

RFC - 19 Clustering data for freshness and query performance

Proposers

- [liwei](#)
- [Satish Kotha](#)


Approvers

Status

Current state:

	Current State
UNDER DISCUSSION	
IN PROGRESS	
ABANDONED	
COMPLETED	✓
INACTIVE	

Discussion thread:

JIRA: [HUDI-897](#)  [HUDI-957](#) - Umbrella ticket for sequencing common tasks required to progress/unblock RFC-08, RFC-15 & RFC-19 CLOSED

Released: <Hudi Version>

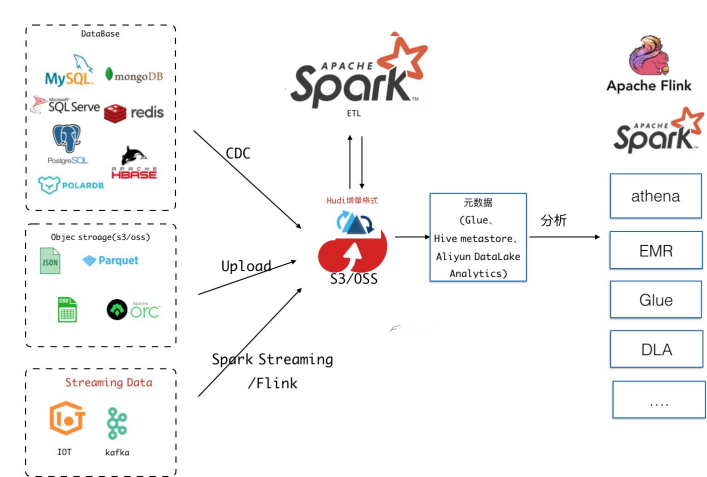
- [Proposers](#)
- [Approvers](#)
- [Status](#)
- [Abstract](#)
- [Implementation](#)
 - [COW Table timeline](#)
 - [MOR Table timeline](#)
 - [Clustering steps](#)
 - [Scheduling clustering](#)
 - [Running clustering](#)
 - [Commands to schedule and run clustering](#)
 - [Quick start using Inline Clustering](#)
 - [Setup for Async clustering Job](#)
 - [Some caveats](#)
- [Performance Evaluation](#)
 - [Summary](#)
- [Rollout/Adoption Plan](#)
- [Test Plan](#)

Abstract

The business scenarios of the data lake mainly include analysis of databases, logs, and files. One of the key trade-offs in managing a data lake is to choose between write throughput and query performance. For better write throughput, it is usually better to write incoming data into smaller data files. This will increase parallelism and improve ingestion speed substantially. But, this can create many small files. Also, in this approach, data locality is not optimal. Data is co-located with other records in the incoming batch and not with data that is queried often. Small file sizes and lack of data locality can degrade query performance. In addition, for many file systems including HDFS, performance degrades when there are many small files.

In this proposal, we present clustering framework for improving write throughput without compromising on query performance. Clustering framework can be used for rewriting data in a different way. Some example usecases:

- 1. Improve freshness: Write small files. stitch small files into large files after certain criteria are met (Time elapsed/ number of small files etc)
- 2. Improve query performance: Changing data layout on disk by sorting data on (different) columns.



Implementation

Hoodie write client insert/upsert/bulk_insert will continue to function as before. Users can configure the small file [soft limit](#) to 0 to force new data to go into a new set of file groups. In addition, 'clustering' action is provided to rewrite data in a different format. This clustering can run asynchronously or synchronously and will provide snapshot isolation between readers and writers. The exact steps taken for clustering are listed below for each table type.

COW Table timeline

COW TABLE (Disallow updates to file group)				
t5	t6 (clustering req)	t7	t8 (clustering t6 done)	t9
<div>f0-t0.parquet</div> <div>f1-t1parquet</div> <div>f2-t3.parquet</div> <div>f3-t4.parquet</div> <div>f4-t5.parquet</div>	<div>c1-t6.parquet (subset of f1, f2, f4 at t5) (phantom until t8)</div> <div>c2-t6.parquet (subset of f0, f3, f4 at t5) (phantom until t8)</div>	<div>Updates NOT Allowed on f0,f1,f2,f3,f4</div>	<div>t6.clusteringcommit (contains metadata that f0, f1,f2,f3, f4 are to be ignored)</div> <div>c1-t6.parquet</div> <div>c2-t6.parquet</div>	<div>Updates Allowed</div> <div>c1-t9.parquet</div>

In the example flow chart above, we show a partition state over time (t5 to t9). The sequence of steps taken for writing are listed below.

1. At t5, a partition in table has 5 file groups f0, f1, f2, f3, f4. For simplicity, assume that each file group is 100MB. So the total data in the partition is 500MB.
2. A clustering operation is requested at t6. Similar to compaction, we create a "t6.clustering.requested" file in metadata with 'ClusteringPlan' that includes all the file groups touched by clustering action across all partitions.
 - a. Example contents:
 - b. { partitionPath: {"datestr"}, oldfileGroups: [{fileId: "f0", time: "t0"}, { fileId: "f1", time: "t1"}, ...], newFileGroups: ["c1", "c2"] }
3. Lets say maximum file size after clustering is configured to be 250MB. Clustering would re-distribute all the data in partition into two file groups: c1, c2. These file groups are 'phantom' and invisible to queries until clustering is complete at t8.
4. Also, note that records in a file group can be split into multiple file groups. In this example, some records from the f4 file group go to both new file groups c1, c2.
5. While the clustering is in progress (t6 through t8), any upserts that touch these file groups are rejected.
6. After writing new data files c1-t6.parquet and c2-t6.parquet, if a global index is configured, we add entries in the record level index for all the keys with the new location. The new index entries will not be visible to other writes because there is no commit associated yet.
7. Finally, we create a commit metadata file 't6.commit' that includes file groups modified by this commit (f0,f1,f2,f3,f4).
8. Note that file groups (f0 to f4) are not deleted from disk immediately. Cleaner would clean these files before archiving t6.commit. We also update all views to ignore all file groups mentioned in all the commit metadata files. So readers will not see duplicates.

Note that there is a possible race condition at step 5 if multiple writers are allowed. Another writer could have started upserts just before the 'clustering requested' file is written. In the initial version, for simplicity, we assume there is only a single writer. The writer can either schedule clustering or run ingestion. The actual clustering operation can run asynchronously. When hoodie has multi-writer support(See RFC-22), we can consider making scheduling asynchronous too.

MOR Table timeline

MOR TABLE				
t5	t6 (clustering req)	t7	t8 (t6 clustering done)	t9
<div>f0-t0.parquet</div> <div>f1-t1.parquet</div> <div>f2-t2.log</div> <div>f3-t3.log</div> <div>f4-t4.log</div>	<div>c1-t6.parquet (subset of f1, f2, f4 at t5) (phantom until t8)</div> <div>c2-t6.parquet (subset of f0, f3, f4 at t5) (phantom until t8)</div>	<div>Updates and compaction NOT Allowed on f0,f1,f2,f3,f4</div>	<div>t6.clusteringcommit (contains metadata that f0, f1,f2,f3, f4 are to be ignored)</div> <div>c1-t8.parquet (f1, f2, f4 at t5)</div> <div>c2-t8.parquet (f0, f3, f4 at t5)</div>	<div>Updates and compaction Allowed on f0,f1,f2,f3,f4</div> <div>c1-t9.log</div> <div>c2-t9.log</div>

This is very similar to the COW table. For MOR table, inserts can go into either parquet files or into log files. This approach will continue to support both modes. The output of clustering is always parquet format. Also, compaction and clustering cannot run at the same time on the same file groups. Compaction also needs changes to ignore file groups that are already clustered.

Clustering steps

Overall, there are 2 parts to clustering

1. Scheduling clustering: Create clustering plan
2. Execute clustering: Process the plan. Create new files and replace old files.

Scheduling clustering

Following steps are followed to schedule clustering.

1. Identify files that are eligible for clustering
 - a. Filter specific partitions (based on config to prioritize latest vs older partitions)
 - b. Any files that have size > targetFileSize are not eligible for clustering
 - c. Any files that have pending compaction/clustering scheduled are not eligible for clustering
 - d. Any filegroups that have log files are not eligible for clustering (We could remove this restriction at a later stage.)
2. Group files that are eligible for clustering based on specific criteria. Each group is expected to have data size in multiples of 'targetFileSize'. Grouping is done as part of 'strategy' defined in the plan. We can provide 2 strategies
 - a. Group files based on record key ranges. This is useful because key range is stored in a parquet footer and can be used for certain queries/updates.
 - b. Groups files based on commit time.
 - c. Group files that have overlapping values for custom columns
 - i. As part of clustering, we want to sort data by column(s) in the schema (other than row_key). Among the files that are eligible for clustering, it is better to group files that have overlapping data for the custom columns.
 1. we have to read data to find this which is expensive with way ingestion works. We can consider storing value ranges as part of ingestion (we already do this for record_key). This requires more discussion. Probably, in the short term, we can focus on strategy 2a below (no support for sortBy custom columns).
 2. Example: say the target of clustering is to produce 1GB files. Partition initially has 8 * 512MB files. (After clustering, we expect data to be present in 4 * 1GB files.)
 - ii. Assume among 8 files, say only 2 files have overlapping data for the 'sort column', then these 2 files will be part of one group. Output of the group after clustering is one 1GB file.
 - iii. Assume among 8 files, say 4 files have overlapping data for the 'sort column', then these 4 files will be part of one group. Output of the group after clustering is two 1GB files.
 - d. Group random files
 - e. We could put a cap on group size to improve parallelism and avoid shuffling large amounts of data
3. Filter groups based on specific criteria (akin to orderAndFilter in CompactionStrategy)
4. Finally, the clustering plan is saved to the timeline. Structure of metadata is here: <https://github.com/apache/hudi/blob/master/hudi-common/src/main/avro/HoodieClusteringPlan.avsc>

In the 'metrics' element, we could store 'min' and 'max' for each column in the file for helping with debugging and operations.

Note that this scheduling can be plugged in with custom implementation. In the first version, [following strategy](#) is provided by default.

Running clustering

1. Read the clustering plan, look at the number of 'clusteringGroups'. This gives parallelism.
2. Create inflight clustering file
3. For each group
 - a. Instantiate appropriate strategy class with strategyParams (example: sortColumns)
 - b. Strategy class defines partitioner and we can use it to create buckets and write the data.
4. Create replacecommit. Contents are in [HoodieReplaceCommitMetadata](#)
 - a. operationType is set to 'clustering'.
 - b. We can extend the metadata and store additional fields to help track important information (strategy class can return this 'extra' metadata information)
 - i. strategy used to combine files
 - ii. track replaced files

In the first version, [following strategy](#) based on 'bulk_insert' is provided as default option.

Commands to schedule and run clustering

Quick start using Inline Clustering

```

import org.apache.hudi.QuickstartUtils._
import scala.collection.JavaConversions._
import org.apache.spark.sql.SaveMode._
import org.apache.hudi.DataSourceReadOptions._
import org.apache.hudi.DataSourceWriteOptions._
import org.apache.hudi.config.HoodieWriteConfig._

val tableName = "hudi_trips_cow"
val basePath = "/tmp/hudi_trips_cow"

val dataGen = new DataGenerator(Array("2020/03/11"))
val updates = convertToStringList(dataGen.generateInserts(10))
val df = spark.read.json(spark.sparkContext.parallelize(updates, 1));
df.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, tableName).
    option("hoodie.parquet.small.file.limit", "0").
    option("hoodie.clustering.inline", "true").
    option("hoodie.clustering.inline.max.commits", "4").
    option("hoodie.clustering.plan.strategy.target.file.max.bytes", "1073741824").
    option("hoodie.clustering.plan.strategy.small.file.limit", "629145600").
    option("hoodie.clustering.plan.strategy.sort.columns", ""). //optional, if sorting is needed as part of rewriting data
    mode(Append).
    save(basePath);

```

Setup for Async clustering Job

Clustering can be scheduled and run asynchronously using a SparkJob. The utilities spark job can be found [here](#)

1. prepare the clusering config file:

```

cat /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_with_small_filegroups3/config/clusteringjob.properties
hoodie.clustering.inline.max.commits=2

```

2. Schedule clustering

```
bin/spark-submit \
--master local[4] \
--class org.apache.hudi.utilities.HoodieClusteringJob \
/Users/liwei/work-space/dla/opensource/incubator-hudi/packaging/hudi-utilities-bundle/target/hudi-utilities-bundle_2.11-0.8.0-SNAPSHOT.jar \
--schedule \
--base-path /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_with_small_filegroups3/dest \
--table-name hudi_table_with_small_filegroups3_schedule_clustering \
--props /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_with_small_filegroups3/config/clusteringjob.properties \
--spark-memory 1g
```

you can find the schedule clustering instant time in the spark logs. With the log prefix "The schedule instant time is" ,and the schedule clustering instant time is 20210122190240

```
21/01/22 19:02:43 INFO FileSystemViewHandler: TimeTakenMillis[Total=68, Refresh=1, handle=66, Check=1], Success=true, Query=partition=2016%2
ork-space%2Fspark%2Fspark-2.4.6-bin-hadoop2.7%2Fhudi_table_with_small_filegroups3%2Fdest&lastinstantts=20210122190139&timelinehash=660662fe5
a6f865440f6bc5354, Host=30.225.168.104:64367, synced=false
21/01/22 19:02:43 INFO Executor: Finished task 0.0 in stage 4.0 (TID 4). 1518 bytes result sent to driver
21/01/22 19:02:43 INFO TaskSetManager: Finished task 0.0 in stage 4.0 (TID 4) in 1039 ms on localhost (executor driver) (1/1)
21/01/22 19:02:43 INFO TaskSchedulerImpl: Removed TaskSet 4.0, whose tasks have all completed, from pool
21/01/22 19:02:43 INFO DAGScheduler: ResultStage 4 (collect at HoodieSparkEngineContext.java:78) finished in 1.088 s
21/01/22 19:02:43 INFO DAGScheduler: Job 4 finished: collect at HoodieSparkEngineContext.java:78, took 1.090955 s
21/01/22 19:02:43 INFO HoodieClusteringJob: The schedule instant time is 20210122190240
21/01/22 19:02:43 INFO HoodieClusteringJob: Clustering with basepath: /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_wit
udi_table_with_small_filegroups3_schedule_clustering, runSchedule: true success
21/01/22 19:02:43 INFO SparkUI: Stopped Spark web UI at http://30.225.168.104:4040
21/01/22 19:02:44 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
21/01/22 19:02:44 INFO MemoryStore: MemoryStore cleared
21/01/22 19:02:44 INFO BlockManager: BlockManager stopped
21/01/22 19:02:44 INFO BlockManagerMaster: BlockManagerMaster stopped
21/01/22 19:02:44 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
21/01/22 19:02:44 INFO SparkContext: Successfully stopped SparkContext
```

3. use the schedule instant time "20210122190240" to run clustering

```
bin/spark-submit \
--master local[4] \
--class org.apache.hudi.utilities.HoodieClusteringJob \
/Users/liwei/work-space/dla/opensource/incubator-hudi/packaging/hudi-utilities-bundle/target/hudi-utilities-bundle_2.11-0.8.0-SNAPSHOT.jar \
--base-path /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_with_small_filegroups3/dest \
--instant-time 20210122190240 \
--table-name hudi_table_with_small_filegroups3_clustering \
--props /Users/liwei/work-space/spark/spark-2.4.6-bin-hadoop2.7/hudi_table_with_small_filegroups3/config/clusteringjob.properties \
--spark-memory 1g
```

Some caveats

There is WIP to fix these limitations. But these issues are worth mentioning:

1. This is alpha feature. Although, there is good unit test coverage, there may be some rough edges. Please report any issues.
2. Better support for async clustering is coming soon.
3. Clustering doesn't work with incremental timeline. So disable it by setting "hoodie.filesystem.view.incr.timeline.sync.enable: false"
4. Incremental queries are not supported with clustering. Incremental queries consider all the data written by clustering as new rows.
5. Clustering creates new type of commit "timestamp.replacecommit". There may be some places in code where we only read commits /delta commits and miss replace commits as part of reading valid commits in timeline. This can cause discrepancy in some cases.
6. Clean policy is different for 'replacecommit'. So there may be more versions retained leading to extra storage usage.

Performance Evaluation

Dataset: <https://s3.amazonaws.com/amazon-reviews-pds/readme.html>

Query: select sum(total_votes), product_category from amzn_reviews where review_date > '2007' and review_date < '2009' group by product_category

1. Convert dataset to hoodie format

```
val df = spark.read.option("sep", "\t").option("header", "true").csv(amznReviewsRawDataPath)

val tableName = "reviews"

df.write.format("org.apache.hudi").
  options(getQuickstartWriteConfigs).
  option(PRECOMBINE_FIELD_OPT_KEY, "customer_id").
  option(RECORDKEY_FIELD_OPT_KEY, "review_id").
  option(PARTITIONPATH_FIELD_OPT_KEY, "marketplace").
  option(OPERATION_OPT_KEY, "insert").
  option(TABLE_NAME, tableName).
  mode(Overwrite).
  save(amznReviewHudiPath);

//creates ~500 data files in one partition
```

2. Evaluate query time (No Clustering)

query takes ~10 seconds

```
scala> spark.time(spark.sql("select sum(total_votes), product_category from amzn_reviews where
review_date > '2007' and review_date < '2009' group by product_category").collect())
Time taken: 10018 ms
```

spark

3.0.1

Jobs

Stages

Storage

Environment


Executors

SQL

Spark shell application LR

Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 0.0 min
Locality Level Summary: Any: 271
Input Size / Records: 630.0 MB / 14190257
Shuffle Write Size / Records: 108.2 MB / 10371
Associated Job Id: 2
DAG Visualization



Event Timeline

spark

3.0.1

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application LR

Details for Query 1

Submitted Time: 2021-01-05 19:55:08
Duration: 11 s
Successful Jobs: 2
Show the Stage ID and Task ID that corresponds to the max metric
Details

== Parsed Logical Plan ==
Aggregate [product_category], [sum(cast(total_votes as double)), None], [product_category]
-- Filter [(review_date > 2007) AND (review_date < 2009)]
-- > BroadcastJoin [amzn_reviews]

== Analyzed Logical Plan ==
sum(cast(total_votes as DOUBLE)): double, product_category: string
Aggregate [product_category#1], [sum(cast(total_votes#4 as double)) AS sum(cast(total_votes as DOUBLE))#45, product_category#1]
-- Filter [(review_date#29 > 2007) AND (review_date#29 < 2009)]
-- SubqueryAlias amzn_reviews

Relation_hoodie_comit_time#0_hoodie_comit_seq#01_hoodie_record_key#2_hoodie_partition_path#3_hoodie_file_name#4_marketplace#5_customer_id#6_review_id#7_product_id#8_product_parent#9_product_title#10_product_category#11_star_rating#12_helpful_votes#13_total_votes#14_vine#15_verified_purchase#16_review_headline#17_review_body#18_review_date#19 parquet

== Optimized Logical Plan ==
Aggregate [product_category#1], [sum(cast(total_votes#4 as double)) AS sum(cast(total_votes as DOUBLE))#45, product_category#1]
-- Project [product_category#1, total_votes#14]
-- Filter [(cast(review_date#29 as double) > 2007) AND (review_date#29 < 2009)]

Relation_hoodie_comit_time#0_hoodie_comit_seq#01_hoodie_record_key#2_hoodie_partition_path#3_hoodie_file_name#4_marketplace#5_customer_id#6_review_id#7_product_id#8_product_parent#9_product_title#10_product_category#11_star_rating#12_helpful_votes#13_total_votes#14_vine#15_verified_purchase#16_review_headline#17_review_body#18_review_date#19 parquet

== Physical Plan ==
* C2 HashAggregateExec[product_category#1], functions=[sum(cast(total_votes#4 as double))], output=[sum(cast(total_votes as DOUBLE))#45, product_category#1]
-- Exchange hashpartitioning(product_category#1, 200), true, [id#1]
-- * C1 HashAggregateExec[product_category#1], functions=[sum(cast(total_votes#4 as double))], output=[product_category#1, sum#45]
-- * C0 Project [product_category#1, total_votes#14]
-- * C2 Filter [(cast(review_date#29 as double) > 2007) AND (review_date#29 < 2009)]
-- * C1 ColumnarTableScan
-- * C0 Filter [isnull(review_date), GreaterThan(review_date,2007), LessThan(review_date,2009)], ReadSchema: struct<product_category:string,total_votes:string,review_date:string>
-- FileScan parquet [product_category#1,total_votes#14,review_date#29] Batched: true, DataFilters: [isnull(review_date#29), (review_date#29 > 2007), (review_date#29 < 2009)], Format: Parquet, Location: s3a://newsfiledbdo

```
// create writeClient with overriding following write config:
// "hoodie.clustering.plan.strategy.sort.columns" -> "product_category,review_date"
// "hoodie.clustering.plan.strategy.max.bytes.per.group" -> "107374182400"
// "hoodie.clustering.plan.strategy.max.num.groups" -> "1"

val clusteringInstant = writeClient.scheduleClustering(Option.empty())
val metadata = writeClient.cluster(clusteringInstant.get, true)

//creates ~350 data files and replaces existing ~500 data files one partition
```



Verify replacecommit is created

```
$ hadoop fs -ls $amznReviewHudiPath/.hoodie/
```

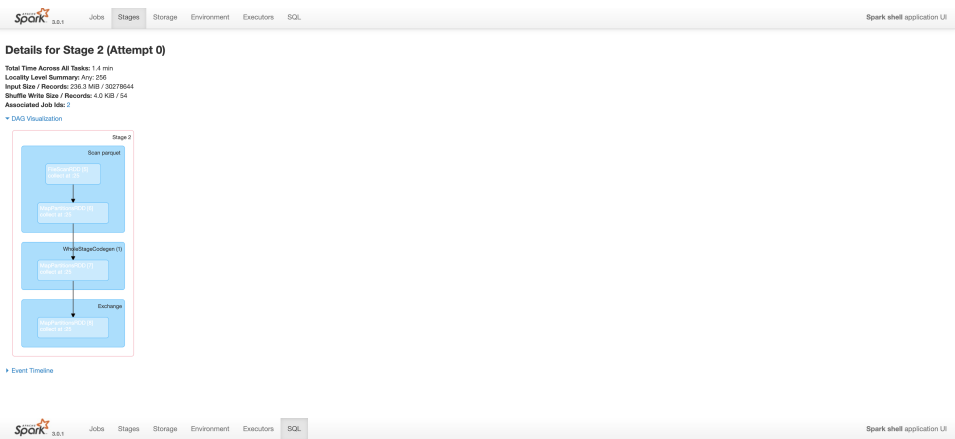
Found 10 items

```
drwxr-xr-x  - satish      0 2021-01-20 18:38 $amznReviewHudiPath/.hoodie/.aux
drwxr-xr-x  - satish      0 2021-01-21 00:49 $amznReviewHudiPath/.hoodie/.temp
-rw-r--r--  3 satish    445621 2021-01-20 18:41 $amznReviewHudiPath/.hoodie/20210120183848.commit
-rw-r--r--  3 satish      0 2021-01-20 18:39 $amznReviewHudiPath/.hoodie/20210120183848.commit.requested
-rw-r--r--  3 satish     979 2021-01-20 18:40 $amznReviewHudiPath/.hoodie/20210120183848.inflight
-rw-r--r--  3 satish    493983 2021-01-21 00:51 $amznReviewHudiPath/.hoodie/20210121004731.replacecommit
-rw-r--r--  3 satish      0 2021-01-21 00:47 $amznReviewHudiPath/.hoodie/20210121004731.replacecommit.inflight
-rw-r--r--  3 satish    131495 2021-01-21 00:47 $amznReviewHudiPath/.hoodie/20210121004731.replacecommit.requested
drwxr-xr-x  - satish      0 2021-01-20 18:38 $amznReviewHudiPath/.hoodie/archived
-rw-r--r--  3 satish     228 2021-01-20 18:38 $amznReviewHudiPath/.hoodie/hoodie.properties
```

4. Evaluate query time (with Clustering). Note that same query in step 2 that took 10 seconds now runs in 4 seconds

query takes ~4 seconds

```
scala> spark.time(spark.sql("select sum(total_votes), product_category from amzn_reviews where
review_date > '2007' and review_date < '2009' group by product_category").collect())
Time taken: 4099 ms
```

Summary

In summary, rewriting the data using clustering can speed up query runtime by ~60%

Rollout/Adoption Plan

- No impact on the existing users because add new function. Note that if you are already using Hudi, It is important to move your readers first to 0.7 release version before upgrading writers. This is because clustering creates a new type of commit and its important query engines recognize this new commit type.

Test Plan

- Unit tests
- Integration tests
- Test on the cluster for a larger dataset.