the 99 row input source and then the 197 row file, it first created single parquet file with a commit metadata for the first time insertion. Then, it created 2 parquet files with the same single commit metadata for the second round of upsert operation that have the 197 rows(insert= 98 rows & update=99 rows) verifying that each updates and inserts to an existing file group produce a new slice for that file group stamped with the commit instant time.

SQL queries running against such a dataset, first check the timeline for that latest commit and filters all but latest file slices of each group. Thus, queries are immune to any write failures and current inflight commits, which makes it only run on committed data.

Case 1: Upserts create new Versioned File

InputSource	Number of rows	Commit time	Num of Parquet file and Name	
Batchjson 1	Insert 197 rows Upsert 0 rows	20190411090958	1	0_20190411090958.parquet
BatchJson 2	Insert 0 rows Upsert 99 rows	20190411144159	1	0_20190411144159.parquet All data inside the file

Case 2: New versioned files for each Insert and Upsert

InputSource	Number of rows	Commit time	Num Parquet file	Parquet file name
Batchjson 2	Insert 99 rows Upsert 0 rows	20190411090958	1	0_20190411090958.parquet
BatchJson 1	Insert 98 rows Upsert 99 rows	20190411144159	2	0_20190411144159.parquet 1_20190411144159.parquet

Figure 4.3: Effects of Hudi Upsert Operation to Number of Parquet Files

As seen from Figure 4.3, the older version of the commits and files are kept after each Hudi upsert operation by setting the Hudi cleaning policy to `KEEP_LATEST_COMMITS`. This setting enables us to do incremental pull and perform time travelling and keep old versions of the parquet files and commits. After setting the `KEEP_LATEST_COMMITS` cleaning policy, we are able to keep all the X commits and perform time traveling to generate reports based on all of our previously committed data. The incremental pull is one key primitive that highlights the incremental processing of Hudi to obtain a change stream or log from a dataset.

Listing 4.1: Hudi Cleaning Policy Setting

```
Property: hudi.cleaner.policy=KEEP_LATEST_COMMITS
```

As the Hudi documentation explains, the Hudi DataSource API provides a fabulous way to incrementally pull data from Hudi dataset and process it via spark with a means to get all and only the updated and new rows since a specified instant time [33].

Figure 4.4 shows a sample incremental pull for obtaining all the records written

since a specified instant of time. The begin and end instant times need to be specified along with the view type to read a Hudi dataset and perform time traveling. Incremental view type enables us to get only updated values from a given instant time. As summarized in the figure, the first incremental pull resulted in 197 rows which were committed at the first ingestion. The second incremental pull resulted in 98 rows of data which show only the updates after the first commit. While the last incremental pull, resulted in 0 rows since nothing was committed after the second ingestion.

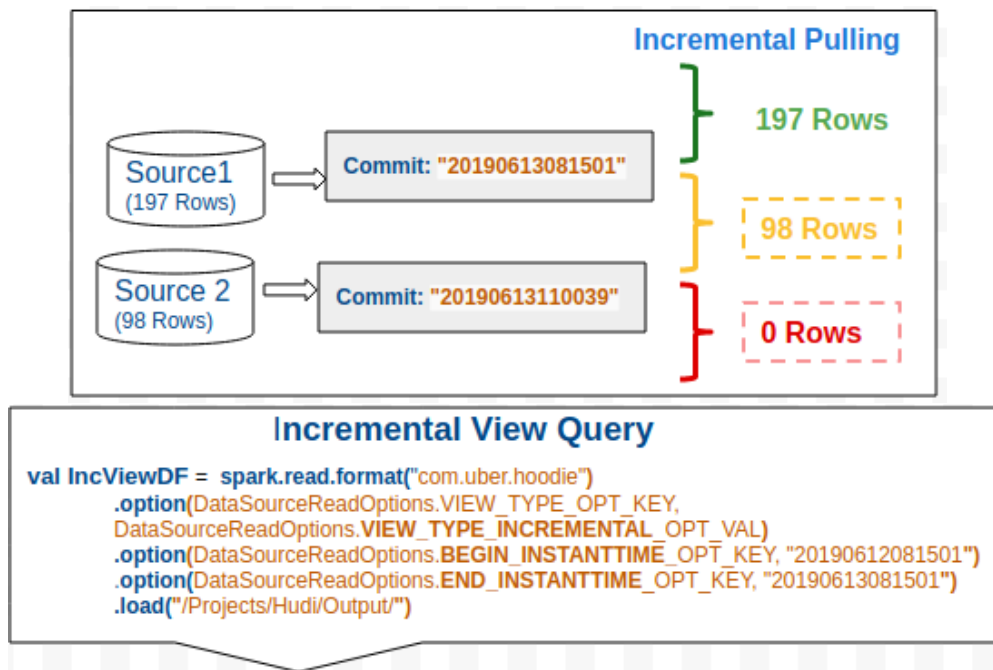


Figure 4.4: Hudi DataSourceRead API for Incremental Pulling

HiveIncremental puller is also another way of realizing incremental view by incrementally extracting changes from large tables via HiveQL instead of full table scans in order to speed up query with special configurations. These special configurations allow the query planning phase to only fetch incremental data out of the dataset.

Listing 4.2: HiveIncremental Puller For Incremental Pull

```
set Hudi.hudi_trial.consume.mode=INCREMENTAL;
set Hudi.hudi_trial.consume.start.timestamp=001;
set Hudi.hudi_trial.consume.max.commits=10;
select `_hoodie_commit_time` from hudi_trial where
  `_hoodie_commit_time` > '20190408140118' limit 100;
```

4.3 Partition and Schema Management

Recently ingested data is only available in the Hudi modeled table until the hive sync tool runs. Let's assume hive sync tool runs once per day during night time and a table called transactions is available where clients get all the transactions which took place over the last 30 days. The 29 days of transactions are synchronized with hive. If the transactions of the 30rd day are not yet synchronized with hive, where should clients issue a query to synchronize the Hudi modeled table with the external hive table?

The purpose of the hive sync tool is for reconciling the Hudi table partitions with the hive partitions so basically it's for schema and partition management. Data management is not handled by the hive sync tool. Hence, until the hive sync tool runs, queries on the hive table won't be able to see the new partition changes in the Hudi modeled table. This is why the sync tool is needed for partition management through giving support to synchronize the partition changes.

4.4 Performance Comparison

Cluster computing frameworks and modern scale-up servers are used for running big data analytics applications with high volume and variety of data. These continuously evolving frameworks are used to provide real time data analysis capabilities. For performance evaluation of Hudi, spark is used as the forefront of big data analytics with large number of machines that can scale up to provide a unified framework for batch and stream processing. Its programming model is based on higher order functions that execute user defined functions in parallel where each stage is splited into tasks that consists of data and computation [34].

In this section, some real world performance comparison between Hudi upsert mechanism and bulk loading is covered. Additionally, it discusses ways to optimize the data ingestion with incremental upserts. At the end of this section, it discusses the effects of cluster provisioning and shuffle parallelism on speeding up ingestion.

4.4.1 Hudi Upsert versus Bulk Loading

Hudi's fast ingestion feature comes from its usage of built in index mechanism. The index mechanism stores a mapping between recordKey and the file group id it belongs to with the help of file ranges and bloom filters. Hudi provides best indexing performance when the recordKey is modeled to be monotonically increasing rather than a random UUID, which leads to range pruning and filtering

out a lot of files for comparison. This is the main reason why ingesting 50GB of data into Hudi modeled table is taking no less than 15-30 minutes in an average case with the proportional cluster provisioning. In addition, Hudi can store a dataset incrementally which opens doors for scheduled ingesting which leads to reduced latency, with a significant efficiency on the overall compute cost.

Hudi data ingestion latency differs according to the cluster setting and spark parameters settings. These settings combine of executor memory, driver memory, and number of executor cores provisioned. For example, supplying high number of executors according to the cluster provisioned can greatly optimize the ingestion latency of a large dataset. This is because spark has a shuffle service that helps shuffle the data and write them to the local file system to be accessed by spark. Using really low number of executors for large dataset forces all the data to be shuffled to those small number of nodes leading to out of disk issue while high number of executors results in optimizing ingestion latency.

One way of optimizing ingestion latency is setting the correct amount of shuffle parallelism. Setting excessive parallelism for low volume of data causes additional spark overhead which in turn increases ingestion latency. When should clients start increasing the parallelism for achieving efficient ingestion latency in accordance to their input data size?

Hudi write operation is an advanced spark job, so scaling tips that apply for spark also apply for Hudi. If the input dataset is larger up to 500GB, the shuffle parallelism of the upsert or insert operation need to be bumped to at least $input_data_size/500MB$. This is because Hudi tends to over-partition input to ensure each spark partition stays within the 2GB limit for inputs up to 500GB. Hence, increasing parallelism by 1 for every 1GB of additional data could help in decreasing high ingestion latency.

Figure 4.5 shows the latency recorded after ingesting Json data into a Hudi modeled table through upsert operation of Hudi in comparison to bulk loading. Bulk loading refers to ingesting database tables all at once in full, unlike Hudi's upsert operation which ingests data incrementally. Simply put, its the difference between fully rewriting your table as you would do it in the pre Hudi world and incrementally rewriting at the file level in the present day of using Hudi.

For this comparison, 32 up to 34 GB of data is used including the updates. In the first comparison, 31.7 GB of data is ingested using Hudi followed by ingesting the 337 MB of update data which took about 4.43 minutes to update the entire datalake. While for bulk loading the 31.7 GB of data with its 337 MB of updates is ingested all together at once. Hence, it took about 8.01 minutes to ingest the 32 GB of data using bulk loading which is approximately as twice of the ingestion

latency it took for Hudi upsert operation with the same amount of data. This particular case indicates that Hudi updates the data incrementally to minimize the ingestion latency which is 50 percent more efficient in terms of ingestion latency in this particular case. Based on the result of our comparison shown in the figure, we can conclude that Hudi's upsert operation outperforms bulk loading as its taking less time to ingest the same amount of data.

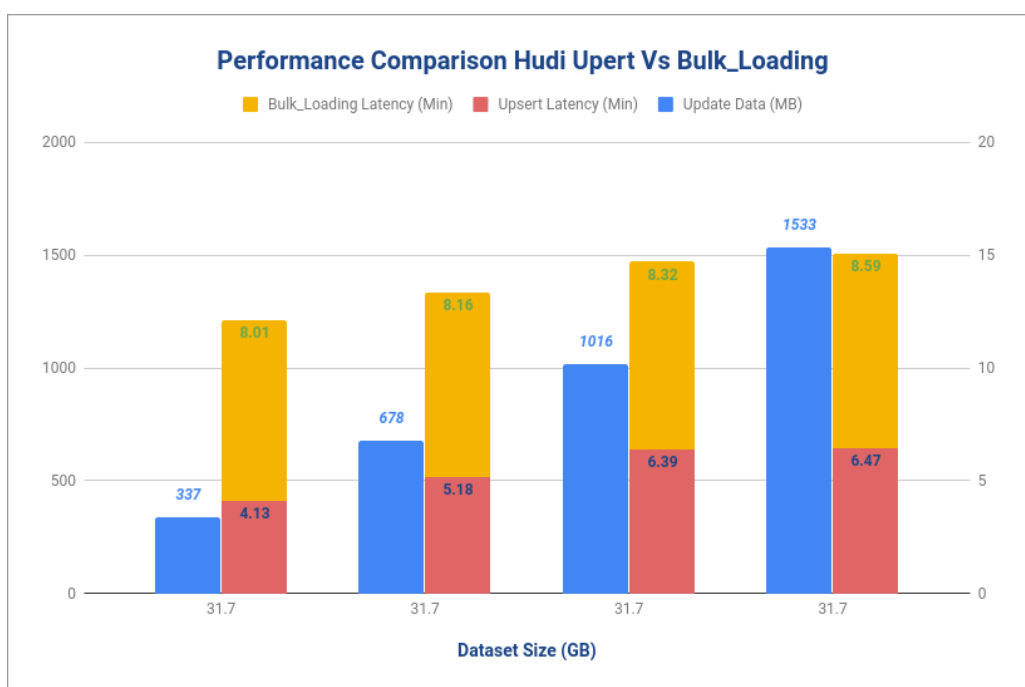


Figure 4.5: Performance Comparison of Hudi Upsert with Bulk Loading

Hudi write performance depends on two things: indexing and writing parquet files (it depends on the schema and CPU cores on the box). Since Hudi writing is a spark job, speed also depends on the parallelism you provide. For a specific dataset with high number of schema columns, the parquet writing is much slower.

If the dataset to be ingested has no updates, then issuing insert or bulk_insert operations of Hudi is better than upsert to completely avoid indexing step and that will gain speed. Difference between insert and bulk_insert is an implementation detail. Insert() caches the input data in memory to do all the cool storage file sizing, while bulk_insert () uses a sort based writing mechanism which can scale to multi terabyte of data in its initial load. In short, the bulk_insert is used to bootstrap the dataset, then insert or upsert are used depending on need.

4.4.2 Feature Store versus Hudi Integrated Feature Store

Hudi organizes datasets into a directory structure under a base path which in turn breaks up a dataset into partitions, which are folders containing data files for that partition. Within each partition, files are organized into file groups, uniquely identified by a file id where each file group contains several file slices, where each slice contains a base columnar parquet file. Uniqueness of Hudi record key is only enforced within each partition. The `_hoodie_record_key`, `_hoodie_commit_time`, `_hoodie_file_name` and `_hoodie_partition_path` are added to every record as meta data columns.

Hudi maps a given hoodie key (record key + partition path) efficiently to a file group, via its indexing mechanism. A monotonically increasing commit data with commit number is created after the task of upserting or inserting a batch of records. This commit data provides atomicity guarantees to Hudi so that only committed data is available for querying. Beside this, it helps in tracking and maintaining timeline of all actions performed on a the dataset and provide efficient time travel to provide instantaneous views of the dataset at different instants of time.

Using Feature store with Hudi as a storage type provides us the advantage of performing incremental processing as compared to the bare implementation of feature store. This gives an adequate answer to the question of how clients fetch data that changed over time. Hudi integration with Feature store makes it possible to handle incremental pulls because the Hudi table format columns such as `_hoodie_record_key`, `_hoodie_commit_time`, `_hoodie_file_name` and `_hoodie_partition_path` are added to every record as meta data columns. These meta data columns handle the pull over a key expression. The harmony among both systems (Hudi and Feature store) further strengths the main goals of Feature store. This is due to the facilitated machine learning model training with features that are incrementally pulled and reuse of these features for operating in machine learning systems.

Listing 4.3: Feature Store SQL Definition

```
CREATE TABLE `stock_1` ( `volume` bigint, `ts` string,
  `symbol` string, `year` int, `month` string, `key`
  string, `day` string) ROW FORMAT SERDE
  'org.apache.hadoop.hive ql.io.orc.OrcSerde'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive ql.io.orc.OrcInputFormat'
```

```

OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'
LOCATION
'hdfs://10.0.2.15:8020/apps/hive/warehouse/hudi.db/stock_1'
TBLPROPERTIES ( 'bucketing_version'='2',
  'transient_lastDdlTime'='1562580582')

```

Listing 4.4: Hudi Integrated Feature Store SQL Definition

```

CREATE EXTERNAL TABLE `stockfeature_1` (
  `_hoodie_commit_time` string, `_hoodie_commit_seqno`
  string, `_hoodie_record_key` string,
  `_hoodie_partition_path` string, `_hoodie_file_name`
  string, `volume` bigint, `ts` string, `symbol` string,
  `year` int, `month` string, `key` string, `day` string)
PARTITIONED BY ( `date` string)
ROW FORMAT SERDE
'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'com.uber.hoodie.hadoop.HoodieInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  'hdfs://10.0.2.15:8020/Projects/Hudi/Output/StockFeature'
TBLPROPERTIES ( 'bucketing_version'='2',
  'last_commit_time_sync'='20190614134937',
  'transient_lastDdlTime'='1560520192')

```

4.5 Qualitative Comparison of Delta Lake, Iceberg and Hudi

Hudi, Delta lake, and Iceberg are open source projects which are built for data in cloud storage with support to snapshot isolation. The evaluation of these systems is done based on the following different dimensions:

- Keeping track of data and management of file based datasets
- Table format
- Support for update and delete
- Support for upsert and incremental pull

- Read older versions of data using time travel
- Support of different file formats
- Support for different engines
- Support for compaction and cleanup

	Hudi	Delta Lake	Iceberg
Keep Track of Data and Management of File Based Datasets	Yes	Yes	Yes
Table Format	Hudi Table	Delta Table	Iceberg
Update and Delete support	Yes	Yes	No
Upsert and Incremental Pull Support	Upsert Incremental Pull	Not yet integrated	Not mentioned
Read Older Versions of data Using Time Travel	Yes	Yes	Not mentioned
Different File Format Support	Parquet Avro	Parquet	Parquet ORC
Engine Support	Spark Presto Hive	Spark Presto	Spark Presto
Compaction Support	Automatic	No	No
Cleanup Support	Automatic	Manual	No

Figure 4.6: Qualitative Comparison of Hudi, Delta Lake and Iceberg

Figure 4.6 summarizes the comparison of the solution systems as of this date. Hudi seemed more advanced with regard to support to different engines as it can be coupled with Spark, Presto, and Hive. In addition to that, it gives support for automatic compaction and cleanup. While Delta lake and Iceberg seem to be promising systems which can manage and keep track of data without having additional support to upsert and incremental pull as Hudi.

Chapter 5

Conclusion and Future Work

This chapter goes through summarizing the main features implemented in this project. It finally concludes by suggesting some possible future works to be investigated on.

5.1 Conclusions and Recommendations

In this thesis, we successfully integrated and extended Hudi to work with Hopsworks platform mainly with HiveonHops and Feature store. With this implementation, users can create incremental data pipelines that yield greater efficiency and optimum latency for their data lake than their typical batch alternatives. This provides ability to atomically upsert datasets with new values in near real time and consume the sequence of changes to a dataset from a given point in time. This enables incremental data processing in addition to making data available quickly to existing query engines like hive and spark.

Moreover, a large volume of data can be ingested from numerous data sources in an efficient and effective manner with the power to maintain, modify and format the data for analytics and storage purposes. This approach helps Feature store to adequately train its machine learning model with the incrementally pulled datasets and perform time travel on the stored features over a time expression drawn from the monotonically increasing commit timeline.

5.2 Future Work

This thesis sets the basics for Hudi on hops. As explained in section 2.1.3, the merge-on-read storage type of Hudi is currently underdevelopment. Hence, our next goal is to investigate and integrate the merge on read feature of Hudi with the

Hopsworks platform and provide real time view by merging incoming and on-disk data on-the-fly. In addition, the performance comparisons of Hudi primitives with the regular ingestion tools is limited to our small cluster provisioning. Hence, further investigation will be held with large volumes of data in accordance to our provisioned cluster.

Bibliography

- [1] Hudi, “Apache hudi documentation,” 2019. [Online]. Available: <https://hudi.incubator.apache.org/>
- [2] DeltaLake, “Delta lake documentation,” 2019. [Online]. Available: <https://docs.delta.io/latest/index.html>
- [3] N. Narkhede, G. Shapira, and T. Palino, “Kafka: The definitive guide,” 2017. [Online]. Available: <https://book.huihoo.com/pdf/confluent-kafka-definitive-guide-complete.pdf>
- [4] J. Wang, W. Zhang, Y. Shi, and S. Duan, “Industrial big data analytics: Challenges, methodologies, and applications,” *IEEE*, 2018. [Online]. Available: <https://arxiv.org/pdf/1807.01016.pdf>
- [5] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, “Big data: The next frontier for innovation, competition, and productivity,” *McKinsey Global Institute*, 2012.
- [6] B. Courtney, “Industrial big data analytics: the present and future,” 2014. [Online]. Available: <https://www.isa.org/intech/20140801/>
- [7] S. Ghemawat, H. Gombioff, and S.-T. Leung, “The google file system,” *ACM SIGOPS Operating Systems Review, Google*, vol. 37, pp. 29–43, 2003. [Online]. Available: <https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf>
- [8] “Apache™ hadoop,” *Apache Software Foundation*, pp. 1–3, 2014. [Online]. Available: <https://hadoop.apache.org/old/index.pdf>
- [9] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters.” *Google*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: <https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf>

- [10] “Apache™ storm,” 2019. [Online]. Available: <https://storm.apache.org/releases/2.0.0/Concepts.html>
- [11] “Apache flink,” 2019. [Online]. Available: <https://flink.apache.org/flink-architecture.html>
- [12] V. Gulisano, R. Jimenez-Peris, M. P. Martinez, C. Soriente, and P. Valduriez, “Streamcloud: an elastic and scalable data streaming system,” *INRIA and LIRMM*, 2011. [Online]. Available: <http://www-sop.inria.fr/members/Patrick.Valduriez/pmwiki/Patrick/uploads/Publications/pds2011>
- [13] T. Weise, “Hudi proposal,” 2019. [Online]. Available: <https://wiki.apache.org/incubator/HudiProposal>
- [14] Z. Jacikevicius, “The hadoop ecosystem: Hdfs, yarn, hive, pig, hbase and growing,” *London, United Kingdom*. [Online]. Available: <https://www.datasciencecentral.com/profiles/blogs/the-hadoop-ecosystem-hdfs-yarn-hive-pig-hbase-and-growing>
- [15] H. Isah and F. Zulkernine, “A scalable and robust framework for data stream ingestion,” 2018. [Online]. Available: <https://arxiv.org/pdf/1812.04197.pdf>
- [16] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du, “Data ingestion for the connected world,” 2017. [Online]. Available: <https://pdfs.semanticscholar.org/78d4/fc136ffcfa192e2bdb4c08cdd56a0a0e2f64.pdf>
- [17] O. Ventura, “Building zero-latency data lake using change data capture,” *Yotpo Engineering*, 2019. [Online]. Available: <https://medium.com/yotpoengineering/building-zero-latency-data-lake-using-change-data-capture-f93ef50eb066>
- [18] Hopsworks, “Hopsworks documentation,” 2018. [Online]. Available: <https://hopsworks.readthedocs.io/en/0.10/overview/introduction/what-hopsworks.html>
- [19] A. Spark, “Apache spark.” [Online]. Available: <https://spark.apache.org/docs/latest/>
- [20] Tensorflow, “Tensorflow.” [Online]. Available: <https://www.tensorflow.org/tutorials/>
- [21] K. Hammar, “Feature store: the missing data layer in ml pipelines?” 2018. [Online]. Available: <https://www.logicalclocks.com/feature-store/>

- [22] A. HIVE, “Apache hive.” [Online]. Available: <https://hive.apache.org/>
- [23] P. Bock, “Getting it right: R&d methods for science and engineering,” *Academic Press*.
- [24] Iceberg, “Iceberg documentation.” [Online]. Available: <https://iceberg.apache.org/>
- [25] IPUMS, “Ipums usa dataset.” [Online]. Available: <https://usa.ipums.org/usa/>
- [26] V.-T. TRAN, “Introduction to distributed systems,” 2014. [Online]. Available: <https://www.slideshare.net/microlife/introduction-to-distributed-file-systems>
- [27] F. Buso, “Sql on hops,” 2017. [Online]. Available: <http://kth.diva-portal.org/smash/get/diva2:1149002/FULLTEXT01.pdf>
- [28] Hudi, “Writing hudi datasets,” 2019. [Online]. Available: https://hudi.apache.org/writing_data.html
- [29] P. Chockalingam, “Open sourcing delta lake,” 2019. [Online]. Available: <https://databricks.com/blog/2019/04/24/open-sourcing-delta-lake.html>
- [30] Kafka, “Class kafkaconsumer.” [Online]. Available: <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html>
- [31] T. Sell and W. Pienaar, “Introducing feast: an open source feature store for machine learning,” *AI & MACHINE LEARNING*, 2019. [Online]. Available: <https://cloud.google.com/blog/products/ai-machine-learning/introducing-feast-an-open-source-feature-store-for-machine-learning>
- [32] P. Rajaperumal and V. Chandar, “Hudi: Uber engineering’s incremental processing framework on apache hadoop,” *Uber Engineering*, 2019. [Online]. Available: <https://eng.uber.com/hoodie/>
- [33] “Hudi documentation: queryingdata.” [Online]. Available: https://github.com/apache/incubator-hudi/blob/asf-site/docs/querying_data.md#incremental-pulling-spark-incr-pull
- [34] A. Awan, “Architectural impact on performance of in-memory data processing,” 2016. [Online]. Available: <https://kth.diva-portal.org/smash/get/diva2:922527/FULLTEXT01.pdf>

TRITA TRITA-EECS-EX-2019:809