



## **Overview on Apache Hudi V 1.1**

**By Arihant Shashank (Lead data Engineer Ericsson)**

### **1. What is Apache Hudi**

- 1.1. Where do we need Apache HUDI?**
- 1.2. How Apache HUDI has been developed**
- 1.3. Where in Ericsson we can utilize Hudi datasets**
- 1.4. Hudi overview and architecture**

### **2. Storage type**

- 2.1 Copy on write**
- 2.2 Merge On Read:**

- 3 How to write HUDI datasets?**
- 4 How data is being inserted USING hudi**
- 5 Hudi Updating data**
- 6 Hudi deleting data**  
**Hudi rollback-> This has to be discussed**
- 7 More areas where HUDI can be used.**

## 1. What is Apache HUDI?

Apache Hudi is an open-source data management framework used to simplify incremental data processing and data pipeline development. This framework more efficiently manages business requirements like data lifecycle and improves data quality.

### 1. 1 Where do we need apache HUDI? (In very simple layman terms)

- **Take an example of GDPR implementations:** This law gives privilege to the customers to delete their data if they are no longer associated to the firm, And deletion or masking of those data becomes very challenging at times because going finding those specific records in the particular data lake and particular partition then deleting it, it will take lot of I/O computation, hence not very optimal thing (but we do that in day to day for GDPR)
- **CDC (Change data capture):** initial Bulk ingestion of the data from any traditional RDBMS to S3/GS/HDFS/ADLS is very easy, but when it comes to CDC , its very challenging to keep up with the latest data of RDBMS (Challenge is to keep data fresh in the new env). (Here Apache Hudi will show what has changed in RDBMS and upsert only those in new data lake using apache HUDI)
- **Streaming Data ingestion: TDW**

**How Apache HUDI has been developed? (I am explaining this because we can relate this use case with our day-to-day work)**

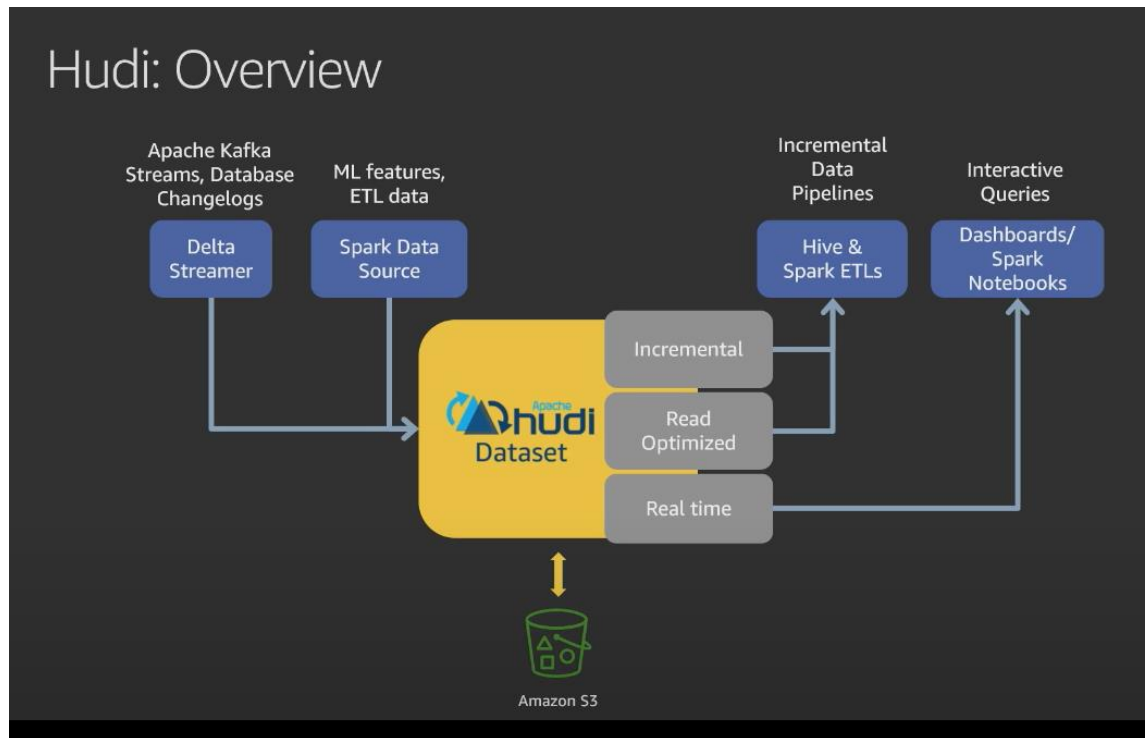
#### Updates in older partitions

- Sometimes you booked a uber cab and you didn't like the trip and after a month you have complained, and the fare got adjusted that needs a update in older partition
- Sometimes you have opened an app and rated a 3 month older trip, that also needs to be updated in that older partition
- You probably used Credit card and it didn't work that time and later you tried with some other Credit card
- Some trip tables have fare in local currency and if the conversion got changed then you need to implement that change in all the derived tables (Just imagine the I/O computation here and the stakes)

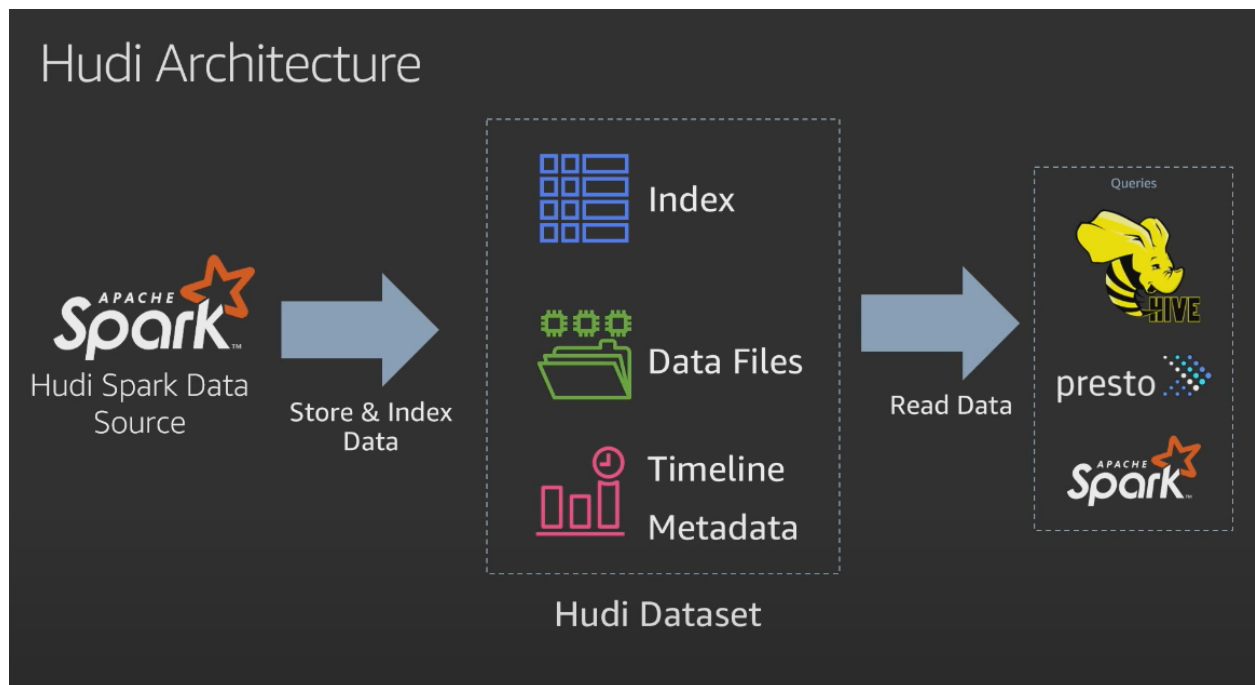
And as these are very critical datasets, so everyone wanted to apply these updates as quickly as possible.

And traditionally its very challenging as the spread of the partitioned data is high and data is spread across, and rewriting tons of data will compute lot of I/O etc.

## Hudi Overview:



## HUDI Architecture



Basically, there are three big moving pieces with HUDI datasets

- 1) Index- it will allow to match individual records to the specific data files that those records are contained in
- 2) Timeline metadata – All the specific action which are performed in specific Hudi dataset

Note: There is no persistent separate daemons in Hudi, All these things are stored with Data in s3 and other layers

Hive, presto, apache spark can be used to query the data

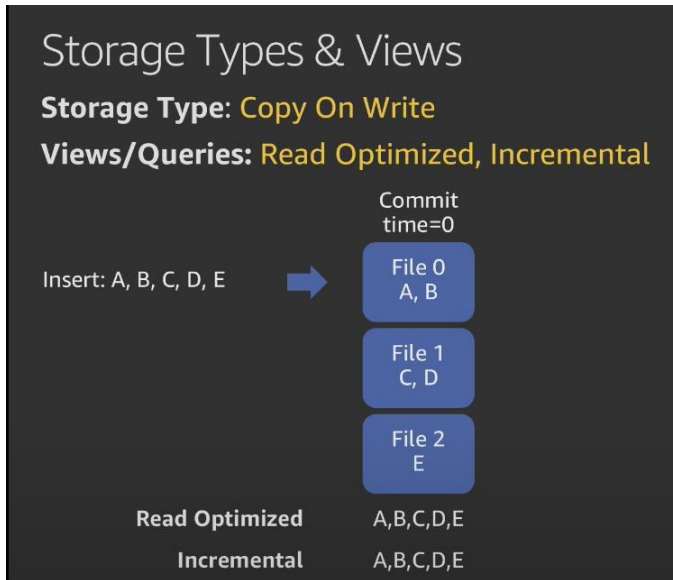
## 2. Storage types of Hudi:

- 1) Copy on Write (Read heavy)
- 2) Merge on Read (Write heavy like streaming data)

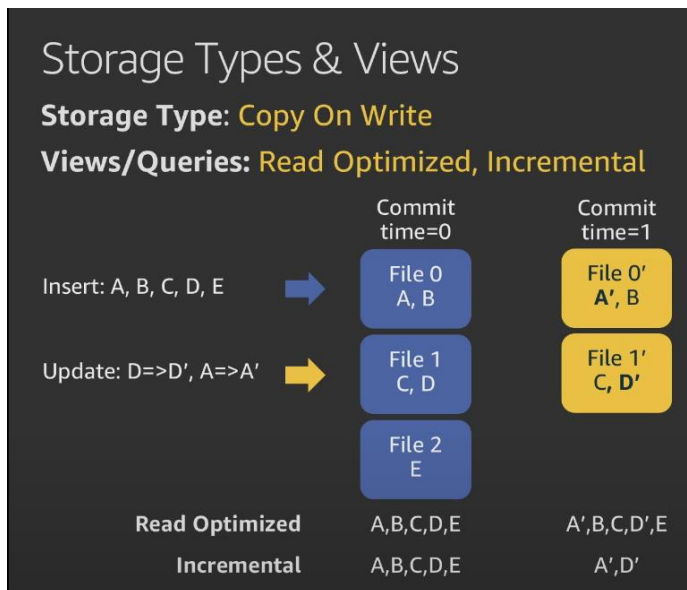
### 7.1 Copy on write

Step 1: Lets start with empty dataset A,B,C,D,E and internally they got distributed in 3 base files (Insert)

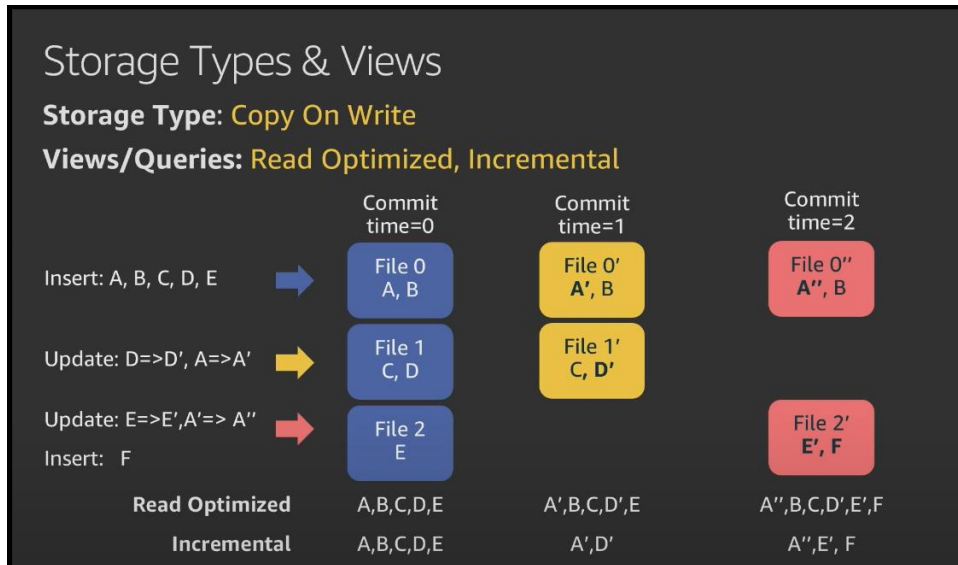
If we do select \* from whole dataset we will get all the data



Step2: Lets update the data of A to A' and D to D', it will generate another commit 1 (Update)



Step 3: Upsert, Here we are updating E to E' and A' to A'' and insert F (Here as usual Read optimized will give A'',B,C,D',E',F) and incremental will give A'',E',F



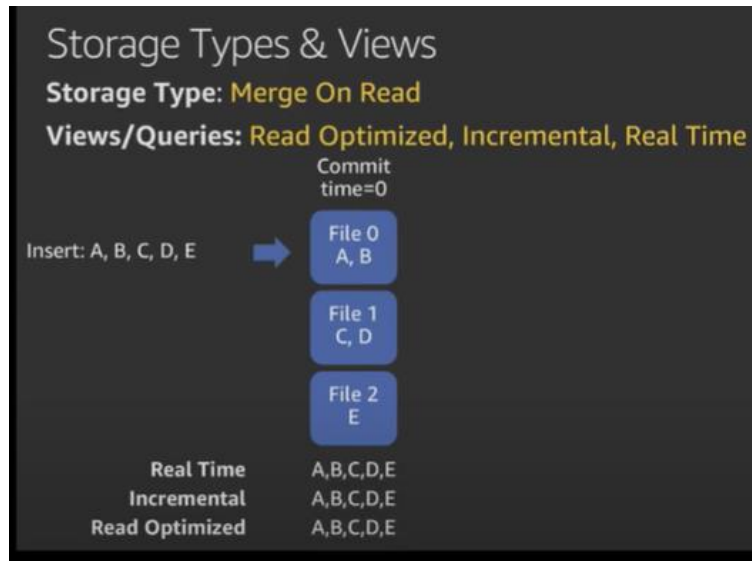
When do we use this?

- 1) If your current job is rewriting entire table/partition to deal with updates.  
(By using this only selective data will be updated and we will refrain using full table scan)
- 2) Your workload is fairly well understood and does not have sudden bursts (Because in this case lot of versioned files will be created)
- 3) You are already using parquet files for your tables
- 4) You want to keep things operationally simple (Batch job where updates are required)

## 2.2 Merge On Read:

Same example

- 1) Take the same data as above A,D,B,C,D , we have additional read optimized view extra in this solution , we will talk on this in below slides .

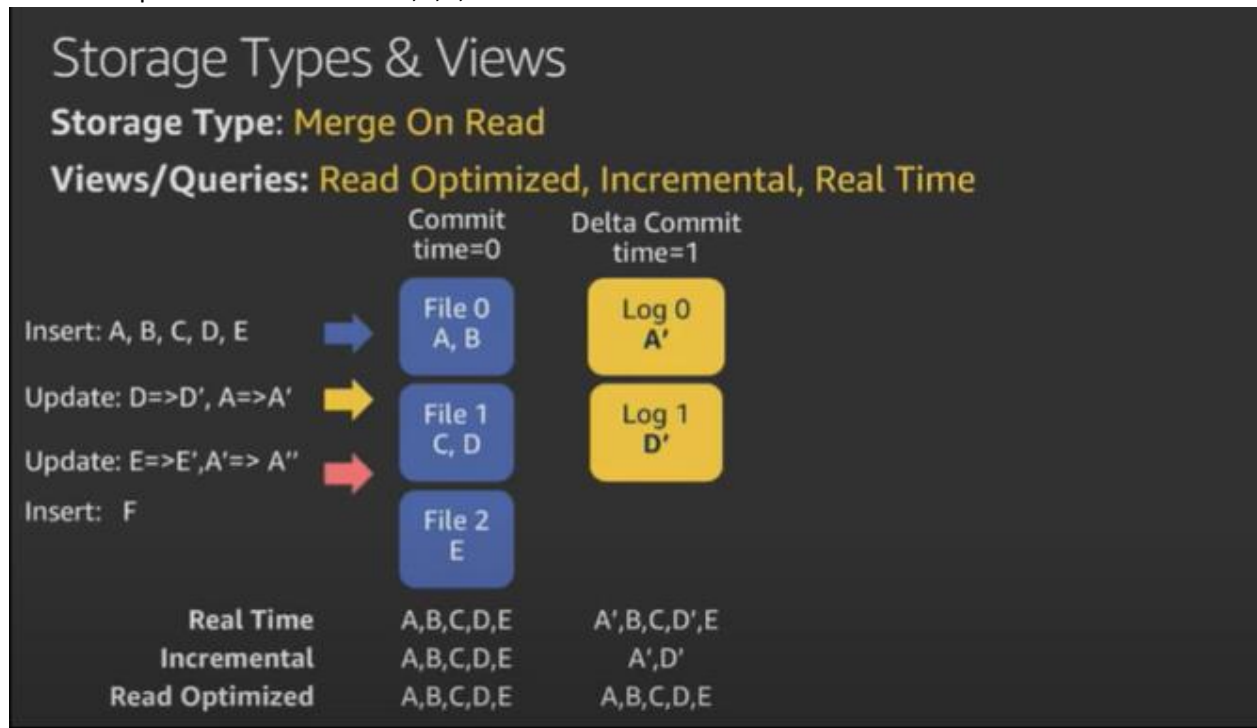


- 2) Lets Introduce the update  
 D to D' and A to A', in this merge on read we do a special commit called delta commit , so instead of entirely rewriting the we create the log of the updated value and updated value will be written in the logs , now when you query the data using real time view it basically merge the files the actual and the log files and give the latest value like A' and D' from the log with commit time 1 and rest from commit 0 the actual file

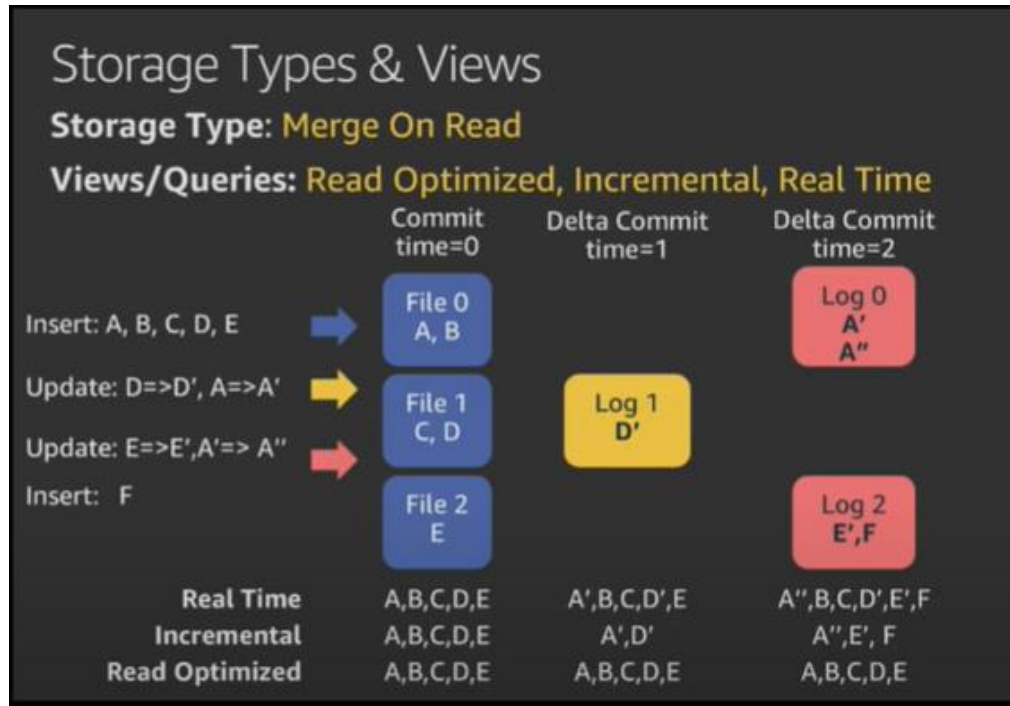
So real time view will be like A',B,C,D'

Incremental view will be A' and D'

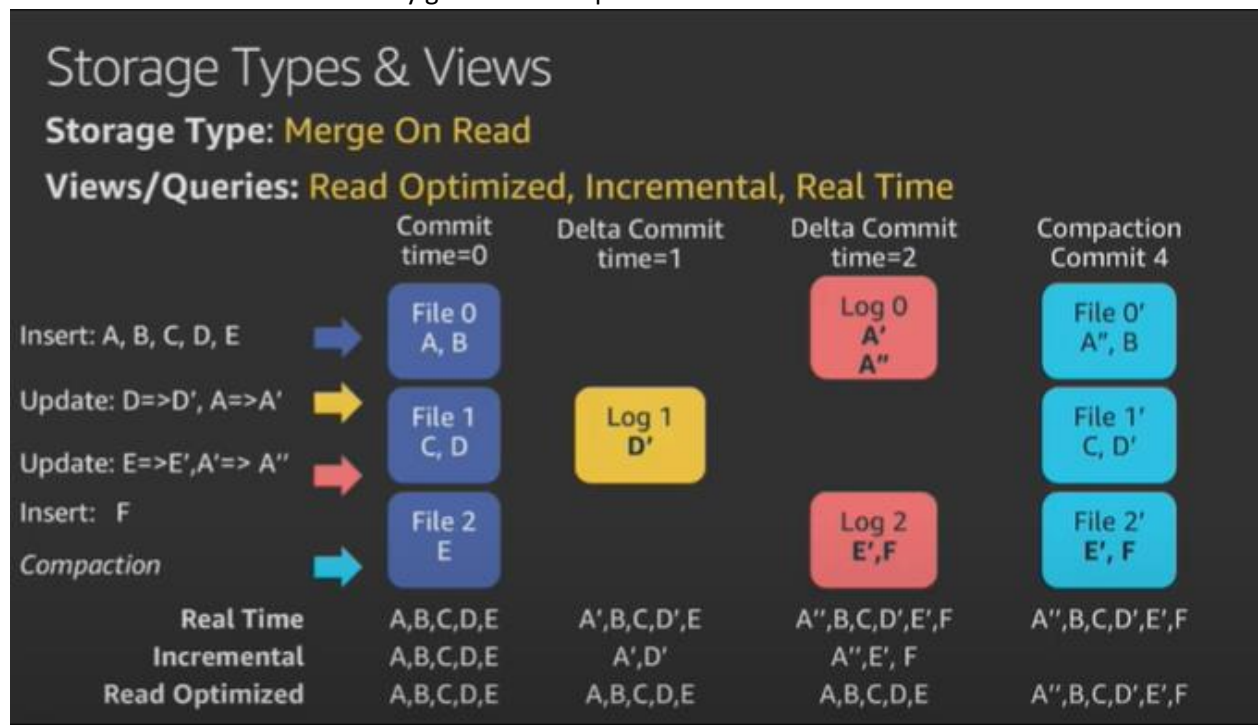
Read optimized view will be A,B,C,D it will be from the base files .



- 3) Lets do the upserts  $E \Rightarrow E'$ ,  $A' \Rightarrow A''$  and insert F  
 Similarly on the fly the logs and the actual files will get combined and below will be the result



- 4) Now when the compaction runs , it merges the logs and files and here the read optimized view and real time view finally gets catches up .





When to use this (Merge On Read) ?

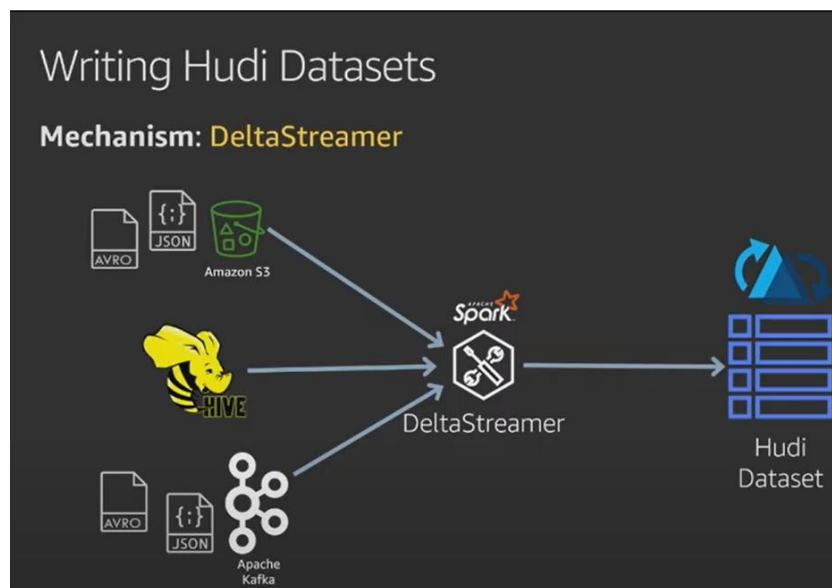
- 1) When you want to ingest the data for query or load the data as fast as possible
- 2) Your workload can have spikes or changes in pattern

**Example:** bulk updates to older transaction database cause updates to old partition in s3  
Any GDPR implementations

### 3. How to write HUDI datasets?

Below are the different option to write data in HUDI datasets

- 1) Delta Steamer:



In deltaStreamer allows you to watch any specific s3 location for avro or json files , and if any additional files will arrive on those buckets the deltastream will pick those up and parse them and update them on the Hudi dataset

Similar in kafka side too, it will be connected to any kafka topic and listen for the events in those kafka topics and stream those data avro or json files into hudi datasets .

There is also an hive incremental pull

It also allows you to connect to the existing hive table and the it can parse to hudi dataset

When the deltastream should be used ?

- ➔ You want to self-managed ingestion tool with automatic compactions, checkpoints

- ➔ You want a simple solution to ingest data from s3/kafka
- ➔ You need to support sql-based transformation of ingested data.

Using the spark data Source api , this will be really helpful if you have existing spark based ETL and instead of writing the parquet files directly , you can change some data format options and use this.

## Writing Hudi Datasets

```
inputDF.write()
    .format("org.apache.hudi")
    .options(opts)
    .option(TABLE_NAME, "sales")
    .mode(SaveMode.Append)
    .save("s3://bucket/sales");
```

When we can write the HUDI datasets ?

Mechanism: Spark data source

- ➔ You want to create derived tables from other data source
- ➔ It will be helpful when you want to read and write into both managed and non managed hudi datasets
- ➔ You want the spark structured streaming output sink

Querying Hudi datasets

1) Spark sql

How to use spark sql with hudi datasets

**Read optimized** spark.sql("select \* from sales\_rt")

**Real time** spark.sql("select \* from sales\_rt")

**Incremental** : import org.apache.hudi.DataSourceReadOptions.\_

```

Spark.read.format("org.apache.hudi")

.option("VIEW_TYPE_OPT_KEY","VIEW_TYPE_INCREMENTAL_OPT_VAL")

.option("BEGIN_INSTANTTIME,OPT_KEY",20220101064621)

// this is the time from where you need the incremental data

.load("s3://<buckets>")

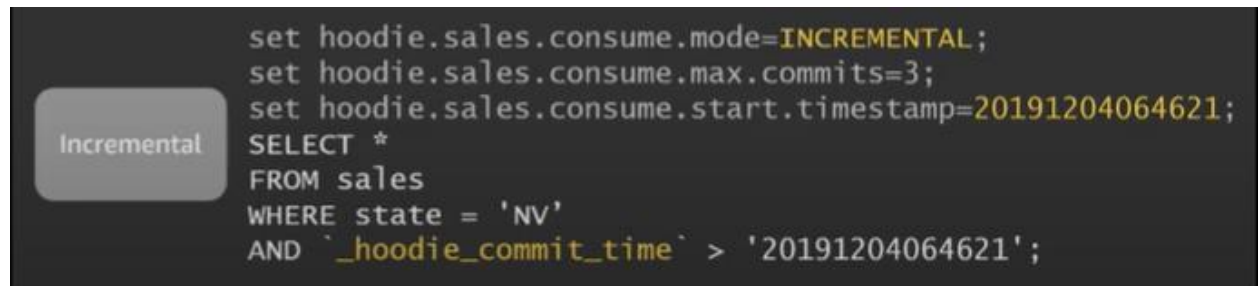
```

## 2) Hive

Read optimized : select \* from sales;

Real time : select \* from sales;

Incremental :



```

set hoodie.sales.consume.mode=INCREMENTAL;
set hoodie.sales.consume.max.commits=3;
set hoodie.sales.consume.start.timestamp=20191204064621;
SELECT *
FROM sales
WHERE state = 'NV'
AND `_hoodie_commit_time` > '20191204064621';

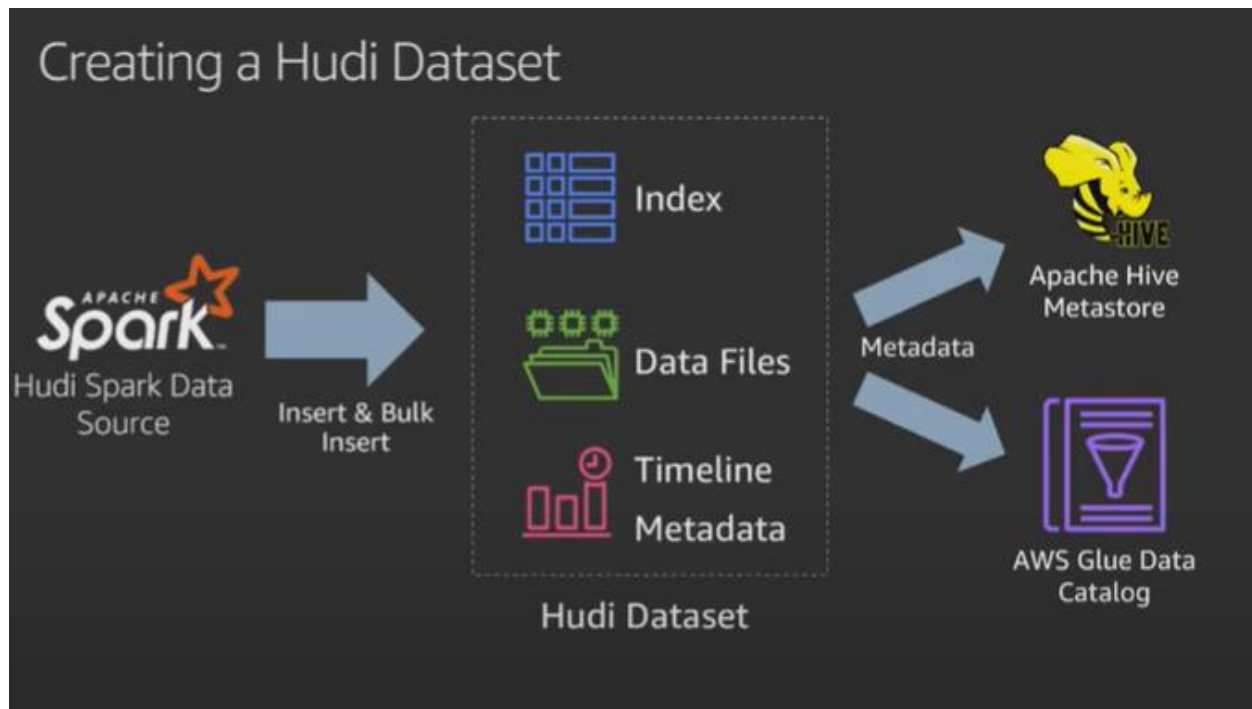
```

Here you need to give more inputs where you need to mention the time from when you need the data of sales from state NV

## 3) Presto:

It just supports read optimized view as of now

## 4. How data is being inserted USING hudi



Above picture depicts the 1<sup>st</sup> setup of the data incoming, all the indexing and metastore will be getting created and register the table in hive

Step1 : here we have taken the 10 data and parse it as json

Step2: reading those data using spark

Step3: now writing those data using hudi as below, certain parameters we need to write these data as below

You can refer the below diagram

# Inserting: Ingesting Transactionally



```
val inserts1 = convertToStringList(dataGen.generateInserts(10))
val df = spark.read.json(spark.sparkContext.parallelize(inserts1, 2))
df.write.format("org.apache.hudi").options(getQuickstartWriteConfigs).
  option(PRECOMBINE_FIELD_OPT_KEY, "ts").
  option(RECORDKEY_FIELD_OPT_KEY, "uuid").// ← ID for each trip
  option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
  option(TABLE_NAME, "trips_changelogs").// ← prefix for Hive tables
  option(OPERATION_OPT_KEY, "bulk_insert").// ← write operation to issue
  mode(Overwrite). // ← Use first time when creating table
  save("s3://bucket/hudi-trips-changelogs");
```

In the 3<sup>rd</sup> step

We need to tell the what is uuid in hudi , basically a primary keys ,Specify how to partition the data, Also specify the pre combine key (it will help to deduplicate within a given batch (multiple logs will be there so pre combine key will help to get the latest log value, kind of rank the deduplicate the records within the batch) ), then we issue an operation like insert or upserts

And then pat to save the files in the s3 location

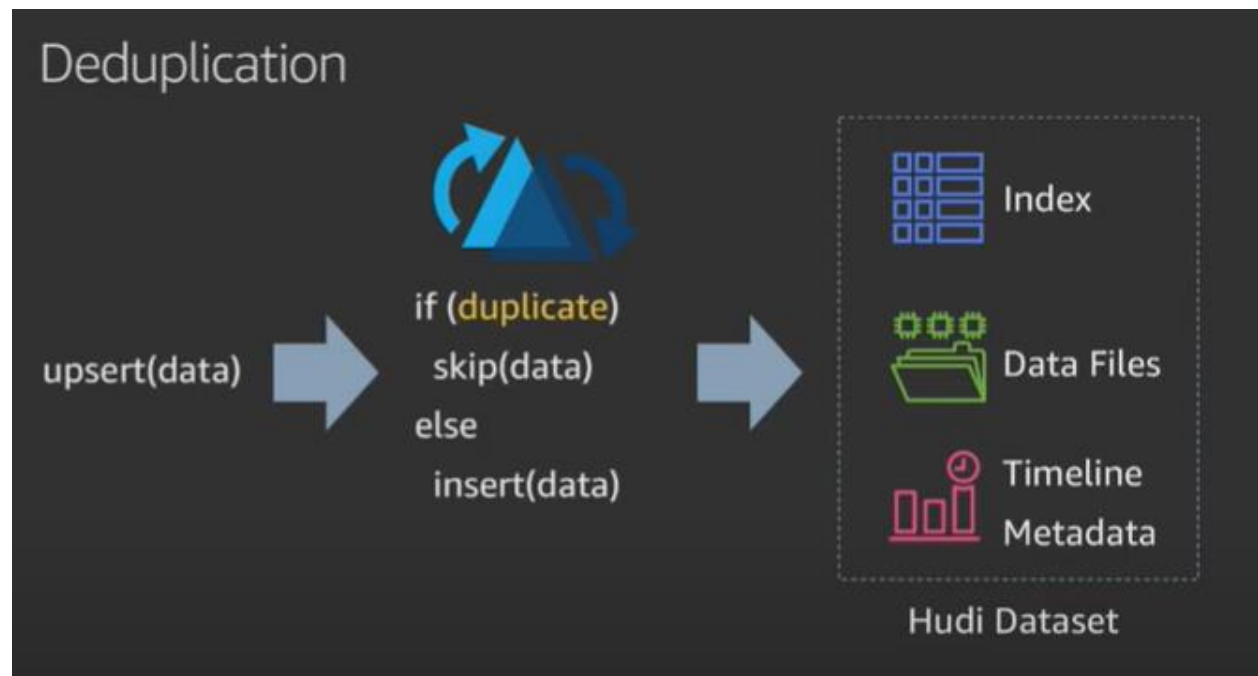
Step 4: After the step 3 is done then it will create the underlying files under the path

## Inserting: Ingesting Transactionally

```
scala> import sys.process._  
  
scala> "hdfs dfs ls s3://bucket/hudi-trips-changelogs/.hoodie" !  
  
total 48  
  
-rw-r--r--  1 hadoop  hadoop   1286 oct 16 13:19 20191016131911.clean  
-rw-r--r--  1 hadoop  hadoop  15718 oct 16 13:19 20191016131911.commit  
drwxr-xr-x  2 hadoop  hadoop    64 oct 16 13:19 archived  
-rw-r--r--  1 hadoop  hadoop    181 oct 16 13:19 hoodie.properties
```

Let's deduplicate the data:

Often while connecting to any kafka topic , we face an issue of duplication of the data and even in batch processing parts too , so below option can be introduce in your data frame to skip the dupes as simple as that 😊



## Inserting: Adding deduplication



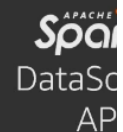
```
val inserts2 = convertToStringList(dataGen.generateInserts(10))
inserts2.addAll(inserts1)

val df = spark.read.json(spark.sparkContext.parallelize(inserts2, 2))
df.write.format("org.apache.hudi").options(getQuickstartWriteConfigs).
  option(PRECOMBINE_FIELD_OPT_KEY, "ts").// ← How to de-dupe within input
  option(RECORDKEY_FIELD_OPT_KEY, "uuid").
  option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
  option(TABLE_NAME, "trips_changelogs").option(OPERATION_OPT_KEY, "bulk_insert").
  option(INSERT_DROP_DUPS_OPT_KEY, "true").// ← De-dupe input against storage
  mode(Append).// ← Use Append always after first write
  save("s3://bucket/hudi-trips-changelogs");
```

Inserting : incremental Streams

Basically in the below pic it will return all the data after the first commit , hence you can see the commit hoodie time in the 2<sup>nd</sup> step

## Inserting: Incremental Streams



```
scala> val incViewDF = spark.read.format("org.apache.hudi").
  option(VIEW_TYPE_OPT_KEY, VIEW_TYPE_INCREMENTAL_OPT_VAL).
  option(BEGIN_INSTANTTIME_OPT_KEY, "20191016131911").// ← first commit time
  load("s3://bucket/hudi-trips-changelogs");

scala> incViewDF.count
res39: Long = 10

scala> incViewDF.select("_hoodie_commit_time").distinct.show
+-----+
| 20191016131935| // ← Records in second commit only
+-----+
```

## 5. Hudi Updating data

By updating the data , i.e if the result already exists then update(data) else insert(data)



For updating data just like bulk insert in insert part , we need to change it to upsert

Step 1 :

Use "upsert" in the OPERATION\_OPT\_KEY

## Updating: Change Data Capture/Apply



```
val datagen = new DataGenerator()
val inserts = convertToStringList(dataGen.generateInserts(10))
val insertdf = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
insertdf.write.format("org.apache.hudi").options(getQuickstartWriteConfigs()).
    option(PRECOMBINE_FIELD_OPT_KEY, "ts").
    option(RECORDKEY_FIELD_OPT_KEY, "uuid").
    option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
    option(TABLE_NAME, "trips_snapshot").
    option(OPERATION_OPT_KEY, "upsert"). // ← Default write operation
    mode(Overwrite). // ← Use first time when creating table
    save("s3://bucket/hudi-trips-changelogs");
```



Step 2:

From previous step of 10 inserts lets pick 5 of those and update for them

## Updating: Change Data Capture/Apply



```
val upserts = convertToStringList(dataGen.generateUpdates(5))
val upsertdf = spark.read.json(spark.sparkContext.parallelize(upserts, 2))
upsertdf.write.format("org.apache.hudi").options(getQuickstartWriteConfigs).
    option(PRECOMBINE_FIELD_OPT_KEY, "ts").
    option(RECORDKEY_FIELD_OPT_KEY, "uuid").
    option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
    option(TABLE_NAME, "trips_snapshot").
    option(OPERATION_OPT_KEY, "upsert").
    mode(Append).
    save("s3://bucket/hudi-trips-changelogs");
```

Step 3:

Now after the update the new 5 data then now if you group by on update\_time\_stamp then you can see the data

## Updating: Querying Snapshot



```
scala> val readDF = spark.read.format("org.apache.hudi").
    load("s3://bucket/hudi-trips-changelogs/**/*.json")

scala> readDF.select("uuid").distinct.count
res7: Long = 10

scala> readDF.groupBy("_hoodie_commit_time").count.show
+-----+-----+
| 20191016140332 | 5 |
| 20191016140320 | 5 |
+-----+-----+
```

## 6: Deletes

There are basically two types of deletes

- ➔ Soft deletes – Record key is retained, and other data is removed
- ➔ Hard deletes – Entire record delete or removed

## 6.1 How to implement the hard delete ?

Note : This needs to be elaborated more .

### Deleting: Enforcing Data Retention



```
val deleteDF = readDF.sample(0.4f)
deleteDF.cache

val deleteCount = deleteDF.count

deleteDF.write.format("org.apache.hudi").options(getQuickstartWriteConfigs).
  option(PRECOMBINE_FIELD_OPT_KEY, "ts").option(RECORDKEY_FIELD_OPT_KEY, "uuid").
  option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
  option(TABLE_NAME, "trips_snapshot").
  option(OPERATION_OPT_KEY, "upsert").
  option(PAYLOAD_CLASS_OPT_KEY, "org.apache.hudi.EmptyHoodieRecordPayload").
  mode(Append).
  save("s3://bucket/hudi-trips-changelogs");
```

Note: Hudi is available above 5.28 EMR and its already there

## 7. Some area where HUDI can be used

- ➔ Data privacy law compliance
- ➔ Consuming real time data streams and applying CDC logs
- ➔ Reinstating late arriving data
- ➔ Tracking data changes and rollback using CLI
- ➔ Simplifying file management on Amazon S3