Carbon Pre-aggregate Table

Author: Ravindra Pesala, Kumar Vishal, Jacky Li
Version: 1.1
Date: 2017-10-27

Changes in version 1.1:
1. Aggregate table creation syntax is changed
2. Table property is changed for aggregate table for timeseries data

Changes in version 1.3:
1. Optimize syntax for creating timeseries pre-aggregate table

Background

Currently Carbondata has standard SQL capability on distributed data sets. This document explains about pre-aggregating tables on carbondata to support OLAP style query to improve query performance.

Creating of aggregation tables

Supporting pre-aggregate table on existing main table

Suppose there are a main table called sales

```
CREATE TABLE sales (
      order_time timestamp,
      user_id string,
      sex string,
      country string,
      quantity int,
      price bigint)
STORED BY 'carbondata'
```

User can create pre-aggregation table using following DDL.

```
CREATE DATAMAP agg_sales
ON TABLE sales
USING "preaggregate"
AS
     SELECT country, sex, sum(quantity), avg(price)
     FROM sales
     GROUP BY country, sex
```

As shown, pre-aggregate table is a datamap. By using this, user can create as many datamap as he wants to improve the query performance, as long as the storage requirement and loading speed is acceptable to the user.

Supporting rollup table for timeseries data

For timeseries data, carbon supports an easy-to-use DDL to make it easier to create pre-aggregate tables in rollup manner. To support this, carbon has built-in understanding different granularity levels: year, month, day, hour, minute.

```
CREATE DATAMAP agg_hour
ON TABLE sales
USING ‘TIMESERIES’
DMPROPERTIES (
  ‘event_time’=’order_time’,
  ‘hour_granularity’=’1’,
) AS
     SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), 
sum(price), avg(price)
     FROM sales
     GROUP BY order_time, country, sex
```

In the above case, user defines a rollup table for main table (sales), since the rollup granularity is “hour”, carbondata will generate rollup table for each hour. For example, carbondata will aggregate data between ‘2018-02-02 16:00:00’ and ‘2018-02-02 16:59:59’, and the key is ‘2018-02-02 16:00:00’ in aggregate table. Their timeseries pre-aggregate table name will be sales_ agg_hour.

Datamap property	Mandatory	Default value	Description
event_time	Yes	(none)	The event time column in the schema, which will be used for rollup. The column need be timestamp type.
Time granularity	Yes	(none)	The aggregate dimension hierarchy. The property value is a key value pair specifying the aggregate granularity for the time level. Carbon support “year_granularity”, “month_granularity”, “day_granularity”, “hour_granularity”, “minute_granularity”, “second_granularity”. Granularity only support 1 when creating datamap. For example, ‘hour_granularity’=’1’ means aggregate every 1 hour. Now the value only support 1.

After user creates the pre-aggregate tables by above syntax for time series data, user can still add more pre-aggregation tables by syntax for normal aggregate table defined in previous section.

[Implementation]
When pre-aggregation table is created, carbon will associate the pre-aggregate table and the main table by update main table's metadata to include this information.
When query is issued, carbon will check whether the table is associated with any pre-aggregate table. If yes, it will check whether there are pre-aggregate table satisfy the query condition and do corresponding query plan transform to use that pre-aggregate table.

In internal implementation, carbon will create these table with SORT_COLUMNS='group by column defined above', so that filter group by query on main table will be faster because it can leverage the index in pre-aggregate tables.

[Timeseries UDF Syntax]
timeseries(timeseries column name, 'aggregation level').

[Example]
Select timeseries(**order_time*****, 'hour'), sum(*****quantity*****) from <table> group by timeseries(*****order_time*****,'hour')***

Loading data to pre-aggregated tables

Loading aggregate tables for existing table

For existing table with loaded data, data load to pre-aggregate table will be trigger by the CREATE DATAMAP statement when user creates the pre-aggregate table.
Note that in this case, the pre-aggregate table will have only one segment while the main table may have multiple segments.

[Implementation]
For aggregate table, the datamap creation is handled by org.apache.carbondata.datamap.AggregateDataMapHandler, it should treat the command like CTAS (CREATE TABLE AS SELECT) statement, and one difference is that the datamap is not visible to the user.

Loading aggregate tables for new load

For any new load, aggregate tables loading initiates as soon as main table loading finishes. The loading of main table and aggregate table is an atomic operation, meaning that main table and aggregate tables are only visible to the user after all tables are loaded. So that data loading will take more time than the case of there is no pre-aggregate table.

[Implementation]

The data load process will be modified to include pre-aggregate table loading. The loading of main table and pre-aggregate tables are in sequence, one after another. The whole process is protected by main table’s table status file to ensure ACID.

Loading aggregate tables for streaming data

The initial idea is that we can trigger the aggregate table based on the size. That means we can finish the segment load after receives configurable amount of data and create a new segment to load new streaming data. While finishing the segment we can trigger the agg tables on it. So there would not be any agg tables on ongoing streaming segment but querying can be done on the streaming segment of actual table.
For example user configures stream_segment size as 1 GB, so for every 1 GB of stream data it receives, it creates a new segment and finishes the current segment. While finishing the current segment we can trigger agg table loading and compaction of segments.
While querying of data we change the query plan to apply union of agg table and streaming segment of actual table to get the current data.
For more detail, see the streaming ingest design document

Aggregation types support and handling

SUM, MAX, MIN

For this aggtype we can choose the related agg table directly just by replacing the main table with agg table.
Rollup can be supported for this agg types.

For example:
User creates pre-aggregate as following

```
CREATE TABLE agg_sales TBLPROPERTIES (parent_table=”xx”)
AS SELECT c1, c2, sum(c3) FROM source GROUP BY c1, c2
```

For following query, the query planner will change query plan to query on the pre-aggregate table:
Q1: SELECT c1, c2, sum(c3) FROM source GROUP BY c1, c2
Q2: SELECT c1, sum(c3) FROM source GROUP BY c1

COUNT

We should choose SUM when we select agg table.
Rollup can be supported for this agg type.

AVG

While creating the agg table we split avg to two columns sum and count, so while choosing aggregation table we can change the plan from AVG to SUM/COUNT

Rollup can be supported for this agg type.

DISTINCT_COUNT

For this aggtype, carbon will store the individual value of the aggregate column, and when come to query, it can choose the related agg table directly just by replacing the main table with agg table.
Rollup cannot be supported for this agg type.

For example:
User creates pre-aggregate as following

```
CREATE TABLE agg_sales TBLPROPERTIES (parent_table="xx")
AS SELECT c1, c2, count(distinct c3) FROM source GROUP BY c1, c2
```

For following queries, Q1 can leverage pre-aggregate table, while Q2 can not leverage pre-aggregate table and will query on main table, since distict_count does not support rollup.
Q1: SELECT c1, c2, count(distinct c3) FROM source GROUP BY c1, c2
Q2: SELECT c1, count(distinct c3) FROM source GROUP BY c1

Compaction

Compaction is an optional operation for pre-aggregate table. If compaction is performed on main table but not performed on pre-aggregate table, all queries still can benefit from pre-aggregate table.
If user wants to further improve performance on pre-aggregate table, he can trigger compaction on pre-aggregate table directly, it will further merge the segments inside pre-aggregation table. To do that, use ALTER TABLE COMPACT command on the pre-aggregate table just like the main table.

For implementation, there are two kinds of implementation for compaction.

1. Mergable pre-aggregate tables: if aggregate functions are count, max, min, sum, avg, the pre-aggregate table segments can be merged directly without re-computing it.
2. Non-mergable pre-aggregate tables: if aggregate function include distinct_count, it needs to re-compute when doing compaction on pre-aggregate table.

Update/Delete Operations

Data update and delete on main table will be rejected if agg tables are present.
To do data update and delete, user should drop all pre-aggregate tables associated with the main table, then he can perform data update and delete on the main table. Pre-aggregate table can be rebuild manually after update operation is finished.

Delete Segment Operations

Delete segment on main table will be rejected if agg tables are present.

To delete segment, user should drop all pre-aggregate tables associated with the main table, then he can delete segment on the main table. Pre-aggregate table can be rebuild manually after delete segment operation is finished

Alter Table Operations

Adding of new column will not impact agg table.
Deleting or renaming existing column may invalidate agg tables, if it invalidate, the operation will be rejected.
User can manually perform this operation and rebuild pre-aggregate table as update scenario.