FAQ

err

General

When is Hudi a useful for me or my organization

If you are looking to quickly ingest data onto HDFS or cloud storage, Hudi can provide you tools to help. Also, if you have ETL/hive/spark jobs which are slow/taking up a lot of resources, Hudi can potentially help by providing an incremental approach to reading and writing data.

As an organization, Hudi can help you build an efficient data lake, solving some of the most complex, low-level storage management problems, while putting data into hands of your data analysts, engineers and scientists much quicker.

What are some non-goals for Hudi

Hudi is not designed for any OLTP use-cases, where typically you are using existing NoSQL/RDBMS data stores. Hudi cannot replace your in-memory analytical database (at-least not yet!). Hudi support near-real time ingestion in the order of few minutes, trading off latency for efficient batching. If you truly desirable sub-minute processing delays, then stick with your favorite stream processing solution.

What is incremental processing? Why does Hudi docs/talks keep talking about it

Incremental processing was first introduced by Vinoth Chandar, in the O'reilly blog, that set off most of this effort. In purely technical terms, incremental processing merely refers to writing mini-batch programs in streaming processing style. Typical batch jobs consume **all input** and recompute **all output**, every few hours. Typical stream processing jobs consume some **new input** and recompute **new/changes to output**, continuously/every few seconds. While recomputing all output in batch fashion can be simpler, it's wasteful and resource expensive. Hudi brings ability to author the same batch pipelines in streaming fashion, run every few minutes.

While we can merely refer to this as stream processing, we call it *incremental processing*, to distinguish from purely stream processing pipelines built using Apache Flink, Apache Apex or Apache Kafka Streams.

What is the difference between copy-on-write (COW) vs merge-on-read (MOR) storage types

Copy On Write - This storage type enables clients to ingest data on columnar file formats, currently parquet. Any new data that is written to the Hudi dataset using COW storage type, will write new parquet files. Updating an existing set of rows will result in a rewrite of the entire parquet files that collectively contain the affected rows being updated. Hence, all writes to such datasets are limited by parquet writing performance, the larger the parquet file, the higher is the time taken to ingest the data.

Merge On Read - This storage type enables clients to ingest data quickly onto row based data format such as avro. Any new data that is written to the Hudi dataset using MOR table type, will write new log/delta files that internally store the data as avro encoded bytes. A compaction process (configured as inline or asynchronous) will convert log file format to columnar file format (parquet). Two different InputFormats expose 2 different views of this data, Read Optimized view exposes columnar parquet reading performance while Realtime View exposes columnar and/or log reading performance respectively. Updating an existing set of rows will result in either a) a companion log/delta file for an existing base parquet file generated from a previous compaction or b) an update written to a log/delta file in case no compaction ever happened for it. Hence, all writes to such datasets are limited by avro/log file writing performance, much faster than parquet. Although, there is a higher cost to pay to read log/delta files vs columnar (parquet) files.

More details can be found here and also Design And Architecture

How do I choose a storage type for my workload

A key goal of Hudi is to provide ***`upsert`*** functionality that is orders of magnitude faster than rewriting entire tables or partitions.

Choose Copy-on-write storage if :

- You are looking for a simple alternative, that replaces your existing parquet tables without any need for real-time data.
- Your current job is rewriting entire table/partition to deal with updates, while only a few files actually change in each partition.
- You are happy keeping things operationally simpler (no compaction etc), with the ingestion/write performance bound by the parquet file size and the number of such files affected/dirtied by updates
- Your workload is fairly well-understood and does not have sudden bursts of large amount of update or inserts to older partitions. COW absorbs all the merging cost on the writer side and thus these sudden changes can clog up your ingestion and interfere with meeting normal mode ingest latency targets.

Choose merge-on-read storage if :

- You want the data to be ingested as quickly & queryable as much as possible.
- Your workload can have sudden spikes/changes in pattern (e.g bulk updates to older transactions in upstream database causing lots of updates to old partitions on DFS). Asynchronous compaction helps amortize the write amplification caused by such scenarios, while normal ingestion keeps up with incoming stream of changes.

Immaterial of what you choose, Hudi provides

- Snapshot isolation and atomic write of batch of records
- Incremental pulls
- Ability to de-duplicate data

Find more here.

Is Hudi an analytical database

A typical database has a bunch of long running storage servers always running, which takes writes and reads. Hudi's architecture is very different and for good reasons. It's highly decoupled where writes and queries/reads can be scaled independently to be able to handle the scale challenges. So, it may not always seems like a database.

Nonetheless, Hudi is designed very much like a database and provides similar functionality (upserts, change capture) and semantics (transactional writes, snapshot isolated reads).

How do I model the data stored in Hudi

When writing data into Hudi, you model the records like how you would on a key-value store - specify a key field (unique for a single partition/across dataset), a partition field (denotes partition to place key into) and preCombine/combine logic that specifies how to handle duplicates in a batch of records written. This model enables Hudi to enforce primary key constraints like you would get on a database table. See here for an example.

When querying/reading data, Hudi just presents itself as a json-like hierarchical table, everyone is used to querying using Hive/Spark/Presto over Parquet /Json/Avro.

Does Hudi support cloud storage/object stores

Yes. Generally speaking, Hudi is able to provide its functionality on any Hadoop FileSystem implementation and thus can read and write datasets on Cloud stores (Amazon S3 or Microsoft Azure or Google Cloud Storage). Over time, Hudi has also incorporated specific design aspects that make building Hudi datasets on the cloud easy, such as consistency checks for s3, Zero moves/renames involved for data files.

What versions of Hive/Spark/Hadoop are support by Hudi

As of September 2019, Hudi can support Spark 2.1+, Hive 2.x, Hadoop 2.7+ (not Hadoop 3)

How does Hudi actually store data inside a dataset

At a high level, Hudi is based on MVCC design that writes data to versioned parquet/base files and log files that contain changes to the base file. All the files are stored under a partitioning scheme for the dataset, which closely resembles how Apache Hive tables are laid out on DFS. Please refer here for more details.

Using Hudi

What are some ways to write a Hudi dataset

Typically, you obtain a set of partial updates/inserts from your source and issue write operations against a Hudi dataset. If you ingesting data from any of the standard sources like Kafka, or tailing DFS, the delta streamer tool is invaluable and provides an easy, self-managed solution to getting data written into Hudi. You can also write your own code to capture data from a custom source using the Spark datasource API and use a Hudi datasource to write into Hudi.

How is a Hudi job deployed

The nice thing about Hudi writing is that it just runs like any other spark job would on a YARN/Mesos or even a K8S cluster. So you could simply use the Spark UI to get visibility into write operations.

How can I now query the Hudi dataset I just wrote

Unless Hive sync is enabled, the dataset written by Hudi using one of the methods above can simply be queries via the Spark datasource like any other source.

```
val hoodieROView = spark.read.format("org.apache.hudi").load(basePath + "/path/to/partitions/*")
val hoodieIncViewDF = spark.read().format("org.apache.hudi")
     .option(DataSourceReadOptions.VIEW_TYPE_OPT_KEY(), DataSourceReadOptions.VIEW_TYPE_INCREMENTAL_OPT_VAL())
     .option(DataSourceReadOptions.BEGIN_INSTANTTIME_OPT_KEY(), <beginInstantTime>)
     .load(basePath);
```

Limitations

Note that currently the reading realtime view natively out of the Spark datasource is not supported. Please use the Hive path below

if Hive Sync is enabled in the deltastreamer tool or datasource, the dataset is available in Hive as a couple of tables, that can now be read using HiveQL, Presto or SparkSQL. See here for more.

How does Hudi handle duplicate record keys in an input

When issuing an `upsert` operation on a dataset and the batch of records provided contains multiple entries for a given key, then all of them are reduced into a single final value by repeatedly calling payload class's `preCombine()` method . By default, we pick the record with the greatest value (determined by calling `.compareTo()`) giving latest-write-wins style semantics. This FAQ entry shows the interface for HoodieRecordPayload if you are interested.

For an `insert` or `bulk_insert` operation, no such pre-combining is performed. Thus, if your input contains duplicates, the dataset would also contain duplicates. If you don't want duplicate records either issue an `upsert` or consider specifying option to de-duplicate input in either datasource or deltastreamer.

Can I implement my own logic for how input records are merged with record on storage

Here is the payload interface that is used in Hudi to represent any hudi record.

```
public interface HoodieRecordPayload<T extends HoodieRecordPayload> extends Serializable {
 /**
   * When more than one HoodieRecord have the same HoodieKey, this function combines them before attempting to
insert/upsert by taking in a property map.
   * Implementation can leverage the property to decide their business logic to do preCombine.
   * @param another instance of another {@link HoodieRecordPayload} to be combined with.
   * @param properties Payload related properties. For example pass the ordering field(s) name to extract from
value in storage.
   * @return the combined value
   */
  default T preCombine(T another, Properties properties);

/**
   * This methods lets you write custom merging/combining logic to produce new values as a function of current
value on storage and whats contained
   * in this object. Implementations can leverage properties if required.
   * <p>
   * eg:
   * 1) You are updating counters, you may want to add counts to currentValue and write back updated counts
   * 2) You may be reading DB redo logs, and merge them with current image for a database row on storage
   * </p>
   *
   * @param currentValue Current value in storage, to merge/combine this payload with
   * @param schema Schema used for record
   * @param properties Payload related properties. For example pass the ordering field(s) name to extract from
value in storage.
   * @return new combined/merged value to be written back to storage. EMPTY to skip writing this record.
   */
  default Option<IndexedRecord> combineAndGetUpdateValue(IndexedRecord currentValue, Schema schema, Properties
properties) throws IOException;

/**
   * Generates an avro record out of the given HoodieRecordPayload, to be written out to storage. Called when
writing a new value for the given
   * HoodieKey, wherein there is no existing record in storage to be combined against. (i.e insert) Return
EMPTY to skip writing this record.
   * Implementations can leverage properties if required.
   * @param schema Schema used for record
   * @param properties Payload related properties. For example pass the ordering field(s) name to extract from
value in storage.
   * @return the {@link IndexedRecord} to be inserted.
   */
  @PublicAPIMethod(maturity = ApiMaturityLevel.STABLE)
  default Option<IndexedRecord> getInsertValue(Schema schema, Properties properties) throws IOException;

/**
   * This method can be used to extract some metadata from HoodieRecordPayload. The metadata is passed to
{@code WriteStatus.markSuccess()} and
   * {@code WriteStatus.markFailure()} in order to compute some aggregate metrics using the metadata in the
context of a write success or failure.
   * @return the metadata in the form of Map<String, String> if any.
   */
  @PublicAPIMethod(maturity = ApiMaturityLevel.STABLE)
  default Option<Map<String, String>> getMetadata() {
    return Option.empty();
  }

}
```

As you could see, (`combineAndGetUpdateValue(), getInsertValue()`) that control how the record on storage is combined with the incoming update/insert to generate the final value to be written back to storage. preCombine() is used to merge records within the same incoming batch.

How do I delete records in the dataset using Hudi

GDPR has made deletes a must-have tool in everyone's data management toolbox. Hudi supports both soft and hard deletes. For details on how to actually perform them, see here.

Does deleted records appear in Hudi's incremental query results ?

Soft Deletes (unlike hard deletes) do appear in the incremental pull query results. So, if you need a mechanism to propagate deletes to downstream tables, you can use Soft deletes.

How do I migrate my data to Hudi

Hudi provides built in support for rewriting your entire dataset into Hudi one-time using the `HDFSParquetImporter` tool available from the hudi-cli . You could also do this via a simple read and write of the dataset using the Spark datasource APIs. Once migrated, writes can be performed using normal means discussed here. This topic is discussed in detail here, including ways to doing partial migrations.

How can I pass hudi configurations to my spark job

Hudi configuration options covering the datasource and low level Hudi write client (which both deltastreamer & datasource internally call) are here. Invoking `--help` on any tool such as DeltaStreamer would print all the usage options. A lot of the options that control upsert, file sizing behavior are defined at the write client level and below is how we pass them to different options available for writing data.

1. For Spark DataSource, you can use the "options" API of DataFrameWriter to pass in these configs.

```
inputDF.write().format("org.apache.hudi")
  .options(clientOpts) // any of the Hudi client opts can be passed in as well
  .option(DataSourceWriteOptions.RECORDKEY_FIELD_OPT_KEY(), "_row_key")
  ...
```

2. When using `HoodieWriteClient` directly, you can simply construct HoodieWriteConfig object with the configs in the link you mentioned.

3. When using HoodieDeltaStreamer tool to ingest, you can set the configs in properties file and pass the file as the cmdline argument "`--props`"

How to create Hive style partition folder structure?

By default Hudi creates the partition folders with just the partition values, but if would like to create partition folders similar to the way Hive will generate the structure, with paths that contain key value pairs, like country=us/… or datestr=2021-04-20. This is Hive style (or format) partitioning. The paths include both the names of the partition keys and the values that each path represents.

To enable hive style partitioning, you need to add this hoodie config when you write your data:

```
hoodie.datasource.write.hive_style_partitioning: true
```

How do I pass hudi configurations to my beeline Hive queries?

If Hudi's input format is not picked the returned results may be incorrect. To ensure correct inputformat is picked, please use `*org.apache.hadoop.hive.ql.io.HiveInputFormat*` or `*org.apache.hudi.hadoop.hive.HoodieCombineHiveInputFormat*` for `*hive.input.format*` config. This can be set like shown below:

```
set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat
```

or

```
set hive.input.format=org.apache.hudi.hadoop.hive.HoodieCombineHiveInputFormat
```

Can I register my Hudi dataset with Apache Hive metastore

Yes. This can be performed either via the standalone Hive Sync tool or using options in deltastreamer tool or datasource.

How does the Hudi indexing work & what are its benefits?

The indexing component is a key part of the Hudi writing and it maps a given recordKey to a fileGroup inside Hudi consistently. This enables faster identification of the file groups that are affected/dirtied by a given write operation.

Hudi supports a few options for indexing as below

- `HoodieBloomIndex (default)` : Uses a bloom filter and ranges information placed in the footer of parquet/base files (and soon log files as well)

- *HoodieGlobalBloomIndex* : The default indexing only enforces uniqueness of a key inside a single partition i.e the user is expected to know the partition under which a given record key is stored. This helps the indexing scale very well for even very large datasets. However, in some cases, it might be necessary instead to do the de-duping/enforce uniqueness across all partitions and the global bloom index does exactly that. If this is used, incoming records are compared to files across the entire dataset and ensure a recordKey is only present in one partition.
- HBaseIndex : Apache HBase is a key value store, typically found in close proximity to HDFS 🙂. You can also store the index inside HBase, which could be handy if you are already operating HBase.

You can implement your own index if you'd like, by subclassing the *HoodieIndex* class and configuring the index class name in configs.

What does the Hudi cleaner do

The Hudi cleaner process often runs right after a commit and deltacommit and goes about deleting old files that are no longer needed. If you are using the incremental pull feature, then ensure you configure the cleaner to retain sufficient amount of last commits to rewind. Another consideration is to provide sufficient time for your long running jobs to finish running. Otherwise, the cleaner could delete a file that is being or could be read by the job and will fail the job. Typically, the default configuration of 10 allows for an ingestion running every 30 mins to retain up-to 5 hours worth of data. If you run ingestion more frequently or if you want to give more running time for a query, consider increasing the value for the config : hoodie.cleaner.commits.retained

What's Hudi's schema evolution story

Hudi uses Avro as the internal canonical representation for records, primarily due to its nice schema compatibility & evolution properties. This is a key aspect of having reliability in your ingestion or ETL pipelines. As long as the schema passed to Hudi (either explicitly in DeltaStreamer schema provider configs or implicitly by Spark Datasource's Dataset schemas) is backwards compatible (e.g no field deletes, only appending new fields to schema), Hudi will seamlessly handle read/write of old and new data and also keep the Hive schema up-to date.

How do I run compaction for a MOR dataset

Simplest way to run compaction on MOR dataset is to run the compaction inline, at the cost of spending more time ingesting; This could be particularly useful, in common cases where you have small amount of late arriving data trickling into older partitions. In such a scenario, you may want to just aggressively compact the last N partitions while waiting for enough logs to accumulate for older partitions. The net effect is that you have converted most of the recent data, that is more likely to be queried to optimized columnar format.

That said, for obvious reasons of not blocking ingesting for compaction, you may want to run it asynchronously as well. This can be done either via a separate compaction job that is scheduled by your workflow scheduler/notebook independently. If you are using delta streamer, then you can run in continuous mode where the ingestion and compaction are both managed concurrently in a single spark run time.

What performance/ingest latency can I expect for Hudi writing

The speed at which you can write into Hudi depends on the write operation and some trade-offs you make along the way like file sizing. Just like how databases incur overhead over direct/raw file I/O on disks, Hudi operations may have overhead from supporting database like features compared to reading/writing raw DFS files. That said, Hudi implements advanced techniques from database literature to keep these minimal. User is encouraged to have this perspective when trying to reason about Hudi performance. As the saying goes : there is no free lunch 🙂 (not yet atleast)

Storage Type	Type of workload	Performance	Tips
copy on write	bulk_insert	Should match vanilla spark writing + an additional sort to properly size files	properly size bulk insert parallelism to get right number of files. use *insert* if you want this auto tuned
copy on write	insert	Similar to bulk insert, except the file sizes are auto tuned requiring input to be cached into memory and custom partitioned.	Performance would be bound by how parallel you can write the ingested data. Tune this limit up, if you see that writes are happening from only a few executors.
copy on write	upsert/ de-duplicate & insert	Both of these would involve index lookup. Compared to naively using Spark (or similar framework)'s JOIN to identify the affected records, Hudi indexing is often 7-10x faster as long as you have ordered keys (discussed below) or <50% updates. Compared to naively overwriting entire partitions, Hudi write can be several magnitudes faster depending on how many files in a given partition is actually updated. For e.g, if a partition has 1000 files out of which only 100 is dirtied every ingestion run, then Hudi would only read/merge a total of 100 files and thus 10x faster than naively rewriting entire partition.	Ultimately performance would be bound by how quickly we can read and write a parquet file and that depends on the size of the parquet file, configured here . Also be sure to properly tune your bloom filters. **HUDI-56** - Getting issue details... STATUS will auto-tune this.

merge on read	bulk insert	Currently new data only goes to parquet files and thus performance here should be similar to copy_on_write bulk insert. This has the nice side-effect of getting data into parquet directly for query performance. **HUDI-86** - Getting issue details... STATUS will add support for logging inserts directly and this up drastically.	
merge on read	insert	Similar to above.	
merge on read	upsert/ de-duplicate & insert	Indexing performance would remain the same as copy-on-write, while ingest latency for updates (costliest I/O operation in copy_on_write) are sent to log files and thus with asynchronous compaction provides very very good ingest performance with low write amplification.	

Like with many typical system that manage time-series data, Hudi performs much better if your keys have a timestamp prefix or monotonically increasing /decreasing. You can almost always achieve this. Even if you have UUID keys, you can follow tricks like this to get keys that are ordered. See also Tuning Guide for more tips on JVM and other configurations.

What performance can I expect for Hudi reading/queries

- For ReadOptimized views, you can expect the same best in-class columnar query performance as a standard parquet table in Hive/Spark/Presto
- For incremental views, you can expect speed up relative to how much data usually changes in a given time window and how much time your entire scan takes. For e.g, if only 100 files changed in the last hour in a partition of 1000 files, then you can expect a speed of 10x using incremental pull in Hudi compared to full scanning the partition to find out new data.
- For real time views, you can expect performance similar to the same avro backed table in Hive/Spark/Presto

How do I to avoid creating tons of small files

A key design decision in Hudi was to avoid creating small files and always write properly sized files.

There are 2 ways to avoid creating tons of small files in Hudi and both of them have different trade-offs:

1. **Auto Size small files during ingestion:** This solution trades ingest/writing time to keep queries always efficient. Common approaches to writing very small files and then later stitching them together only solve for system scalability issues posed by small files and also let queries slow down by exposing small files to them anyway.

Hudi has the ability to maintain a configured target file size, when performing **upsert/insert** operations. (Note: **bulk_insert** operation does not provide this functionality and is designed as a simpler replacement for normal `spark.write.parquet`)

For **copy-on-write**, this is as simple as configuring the maximum size for a base/parquet file and the soft limit below which a file should be considered a small file. For the initial bootstrap to Hudi table, tuning record size estimate is also important to ensure sufficient records are bin-packed in a parquet file. For subsequent writes, Hudi automatically uses average record size based on previous commit. Hudi will try to add enough records to a small file at write time to get it to the configured maximum limit. For e.g , with `compactionSmallFileSize=100MB` and `limitFileSize=120MB`, Hudi will pick all files < 100MB and try to get them upto 120MB.

For **merge-on-read**, there are few more configs to set. MergeOnRead works differently for different INDEX choices.

1. Indexes with **canIndexLogFiles = true** : Inserts of new data go directly to log files. In this case, you can configure the maximum log size and a fac tor that denotes reduction in size when data moves from avro to parquet files.
2. Indexes with **canIndexLogFiles = false** : Inserts of new data go only to parquet files. In this case, the same configurations as above for the COPY_ON_WRITE case applies.

NOTE : In either case, small files will be auto sized only if there is no PENDING compaction or associated log file for that particular file slice. For example, for case 1: If you had a log file and a compaction C1 was scheduled to convert that log file to parquet, no more inserts can go into that log file. For case 2: If you had a parquet file and an update ended up creating an associated delta log file, no more inserts can go into that parquet file. Only after the compaction has been performed and there are NO log files associated with the base parquet file, can new inserts be sent to auto size that parquet file.

2. **Clustering** : This is a feature in Hudi to group small files into larger ones either synchronously or asynchronously. Since first solution of auto-sizing small files has a tradeoff on ingestion speed (since the small files are sized during ingestion), if your use-case is very sensitive to ingestion latency where you don't want to compromise on ingestion speed which may end up creating a lot of small files, clustering comes to the rescue. Clustering can be scheduled through the ingestion job and an asynchronus job can stitch small files together in the background to generate larger files. NOTE that during this, ingestion can continue to run concurrently.

Please note that Hudi always creates immutable files on disk. To be able to do auto-sizing or clustering, Hudi will always create a newer version of the smaller file, resulting in 2 versions of the same file. The cleaner service will later kick in and delte the older version small file and keep the latest one.

Why does Hudi retain at-least one previous commit even after setting hoodie.cleaner.commits.retained': 1 ?

Hudi runs cleaner to remove old file versions as part of writing data either in inline or in asynchronous mode (0.6.0 onwards). Hudi Cleaner retains at-least one previous commit when cleaning old file versions. This is to prevent the case when concurrently running queries which are reading the latest file versions suddenly see those files getting deleted by cleaner because a new file version got added . In other words, retaining at-least one previous commit is needed for ensuring snapshot isolation for readers.

How do I use DeltaStreamer or Spark DataSource API to write to a Non-partitioned Hudi dataset ?

Hudi supports writing to non-partitioned datasets. For writing to a non-partitioned Hudi dataset and performing hive table syncing, you need to set the below configurations in the properties passed:

```
hoodie.datasource.write.keygenerator.class=org.apache.hudi.keygen.NonpartitionedKeyGenerator
hoodie.datasource.hive_sync.partition_extractor_class=org.apache.hudi.hive.NonPartitionedExtractor
```

Why do we have to set 2 different ways of configuring Spark to work with Hudi?

Non-Hive engines tend to do their own listing of DFS to query datasets. For e.g Spark starts reading the paths direct from the file system (HDFS or S3).

From Spark the calls would be as below:
- org.apache.spark.rdd.NewHadoopRDD.getPartitions
- org.apache.parquet.hadoop.ParquetInputFormat.getSplits
- org.apache.hadoop.mapreduce.lib.input.FileInputFormat.getSplits

Without understanding of Hudi's file layout, engines would just plainly reading all the parquet files and displaying the data within them, with massive amounts of duplicates in the result.

At a high level, there are two ways of configuring a query engine to properly read Hudi datasets

A) Making them invoke methods in `HoodieParquetInputFormat#getSplits` and `HoodieParquetInputFormat#getRecordReader`

- Hive does this natively, since the InputFormat is the abstraction in Hive to plugin new table formats. HoodieParquetInputFormat extends MapredParquetInputFormat which is nothing but a input format for hive and we register Hudi tables to Hive metastore backed by these input formats
- Presto also falls back to calling the input format when it sees a `UseFileSplitsFromInputFormat` annotation, to just obtain splits, but then goes on to use its own optimized/vectorized parquet reader for queries on Copy-on-Write tables
- Spark can be forced into falling back to the HoodieParquetInputFormat class, using `--conf spark.sql.hive.convertMetastoreParquet=false` `

B) Making the engine invoke a path filter or other means to directly call Hudi classes to filter the files on DFS and pick out the latest file slice

- Even though we can force Spark to fallback to using the InputFormat class, we could lose ability to use Spark's optimized parquet reader path by doing so.
- To keep benefits of native parquet read performance, we set the `*HoodieROTablePathFilter*` as a path filter, explicitly set this in the Spark Hadoop Configuration.There is logic in the file: to ensure that folders (paths) or files for Hoodie related files always ensures that latest file slice is selected. This filters out duplicate entries and shows latest entries for each record.

I have an existing dataset and want to evaluate Hudi using portion of that data ?

You can bulk import portion of that data to a new hudi table. For example, if you want to try on a month of data -

```
spark.read.parquet("your_data_set/path/to/month")
     .write.format("org.apache.hudi")
     .option("hoodie.datasource.write.operation", "bulk_insert")
     .option("hoodie.datasource.write.storage.type", "storage_type") // COPY_ON_WRITE or MERGE_ON_READ
     .option(RECORDKEY_FIELD_OPT_KEY, "<your key>").
     .option(PARTITIONPATH_FIELD_OPT_KEY, "<your_partition>")
     ...
     .mode(SaveMode.Append)
     .save(basePath);
```

Once you have the initial copy, you can simply run upsert operations on this by selecting some sample of data every round

```
spark.read.parquet("your_data_set/path/to/month").limit(n) // Limit n records
     .write.format("org.apache.hudi")
     .option("hoodie.datasource.write.operation", "upsert")
     .option(RECORDKEY_FIELD_OPT_KEY, "<your key>").
     .option(PARTITIONPATH_FIELD_OPT_KEY, "<your_partition>")
     ...
     .mode(SaveMode.Append)
     .save(basePath);
```

For merge on read table, you may want to also try scheduling and running compaction jobs. You can run compaction directly using spark submit on org.apache.hudi.utilities.HoodieCompactor or by using HUDI CLI

If I keep my file versions at 1, with this configuration will i be able to do a roll back (to the last commit) when write fail?

Yes, Commits happen before cleaning. Any failed commits will not cause any side-effects and Hudi will guarantee snapshot isolation.

Does AWS GLUE support Hudi ?

AWS Glue jobs can write, read and update Glue Data Catalog for hudi tables. In order to successfully integrate with Glue Data Catalog, you need to subscribe to one of the AWS provided Glue connectors named "AWS Glue Connector for Apache Hudi". Glue job needs to have "Use Glue data catalog as the Hive metastore" option ticked. Detailed steps with a sample scripts is available on this article provided by AWS - https://aws.amazon.com/blogs/big-data/writing-to-apache-hudi-tables-using-aws-glue-connector/.

In case if your using either notebooks or Zeppelin through Glue dev-endpoints, your script might not be able to integrate with Glue DataCatalog when writing to hudi tables.

Why partition fields are also stored in parquet files in addition to the partition path ?

Hudi supports customizable partition values which could be a derived value of another field. Also, storing the partition value only as part of the field results in losing type information when queried by various query engines.

I am seeing lot of archive files. How do I control the number of archive commit files generated?

Please note that in cloud stores that do not support log append operations, Hudi is forced to create new archive files to archive old metadata operations. You can increase hoodie.commits.archival.batch moving forward to increase the number of commits archived per archive file. In addition, you can increase the difference between the 2 watermark configurations : hoodie.keep.max.commits (default : 30) and hoodie.keep.min.commits (default : 20). This way, you can reduce the number of archive files created and also at the same time increase the number of metadata archived per archive file. Note that post 0.7.0 release where we are adding consolidated Hudi metadata (RFC-15), the follow up work would involve re-organizing archival metadata so that we can do periodic compactions to control file-sizing of these archive files.

How do I configure Bloom filter (when Bloom/Global_Bloom index is used)?

Bloom filters are used in bloom indexes to look up the location of record keys in write path. Bloom filters are used only when the index type is chosen as “BLOOM” or “GLOBAL_BLOOM”. Hudi has few config knobs that users can use to tune their bloom filters.

On a high level, hudi has two types of blooms: Simple and Dynamic.

Simple, as the name suggests, is simple. Size is statically allocated based on few configs.

“hoodie.bloom.index.filter.type”: SIMPLE

“hoodie.index.bloom.num_entries” refers to the total number of entries per bloom filter, which refers to one file slice. Default value is 60000.

“hoodie.index.bloom.fpp” refers to the false positive probability with the bloom filter. Default value: 1*10^-9.

Size of the bloom filter depends on these two values. This is statically allocated and here is the formula that determines the size of bloom. Until the total number of entries added to the bloom is within the configured *“hoodie.index.bloom.num_entries”* value, the fpp will be honored. i.e. with default values of 60k and 1*10^-9, bloom filter serialized size = 430kb. But if more entries are added, then the false positive probability will not be honored. Chances that more false positives could be returned if you add more number of entries than the configured value. So, users are expected to set the right values for both num_entries and fpp.

Hudi suggests to have roughly 100 to 120 mb sized files for better query performance. So, based on the record size, one could determine how many records could fit into one data file.

Lets say your data file max size is 128Mb and default avg record size is 1024 bytes. Hence, roughly this translates to 130k entries per data file. For this config, you should set num_entries to ~130k.

Dynamic bloom filter:

"hoodie.bloom.index.filter.type" : DYNAMIC

This is an advanced version of the bloom filter which grows dynamically as the number of entries grows. So, users are expected to set two values wrt num_entires. *"hoodie.index.bloom.num_entries"* will determine the starting size of the bloom. *"hoodie.bloom.index.filter.dynamic.max.entries"* will determine the max size to which the bloom can grow upto. And fpp needs to be set similar to "Simple" bloom filter. Bloom size will be allotted based on the first config *"hoodie.index.bloom.num_entries"*. Once the number of entries reaches this value, bloom will dynamically grow its size to 2X. This will go on until the size reaches a max of *"hoodie.bloom.index.filter.dynamic.max.entries"* value. Until the size reaches this max value, fpp will be honored. If the entries added exceeds the max value, then the fpp may not be honored.

How to tune shuffle parallelism of Hudi jobs ?

First, let's understand what the term parallelism means in the context of Hudi jobs. For any Hudi job using Spark, parallelism equals to the number of spark partitions that should be generated for a particular stage in the DAG. To understand more about spark partitions, read this article. In spark, each spark partition is mapped to a spark task that can be executed on an executor. Typically, for a spark application the following hierarchy holds true

(Spark Application N Spark Jobs M Spark Stages T Spark Tasks) on (E executors with C cores)

A spark application can be given E number of executors to run the spark application on. Each executor might hold 1 or more spark cores. Every spark task will require atleast 1 core to execute, so imagine T number of tasks to be done in Z time depending on C cores. The higher C, Z is smaller.

With this understanding, if you want your DAG stage to run faster, *bring T as close or higher to C*. Additionally, this parallelism finally controls the number of output files you write using a Hudi based job. Let's understand the different kinds of knobs available:

BulkInsertParallelism This is used to control the parallelism with which output files will be created by a Hudi job. The higher this parallelism, the more number of tasks are created and hence the more number of output files will eventually be created. Even if you define parquet-max-file-size to be of a high value, if you make parallelism really high, the max file size cannot be honored since the spark tasks are working on smaller amounts of data.

Upsert/Insert Parallelism This is used to control how fast the read process should be when reading data into the job. Find more details here.

INT96, INT64 and timestamp compatibility

https://hudi.apache.org/docs/configurations.html#HIVE_SUPPORT_TIMESTAMP

Contributing to FAQ

A good and usable FAQ should be community-driven and crowd source questions/thoughts across everyone.

You can improve the FAQ by the following processes

- Comment on the text to spot inaccuracies, typos and leave suggestions.
- Propose new questions with answers under the comments section at the bottom of the page
- Lean towards making it very understandable and simple, and heavily link to parts of documentation as needed
- One committer on the project will review new questions and incorporate them upon review.