Apache CarbonData

Ver 1.4.1
Documentation

The Apache Software Foundation **2018-08-17**

Table of Contents

1 Quick Start

Quick Start

This tutorial provides a quick introduction to using CarbonData.

1.1 Prerequisites

- Installation and building CarbonData.
- Create a sample.csv file using the following commands. The CSV file is required for loading data into CarbonData.

```
cd carbondata cat > sample.csv << EOF id,name,city,age 1,david,shenzhen,31
2,eason,shenzhen,27 3,jarry,wuhan,35 EOF
```

1.2 Interactive Analysis with Spark Shell Version 2.1

Apache Spark Shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. Please visit Apache Spark Documentation for more details on Spark shell.

1.2.1.1 Basics

Start Spark shell by running the following command in the Spark directory:

```
./bin/spark-shell --jars <carbondata assembly jar path>
```

NOTE: Assembly jar will be available after building CarbonData and can be copied from `./assembly/target/scala-2.1x/carbondata_xxx.jar`

In this shell, SparkSession is readily available as `spark` and Spark context is readily available as `sc`.

In order to create a CarbonSession we will have to configure it explicitly in the following manner :

- Import the following :

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.CarbonSession._
```

- Create a CarbonSession :

```
val carbon = SparkSession.builder().config(sc.getConf)
             .getOrCreateCarbonSession("<hdfs store path>")
```

NOTE: By default metastore location is pointed to `../carbon.metastore`,
user can provide own metastore location to CarbonSession like
`SparkSession.builder().config(sc.getConf) .getOrCreateCarbonSession("<hdfs store path>", "<local metastore path>")`

1.2.1.2 Executing Queries

1.Creating a Table

```
scala>carbon.sql("CREATE TABLE
                        IF NOT EXISTS test_table(
                                  id string,
                                  name string,
                                  city string,
                                  age Int)
                       STORED BY 'carbondata'")
```

1.Loading Data to a Table

```
scala>carbon.sql("LOAD DATA INPATH '/path/to/sample.csv'
                  INTO TABLE test_table")
```

NOTE: Please provide the real file path of `sample.csv` for the above script. If you get "tablestatus.lock" issue, please refer to troubleshooting

1.Query Data from a Table

```
scala>carbon.sql("SELECT * FROM test_table").show()

scala>carbon.sql("SELECT city, avg(age), sum(age)
                  FROM test_table
                  GROUP BY city").show()
```

2 CarbonData File Structure

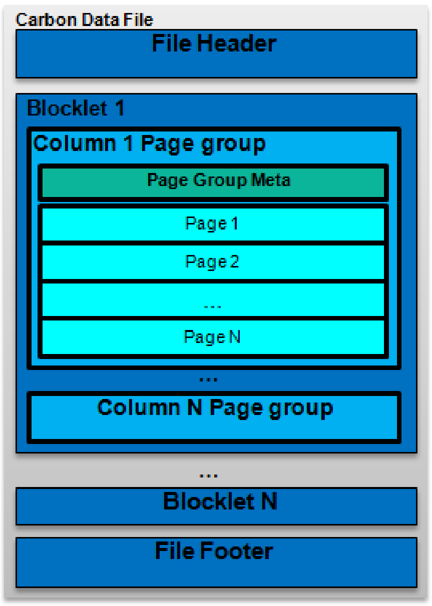
CarbonData File Structure

CarbonData files contain groups of data called blocklets, along with all required information like schema, offsets and indices etc, in a file header and footer, co-located in HDFS.

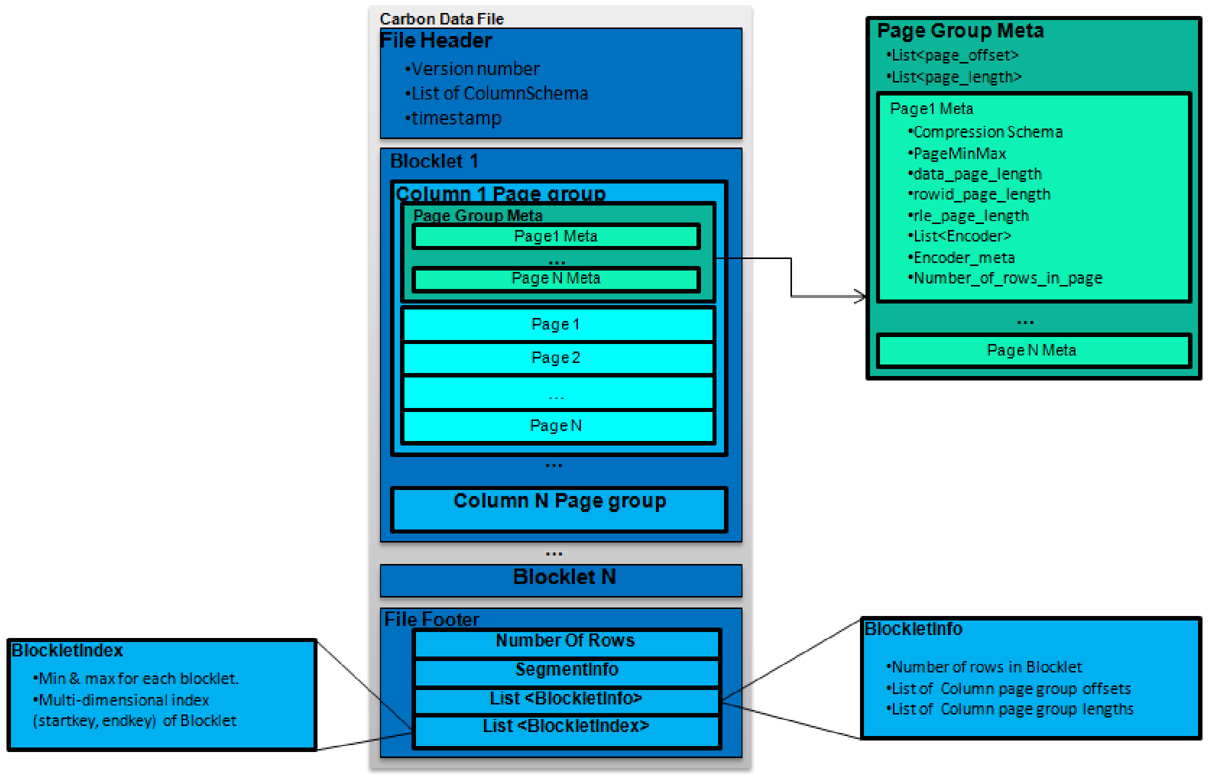
The file footer can be read once to build the indices in memory, which can be utilized for optimizing the scans and processing for all subsequent queries.

2.1.1 Understanding CarbonData File Structure

- Block : It would be as same as HDFS block, CarbonData creates one file for each data block, user can specify TABLE_BLOCKSIZE during creation table. Each file contains File Header, Blocklets and File Footer.



- File Header : It contains CarbonData file version number, list of column schema and schema updation timestamp.
- File Footer : it contains Number of rows, segmentinfo ,all blocklets' info and index, you can find the detail from the below diagram.
- Blocklet : Rows are grouped to form a blocklet, the size of the blocklet is configurable and default size is 64MB, Blocklet contains Column Page groups for each column.
- Column Page Group : Data of one column and it is further divided into pages, it is guaranteed to be contiguous in file.
- Page : It has the data of one column and the number of row is fixed to 32000 size.



2.1.2 Each page contains three types of data

- Data Page: Contains the encoded data of a column of columns.
- Row ID Page (optional): Contains the row ID mappings used when the data page is stored as an inverted index.
- RLE Page (optional): Contains additional metadata used when the data page is RLE coded.

3 Data Types

Data Types

3.1.1.1 CarbonData supports the following data types:

- Numeric Types
  - SMALLINT
  - INT/INTEGER
  - BIGINT
  - DOUBLE
  - DECIMAL
- Date/Time Types
  - TIMESTAMP
  - DATE
- String Types
  - STRING
  - CHAR
  - VARCHAR

  NOTE: For string longer than 32000 characters, use `LONG_STRING_COLUMNS` in table property. Please refer to TBLProperties in CreateTable for more information.
- Complex Types
  - arrays: ARRAY `<data_type>`
  - structs: STRUCT `<col_name : data_type COMMENT col_comment, ...>`

  NOTE: Only 2 level complex type schema is supported for now.
- Other Types
  - BOOLEAN

4 Data Management on CarbonData

Data Management on CarbonData

This tutorial is going to introduce all commands and data operations on CarbonData.

- CREATE TABLE
- CREATE DATABASE
- TABLE MANAGEMENT
- LOAD DATA
- UPDATE AND DELETE
- COMPACTION
- PARTITION
- BUCKETING
- SEGMENT MANAGEMENT

4.1 CREATE TABLE

This command can be used to create a CarbonData table by specifying the list of fields along with the table properties. You can also specify the location where the table needs to be stored.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name[(col_name
data_type , ...)] STORED AS carbondata [TBLPROPERTIES
(property_name=property_value, ...)] [LOCATION 'path']
```

NOTE: CarbonData also supports “STORED AS carbondata” and “USING carbondata”. Find example code at CarbonSessionExample in the CarbonData repo.

4.1.1 Usage Guidelines

Following are the guidelines for TBLPROPERTIES, CarbonData’s additional table options can be set via carbon.properties.

- **Dictionary Encoding Configuration**

  Dictionary encoding is turned off for all columns by default from 1.3 onwards, you can use this command for including or excluding columns to do dictionary encoding. Suggested use cases : do dictionary encoding for low cardinality columns, it might help to improve data compression ratio and performance.

  `TBLPROPERTIES ('DICTIONARY_INCLUDE'='column1, column2')` NOTE: Dictionary Include/Exclude for complex child columns is not supported.

- **Inverted Index Configuration**

  By default inverted index is enabled, it might help to improve compression ratio and query speed, especially for low cardinality columns which are in reward position. Suggested use cases : For high cardinality columns, you can disable the inverted index for improving the data loading performance.

  `TBLPROPERTIES ('NO_INVERTED_INDEX'='column1, column3')`

- **Sort Columns Configuration**

  This property is for users to specify which columns belong to the MDK(Multi-Dimensions-Key) index. * If users don’t specify “SORT_COLUMN” property, by default MDK index be built by using all dimension columns except complex data type column. * If this property is specified but with empty argument, then the table will be loaded without sort. * This supports only string, date, timestamp, short, int, long, and boolean data types. Suggested use cases : Only build MDK index for required columns,it might help to improve the data loading performance.

`TBLPROPERTIES ('SORT_COLUMNS'='column1, column3') OR TBLPROPERTIES ('SORT_COLUMNS'='')` NOTE: Sort_Columns for Complex datatype columns is not supported.

- **Sort Scope Configuration**

  This property is for users to specify the scope of the sort during data load, following are the types of sort scope.

  - LOCAL_SORT: It is the default sort scope.
  - NO_SORT: It will load the data in unsorted manner, it will significantly increase load performance.
  - BATCH_SORT: It increases the load performance but decreases the query performance if identified blocks > parallelism.
  - GLOBAL_SORT: It increases the query performance, especially high concurrent point query. And if you care about loading resources isolation strictly, because the system uses the spark GroupBy to sort data, the resource can be controlled by spark.

4.1.2 Example:

```
CREATE TABLE IF NOT EXISTS productSchema.productSalesTable
( productNumber INT, productName STRING, storeCity STRING,
storeProvince STRING, productCategory STRING, productBatch STRING,
saleQuantity INT, revenue INT) STORED BY 'carbondata' TBLPROPERTIES
('SORT_COLUMNS'='productName,storeCity', 'SORT_SCOPE'='NO_SORT')
```

NOTE: CarbonData also supports “using carbondata”. Find example code at SparkSessionExample in the CarbonData repo.

- **Table Block Size Configuration**

  This command is for setting block size of this table, the default value is 1024 MB and supports a range of 1 MB to 2048 MB.

  `TBLPROPERTIES ('TABLE_BLOCKSIZE'='512')` **NOTE:** 512 or 512M both are accepted.

- **Table Compaction Configuration**

  These properties are table level compaction configurations, if not specified, system level configurations in carbon.properties will be used. Following are 5 configurations:

  - MAJOR_COMPACTION_SIZE: same meaning as carbon.major.compaction.size, size in MB.
  - AUTO_LOAD_MERGE: same meaning as carbon.enable.auto.load.merge.
  - COMPACTION_LEVEL_THRESHOLD: same meaning as carbon.compaction.level.threshold.
  - COMPACTION_PRESERVE_SEGMENTS: same meaning as carbon.numberof.preserve.segments.
  - ALLOWED_COMPACTION_DAYS: same meaning as carbon.allowed.compaction.days.

  ```
  TBLPROPERTIES ('MAJOR_COMPACTION_SIZE'='2048',
  'AUTO_LOAD_MERGE'='true', 'COMPACTION_LEVEL_THRESHOLD'='5,6',
  'COMPACTION_PRESERVE_SEGMENTS'='10', 'ALLOWED_COMPACTION_DAYS'='5')
  ```

- **Streaming**

CarbonData supports streaming ingestion for real-time data. You can create the ‘streaming’ table using the following table properties.

```
TBLPROPERTIES ('streaming'='true')
```

- **Local Dictionary Configuration**

Columns for which dictionary is not generated needs more storage space and in turn more IO. Also since more data will have to be read during query, query performance also would suffer.Generating dictionary per blocklet for such columns would help in saving storage space and assist in improving query performance as carbondata is optimized for handling dictionary encoded columns more effectively.Generating dictionary internally per blocklet is termed as local dictionary. Please refer to File structure of Carbondata for understanding about the file structure of carbondata and meaning of terms like blocklet.

Local Dictionary helps in: 1. Getting more compression. 2. Filter queries and full scan queries will be faster as filter will be done on encoded data. 3. Reducing the store size and memory footprint as only unique values will be stored as part of local dictionary and corresponding data will be stored as encoded data. 4. Getting higher IO throughput.

NOTE:

- Following Data Types are Supported for Local Dictionary:
  - STRING
  - VARCHAR
  - CHAR
- Following Data Types are not Supported for Local Dictionary:
  - SMALLINT
  - INTEGER
  - BIGINT
  - DOUBLE
  - DECIMAL
  - TIMESTAMP
  - DATE
  - BOOLEAN
- In case of multi-level complex dataType columns, primitive string/varchar/char columns are considered for local dictionary generation.

Local dictionary will have to be enabled explicitly during create table or by enabling the system property ‘carbon.local.dictionary.enable’. By default, Local Dictionary will be disabled for the carbondata table.

Local Dictionary can be configured using the following properties during create table command:

| Properties | Default value | Description | | ——– | ———– | ——— | | LOCAL_DICTIONARY_ENABLE | false | Whether to enable local dictionary generation. **NOTE:** If this property is defined, it will override the value configured at system level by ‘carbon.local.dictionary.enable’ | | LOCAL_DICTIONARY_THRESHOLD | 10000 | The maximum cardinality of a column upto which carbondata can try to generate local dictionary (maximum - 100000) | | LOCAL_DICTIONARY_INCLUDE | string/varchar/char columns| Columns for which Local Dictionary has to be generated. **NOTE:** Those string/varchar/char columns which are added into DICTIONARY_INCLUDE option will not be considered for local dictionary generation.| | LOCAL_DICTIONARY_EXCLUDE | none | Columns for which Local Dictionary need not be generated. |

Fallback behavior:

- When the cardinality of a column exceeds the threshold, it triggers a fallback and the generated dictionary will be reverted and data loading will be continued without dictionary encoding.

NOTE: When fallback is triggered, the data loading performance will decrease as encoded data will be discarded and the actual data is written to the temporary sort files.

Points to be noted:

1. Reduce Block size:

   Number of Blocks generated is less in case of Local Dictionary as compression ratio is high. This may reduce the number of tasks launched during query, resulting in degradation of query performance if the pruned blocks are less compared to the number of parallel tasks which can be run. So it is recommended to configure smaller block size which in turn generates more number of blocks.
2. All the page-level data for a blocklet needs to be maintained in memory until all the pages encoded for local dictionary is processed in order to handle fallback. Hence the memory required for local dictionary based table is more and this memory increase is proportional to number of columns.

4.1.3 Example:

``` CREATE TABLE carbontable(

```
           column1 string,

           column2 string,

           column3 LONG)

STORED BY 'carbondata'
TBLPROPERTIES('LOCAL_DICTIONARY_ENABLE'='true','LOCAL_DICTIONARY_THRESHOLD'='1000'
'LOCAL_DICTIONARY_INCLUDE'='column1','LOCAL_DICTIONARY_EXCLUDE'='column2')
```

```

NOTE:

- We recommend to use Local Dictionary when cardinality is high but is distributed across multiple loads
- On a large cluster, decoding data can become a bottleneck for global dictionary as there will be many remote reads. In this scenario, it is better to use Local Dictionary.
- When cardinality is less, but loads are repetitive, it is better to use global dictionary as local dictionary generates multiple dictionary files at blocklet level increasing redundancy.
- **Caching Min/Max Value for Required Columns** By default, CarbonData caches min and max values of all the columns in schema. As the load increases, the memory required to hold the min and max values increases considerably. This feature enables you to configure min and max values only for the required columns, resulting in optimized memory usage.

  Following are the valid values for COLUMN_META_CACHE: * If you want no column min/max values to be cached in the driver.

  `COLUMN_META_CACHE=’’`

  - If you want only col1 min/max values to be cached in the driver.

  `COLUMN_META_CACHE=’col1’`

- If you want min/max values to be cached in driver for all the specified columns.

```
COLUMN_META_CACHE=’col1,col2,col3,…’
```

Columns to be cached can be specified either while creating table or after creation of the table. During create table operation; specify the columns to be cached in table properties.

Syntax:

```
CREATE TABLE [dbName].tableName (col1 String, col2 String, col3 int,…)
STORED BY ‘carbondata’ TBLPROPERTIES (‘COLUMN_META_CACHE’=’col1,col2,
…’)
```

Example:

```
CREATE TABLE employee (name String, city String, id int) STORED BY
‘carbondata’ TBLPROPERTIES (‘COLUMN_META_CACHE’=’name’)
```

After creation of table or on already created tables use the alter table command to configure the columns to be cached.

Syntax:

```
ALTER TABLE [dbName].tableName SET TBLPROPERTIES
(‘COLUMN_META_CACHE’=’col1,col2,…’)
```

Example:

```
ALTER TABLE employee SET TBLPROPERTIES (‘COLUMN_META_CACHE’=’city’)
```

- **Caching at Block or Blocklet Level**

  This feature allows you to maintain the cache at Block level, resulting in optimized usage of the memory. The memory consumption is high if the Blocklet level caching is maintained as a Block can have multiple Blocklet.

  Following are the valid values for CACHE_LEVEL:

  Configuration for caching in driver at Block level (default value).

```
CACHE_LEVEL= ‘BLOCK’
```

  Configuration for caching in driver at Blocklet level.

```
CACHE_LEVEL= ‘BLOCKLET’
```

  Cache level can be specified either while creating table or after creation of the table. During create table operation specify the cache level in table properties.

  Syntax:

```
CREATE TABLE [dbName].tableName (col1 String, col2 String, col3 int,…)
STORED BY ‘carbondata’ TBLPROPERTIES (‘CACHE_LEVEL’=’Blocklet’)
```

  Example:

```
CREATE TABLE employee (name String, city String, id int) STORED BY
‘carbondata’ TBLPROPERTIES (‘CACHE_LEVEL’=’Blocklet’)
```

  After creation of table or on already created tables use the alter table command to configure the cache level.

  Syntax:

```
ALTER TABLE [dbName].tableName SET TBLPROPERTIES
(‘CACHE_LEVEL’=’Blocklet’)
```

Example:

```
ALTER TABLE employee SET TBLPROPERTIES (‘CACHE_LEVEL’=’Blocklet’)
```

- **Support Flat folder same as Hive/Parquet**

This feature allows all carbondata and index files to keep directy under tablepath. Currently all carbondata/carbonindex files written under tablepath/Fact/Part0/Segment_NUM folder and it is not same as hive/parquet folder structure. This feature makes all files written will be directly under tablepath, it does not maintain any segment folder structure.This is useful for interoperability between the execution engines and plugin with other execution engines like hive or presto becomes easier.

Following table property enables this feature and default value is false. `'flat_folder'='true'` Example: `CREATE TABLE employee (name String, city String, id int) STORED BY ‘carbondata’ TBLPROPERTIES ('flat_folder'='true')`

- **String longer than 32000 characters**

In common scenarios, the length of string is less than 32000, so carbondata stores the length of content using Short to reduce memory and space consumption. To support string longer than 32000 characters, carbondata introduces a table property called `LONG_STRING_COLUMNS`. For these columns, carbondata internally stores the length of content using Integer.

You can specify the columns as ‘long string column’ using below tblProperties:

```
// specify col1, col2 as long string columns TBLPROPERTIES
('LONG_STRING_COLUMNS'='col1,col2')
```

Besides, you can also use this property through DataFrame by `df.format("carbondata") .option("tableName", "carbonTable") .option("long_string_columns", "col1, col2") .save()`

If you are using Carbon-SDK, you can specify the datatype of long string column as `varchar`. You can refer to SDKwriterTestCase for example.

NOTE: The LONG_STRING_COLUMNS can only be string/char/varchar columns and cannot be dictionary_include/sort_columns/complex columns.

4.2 CREATE TABLE AS SELECT

This function allows user to create a Carbon table from any of the Parquet/Hive/Carbon table. This is beneficial when the user wants to create Carbon table from any other Parquet/Hive table and use the Carbon query engine to query and achieve better query results for cases where Carbon is faster than other file formats. Also this feature can be used for backing up the data.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name STORED BY
'carbondata' [TBLPROPERTIES (key1=val1, key2=val2, ...)] AS
select_statement;
```

4.2.1 Examples

``` carbon.sql(“CREATE TABLE source_table( id INT, name STRING, city STRING, age INT) STORED AS parquet”) carbon.sql(“INSERT INTO source_table SELECT 1,‘bob’,‘shenzhen’,27”) carbon.sql(“INSERT INTO source_table SELECT 2,‘david’,‘shenzhen’,31”)

carbon.sql(“CREATE TABLE target_table STORED BY ‘carbondata’ AS SELECT city,avg(age) FROM source_table GROUP BY city”)
```

carbon.sql(“SELECT * FROM target_table”).show // results: // +———+———+ // | city|avg(age)| // +———+———+ // |shenzhen| 29.0| // +———+———+
```

## 4.3 CREATE EXTERNAL TABLE

This function allows user to create external table by specifying location. `CREATE EXTERNAL TABLE [IF NOT EXISTS] [db_name.]table_name STORED BY 'carbondata' LOCATION ‘$FilesPath’`

### 4.3.1 Create external table on managed table data location.

Managed table data location provided will have both FACT and Metadata folder. This data can be generated by creating a normal carbon table and use this path as $FilesPath in the above syntax.

**Example:** ``` sql(“CREATE TABLE origin(key INT, value STRING) STORED BY ‘carbondata’”) sql(“INSERT INTO origin select 100,‘spark’”) sql(“INSERT INTO origin select 200,‘hive’”) // creates a table in $storeLocation/origin

sql(s""“ |CREATE EXTERNAL TABLE source |STORED BY ‘carbondata’ |LOCATION ‘$storeLocation/origin’ ”"“.stripMargin) checkAnswer(sql(”SELECT count( *) from source“), sql(”SELECT count(*) from origin")) ```

### 4.3.2 Create external table on Non-Transactional table data location.

Non-Transactional table data location will have only carbondata and carbonindex files, there will not be a metadata folder (table status and schema). Our SDK module currently support writing data in this format.

**Example:** `sql( s"""CREATE EXTERNAL TABLE sdkOutputTable STORED BY 'carbondata' LOCATION |'$writerPath' """.stripMargin)`

Here writer path will have carbondata and index files. This can be SDK output. Refer SDK Writer Guide.

**Note:** 1. Dropping of the external table should not delete the files present in the location. 2. When external table is created on non-transactional table data, external table will be registered with the schema of carbondata files. If multiple files with different schema is present, exception will be thrown. So, If table registered with one schema and files are of different schema, suggest to drop the external table and create again to register table with new schema.

## 4.4 CREATE DATABASE

This function creates a new database. By default the database is created in Carbon store location, but you can also specify custom location. `CREATE DATABASE [IF NOT EXISTS] database_name [LOCATION path];`

### 4.4.1 Example

```
CREATE DATABASE carbon LOCATION “hdfs://name_cluster/dir1/carbonstore”;
```
```

4.5 TABLE MANAGEMENT

4.5.1 SHOW TABLE

This command can be used to list all the tables in current database or all the tables of a specific database. `SHOW TABLES [IN db_Name]`

Example: `SHOW TABLES OR SHOW TABLES IN defaultdb`

4.5.2 ALTER TABLE

The following section introduce the commands to modify the physical or logical state of the existing table(s).

- **RENAME TABLE**

  This command is used to rename the existing table. `ALTER TABLE [db_name.]table_name RENAME TO new_table_name`

  Examples: `ALTER TABLE carbon RENAME TO carbonTable OR ALTER TABLE test_db.carbon RENAME TO test_db.carbonTable`

- **ADD COLUMNS**

  This command is used to add a new column to the existing table. `ALTER TABLE [db_name.]table_name ADD COLUMNS (col_name data_type,...) TBLPROPERTIES('DICTIONARY_INCLUDE'='col_name,...', 'DEFAULT.VALUE.COLUMN_NAME'='default_value')`

  Examples: `ALTER TABLE carbon ADD COLUMNS (a1 INT, b1 STRING)`

  `ALTER TABLE carbon ADD COLUMNS (a1 INT, b1 STRING) TBLPROPERTIES('DICTIONARY_INCLUDE'='a1')`

  `ALTER TABLE carbon ADD COLUMNS (a1 INT, b1 STRING) TBLPROPERTIES('DEFAULT.VALUE.a1'='10')` NOTE: Add Complex datatype columns is not supported.

Users can specify which columns to include and exclude for local dictionary
generation after adding new columns. These will be appended with the already
existing local dictionary include and exclude columns of main table respectively.
`ALTER TABLE carbon ADD COLUMNS (a1 STRING, b1 STRING) TBLPROPERTIES('LOCAL_DICTIONARY_INCLUDE'='a1','LOCAL_DICTIONARY_EXCLUDE'='b1')`

- **DROP COLUMNS**

  This command is used to delete the existing column(s) in a table. `ALTER TABLE [db_name.]table_name DROP COLUMNS (col_name, ...)`

  Examples: ``` ALTER TABLE carbon DROP COLUMNS (b1) OR ALTER TABLE test_db.carbon DROP COLUMNS (b1)

  ALTER TABLE carbon DROP COLUMNS (c1,d1) ``` NOTE: Drop Complex child column is not supported.

- **CHANGE DATA TYPE**

  This command is used to change the data type from INT to BIGINT or decimal precision from lower to higher. Change of decimal data type from lower precision to higher precision will only be supported for cases where there is no data loss. `ALTER TABLE [db_name.]table_name CHANGE col_name col_name changed_column_type`

  Valid Scenarios - Invalid scenario - Change of decimal precision from (10,2) to (10,5) is invalid as in this case only scale is increased but total number of digits remains the same. - Valid scenario - Change of decimal precision from (10,2) to (12,3) is valid as the total number of digits are increased by 2 but scale is increased only by 1 which will not lead to any data loss. - **NOTE:** The allowed range is 38,38 (precision, scale) and is a valid upper case scenario which is not resulting in data loss.

Example1:Changing data type of column a1 from INT to BIGINT. `ALTER TABLE test_db.carbon CHANGE a1 a1 BIGINT`

Example2:Changing decimal precision of column a1 from 10 to 18. `ALTER TABLE test_db.carbon CHANGE a1 a1 DECIMAL(18,2)`

- **MERGE INDEX**

  This command is used to merge all the CarbonData index files (.carbonindex) inside a segment to a single CarbonData index merge file (.carbonindexmerge). This enhances the first query performance. `ALTER TABLE [db_name.]table_name COMPACT 'SEGMENT_INDEX'`

  Examples: `ALTER TABLE test_db.carbon COMPACT 'SEGMENT_INDEX'` **NOTE:** * Merge index is not supported on streaming table.

- **SET and UNSET for Local Dictionary Properties**

When set command is used, all the newly set properties will override the corresponding old properties if exists.

Example to SET Local Dictionary Properties: `ALTER TABLE tablename SET TBLPROPERTIES('LOCAL_DICTIONARY_ENABLE'='false','LOCAL_DICTIONARY_THRESHOLD'='1000','LO` When Local Dictionary properties are unset, corresponding default values will be used for these properties.

Example to UNSET Local Dictionary Properties: `ALTER TABLE tablename UNSET TBLPROPERTIES('LOCAL_DICTIONARY_ENABLE','LOCAL_DICTIONARY_THRESHOLD','LOCAL_DICTIONARY_`

NOTE: For old tables, by default, local dictionary is disabled. If user wants local dictionary for these tables, user can enable/disable local dictionary for new data at their discretion. This can be achieved by using the alter table set command.

4.5.3 DROP TABLE

This command is used to delete an existing table. `DROP TABLE [IF EXISTS] [db_name.]table_name`

Example: `DROP TABLE IF EXISTS productSchema.productSalesTable`

4.5.4 REFRESH TABLE

This command is used to register Carbon table to HIVE meta store catalogue from existing Carbon table data. `REFRESH TABLE $db_NAME.$table_NAME`

Example: `REFRESH TABLE dbcarbon.productSalesTable`

NOTE: * The new database name and the old database name should be same. * Before executing this command the old table schema and data should be copied into the new database location. * If the table is aggregate table, then all the aggregate tables should be copied to the new database location. * For old store, the time zone of the source and destination cluster should be same. * If old cluster used HIVE meta store to store schema, refresh will not work as schema file does not exist in file system.

4.5.5 Table and Column Comment

You can provide more information on table by using table comment. Similarly you can provide more information about a particular column using column comment. You can see the column comment of an existing table using describe formatted command.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name[(col_name data_type
[COMMENT col_comment], ...)] [COMMENT table_comment] STORED BY
'carbondata' [TBLPROPERTIES (property_name=property_value, ...)]
```

Example: `CREATE TABLE IF NOT EXISTS productSchema.productSalesTable ( productNumber Int COMMENT 'unique serial number for product') COMMENT “This is table comment” STORED BY 'carbondata' TBLPROPERTIES ('DICTIONARY_INCLUDE'='productNumber')` You can also SET and UNSET table comment using ALTER command.

Example to SET table comment:

```
ALTER TABLE carbon SET TBLPROPERTIES ('comment'='this table comment is
modified');
```

Example to UNSET table comment:

```
ALTER TABLE carbon UNSET TBLPROPERTIES ('comment');
```

4.6 LOAD DATA

4.6.1 LOAD FILES TO CARBONDATA TABLE

This command is used to load csv files to carbondata, OPTIONS are not mandatory for data loading process. Inside OPTIONS user can provide any options like DELIMITER, QUOTECHAR, FILEHEADER, ESCAPECHAR, MULTILINE as per requirement.

```
LOAD DATA [LOCAL] INPATH 'folder_path' INTO TABLE [db_name.]table_name
OPTIONS(property_name=property_value, ...)
```

You can use the following options to load data:

- **DELIMITER:** Delimiters can be provided in the load command.

  ```
  OPTIONS('DELIMITER'=',')
  ```

- **QUOTECHAR:** Quote Characters can be provided in the load command.

  ```
  OPTIONS('QUOTECHAR'='"')
  ```

- **COMMENTCHAR:** Comment Characters can be provided in the load command if user want to comment lines.

  ```
  OPTIONS('COMMENTCHAR'='#')
  ```

- **HEADER:** When you load the CSV file without the file header and the file header is the same with the table schema, then add ‘HEADER’=‘false’ to load data SQL as user need not provide the file header. By default the value is ‘true’. false: CSV file is without file header. true: CSV file is with file header.

```
OPTIONS('HEADER'='false')
```

NOTE: If the HEADER option exist and is set to ‘true’, then the FILEHEADER option is not required.

- **FILEHEADER:** Headers can be provided in the LOAD DATA command if headers are missing in the source files.

```
OPTIONS('FILEHEADER'='column1,column2')
```

- **MULTILINE:** CSV with new line character in quotes.

```
OPTIONS('MULTILINE'='true')
```

- **ESCAPECHAR:** Escape char can be provided if user want strict validation of escape character in CSV files.

```
OPTIONS('ESCAPECHAR'='\')
```

- **SKIP_EMPTY_LINE:** This option will ignore the empty line in the CSV file during the data load.

```
OPTIONS('SKIP_EMPTY_LINE'='TRUE/FALSE')
```

- **COMPLEX_DELIMITER_LEVEL_1:** Split the complex type data column in a row (eg., a$b $c –> Array = {a,b,c}).

```
OPTIONS('COMPLEX_DELIMITER_LEVEL_1'='$')
```

- **COMPLEX_DELIMITER_LEVEL_2:** Split the complex type nested data column in a row. Applies level_1 delimiter & applies level_2 based on complex data type (eg., a:b$c:d –> Array> = {{a,b},{c,d}}).

```
OPTIONS('COMPLEX_DELIMITER_LEVEL_2'=':')
```

- **ALL_DICTIONARY_PATH:** All dictionary files path.

```
OPTIONS('ALL_DICTIONARY_PATH'='/opt/alldictionary/data.dictionary')
```

- **COLUMNDICT:** Dictionary file path for specified column.

```
OPTIONS('COLUMNDICT'='column1:dictionaryFilePath1,column2:dictionaryFilePath2')
```

NOTE: ALL_DICTIONARY_PATH and COLUMNDICT can’t be used together.

- **DATEFORMAT/TIMESTAMPFORMAT:** Date and Timestamp format for specified column.

```
OPTIONS('DATEFORMAT' = 'yyyy-MM-dd','TIMESTAMPFORMAT'='yyyy-MM-dd HH:mm:ss')
```

NOTE: Date formats are specified by date pattern strings. The date pattern letters in CarbonData are same as in JAVA. Refer to SimpleDateFormat.

- **SORT COLUMN BOUNDS:** Range bounds for sort columns.

  Suppose the table is created with 'SORT_COLUMNS'='name,id' and the range for name is aaa~zzz, the value range for id is 0~1000. Then during data loading, we can specify the following option to enhance data loading performance. `OPTIONS('SORT_COLUMN_BOUNDS'='f,250;l,500;r,750')` Each bound is separated by ';' and each field value in bound is separated by ','. In the example above, we provide 3 bounds to distribute records to 4 partitions. The values 'f','l','r' can evenly distribute the records. Inside carbondata, for a record we compare the value of sort columns with that of the bounds and decide which partition the record will be forwarded to.

  NOTE: * SORT_COLUMN_BOUNDS will be used only when the SORT_SCOPE is 'local_sort'. * Carbondata will use these bounds as ranges to process data concurrently during the final sort percedure. The records will be sorted and written out inside each partition. Since the partition is sorted, all records will be sorted. * Since the actual order and literal order of the dictionary column are not necessarily the same, we do not recommend you to use this feature if the first sort column is 'dictionary_include'. * The option works better if your CPU usage during loading is low. If your system is already CPU tense, better not to use this option. Besides, it depends on the user to specify the bounds. If user does not know the exactly bounds to make the data distributed evenly among the bounds, loading performance will still be better than before or at least the same as before. * Users can find more information about this option in the description of PR1953.

- **SINGLE_PASS:** Single Pass Loading enables single job to finish data loading with dictionary generation on the fly. It enhances performance in the scenarios where the subsequent data loading after initial load involves fewer incremental updates on the dictionary.

This option specifies whether to use single pass for loading data or not. By default this option is set to FALSE.

```
OPTIONS('SINGLE_PASS'='TRUE')
```

NOTE: * If this option is set to TRUE then data loading will take less time. * If this option is set to some invalid value other than TRUE or FALSE then it uses the default value.

Example:

```
LOAD DATA local inpath '/opt/rawdata/data.csv'
INTO table carbontable options('DELIMITER'=',',
'QUOTECHAR'='"','COMMENTCHAR'='#', 'HEADER'='false',
'FILEHEADER'='empno,empname,designation,doj,workgroupcategory,
workgroupcategoryname,deptno,deptname,projectcode,
projectjoindate,projectenddate,attendance,utilization,salary',
'MULTILINE'='true','ESCAPECHAR'='\','COMPLEX_DELIMITER_LEVEL_1'='$',
'COMPLEX_DELIMITER_LEVEL_2'=':', 'ALL_DICTIONARY_PATH'='/opt/alldictionary/
data.dictionary', 'SINGLE_PASS'='TRUE')
```

- **BAD RECORDS HANDLING:** Methods of handling bad records are as follows:

  - Load all of the data before dealing with the errors.
  - Clean or delete bad records before loading data or stop the loading when bad records are found.

```
OPTIONS('BAD_RECORDS_LOGGER_ENABLE'='true', 'BAD_RECORD_PATH'='hdfs://hacluster/
```

NOTE: * BAD_RECORDS_ACTION property can have four type of actions for bad records FORCE, REDIRECT, IGNORE and FAIL. * FAIL option is its Default value. If the FAIL option is used, then data loading fails if any bad records are found. * If the REDIRECT option is used, CarbonData will add all bad records in to a separate CSV file. However, this file must not be used for subsequent data loading because the content may not exactly match the source record. You are advised to cleanse the original source record for further data ingestion. This option is used to remind you which records are bad records. * If the FORCE option is used, then it auto-converts the data by storing the bad records as NULL before Loading data. * If the IGNORE option is used, then bad records are neither loaded nor written to the separate CSV file. * In loaded data, if all records are bad records, the BAD_RECORDS_ACTION is invalid and the load operation fails. * The default maximum number of characters per column is 32000. If there are more than 32000 characters in a column, please refer to *String longer than 32000 characters* section. * Since Bad Records Path can be specified in create, load and carbon properties. Therefore, value specified in load will have the highest priority, and value specified in carbon properties will have the least priority.

Bad Records Path:

This property is used to specify the location where bad records would be written.

```
TBLPROPERTIES('BAD_RECORDS_PATH'='/opt/badrecords'')
```

Example:

```
LOAD DATA INPATH 'filepath.csv' INTO TABLE tablename
OPTIONS('BAD_RECORDS_LOGGER_ENABLE'='true','BAD_RECORD_PATH'='hdfs://
hacluster/tmp/carbon',
'BAD_RECORDS_ACTION'='REDIRECT','IS_EMPTY_DATA_BAD_RECORD'='false')
```

- **GLOBAL_SORT_PARTITIONS:** If the SORT_SCOPE is defined as GLOBAL_SORT, then user can specify the number of partitions to use while shuffling data for sort using GLOBAL_SORT_PARTITIONS. If it is not configured, or configured less than 1, then it uses the number of map task as reduce task. It is recommended that each reduce task deal with 512MB-1GB data.

```
OPTIONS('GLOBAL_SORT_PARTITIONS'='2')
```

NOTE: * GLOBAL_SORT_PARTITIONS should be Integer type, the range is [1,Integer.MaxValue]. * It is only used when the SORT_SCOPE is GLOBAL_SORT.

4.6.2 INSERT DATA INTO CARBONDATA TABLE

This command inserts data into a CarbonData table, it is defined as a combination of two queries Insert and Select query respectively. It inserts records from a source table into a target CarbonData table, the source table can be a Hive table, Parquet table or a CarbonData table itself. It comes with the functionality to aggregate the records of a table by performing Select query on source table and load its corresponding resultant records into a CarbonData table.

```
INSERT INTO TABLE <CARBONDATA TABLE> SELECT * FROM sourceTableName [ WHERE
{ <filter_condition> } ]
```

You can also omit the `table` keyword and write your query as:

```
INSERT INTO <CARBONDATA TABLE> SELECT * FROM sourceTableName [ WHERE
{ <filter_condition> } ]
```

Overwrite insert data: `INSERT OVERWRITE TABLE <CARBONDATA TABLE> SELECT * FROM sourceTableName [ WHERE { <filter_condition> } ]`

NOTE: * The source table and the CarbonData table must have the same table schema. * The data type of source and destination table columns should be same * INSERT INTO command does not support partial success if bad records are found, it will fail. * Data cannot be loaded or updated in source table while insert from source table to target table is in progress.

Examples `INSERT INTO table1 SELECT item1, sum(item2 + 1000) as result FROM table2 group by item1`

```
INSERT INTO table1 SELECT item1, item2, item3 FROM table2 where item2='xyz'
```

```
INSERT OVERWRITE TABLE table1 SELECT * FROM TABLE2
```

4.7 UPDATE AND DELETE

4.7.1 UPDATE

This command will allow to update the CarbonData table based on the column expression and optional filter conditions.

```
UPDATE <table_name> SET (column_name1, column_name2, ... column_name n) =
(column1_expression , column2_expression, ... column n_expression ) [ WHERE
{ <filter_condition> } ]
```

alternatively the following command can also be used for updating the CarbonData Table :

```
UPDATE <table_name> SET (column_name1, column_name2) =(select
sourceColumn1, sourceColumn2 from sourceTable [ WHERE
{ <filter_condition> } ] ) [ WHERE { <filter_condition> } ]
```

NOTE: The update command fails if multiple input rows in source table are matched with single row in destination table.

Examples: `UPDATE t3 SET (t3_salary) = (t3_salary + 9) WHERE t3_name = 'aaa1'`

```
UPDATE t3 SET (t3_date, t3_country) = ('2017-11-18', 'india') WHERE
t3_salary < 15003
```

```
UPDATE t3 SET (t3_country, t3_name) = (SELECT t5_country, t5_name FROM t5
WHERE t5_id = 5) WHERE t3_id < 5
```

```
UPDATE t3 SET (t3_date, t3_serialname, t3_salary) = (SELECT '2099-09-09',
t5_serialname, '9999' FROM t5 WHERE t5_id = 5) WHERE t3_id < 5
```

`UPDATE t3 SET (t3_country, t3_salary) = (SELECT t5_country, t5_salary FROM t5 FULL JOIN t3 u WHERE u.t3_id = t5_id and t5_id=6) WHERE t3_id >6` NOTE: Update Complex datatype columns is not supported.

4.7.2 DELETE

This command allows us to delete records from CarbonData table. `DELETE FROM table_name [WHERE expression]`

Examples:

```
DELETE FROM carbontable WHERE column1 = 'china'
```

```
DELETE FROM carbontable WHERE column1 IN ('china', 'USA')
```

```
DELETE FROM carbontable WHERE column1 IN (SELECT column11 FROM
sourceTable2)
```

```
DELETE FROM carbontable WHERE column1 IN (SELECT column11 FROM sourceTable2
WHERE column1 = 'USA')
```

4.8 COMPACTION

Compaction improves the query performance significantly.

There are several types of compaction.

```
ALTER TABLE [db_name.]table_name COMPACT 'MINOR/MAJOR/CUSTOM'
```

- **Minor Compaction**

In Minor compaction, user can specify the number of loads to be merged. Minor compaction triggers for every data load if the parameter carbon.enable.auto.load.merge is set to true. If any segments are available to be merged, then compaction will run parallel with data load, there are 2 levels in minor compaction: * Level 1: Merging of the segments which are not yet compacted. * Level 2: Merging of the compacted segments again to form a larger segment.

```
ALTER TABLE table_name COMPACT 'MINOR'
```

- **Major Compaction**

In Major compaction, multiple segments can be merged into one large segment. User will specify the compaction size until which segments can be merged, Major compaction is usually done during the off-peak time. Configure the property carbon.major.compaction.size with appropriate value in MB.

This command merges the specified number of segments into one segment:

```
ALTER TABLE table_name COMPACT 'MAJOR'
```

- **Custom Compaction**

In Custom compaction, user can directly specify segment ids to be merged into one large segment. All specified segment ids should exist and be valid, otherwise compaction will fail. Custom compaction is usually done during the off-peak time.

`ALTER TABLE table_name COMPACT 'CUSTOM' WHERE SEGMENT.ID IN (2,3,4)` NOTE: Compaction is unsupported for table containing Complex columns.

- **CLEAN SEGMENTS AFTER Compaction**

Clean the segments which are compacted: `CLEAN FILES FOR TABLE carbon_table`

4.9 PARTITION

4.9.1 STANDARD PARTITION

The partition is similar as spark and hive partition, user can use any column to build partition:

4.9.1.1 Create Partition Table

This command allows you to create table with partition.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name [(col_name
data_type , ...)] [COMMENT table_comment] [PARTITIONED BY
(col_name data_type , ...)] [STORED BY file_format] [TBLPROPERTIES
(property_name=property_value, ...)]
```

Example: `CREATE TABLE IF NOT EXISTS productSchema.productSalesTable ( productNumber INT, productName STRING, storeCity STRING, storeProvince STRING, saleQuantity INT, revenue INT) PARTITIONED BY (productCategory STRING, productBatch STRING) STORED BY 'carbondata'` NOTE: Hive partition is not supported on complex datatype columns.

4.9.1.2 Load Data Using Static Partition

This command allows you to load data using static partition.

```
LOAD DATA [LOCAL] INPATH 'folder_path' INTO TABLE [db_name.]table_name
PARTITION (partition_spec) OPTIONS(property_name=property_value, ...)
INSERT INTO INTO TABLE [db_name.]table_name PARTITION (partition_spec)
<SELECT STATEMENT>
```

Example: `LOAD DATA LOCAL INPATH '${env:HOME}/staticinput.csv' INTO TABLE locationTable PARTITION (country = 'US', state = 'CA') INSERT INTO TABLE locationTable PARTITION (country = 'US', state = 'AL') SELECT <columns list excluding partition columns> FROM another_user`

4.9.1.3 Load Data Using Dynamic Partition

This command allows you to load data using dynamic partition. If partition spec is not specified, then the partition is considered as dynamic.

Example: `LOAD DATA LOCAL INPATH '${env:HOME}/staticinput.csv' INTO TABLE locationTable INSERT INTO TABLE locationTable SELECT <columns list excluding partition columns> FROM another_user`

4.9.1.4 Show Partitions

This command gets the Hive partition information of the table

```
SHOW PARTITIONS [db_name.]table_name
```

4.9.1.5 Drop Partition

This command drops the specified Hive partition only. `ALTER TABLE table_name DROP [IF EXISTS] PARTITION (part_spec, ...)`

Example: `ALTER TABLE locationTable DROP PARTITION (country = 'US');`

4.9.1.6 Insert OVERWRITE

This command allows you to insert or load overwrite on a specific partition.

```
INSERT OVERWRITE TABLE table_name PARTITION (column = 'partition_name')
select_statement
```

Example: `INSERT OVERWRITE TABLE partitioned_user PARTITION (country = 'US') SELECT * FROM another_user au WHERE au.country = 'US';`

4.9.2 CARBONDATA PARTITION(HASH,RANGE,LIST) – Alpha feature, this partition feature does not support update and delete data.

The partition supports three type:(Hash,Range,List), similar to other system’s partition features, CarbonData’s partition feature can be used to improve query performance by filtering on the partition column.

4.9.3 Create Hash Partition Table

This command allows us to create hash partition.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name [(col_name
data_type , ...)] PARTITIONED BY (partition_col_name data_type)
STORED BY 'carbondata' [TBLPROPERTIES ('PARTITION_TYPE'='HASH',
'NUM_PARTITIONS'='N' ...)]
```
NOTE: N is the number of hash partitions

Example: `CREATE TABLE IF NOT EXISTS hash_partition_table( col_A STRING, col_B INT, col_C LONG, col_D DECIMAL(10,2), col_F`

```
TIMESTAMP ) PARTITIONED BY (col_E LONG) STORED BY 'carbondata'
TBLPROPERTIES('PARTITION_TYPE'='HASH','NUM_PARTITIONS'='9')
```

4.9.4 Create Range Partition Table

This command allows us to create range partition. `CREATE TABLE [IF NOT EXISTS] [db_name.]table_name [(col_name data_type , ...)] PARTITIONED BY (partition_col_name data_type) STORED BY 'carbondata' [TBLPROPERTIES ('PARTITION_TYPE'='RANGE', 'RANGE_INFO'='2014-01-01, 2015-01-01, 2016-01-01, ...')]`

NOTE: * The 'RANGE_INFO' must be defined in ascending order in the table properties. * The default format for partition column of Date/Timestamp type is yyyy-MM-dd. Alternate formats for Date/Timestamp could be defined in CarbonProperties.

Example: `CREATE TABLE IF NOT EXISTS range_partition_table( col_A STRING, col_B INT, col_C LONG, col_D DECIMAL(10,2), col_E LONG ) partitioned by (col_F Timestamp) PARTITIONED BY 'carbondata' TBLPROPERTIES('PARTITION_TYPE'='RANGE', 'RANGE_INFO'='2015-01-01, 2016-01-01, 2017-01-01, 2017-02-01')`

4.9.5 Create List Partition Table

This command allows us to create list partition. `CREATE TABLE [IF NOT EXISTS] [db_name.]table_name [(col_name data_type , ...)] PARTITIONED BY (partition_col_name data_type) STORED BY 'carbondata' [TBLPROPERTIES ('PARTITION_TYPE'='LIST', 'LIST_INFO'='A, B, C, ...')]` **NOTE:** List partition supports list info in one level group.

Example: `CREATE TABLE IF NOT EXISTS list_partition_table( col_B INT, col_C LONG, col_D DECIMAL(10,2), col_E LONG, col_F TIMESTAMP ) PARTITIONED BY (col_A STRING) STORED BY 'carbondata' TBLPROPERTIES('PARTITION_TYPE'='LIST', 'LIST_INFO'='aaaa, bbbb, (cccc, dddd), eeee')`

4.9.6 Show Partitions

The following command is executed to get the partition information of the table

```
SHOW PARTITIONS [db_name.]table_name
```

4.9.7 Add a new partition

```
ALTER TABLE [db_name].table_name ADD PARTITION('new_partition')
```

4.9.8 Split a partition

```
ALTER TABLE [db_name].table_name SPLIT PARTITION(partition_id)
INTO('new_partition1', 'new_partition2'...)
```

4.9.9 Drop a partition

Only drop partition definition, but keep data `ALTER TABLE [db_name].table_name DROP PARTITION(partition_id)`

Drop both partition definition and data `ALTER TABLE [db_name].table_name DROP PARTITION(partition_id) WITH DATA`

NOTE: * Hash partition table is not supported for ADD, SPLIT and DROP commands. * Partition Id: in CarbonData like the hive, folders are not used to divide partitions instead partition id is used to replace the task id. It could make use of the characteristic and meanwhile reduce some metadata.

```
SegmentDir/0_batchno0-0-1502703086921.carbonindex ^ SegmentDir/
part-0-0_batchno0-0-1502703086921.carbondata ^
```

Here are some useful tips to improve query performance of carbonData partition table: * The partitioned column can be excluded from SORT_COLUMNS, this will let other columns to do the efficient sorting. * When writing SQL on a partition table, try to use filters on the partition column.

4.10 BUCKETING

Bucketing feature can be used to distribute/organize the table/partition data into multiple files such that similar records are present in the same file. While creating a table, user needs to specify the columns to be used for bucketing and the number of buckets. For the selection of bucket the Hash value of columns is used.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
[(col_name data_type, ...)] STORED BY 'carbondata'
TBLPROPERTIES('BUCKETNUMBER'='noOfBuckets', 'BUCKETCOLUMNS'='columnname')
```

NOTE: * Bucketing cannot be performed for columns of Complex Data Types. * Columns in the BUCKETCOLUMN parameter must be dimensions. The BUCKETCOLUMN parameter cannot be a measure or a combination of measures and dimensions.

Example: `CREATE TABLE IF NOT EXISTS productSchema.productSalesTable (productNumber INT, saleQuantity INT, productName STRING, storeCity STRING, storeProvince STRING, productCategory STRING, productBatch STRING, revenue INT) STORED BY 'carbondata' TBLPROPERTIES ('BUCKETNUMBER'='4', 'BUCKETCOLUMNS'='productName')`

4.11 SEGMENT MANAGEMENT

4.11.1 SHOW SEGMENT

This command is used to list the segments of CarbonData table.

```
SHOW [HISTORY] SEGMENTS FOR TABLE [db_name.]table_name LIMIT
number_of_segments
```

Example: Show visible segments `SHOW SEGMENTS FOR TABLE CarbonDatabase.CarbonTable LIMIT 4` Show all segments, include invisible segments `SHOW HISTORY SEGMENTS FOR TABLE CarbonDatabase.CarbonTable LIMIT 4`

4.11.2 DELETE SEGMENT BY ID

This command is used to delete segment by using the segment ID. Each segment has a unique segment ID associated with it. Using this segment ID, you can remove the segment.

The following command will get the segmentID.

```
SHOW SEGMENTS FOR TABLE [db_name.]table_name LIMIT number_of_segments
```

After you retrieve the segment ID of the segment that you want to delete, execute the following command to delete the selected segment.

```
DELETE FROM TABLE [db_name.]table_name WHERE SEGMENT.ID IN (segment_id1,
segments_id2, ...)
```

Example:

```
DELETE FROM TABLE CarbonDatabase.CarbonTable WHERE SEGMENT.ID IN (0) DELETE
FROM TABLE CarbonDatabase.CarbonTable WHERE SEGMENT.ID IN (0,5,8)
```

4.11.3 DELETE SEGMENT BY DATE

This command will allow to delete the CarbonData segment(s) from the store based on the date provided by the user in the DML command. The segment created before the particular date will be removed from the specific stores.

```
DELETE FROM TABLE [db_name.]table_name WHERE SEGMENT.STARTTIME BEFORE
DATE_VALUE
```

Example: `DELETE FROM TABLE CarbonDatabase.CarbonTable WHERE SEGMENT.STARTTIME BEFORE '2017-06-01 12:05:06'`

4.11.4 QUERY DATA WITH SPECIFIED SEGMENTS

This command is used to read data from specified segments during CarbonScan.

Get the Segment ID: `SHOW SEGMENTS FOR TABLE [db_name.]table_name LIMIT number_of_segments`

Set the segment IDs for table `SET carbon.input.segments.<database_name>.<table_name> = <list of segment IDs>`

NOTE: carbon.input.segments: Specifies the segment IDs to be queried. This property allows you to query specified segments of the specified table. The CarbonScan will read data from specified segments only.

If user wants to query with segments reading in multi threading mode, then CarbonSession. threadSet can be used instead of SET query. `CarbonSession.threadSet ("carbon.input.segments.<database_name>.<table_name>","<list of segment IDs>");`

Reset the segment IDs `SET carbon.input.segments.<database_name>.<table_name> = *;`

If user wants to query with segments reading in multi threading mode, then CarbonSession. threadSet can be used instead of SET query. `CarbonSession.threadSet ("carbon.input.segments.<database_name>.<table_name>","*");`

Examples:

- Example to show the list of segment IDs,segment status, and other required details and then specify the list of segments to be read.

``` SHOW SEGMENTS FOR carbontable1;

SET carbon.input.segments.db.carbontable1 = 1,3,9; ```

- Example to query with segments reading in multi threading mode:

```
CarbonSession.threadSet
("carbon.input.segments.db.carbontable_Multi_Thread","1,3");
```

- Example for threadset in multithread environment (following shows how it is used in Scala code):

```
def main(args: Array[String]) { Future { CarbonSession.threadSet
("carbon.input.segments.db.carbontable_Multi_Thread","1") spark.sql("select
```
```

```
count(empno) from
carbon.input.segments.db.carbontable_Multi_Thread").show(); } }
```

5 Installation

Installation Guide

This tutorial guides you through the installation and configuration of CarbonData in the following two modes :

- Installing and Configuring CarbonData on Standalone Spark Cluster
- Installing and Configuring CarbonData on Spark on YARN Cluster

followed by :

- Query Execution using CarbonData Thrift Server

5.1 Installing and Configuring CarbonData on Standalone Spark Cluster

5.1.1 Prerequisites

- Hadoop HDFS and Yarn should be installed and running.
- Spark should be installed and running on all the cluster nodes.
- CarbonData user should have permission to access HDFS.

5.1.2 Procedure

1. Build the CarbonData project and get the assembly jar from `./assembly/target/scala-2.1x/carbondata_xxx.jar`.
2. Copy `./assembly/target/scala-2.1x/carbondata_xxx.jar` to `$SPARK_HOME/carbonlib` folder.

   NOTE: Create the carbonlib folder if it does not exist inside `$SPARK_HOME` path.
3. Add the carbonlib folder path in the Spark classpath. (Edit `$SPARK_HOME/conf/spark-env.sh` file and modify the value of `SPARK_CLASSPATH` by appending `$SPARK_HOME/carbonlib/*` to the existing value)
4. Copy the `./conf/carbon.properties.template` file from CarbonData repository to `$SPARK_HOME/conf/` folder and rename the file to `carbon.properties`.
5. Repeat Step 2 to Step 5 in all the nodes of the cluster.
6. In Spark node[master], configure the properties mentioned in the following table in `$SPARK_HOME/conf/spark-defaults.conf` file.

Property	Value	Description
spark.driver.extraJavaOptions	`-Dcarbon.properties.filepa = $SPARK_HOME/conf/carbon.properties`	A string of extra JVM options to pass to the driver. For instance, GC settings or other logging.
spark.executor.extraJavaOptions	`-Dcarbon.properties.filepa = $SPARK_HOME/conf/carbon.properties`	A string of extra JVM options to pass to executors. For instance, GC settings or other logging. **NOTE**: You can enter multiple values separated by space.

1. Add the following properties in `$SPARK_HOME/conf/carbon.properties` file:

Property	Required	Description	Example	Remark

carbon.storelocation	NO	Location where data CarbonData will create the store and write the data in its own format. If not specified then it takes spark.sql.warehouse.c path.	hdfs:// HOSTNAME:PORT/ Opt/CarbonStore	Propose to set HDFS directory

1. Verify the installation. For example:

```
./spark-shell --master spark://HOSTNAME:PORT --total-executor-cores 2
--executor-memory 2G
```

NOTE: Make sure you have permissions for CarbonData JARs and files through which driver and executor will start.

To get started with CarbonData : Quick Start, Data Management on CarbonData

5.2 Installing and Configuring CarbonData on Spark on YARN Cluster

This section provides the procedure to install CarbonData on “Spark on YARN” cluster.

5.2.1 Prerequisites

- Hadoop HDFS and Yarn should be installed and running.
- Spark should be installed and running in all the clients.
- CarbonData user should have permission to access HDFS.

5.2.2 Procedure

The following steps are only for Driver Nodes. (Driver nodes are the one which starts the spark context.)

1. Build the CarbonData project and get the assembly jar from `./assembly/target/scala-2.1x/carbondata_xxx.jar` and copy to `$SPARK_HOME/carbonlib` folder.

   NOTE: Create the carbonlib folder if it does not exists inside `$SPARK_HOME` path.
2. Copy the `./conf/carbon.properties.template` file from CarbonData repository to `$SPARK_HOME/conf/` folder and rename the file to `carbon.properties`.
3. Create `tar.gz` file of carbonlib folder and move it inside the carbonlib folder.

```
cd $SPARK_HOME
tar -zcvf carbondata.tar.gz carbonlib/
mv carbondata.tar.gz carbonlib/
```

1. Configure the properties mentioned in the following table in `$SPARK_HOME/conf/spark-defaults.conf` file.

Property	Description	Value

spark.master	Set this value to run the Spark in yarn cluster mode.	Set yarn-client to run the Spark in yarn cluster mode.
spark.yarn.dist.files	Comma-separated list of files to be placed in the working directory of each executor.	`$SPARK_HOME/conf/carbon.properties`
spark.yarn.dist.archives	Comma-separated list of archives to be extracted into the working directory of each executor.	`$SPARK_HOME/carbonlib/carbondata.tar.gz`
spark.executor.extraJavaOptions	A string of extra JVM options to pass to executors. For instance **NOTE**: You can enter multiple values separated by space.	`-Dcarbon.properties.filepath = carbon.properties`
spark.executor.extraClassPath	Extra classpath entries to prepend to the classpath of executors. **NOTE**: If SPARK_CLASSPATH is defined in spark-env.sh, then comment it and append the values in below parameter spark.driver.extraClassPath	`carbondata.tar.gz/carbonlib/*`
spark.driver.extraClassPath	Extra classpath entries to prepend to the classpath of the driver. **NOTE**: If SPARK_CLASSPATH is defined in spark-env.sh, then comment it and append the value in below parameter spark.driver.extraClassPath.	`$SPARK_HOME/carbonlib/*`
spark.driver.extraJavaOptions	A string of extra JVM options to pass to the driver. For instance, GC settings or other logging.	`-Dcarbon.properties.filepath = $SPARK_HOME/conf/carbon.properties`

1. Add the following properties in `$SPARK_HOME/conf/carbon.properties`:

Property	Required	Description	Example	Default Value
carbon.storelocation	NO	Location where CarbonData will create the store and write the data in its own format. If not specified then it takes spark.sql.warehouse.c path.	hdfs://HOSTNAME:PORT/Opt/CarbonStore	Propose to set HDFS directory

1. Verify the installation.

```
./bin/spark-shell --master yarn-client --driver-memory 1g
--executor-cores 2 --executor-memory 2G
```

NOTE: Make sure you have permissions for CarbonData JARs and files through which driver and executor will start.

Getting started with CarbonData : Quick Start, Data Management on CarbonData

5.3 Query Execution Using CarbonData Thrift Server

5.3.1 Starting CarbonData Thrift Server.

a. cd `$SPARK_HOME`

b. Run the following command to start the CarbonData thrift server.

```
./bin/spark-submit
--class org.apache.carbondata.spark.thriftserver.CarbonThriftServer
$SPARK_HOME/carbonlib/$CARBON_ASSEMBLY_JAR <carbon_store_path>
```

Parameter	Description	Example
CARBON_ASSEMBLY_JAR	CarbonData assembly jar name present in the `$SPARK_HOME/carbonlib/` folder.	carbondata_2.xx-x.x.x-SNAPSHOT-shade-hadoop2.7.2.jar
carbon_store_path	This is a parameter to the CarbonThriftServer class. This a HDFS path where CarbonData files will be kept. Strongly Recommended to put same as carbon.storelocation parameter of carbon.properties. If not specified then it takes spark.sql.warehouse.dir path.	`hdfs://<host_name>:port/user/hive/warehouse/carbon.store`

NOTE: From Spark 1.6, by default the Thrift server runs in multi-session mode. Which means each JDBC/ODBC connection owns a copy of their own SQL configuration and temporary function registry. Cached tables are still shared though. If you prefer to run the Thrift server in single-session mode and share all SQL configuration and temporary function registry, please set option `spark.sql.hive.thriftServer.singleSession` to `true`. You may either add this option to `spark-defaults.conf`, or pass it to `spark-submit.sh` via `--conf`:

```
./bin/spark-submit
--conf spark.sql.hive.thriftServer.singleSession=true
--class org.apache.carbondata.spark.thriftserver.CarbonThriftServer
$SPARK_HOME/carbonlib/$CARBON_ASSEMBLY_JAR <carbon_store_path>
```

But in single-session mode, if one user changes the database from one connection, the database of the other connections will be changed too.

Examples

- Start with default memory and executors.

```
./bin/spark-submit
--class org.apache.carbondata.spark.thriftserver.CarbonThriftServer
$SPARK_HOME/carbonlib
/carbondata_2.xx-x.x.x-SNAPSHOT-shade-hadoop2.7.2.jar
hdfs://<host_name>:port/user/hive/warehouse/carbon.store
```

- Start with Fixed executors and resources.

```
./bin/spark-submit
--class org.apache.carbondata.spark.thriftserver.CarbonThriftServer
--num-executors 3 --driver-memory 20g --executor-memory 250g
--executor-cores 32
/srv/OSCON/BigData/HACluster/install/spark/sparkJdbc/lib
/carbondata_2.xx-x.x.x-SNAPSHOT-shade-hadoop2.7.2.jar
hdfs://<host_name>:port/user/hive/warehouse/carbon.store
```

5.3.2 Connecting to CarbonData Thrift Server Using Beeline.

```
     cd $SPARK_HOME
     ./sbin/start-thriftserver.sh
     ./bin/beeline -u jdbc:hive2://<thriftserver_host>:port

     Example
     ./bin/beeline -u jdbc:hive2://10.10.10.10:10000
```

6 Configuring CarbonData

Configuring CarbonData

This tutorial guides you through the advanced configurations of CarbonData :

- System Configuration
- Performance Configuration
- Miscellaneous Configuration
- Spark Configuration
- Dynamic Configuration In CarbonData Using SET-RESET

6.1 System Configuration

This section provides the details of all the configurations required for the CarbonData System.

System Configuration in carbon.properties

Property	Default Value	Description
carbon.storelocation		Location where CarbonData will create the store, and write the data in its own format. If not specified then it takes spark.sql.warehouse.dir path. NOTE: Store location should be in HDFS.
carbon.ddl.base.hdfs.url		This property is used to configure the HDFS relative path, the path configured in carbon.ddl.base.hdfs.url will be appended to the HDFS path configured in fs.defaultFS. If this path is configured, then user need not pass the complete path while dataload. For example: If absolute path of the csv file is hdfs://10.18.101.155:54310/data/cnbc/2016/xyz.csv, the path " hdfs://10.18.101.155:54310" will come from property fs.defaultFS and user can configure the /data/cnbc/ as carbon.ddl.base.hdfs.url. Now while dataload user can specify the csv path as /2016/xyz.csv.
carbon.badRecords.location		Path where the bad records are stored.
carbon.data.file.version	V3	If this parameter value is set to 1, then CarbonData will support the data load which is in old format(0.x version). If the value is set to 2(1.x onwards version), then CarbonData will support the data load of new format only.

carbon.streaming.auto.handoff.enabl	true	If this parameter value is set to true, auto trigger handoff function will be enabled.
carbon.streaming.segment.max.size	1024000000	This parameter defines the maximum size of the streaming segment. Setting this parameter to appropriate value will avoid impacting the streaming ingestion. The value is in bytes.
carbon.query.show.datamaps	true	If this parameter value is set to true, show tables command will list all the tables including datatmaps(eg: Preaggregate table), else datamaps will be excluded from the table list.
carbon.segment.lock.files.preserve.h	48	This property value indicates the number of hours the segment lock files will be preserved after dataload. These lock files will be deleted with the clean command after the configured number of hours.
carbon.unsafe.working.memory.in.mb	512	Specifies the size of executor unsafe working memory. Used for sorting data, storing column pages,etc. This value is expressed in MB.
carbon.unsafe.driver.working.memory	512	Specifies the size of driver unsafe working memory. Used for storing block or blocklet datamap cache. If not configured then carbon.unsafe.working.memory.in.mb value is considered. This value is expressed in MB.

6.2 Performance Configuration

This section provides the details of all the configurations required for CarbonData Performance Optimization.

Performance Configuration in carbon.properties

- **Data Loading Configuration**

Parameter	Default Value	Description	Range
carbon.number.of.cores.whi	2	Number of cores to be used while loading data.	
carbon.sort.size	100000	Record count to sort and write intermediate files to temp.	

carbon.max.driver.lru.cache	-1	Max LRU cache size upto which data will be loaded at the driver side. This value is expressed in MB. Default value of -1 means there is no memory limit for caching. Only integer values greater than 0 are accepted.
carbon.max.executor.lru.cac	-1	Max LRU cache size upto which data will be loaded at the executor side. This value is expressed in MB. Default value of -1 means there is no memory limit for caching. Only integer values greater than 0 are accepted. If this parameter is not configured, then the carbon.max.driver.lru.cache value will be considered.
carbon.merge.sort.prefetch	true	Enable prefetch of data during merge sort while reading data from sort temp files in data loading.
carbon.insert.persist.enable	false	Enabling this parameter considers persistent data. If we are executing insert into query from source table using select statement & loading the same source table concurrently, when select happens on source table during the data load, it gets new record for which dictionary is not generated, so there will be inconsistency. To avoid this condition we can persist the dataframe into MEMORY_AND_DISK(defa value) and perform insert into operation. By default this value will be false because no need to persist the dataframe in all cases. If user wants to run load and insert queries on source table concurrently then user can enable this parameter.

carbon.insert.storage.level	MEMORY_AND_DISK	Which storage level to persist dataframe when ‘carbon.insert.persist.enable if user’s executor has less memory, set this parameter to ‘MEMORY_AND_DISK_SE or other storage level to correspond to different environment. See detail.
carbon.update.persist.enabl	true	Enabling this parameter considers persistent data. Enabling this will reduce the execution time of UPDATE operation.
carbon.update.storage.level	MEMORY_AND_DISK	Which storage level to persist dataframe when ‘carbon.update.persist.enab if user’s executor has less memory, set this parameter to ‘MEMORY_AND_DISK_SE or other storage level to correspond to different environment. See detail.
carbon.global.sort.rdd.stora	MEMORY_ONLY	Which storage level to persist rdd when loading data with ‘sort_scope’=‘global_sort’, if user’s executor has less memory, set this parameter to ‘MEMORY_AND_DISK_SE or other storage level to correspond to different environment. See detail.
carbon.load.global.sort.parti	0	The Number of partitions to use when shuffling data for sort. If user don’t configurate or configurate it less than 1, it uses the number of map tasks as reduce tasks. In general, we recommend 2-3 tasks per CPU core in your cluster.
carbon.options.bad.records.	false	Whether to create logs with details about bad records.

carbon.bad.records.action	FORCE	This property can have four types of actions for bad records FORCE, REDIRECT, IGNORE and FAIL. If set to FORCE then it auto-corrects the data by storing the bad records as NULL. If set to REDIRECT then bad records are written to the raw CSV instead of being loaded. If set to IGNORE then bad records are neither loaded nor written to the raw CSV. If set to FAIL then data loading fails if any bad records are found.
carbon.options.is.empty.dat	false	If false, then empty ("" or '' or ,,) data will not be considered as bad record and vice versa.
carbon.options.bad.record.p		Specifies the HDFS path where bad records are stored. By default the value is Null. This path must to be configured by the user if bad record logger is enabled or bad record action redirect.

carbon.enable.vector.reade	true	This parameter increases the performance of select queries as it fetch columnar batch of size 4 *1024 rows instead of fetching data row by row. | | | carbon.blockletgroup.size.in | 64 MB | The data are read as a group of blocklets which are called blocklet groups. This parameter specifies the size of the blocklet group. Higher value results in better sequential IO access.The minimum value is 16MB, any value lesser than 16MB will reset to the default value (64MB). | | | carbon.task.distribution | block | block : Setting this value will launch one task per block. This setting is suggested in case of concurrent queries and queries having big shuffling scenarios. custom : Setting this value will group the blocks and distribute it uniformly to the available resources in the cluster. This enhances the query performance but not suggested in case of concurrent queries and queries having big shuffling scenarios. blocklet : Setting this value will launch one task per blocklet. This setting is suggested in case of concurrent queries and queries having big shuffling scenarios. merge_small_files : Setting this value will merge all the small partitions to a size of (128 MB is the default value of “spark.sql.files.maxPartition is configurable) during querying. The small partitions are combined to a map task to reduce the number of read task. This enhances the performance. | | | carbon.load.sortmemory.sp | 0 | If we use unsafe memory during data loading, this configuration will be used to control the behavior of spilling inmemory pages to disk. Internally in Carbondata, during sorting carbondata will sort data in pages*	Integer values between 0 and 100

- **Compaction Configuration**

Parameter	Default Value	Description	Range
carbon.number.of.cores.whi	2	Number of cores which are used to write data during compaction.	
carbon.compaction.level.thr	4, 3	This property is for minor compaction which decides how many segments to be merged. Example: If it is set as 2, 3 then minor compaction will be triggered for every 2 segments. 3 is the number of level 1 compacted segment which is further compacted to new segment.	Valid values are from 0-100.
carbon.major.compaction.si	1024	Major compaction size can be configured using this parameter. Sum of the segments which is below this threshold will be merged. This value is expressed in MB.	
carbon.horizontal.compactic	true	This property is used to turn ON/OFF horizontal compaction. After every DELETE and UPDATE statement, horizontal compaction may occur in case the delta (DELETE/ UPDATE) files becomes more than specified threshold.	
carbon.horizontal.UPDATE.	1	This property specifies the threshold limit on number of UPDATE delta files within a segment. In case the number of delta files goes beyond the threshold, the UPDATE delta files within the segment becomes eligible for horizontal compaction and compacted into single UPDATE delta file.	Values between 1 to 10000.

carbon.horizontal.DELETE.c	1	This property specifies the threshold limit on number of DELETE delta files within a block of a segment. In case the number of delta files goes beyond the threshold, the DELETE delta files for the particular block of the segment becomes eligible for horizontal compaction and compacted into single DELETE delta file.	Values between 1 to 10000.
carbon.update.segment.par	1	This property specifies the parallelism for each segment during update. If there are segments that contain too many records to update and the spark job encounter data-spill related errors, it is better to increase this property value. It is recommended to set this value to a multiple of the number of executors for balance.	Values between 1 to 1000.
carbon.merge.index.in.segn	true	This property is used to merge all carbon index files (.carbonindex) inside a segment to a single carbon index merge file (.carbonindexmerge).	Values true or false

- **Query Configuration**

Parameter	Default Value	Description	Range
carbon.number.of.cores	4	Number of cores to be used while querying.	
carbon.enable.quick.filter	false	Improves the performance of filter query.	

6.3 Miscellaneous Configuration

Extra Configuration in carbon.properties

- **Time format for CarbonData**

Parameter	Default Format	Description
carbon.timestamp.format	yyyy-MM-dd HH:mm:ss	Timestamp format of input data used for timestamp data type.

- **Dataload Configuration**

Parameter	Default Value	Description
carbon.sort.file.write.buffer.size	16384	File write buffer size used during sorting. Minimum allowed buffer size is 10240 byte and Maximum allowed buffer size is 10485760 byte.
carbon.lock.type	LOCALLOCK	This configuration specifies the type of lock to be acquired during concurrent operations on table. There are following types of lock implementation: - LOCALLOCK: Lock is created on local file system as file. This lock is useful when only one spark driver (thrift server) runs on a machine and no other CarbonData spark application is launched concurrently. - HDFSLOCK: Lock is created on HDFS file system as file. This lock is useful when multiple CarbonData spark applications are launched and no ZooKeeper is running on cluster and HDFS supports file based locking.
carbon.lock.path	TABLEPATH	Locks on the files are used to prevent concurrent operation from modifying the same files. This

configuration specifies the path where lock files have to be created. Recommended to configure HDFS lock path(to this property) in case of S3 file system as locking is not feasible on S3. **Note:** If this property is not set to HDFS location for S3 store, then there is a possibility of data corruption because multiple data manipulation calls might try to update the status file and as lock is not acquired before updation data might get overwritten. | | carbon.sort.intermediate.files.limit | 20 | Minimum number of intermediate files after which merged sort can be started (minValue = 2, maxValue=50). | | carbon.block.meta.size.reserved.percentage | 10 | Space reserved in percentage for writing block meta data in CarbonData file. | | carbon.csv.read.buffersize.byte | 1048576 | csv reading buffer size. | | carbon.merge.sort.reader.thread | 3 | Maximum no of threads used for reading intermediate files for final merging. | | carbon.concurrent.lock.retries | 100 | Specifies the maximum number of retries to obtain the lock for concurrent operations. This is used for concurrent loading. | | carbon.concurrent.lock.retry.timeout.sec | 1 | Specifies the interval between the retries to obtain the lock for concurrent operations. | | carbon.lock.retries | 3 | Specifies the maximum number of retries to obtain the lock for any operations other than load. | | carbon.lock.retry.timeout.sec | 5 | Specifies the interval between the retries to obtain the lock for any operation other than load. | | carbon.skip.empty.line | false | Setting this property ignores the empty lines in the CSV file during the data load | | carbon.enable.calculate.size | true | **For Load Operation**: Setting this property calculates the size of the carbon data file (.carbondata) and carbon index file (.carbonindex) for every load and updates the table status file. **For Describe Formatted**: Setting this property calculates the total size of the carbon data files and carbon index files for the respective table and displays in describe formatted command. |

- **Compaction Configuration**

Parameter	Default Value	Description

carbon.numberof.preserve.segments	0	If the user wants to preserve some number of segments from being compacted then he can set this property. Example: carbon.numberof.preserve.segments = 2 then 2 latest segments will always be excluded from the compaction. No segments will be preserved by default.
carbon.allowed.compaction.days	0	Compaction will merge the segments which are loaded with in the specific number of days configured. Example: If the configuration is 2, then the segments which are loaded in the time frame of 2 days only will get merged. Segments which are loaded 2 days apart will not be merged. This is disabled by default.
carbon.enable.auto.load.merge	false	To enable compaction while data loading.
carbon.enable.page.level.reader.in.cc	true	Enabling page level reader for compaction reduces the memory usage while compacting more number of segments. It allows reading only page by page instead of reading whole blocklet to memory.

- **Query Configuration**

Parameter	Default Value	Description
max.query.execution.time	60	Maximum time allowed for one query to be executed. The value is in minutes.
carbon.enableMinMax	true	Min max is feature added to enhance query performance. To disable this feature, set it false.
carbon.dynamicallocation.schedulerti	5	Specifies the maximum time (unit in seconds) the scheduler can wait for executor to be active. Minimum value is 5 sec and maximum value is 15 sec.
carbon.scheduler.minregisteredresou	0.8	Specifies the minimum resource (executor) ratio needed for starting the block distribution. The default value is 0.8, which indicates 80% of the requested resource is allocated for starting block distribution. The minimum value is 0.1 min and the maximum value is 1.0.

carbon.search.enabled (Alpha Feature)	false	If set to true, it will use CarbonReader to do distributed scan directly instead of using compute framework like spark, thus avoiding limitation of compute framework like SQL optimizer and task scheduling overhead.

- **Global Dictionary Configurations**

Parameter	Default Value	Description
carbon.cutOffTimestamp		Sets the start date for calculating the timestamp. Java counts the number of milliseconds from start of “1970-01-01 00:00:00”. This property is used to customize the start of position. For example “2000-01-01 00:00:00”. The date must be in the form “carbon.timestamp.format”.
carbon.timegranularity	SECOND	The property used to set the data granularity level DAY, HOUR, MINUTE, or SECOND.

6.4 Spark Configuration

Spark Configuration Reference in spark-defaults.conf

Parameter	Default Value	Description
spark.driver.memory	1g	Amount of memory to be used by the driver process.
spark.executor.memory	1g	Amount of memory to be used per executor process.

6.5 Dynamic Configuration In CarbonData Using SET-RESET

SET/RESET commands are used to add, update, display, or reset the carbondata properties dynamically without restarting the driver.

Syntax

- **Add or Update :** This command adds or updates the value of parameter_name.

```
SET parameter_name=parameter_value
```

- Display Property Value: This command displays the value of the specified parameter_name.

```
SET parameter_name
```

- Display Session Parameters: This command displays all the supported session parameters.

```
SET
```

- Display Session Parameters along with usage details: This command displays all the supported session parameters along with their usage details.

```
SET -v
```

- Reset: This command clears all the session parameters.

```
RESET
```

Parameter Description:

Parameter	Description
parameter_name	Name of the property whose value needs to be dynamically added, updated, or displayed.
parameter_value	New value of the parameter_name to be set.

Dynamically Configurable Properties of CarbonData

Properties	Description
carbon.options.bad.records.logger.enable	To enable or disable bad record logger.
carbon.options.bad.records.action	This property can have four types of actions for bad records FORCE, REDIRECT, IGNORE and FAIL. If set to FORCE then it auto-corrects the data by storing the bad records as NULL. If set to REDIRECT then bad records are written to the raw CSV instead of being loaded. If set to IGNORE then bad records are neither loaded nor written to the raw CSV. If set to FAIL then data loading fails if any bad records are found.
carbon.options.is.empty.data.bad.record	If false, then empty ("" or '' or ,,) data will not be considered as bad record and vice versa.
carbon.options.batch.sort.size.inmb	Size of batch data to keep in memory, as a thumb rule it supposed to be less than 45% of sort.inmemory.size.inmb otherwise it may spill intermediate data to disk.
carbon.options.single.pass	Single Pass Loading enables single job to finish data loading with dictionary generation on the fly. It enhances performance in the scenarios where the subsequent data loading after initial load involves fewer incremental updates on the dictionary. This option specifies whether to use single pass for loading data or not. By default this option is set to FALSE.
carbon.options.bad.record.path	Specifies the HDFS path where bad records needs to be stored.

carbon.custom.block.distribution	Specifies whether to use the Spark or Carbon block distribution feature.
enable.unsafe.sort	Specifies whether to use unsafe sort during data loading. Unsafe sort reduces the garbage collection during data load operation, resulting in better performance.

Examples:

- Add or Update:

```
SET enable.unsafe.sort =true
```

- Display Property Value:

```
SET enable.unsafe.sort
```

- Reset:

```
RESET
```

System Response:

- Success will be recorded in the driver log.
- Failure will be displayed in the UI.

7 Streaming Guide

CarbonData Streaming Ingestion

7.1 Quick example

Download and unzip spark-2.2.0-bin-hadoop2.7.tgz, and export $SPARK_HOME

Package carbon jar, and copy assembly/target/scala-2.11/carbondata_2.11-1.3.0-SNAPSHOT-shade-hadoop2.7.2.jar to $SPARK_HOME/jars `shell mvn clean package -DskipTests -Pspark-2.2`

Start a socket data server in a terminal `shell nc -lk 9099` type some CSV rows as following `csv 1,col1 2,col2 3,col3 4,col4 5,col5`

Start spark-shell in new terminal, type :paste, then copy and run the following code. ```scala import java.io.File import org.apache.spark.sql.{CarbonEnv, SparkSession} import org.apache.spark.sql.CarbonSession._ import org.apache.spark.sql.streaming.{ProcessingTime, StreamingQuery} import org.apache.carbondata.core.util.path.CarbonTablePath import org.apache.carbondata.streaming.parser.CarbonStreamParser

val warehouse = new File("./warehouse").getCanonicalPath val metastore = new File("./metastore").getCanonicalPath

val spark = SparkSession .builder() .master("local") .appName("StreamExample") .config("spark.sql.warehouse.dir", warehouse) .getOrCreateCarbonSession(warehouse, metastore)

spark.sparkContext.setLogLevel("ERROR")

// drop table if exists previously spark.sql(s"DROP TABLE IF EXISTS carbon_table") // Create target carbon table and populate with initial data spark.sql(s""" | CREATE TABLE carbon_table (| col1 INT, | col2 STRING |) | STORED BY 'carbondata' | TBLPROPERTIES('streaming'='true')""".stripMargin)

val carbonTable = CarbonEnv.getCarbonTable(Some("default"), "carbon_table")(spark) val tablePath = carbonTable.getTablePath

// batch load var qry: StreamingQuery = null val readSocketDF = spark.readStream .format("socket") .option("host", "localhost") .option("port", 9099) .load()

// Write data from socket stream to carbondata file qry = readSocketDF.writeStream .format("carbondata") .trigger(ProcessingTime("5 seconds")) .option("checkpointLocation", CarbonTablePath.getStreamingCheckpointDir(tablePath)) .option("dbName", "default") .option("tableName", "carbon_table") .option(CarbonStreamParser.CARBON_STREAM_PARSER, CarbonStreamParser.CARBON_STREAM_PARSER_CSV) .start()

// start new thread to show data new Thread() { override def run(): Unit = { do { spark.sql("select * from carbon_table").show(false) Thread.sleep(10000) } while (true) } }.start()

qry.awaitTermination() ```

Continue to type some rows into data server, and spark-shell will show the new data of the table.

7.2 Create table with streaming property

Streaming table is just a normal carbon table with “streaming” table property, user can create streaming table using following DDL. `sql CREATE TABLE streaming_table ( col1 INT, col2 STRING ) STORED BY 'carbondata' TBLPROPERTIES('streaming'='true')`

property name	default	description
streaming	false	Whether to enable streaming ingest feature for this table Value range: true, false

“DESC FORMATTED” command will show streaming property. `sql DESC FORMATTED streaming_table`

7.3 Alter streaming property

For an old table, use ALTER TABLE command to set the streaming property. `sql ALTER TABLE streaming_table SET TBLPROPERTIES('streaming'='true')`

7.4 Acquire streaming lock

At the begin of streaming ingestion, the system will try to acquire the table level lock of streaming.lock file. If the system isn’t able to acquire the lock of this table, it will throw an InterruptedException.

7.5 Create streaming segment

The input data of streaming will be ingested into a segment of the CarbonData table, the status of this segment is streaming. CarbonData call it a streaming segment. The “tablestatus” file will record the segment status and data size. The user can use “SHOW SEGMENTS FOR TABLE tableName” to check segment status.

After the streaming segment reaches the max size, CarbonData will change the segment status to “streaming finish” from “streaming”, and create new “streaming” segment to continue to ingest streaming data.

option	default	description
carbon.streaming.segment.max.size	1024000000	Unit: byte max size of streaming segment

segment status	description
streaming	The segment is running streaming ingestion
streaming finish	The segment already finished streaming ingestion, it will be handed off to a segment in the columnar format

7.6 Change segment status

Use below command to change the status of “streaming” segment to “streaming finish” segment. If the streaming application is running, this command will be blocked. `sql ALTER TABLE streaming_table FINISH STREAMING`

7.7 Handoff “streaming finish” segment to columnar segment

Use below command to handoff “streaming finish” segment to columnar format segment manually. ```sql ALTER TABLE streaming_table COMPACT ‘streaming’

```
## Auto handoff streaming segment
Config the property "carbon.streaming.auto.handoff.enabled" to auto handoff streami

property name | default | description
--- | --- | ---
carbon.streaming.auto.handoff.enabled | true | whether to auto trigger handoff oper

## Stream data parser
Config the property "carbon.stream.parser" to define a stream parser to convert Int

property name | default | description
--- | --- | ---
carbon.stream.parser | org.apache.carbondata.streaming.parser.RowStreamParserImp |

Currently CarbonData support two parsers, as following:

**1. org.apache.carbondata.streaming.parser.CSVStreamParserImp**: This parser gets

**2. org.apache.carbondata.streaming.parser.RowStreamParserImp**: This is the defau
```

case class FileElement(school: Array[String], age: Int) case class StreamData(id: Int, name: String, city: String, salary: Float, file: FileElement) …

var qry: StreamingQuery = null val readSocketDF = spark.readStream .format(“socket”) .option(“host”, “localhost”) .option(“port”, 9099) .load() .as[String] .map(_.split(“,”)) .map { fields => { val tmp = fields(4).split(“\$”) val file = FileElement(tmp(0).split(“:”), tmp(1).toInt) StreamData(fields(0).toInt, fields(1), fields(2), fields(3).toFloat, file) } }

// Write data from socket stream to carbondata file qry = readSocketDF.writeStream .format(“carbondata”) .trigger(ProcessingTime(“5 seconds”)) .option(“checkpointLocation”, tablePath.getStreamingCheckpointDir) .option(“dbName”, “default”) .option(“tableName”, “carbon_table”) .start()

… ```

7.7.1 How to implement a customized stream parser

If user needs to implement a customized stream parser to convert a specific InternalRow to Object[], it needs to implement `initialize` method and `parserRow` method of interface `CarbonStreamParser`, for example:

```
package org.XXX.XXX.streaming.parser

import org.apache.hadoop.conf.Configuration
import org.apache.spark.sql.catalyst.InternalRow
import org.apache.spark.sql.types.StructType

class XXXStreamParserImp extends CarbonStreamParser {

  override def initialize(configuration: Configuration, structType: StructType): U
    // user can get the properties from "configuration"
  }

  override def parserRow(value: InternalRow): Array[Object] = {
    // convert InternalRow to Object[](Array[Object] in Scala)
  }

  override def close(): Unit = {
  }
}
```

and then set the property “carbon.stream.parser” to “org.XXX.XXX.streaming.parser.XXXStreamParserImp”.

7.8 Close streaming table

Use below command to handoff all streaming segments to columnar format segments and modify the streaming property to false, this table becomes a normal table. ```sql ALTER TABLE streaming_table COMPACT ‘close_streaming’
```

## 7.9 Constraint

1. reject set streaming property from true to false.
2. reject UPDATE/DELETE command on the streaming table.
3. reject create pre-aggregation DataMap on the streaming table.
4. reject add the streaming property on the table with pre-aggregation DataMap.
5. if the table has dictionary columns, it will not support concurrent data loading.
6. block delete “streaming” segment while the streaming ingestion is running.
7. block drop the streaming table while the streaming ingestion is running.
```

8 SDK Guide

SDK Guide

In the carbon jars package, there exist a carbondata-store-sdk-x.x.x-SNAPSHOT.jar, including SDK writer and reader.

SDK Writer

This SDK writer, writes carbondata file and carbonindex file at a given path. External client can make use of this writer to convert other format data or live data to create carbondata and index files. These SDK writer output contains just a carbondata and carbonindex files. No metadata folder will be present.

8.1 Quick example

8.1.1 Example with csv format

```
import java.io.IOException;

import org.apache.carbondata.common.exceptions.sql.InvalidLoadOptionException;
import org.apache.carbondata.core.metadata.datatype.DataTypes;
import org.apache.carbondata.core.util.CarbonProperties;
import org.apache.carbondata.sdk.file.CarbonWriter;
import org.apache.carbondata.sdk.file.CarbonWriterBuilder;
import org.apache.carbondata.sdk.file.Field;
import org.apache.carbondata.sdk.file.Schema;

public class TestSdk {

  // pass true or false while executing the main to use offheap memory or not
  public static void main(String[] args) throws IOException, InvalidLoadOptionExce
    if (args.length > 0 && args[0] != null) {
      testSdkWriter(args[0]);
    } else {
      testSdkWriter("true");
    }
  }

  public static void testSdkWriter(String enableOffheap) throws IOException, Inval
    String path = "./target/testCSVSdkWriter";

    Field[] fields = new Field[2];
    fields[0] = new Field("name", DataTypes.STRING);
    fields[1] = new Field("age", DataTypes.INT);

    Schema schema = new Schema(fields);

    CarbonProperties.getInstance().addProperty("enable.offheap.sort", enableOffhea

    CarbonWriterBuilder builder = CarbonWriter.builder().outputPath(path);

    CarbonWriter writer = builder.buildWriterForCSVInput(schema);

    int rows = 5;
    for (int i = 0; i < rows; i++) {
      writer.write(new String[] { "robot" + (i % 10), String.valueOf(i) });
    }
    writer.close();
  }
}
```

8.1.2 Example with Avro format

```
import java.io.IOException;

import org.apache.carbondata.common.exceptions.sql.InvalidLoadOptionException;
import org.apache.carbondata.core.metadata.datatype.DataTypes;
import org.apache.carbondata.sdk.file.AvroCarbonWriter;
import org.apache.carbondata.sdk.file.CarbonWriter;
import org.apache.carbondata.sdk.file.Field;

import org.apache.avro.generic.GenericData;
import org.apache.commons.lang.CharEncoding;

import tech.allegro.schema.json2avro.converter.JsonAvroConverter;

public class TestSdkAvro {

  public static void main(String[] args) throws IOException, InvalidLoadOptionExcep
    testSdkWriter();
  }


  public static void testSdkWriter() throws IOException, InvalidLoadOptionException
    String path = "./AvroCarbonWriterSuiteWriteFiles";
    // Avro schema
    String avroSchema =
        "{" +
            "   \"type\" : \"record\"," +
            "   \"name\" : \"Acme\"," +
            "   \"fields\" : ["
            + "{ \"name\" : \"fname\", \"type\" : \"string\" },"
            + "{ \"name\" : \"age\", \"type\" : \"int\" }]" +
            "}";

    String json = "{\"fname\":\"bob\", \"age\":10}";

    // conversion to GenericData.Record
    JsonAvroConverter converter = new JsonAvroConverter();
    GenericData.Record record = converter.convertToGenericDataRecord(
        json.getBytes(CharEncoding.UTF_8), new org.apache.avro.Schema.Parser().pars

    try {
      CarbonWriter writer = CarbonWriter.builder()
          .outputPath(path)
          .buildWriterForAvroInput(new org.apache.avro.Schema.Parser().parse(avroSc

      for (int i = 0; i < 100; i++) {
        writer.write(record);
      }
      writer.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

8.1.3 Example with Json format

```
import java.io.IOException;

import org.apache.carbondata.common.exceptions.sql.InvalidLoadOptionException;
import org.apache.carbondata.core.metadata.datatype.DataTypes;
import org.apache.carbondata.core.util.CarbonProperties;
import org.apache.carbondata.sdk.file.CarbonWriter;
import org.apache.carbondata.sdk.file.CarbonWriterBuilder;
import org.apache.carbondata.sdk.file.Field;
import org.apache.carbondata.sdk.file.Schema;

public class TestSdkJson {

   public static void main(String[] args) throws InvalidLoadOptionException {
       testJsonSdkWriter();
   }

   public static void testJsonSdkWriter() throws InvalidLoadOptionException {
    String path = "./target/testJsonSdkWriter";

    Field[] fields = new Field[2];
    fields[0] = new Field("name", DataTypes.STRING);
    fields[1] = new Field("age", DataTypes.INT);

    Schema CarbonSchema = new Schema(fields);

    CarbonWriterBuilder builder = CarbonWriter.builder().outputPath(path);

    // initialize json writer with carbon schema
    CarbonWriter writer = builder.buildWriterForJsonInput(CarbonSchema);
    // one row of json Data as String
    String  JsonRow = "{\"name\":\"abcd\", \"age\":10}";

    int rows = 5;
    for (int i = 0; i < rows; i++) {
      writer.write(JsonRow);
    }
    writer.close();
  }
}
```

8.2 Datatypes Mapping

Each of SQL data types are mapped into data types of SDK. Following are the mapping:

SQL DataTypes	Mapped SDK DataTypes
BOOLEAN	DataTypes.BOOLEAN
SMALLINT	DataTypes.SHORT
INTEGER	DataTypes.INT

BIGINT	DataTypes.LONG
DOUBLE	DataTypes.DOUBLE
VARCHAR	DataTypes.STRING
DATE	DataTypes.DATE
TIMESTAMP	DataTypes.TIMESTAMP
STRING	DataTypes.STRING
DECIMAL	DataTypes.createDecimalType(precision, scale)

NOTE: Carbon Supports below logical types of AVRO. a. Date The date logical type represents a date within the calendar, with no reference to a particular time zone or time of day. A date logical type annotates an Avro int, where the int stores the number of days from the unix epoch, 1 January 1970 (ISO calendar). b. Timestamp (millisecond precision) The timestamp-millis logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one millisecond. A timestamp-millis logical type annotates an Avro long, where the long stores the number of milliseconds from the unix epoch, 1 January 1970 00:00:00.000 UTC. c. Timestamp (microsecond precision) The timestamp-micros logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one microsecond. A timestamp-micros logical type annotates an Avro long, where the long stores the number of microseconds from the unix epoch, 1 January 1970 00:00:00.000000 UTC.

```
Currently the values of logical types are not validated by carbon.
Expect that avro record passed by the user is already validated by avro record gene
```

8.3 Run SQL on files directly

Instead of creating table and query it, you can also query that file directly with SQL.

8.3.1 Example

```
SELECT * FROM carbonfile.`$Path`
```

Find example code at DirectSQLExample in the CarbonData repo.

8.4 API List

8.4.1 Class org.apache.carbondata.sdk.file.CarbonWriterBuilder

```
/**
* Sets the output path of the writer builder
* @param path is the absolute path where output files are written
*             This method must be called when building CarbonWriterBuilder
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder outputPath(String path);
```

```
/**
* If set false, writes the carbondata and carbonindex files in a flat folder struct
* @param isTransactionalTable is a boolelan value
*             if set to false, then writes the carbondata and carbonindex files
*                                                               in a flat folder struc
*             if set to true, then writes the carbondata and carbonindex files
*                                                               in segment folder stru
*             By default set to false.
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder isTransactionalTable(boolean isTransactionalTable);
```

```
/**
* to set the timestamp in the carbondata and carbonindex index files
* @param UUID is a timestamp to be used in the carbondata and carbonindex index fil
*             By default set to zero.
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder uniqueIdentifier(long UUID);
```

```
/**
* To set the carbondata file size in MB between 1MB-2048MB
* @param blockSize is size in MB between 1MB to 2048 MB
*                  default value is 1024 MB
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder withBlockSize(int blockSize);
```

```
/**
* To set the blocklet size of carbondata file
* @param blockletSize is blocklet size in MB
*                     default value is 64 MB
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder withBlockletSize(int blockletSize);
```

```
/**
   * @param enableLocalDictionary enable local dictionary  , default is false
   * @return updated CarbonWriterBuilder
   */
public CarbonWriterBuilder enableLocalDictionary(boolean enableLocalDictionary);
```

```
/**
   * @param localDictionaryThreshold is localDictionaryThreshold,default is 10000
   * @return updated CarbonWriterBuilder
   */
public CarbonWriterBuilder localDictionaryThreshold(int localDictionaryThreshold) ;
```

```
/**
* sets the list of columns that needs to be in sorted order
* @param sortColumns is a string array of columns that needs to be sorted.
*                    If it is null or by default all dimensions are selected for so
*                    If it is empty array, no columns are sorted
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder sortBy(String[] sortColumns);
```

```
/**
* If set, create a schema file in metadata folder.
* @param persist is a boolean value, If set to true, creates a schema file in metad
*                By default set to false. will not create metadata folder
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder persistSchemaFile(boolean persist);
```

```
/**
* sets the taskNo for the writer. SDKs concurrently running
* will set taskNo in order to avoid conflicts in file's name during write.
* @param taskNo is the TaskNo user wants to specify.
*               by default it is system time in nano seconds.
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder taskNo(long taskNo);
```

```
/**
* To support the load options for sdk writer
* @param options key,value pair of load options.
*                supported keys values are
*                a. bad_records_logger_enable -- true (write into separate logs), f
*                b. bad_records_action -- FAIL, FORCE, IGNORE, REDIRECT
*                c. bad_record_path -- path
*                d. dateformat -- same as JAVA SimpleDateFormat
*                e. timestampformat -- same as JAVA SimpleDateFormat
*                f. complex_delimiter_level_1 -- value to Split the complexTypeData
*                g. complex_delimiter_level_2 -- value to Split the nested complexT
*                h. quotechar
*                i. escapechar
*
*                Default values are as follows.
*
*                a. bad_records_logger_enable -- "false"
*                b. bad_records_action -- "FAIL"
*                c. bad_record_path -- ""
*                d. dateformat -- "" , uses from carbon.properties file
*                e. timestampformat -- "", uses from carbon.properties file
*                f. complex_delimiter_level_1 -- "$"
*                g. complex_delimiter_level_2 -- ":"
*                h. quotechar -- "\""
*                i. escapechar -- "\\"
*
* @return updated CarbonWriterBuilder
*/
public CarbonWriterBuilder withLoadOptions(Map<String, String> options);
```

```
/**
* Build a {@link CarbonWriter}, which accepts row in CSV format object
* @param schema carbon Schema object {org.apache.carbondata.sdk.file.Schema}
* @return CSVCarbonWriter
* @throws IOException
* @throws InvalidLoadOptionException
*/
public CarbonWriter buildWriterForCSVInput(org.apache.carbondata.sdk.file.Schema sc
```

```
/**
* Build a {@link CarbonWriter}, which accepts Avro format object
* @param avroSchema avro Schema object {org.apache.avro.Schema}
* @return AvroCarbonWriter
* @throws IOException
* @throws InvalidLoadOptionException
*/
public CarbonWriter buildWriterForAvroInput(org.apache.avro.Schema schema) throws I
```

```
/**
* Build a {@link CarbonWriter}, which accepts Json object
* @param carbonSchema carbon Schema object
* @return JsonCarbonWriter
* @throws IOException
* @throws InvalidLoadOptionException
*/
public JsonCarbonWriter buildWriterForJsonInput(Schema carbonSchema);
```

8.4.2 Class org.apache.carbondata.sdk.file.CarbonWriter

```
/**
* Write an object to the file, the format of the object depends on the implementati
* If AvroCarbonWriter, object is of type org.apache.avro.generic.GenericData.Record
*                       which is one row of data.
* If CSVCarbonWriter, object is of type String[], which is one row of data
* If JsonCarbonWriter, object is of type String, which is one row of json
* Note: This API is not thread safe
* @param object
* @throws IOException
*/
public abstract void write(Object object) throws IOException;
```

```
/**
* Flush and close the writer
*/
public abstract void close() throws IOException;
```

```
/**
* Create a {@link CarbonWriterBuilder} to build a {@link CarbonWriter}
*/
public static CarbonWriterBuilder builder() {
    return new CarbonWriterBuilder();
}
```

8.4.3 Class org.apache.carbondata.sdk.file.Field

```
/**
* Field Constructor
* @param name name of the field
* @param type datatype of field, specified in strings.
*/
public Field(String name, String type);
```

```
/**
* Field constructor
* @param name name of the field
* @param type datatype of the field of class DataType
*/
public Field(String name, DataType type);
```

8.4.4 Class org.apache.carbondata.sdk.file.Schema

```
/**
* construct a schema with fields
* @param fields
*/
public Schema(Field[] fields);
```

```
/**
* Create a Schema using JSON string, for example:
* [
*   {"name":"string"},
*   {"age":"int"}
* ]
* @param json specified as string
* @return Schema
*/
public static Schema parseJson(String json);
```

8.4.5 Class org.apache.carbondata.sdk.file.AvroCarbonWriter

```
/**
* converts avro schema to carbon schema, required by carbonWriter
*
* @param avroSchemaString json formatted avro schema as string
* @return carbon sdk schema
*/
public static org.apache.carbondata.sdk.file.Schema getCarbonSchemaFromAvroSchema(S
```

SDK Reader

This SDK reader reads CarbonData file and carbonindex file at a given path. External client can make use of this reader to read CarbonData files without CarbonSession.

8.5 Quick example

```
    // 1. Create carbon reader
    String path = "./testWriteFiles";
    CarbonReader reader = CarbonReader
        .builder(path, "_temp")
        .projection(new String[]{"stringField", "shortField", "intField", "longFiel
                "doubleField", "boolField", "dateField", "timeField", "decimalField
        .build();

    // 2. Read data
    long day = 24L * 3600 * 1000;
    int i = 0;
    while (reader.hasNext()) {
        Object[] row = (Object[]) reader.readNextRow();
        System.out.println(String.format("%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t"
            i, row[0], row[1], row[2], row[3], row[4], row[5],
            new Date((day * ((int) row[6]))), new Timestamp((long) row[7] / 1000),
        ));
        i++;
    }

    // 3. Close this reader
    reader.close();
```

Find example code at CarbonReaderExample in the CarbonData repo.

8.6 API List

8.6.1 Class org.apache.carbondata.sdk.file.CarbonReader

```
   /**
    * Return a new {@link CarbonReaderBuilder} instance
    *
    * @param tablePath table store path
    * @param tableName table name
    * @return CarbonReaderBuilder object
    */
  public static CarbonReaderBuilder builder(String tablePath, String tableName);
```

```
  /**
   * Return a new CarbonReaderBuilder instance
   * Default value of table name is table + tablePath + time
   *
   * @param tablePath table path
   * @return CarbonReaderBuilder object
   */
  public static CarbonReaderBuilder builder(String tablePath);
```

```
/**
 * Return true if has next row
 */
public boolean hasNext();
```

```
/**
 * Read and return next row object
 */
public T readNextRow();
```

```
/**
 * Close reader
 */
public void close();
```

8.6.2 Class org.apache.carbondata.sdk.file.CarbonReaderBuilder

```
/**
 * Construct a CarbonReaderBuilder with table path and table name
 *
 * @param tablePath table path
 * @param tableName table name
 */
CarbonReaderBuilder(String tablePath, String tableName);
```

```
/**
 * Configure the projection column names of carbon reader
 *
 * @param projectionColumnNames projection column names
 * @return CarbonReaderBuilder object
 */
public CarbonReaderBuilder projection(String[] projectionColumnNames);
```

```
/**
 * Configure the transactional status of table
 * If set to false, then reads the carbondata and carbonindex files from a flat f
 * If set to true, then reads the carbondata and carbonindex files from segment f
 * Default value is false
 *
 * @param isTransactionalTable whether is transactional table or not
 * @return CarbonReaderBuilder object
 */
public CarbonReaderBuilder isTransactionalTable(boolean isTransactionalTable);
```

```
/**
 * Configure the filter expression for carbon reader
 *
 * @param filterExpression filter expression
 * @return CarbonReaderBuilder object
 */
 public CarbonReaderBuilder filter(Expression filterExpression);
```

```
/**
 * Set the access key for S3
 *
 * @param key   the string of access key for different S3 type,like: fs.s3a.acces
 * @param value the value of access key
 * @return CarbonWriterBuilder
 */
public CarbonReaderBuilder setAccessKey(String key, String value);
```

```
/**
 * Set the access key for S3.
 *
 * @param value the value of access key
 * @return CarbonWriterBuilder object
 */
public CarbonReaderBuilder setAccessKey(String value);
```

```
/**
 * Set the secret key for S3
 *
 * @param key   the string of secret key for different S3 type,like: fs.s3a.secre
 * @param value the value of secret key
 * @return CarbonWriterBuilder object
 */
public CarbonReaderBuilder setSecretKey(String key, String value);
```

```
/**
 * Set the secret key for S3
 *
 * @param value the value of secret key
 * @return CarbonWriterBuilder object
 */
public CarbonReaderBuilder setSecretKey(String value);
```

```
/**
  * Set the endpoint for S3
  *
  * @param key   the string of endpoint for different S3 type,like: fs.s3a.endpoin
  * @param value the value of endpoint
  * @return CarbonWriterBuilder object
  */
 public CarbonReaderBuilder setEndPoint(String key, String value);
```

```
 /**
  * Set the endpoint for S3
  *
  * @param value the value of endpoint
  * @return CarbonWriterBuilder object
  */
 public CarbonReaderBuilder setEndPoint(String value);
```

```
/**
  * Build CarbonReader
  *
  * @param <T>
  * @return CarbonReader
  * @throws IOException
  * @throws InterruptedException
  */
 public <T> CarbonReader<T> build();
```

8.6.3 Class org.apache.carbondata.sdk.file.CarbonSchemaReader

```
 /**
  * Read schema file and return the schema
  *
  * @param schemaFilePath complete path including schema file name
  * @return schema object
  * @throws IOException
  */
 public static Schema readSchemaInSchemaFile(String schemaFilePath);
```

```
 /**
  * Read carbondata file and return the schema
  *
  * @param dataFilePath complete path including carbondata file name
  * @return Schema object
  * @throws IOException
  */
 public static Schema readSchemaInDataFile(String dataFilePath);
```

```
/**
 * Read carbonindex file and return the schema
 *
 * @param indexFilePath complete path including index file name
 * @return schema object
 * @throws IOException
 */
public static Schema readSchemaInIndexFile(String indexFilePath);
```

8.6.4 Class org.apache.carbondata.sdk.file.Schema

```
/**
 * construct a schema with fields
 * @param fields
 */
public Schema(Field[] fields);
```

```
/**
 * construct a schema with List<ColumnSchema>
 *
 * @param columnSchemaList column schema list
 */
public Schema(List<ColumnSchema> columnSchemaList);
```

```
/**
 * Create a Schema using JSON string, for example:
 * [
 *   {"name":"string"},
 *   {"age":"int"}
 * ]
 * @param json specified as string
 * @return Schema
 */
public static Schema parseJson(String json);
```

```
/**
 * Sort the schema order as original order
 *
 * @return Schema object
 */
public Schema asOriginOrder();
```

8.6.5 Class org.apache.carbondata.sdk.file.Field

```
  /**
   * Field Constructor
   * @param name name of the field
   * @param type datatype of field, specified in strings.
   */
  public Field(String name, String type);
```

```
  /**
   * Construct Field from ColumnSchema
   *
   * @param columnSchema ColumnSchema, Store the information about the column meta
   */
  public Field(ColumnSchema columnSchema);
```

Find S3 example code at SDKS3Example in the CarbonData repo.

Common API List for CarbonReader and CarbonWriter

8.6.6 Class org.apache.carbondata.core.util.CarbonProperties

```
/**
* This method will be responsible to get the instance of CarbonProperties class
*
* @return carbon properties instance
*/
public static CarbonProperties getInstance();
```

```
/**
* This method will be used to add a new property
*
* @param key is a property name to set for carbon.
* @param value is valid parameter corresponding to property.
* @return CarbonProperties object
*/
public CarbonProperties addProperty(String key, String value);
```

```
/**
* This method will be used to get the property value. If property is not
* present, then it will return the default value.
*
* @param key is a property name to get user specified value.
* @return properties value for corresponding key. If not set, then returns null.
*/
public String getProperty(String key);
```

```
/**
* This method will be used to get the property value. If property is not
* present, then it will return the default value.
*
* @param key is a property name to get user specified value..
* @param defaultValue used to be returned by function if corrosponding key not set.
* @return properties value for corresponding key. If not set, then returns specifie
*/
public String getProperty(String key, String defaultValue);
```

Reference : list of carbon properties

9 DataMap Developer Guide

DataMap Developer Guide

9.1.1 Introduction

DataMap is a data structure that can be used to accelerate certain query of the table. Different DataMap can be implemented by developers. Currently, there are two 2 types of DataMap supported: 1. IndexDataMap: DataMap that leveraging index to accelerate filter query 2. MVDataMap: DataMap that leveraging Materialized View to accelerate olap style query, like SPJG query (select, predicate, join, groupby)

9.1.2 DataMap provider

When user issues `CREATE DATAMAP dm ON TABLE main USING 'provider'`, the corresponding DataMapProvider implementation will be created and initialized. Currently, the provider string can be: 1. preaggregate: one type of MVDataMap that do pre-aggregate of single table 2. timeseries: one type of MVDataMap that do pre-aggregate based on time dimension of the table 3. class name IndexDataMapFactory implementation: Developer can implement new type of IndexDataMap by extending IndexDataMapFactory

When user issues `DROP DATAMAP dm ON TABLE main`, the corresponding DataMapProvider interface will be called.

10 CarbonData BloomFilter DataMap (Alpha Feature)

CarbonData BloomFilter DataMap (Alpha Feature)

10.1.1.1 DataMap Management

Creating BloomFilter DataMap `CREATE DATAMAP [IF NOT EXISTS] datamap_name ON TABLE main_table USING 'bloomfilter' DMPROPERTIES ('index_columns'='city, name', 'BLOOM_SIZE'='640000', 'BLOOM_FPP'='0.00001')`

Dropping specified datamap `DROP DATAMAP [IF EXISTS] datamap_name ON TABLE main_table`

Showing all DataMaps on this table `SHOW DATAMAP ON TABLE main_table`

Disable Datamap

The datamap by default is enabled. To support tuning on query, we can disable a specific datamap during query to observe whether we can gain performance enhancement from it. This will only take effect current session.

```
// disable the datamap SET
carbon.datamap.visible.dbName.tableName.dataMapName = false // enable the
datamap SET carbon.datamap.visible.dbName.tableName.dataMapName = true
```

10.2 BloomFilter DataMap Introduction

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Carbondata introduced BloomFilter as an index datamap to enhance the performance of querying with precise value. It is well suitable for queries that do precise match on high cardinality columns(such as Name/ID). Internally, CarbonData maintains a BloomFilter per blocklet for each index column to indicate that whether a value of the column is in this blocklet. Just like the other datamaps, BloomFilter datamap is managed along with main tables by CarbonData. User can create BloomFilter datamap on specified columns with specified BloomFilter configurations such as size and probability.

For instance, main table called **datamap_test** which is defined as:

```
CREATE TABLE datamap_test ( id string, name string, age int, city string,
country string) STORED BY 'carbondata' TBLPROPERTIES('SORT_COLUMNS'='id')
```

In the above example, `id` and `name` are high cardinality columns and we always query on `id` and `name` with precise value. since `id` is in the sort_columns and it is orderd, query on it will be fast because CarbonData can skip all the irrelative blocklets. But queries on `name` may be bad since the blocklet minmax may not help, because in each blocklet the range of the value of `name` may be the same – all from A ~z. In this case, user can create a BloomFilter datamap on column `name`. Moreover, user can also create a BloomFilter datamap on the sort_columns. This is useful if user has too many segments and the range of the value of sort_columns are almost the same.

User can create BloomFilter datamap using the Create DataMap DDL:

```
CREATE DATAMAP dm ON TABLE datamap_test USING 'bloomfilter' DMPROPERTIES
('INDEX_COLUMNS' = 'name,id', 'BLOOM_SIZE'='640000', 'BLOOM_FPP'='0.00001',
'BLOOM_COMPRESS'='true')
```

Properties for BloomFilter DataMap

Property	Is Required	Default Value	Description
INDEX_COLUMNS	YES		Carbondata will generate BloomFilter index on these columns. Queries on these columns are usually like 'COL = VAL'.
BLOOM_SIZE	NO	640000	This value is internally used by BloomFilter as the number of expected insertions, it will affect the size of BloomFilter index. Since each blocklet has a BloomFilter here, so the default value is the approximate distinct index values in a blocklet assuming that each blocklet contains 20 pages and each page contains 32000 records. The value should be an integer.
BLOOM_FPP	NO	0.00001	This value is internally used by BloomFilter as the False-Positive Probability, it will affect the size of bloomfilter index as well as the number of hash functions for the BloomFilter. The value should be in the range (0, 1). In one test scenario, a 96GB TPCH customer table with bloom_size=320000 and bloom_fpp=0.00001 will result in 18 false positive samples.
BLOOM_COMPRESS	NO	true	Whether to compress the BloomFilter index files.

10.3 Loading Data

When loading data to main table, BloomFilter files will be generated for all the index_columns given in DMProperties which contains the blockletId and a BloomFilter for each index column. These index files will be written inside a folder named with datamap name inside each segment folders.

10.4 Querying Data

User can verify whether a query can leverage BloomFilter datamap by executing `EXPLAIN` command, which will show the transformed logical plan, and thus user can check whether the BloomFilter datamap can skip blocklets during the scan. If the datamap does not prune blocklets well, you can try to increase the value of property `BLOOM_SIZE` and decrease the value of property `BLOOM_FPP`.

10.5 Data Management With BloomFilter DataMap

Data management with BloomFilter datamap has no difference with that on Lucene datamap. You can refer to the corresponding section in `CarbonData Lucene DataMap`.

10.6 Useful Tips

- BloomFilter DataMap is suggested to be created on the high cardinality columns. Query conditions on these columns are always simple `equal` or `in`, such as ‘col1=XX’, ‘col1 in (XX, YY)’.
- We can create multiple BloomFilter datamaps on one table, but we do recommend you to create one BloomFilter datamap that contains multiple index columns, because the data loading and query performance will be better.
- `BLOOM_FPP` is only the expected number from user, the actually FPP may be worse. If the BloomFilter datamap does not work well, you can try to increase `BLOOM_SIZE` and decrease `BLOOM_FPP` at the same time. Notice that bigger `BLOOM_SIZE` will increase the size of index file and smaller `BLOOM_FPP` will increase runtime calculation while performing query.
- ‘0’ skipped blocklets of BloomFilter datamap in explain output indicates that BloomFilter datamap does not prune better than Main datamap. (For example since the data is not ordered, a specific value may be contained in many blocklets. In this case, bloom may not work better than Main DataMap.) If this occurs very often, it means that current BloomFilter is useless. You can disable or drop it. Sometimes we cannot see any pruning result about BloomFilter datamap in the explain output, this indicates that the previous datamap has pruned all the blocklets and there is no need to continue pruning.
- In some scenarios, the BloomFilter datamap may not enhance the query performance significantly but if it can reduce the number of spark task, there is still a chance that BloomFilter datamap can enhance the performance for concurrent query.
- Note that BloomFilter datamap will decrease the data loading performance and may cause slightly storage expansion (for datamap index file).

11 CarbonData Lucene DataMap (Alpha Feature)

CarbonData Lucene DataMap (Alpha Feature)

- DataMap Management
- Lucene Datamap
- Loading Data
- Querying Data
- Data Management

11.1.1.1 DataMap Management

Lucene DataMap can be created using following DDL `CREATE DATAMAP [IF NOT EXISTS] datamap_name ON TABLE main_table USING 'lucene' DMPROPERTIES ('index_columns'='city, name', ...)`

DataMap can be dropped using following DDL: `DROP DATAMAP [IF EXISTS] datamap_name ON TABLE main_table` To show all DataMaps created, use: `SHOW DATAMAP ON TABLE main_table` It will show all DataMaps created on main table.

11.2 Lucene DataMap Introduction

Lucene is a high performance, full featured text search engine. Lucene is integrated to carbon as an index datamap and managed along with main tables by CarbonData.User can create lucene datamap to improve query performance on string columns which has content of more length. So, user can search tokenized word or pattern of it using lucene query on text content.

For instance, main table called **datamap_test** which is defined as:

```
CREATE TABLE datamap_test ( name string, age int, city string, country
string) STORED BY 'carbondata'
```

User can create Lucene datamap using the Create DataMap DDL:

```
CREATE DATAMAP dm ON TABLE datamap_test USING 'lucene' DMPROPERTIES
('INDEX_COLUMNS' = 'name, country',)
```

DMProperties 1. INDEX_COLUMNS: The list of string columns on which lucene creates indexes. 2. FLUSH_CACHE: size of the cache to maintain in Lucene writer, if specified then it tries to aggregate the unique data till the cache limit and flush to Lucene. It is best suitable for low cardinality dimensions. 3. SPLIT_BLOCKLET: when made as true then store the data in blocklet wise in lucene , it means new folder will be created for each blocklet, thus, it eliminates storing blockletid in lucene and also it makes lucene small chunks of data.

11.3 Loading data

When loading data to main table, lucene index files will be generated for all the index_columns(String Columns) given in DMProperties which contains information about the data location of index_columns. These index files will be written inside a folder named with datamap name inside each segment folders.

A system level configuration carbon.lucene.compression.mode can be added for best compression of lucene index files. The default value is speed, where the index writing speed will be more. If the value is compression, the index file size will be compressed.

11.4 Querying data

As a technique for query acceleration, Lucene indexes cannot be queried directly. Queries are to be made on main table. when a query with TEXT_MATCH('name:c10') or TEXT_MATCH_WITH_LIMIT('name:n10',10)[the second parameter represents the number of result to be returned, if user does not specify this value, all results will be returned without any limit] is fired, two jobs are fired.The first job writes the temporary files in folder created at table level which contains lucene's seach results and these files will be read in second job to give faster results. These temporary files will be cleared once the query finishes.

User can verify whether a query can leverage Lucene datamap or not by executing `EXPLAIN` command, which will show the transformed logical plan, and thus user can check whether TEXT_MATCH() filter is applied on query or not.

Note: 1. The filter columns in TEXT_MATCH or TEXT_MATCH_WITH_LIMIT must be always in lower case and filter condition like 'AND','OR' must be in upper case.

```
Ex:
```
select * from datamap_test where TEXT_MATCH('name:*10 AND name:*n*')
```
```

1. Query supports only one TEXT_MATCH udf for filter condition and not multiple udfs.

The following query is supported: `select * from datamap_test where TEXT_MATCH('name:*10 AND name:*n*')`

The following query is not supported: `select * from datamap_test where TEXT_MATCH('name:*10) AND TEXT_MATCH(name:*n*')`

Below like queries can be converted to text_match queries as following: ``` select * from datamap_test where name='n10'

select * from datamap_test where name like 'n1%'

select * from datamap_test where name like '%10'

select * from datamap_test where name like '%n%'

select * from datamap_test where name like '%10' and name not like '%n%' `Lucene TEXT_MATCH Queries:` select * from datamap_test where TEXT_MATCH('name:n10')

select * from datamap_test where TEXT_MATCH('name:n1*')

select * from datamap_test where TEXT_MATCH('name:*10')

select * from datamap_test where TEXT_MATCH('name: *n*')

select * from datamap_test where TEXT_MATCH('name: *10 -name: n*') ``` *Note:** For lucene queries and syntax, refer to lucene-syntax

11.5 Data Management with lucene datamap

Once there is lucene datamap is created on the main table, following command on the main table is not supported: 1. Data management command: `UPDATE/DELETE`. 2. Schema management command: `ALTER TABLE DROP COLUMN`, `ALTER TABLE CHANGE DATATYPE`, `ALTER TABLE RENAME`.

Note: Adding a new column is supported, and for dropping columns and change datatype command, CarbonData will check whether it will impact the lucene datamap, if not, the operation is allowed, otherwise operation will be rejected by throwing exception.

1. Partition management command: `ALTER TABLE ADD/DROP PARTITION`.

However, there is still way to support these operations on main table, in current CarbonData release, user can do as following: 1. Remove the lucene datamap by `DROP DATAMAP` command. 2. Carry out the data management operation on main table. 3. Create the lucene datamap again by `CREATE DATAMAP` command. Basically, user can manually trigger the operation by re-building the datamap.

12 CarbonData Pre-aggregate DataMap

CarbonData Pre-aggregate DataMap

12.1 Quick example

Download and unzip spark-2.2.0-bin-hadoop2.7.tgz, and export $SPARK_HOME

Package carbon jar, and copy assembly/target/scala-2.11/carbondata_2.11-x.x.x-SNAPSHOT-shade-hadoop2.7.2.jar to $SPARK_HOME/jars `shell mvn clean package -DskipTests -Pspark-2.2`

Start spark-shell in new terminal, type :paste, then copy and run the following code. ```scala import java.io.File import org.apache.spark.sql.{CarbonEnv, SparkSession} import org.apache.spark.sql.CarbonSession._ import org.apache.spark.sql.streaming.{ProcessingTime, StreamingQuery} import org.apache.carbondata.core.util.path.CarbonStorePath

val warehouse = new File("./warehouse").getCanonicalPath val metastore = new File("./metastore").getCanonicalPath

val spark =
SparkSession .builder() .master("local") .appName("preAggregateExample") .config("spark.sql.warehouse.dir", warehouse) .getOrCreateCarbonSession(warehouse, metastore)

spark.sparkContext.setLogLevel("ERROR")

// drop table if exists previously spark.sql(s"DROP TABLE IF EXISTS sales")

// Create main table spark.sql(s""" | CREATE TABLE sales (| user_id string, | country string, | quantity int, | price bigint) | STORED BY 'carbondata' """.stripMargin)

// Create pre-aggregate table on the main table // If main table already have data, following command // will trigger one immediate load to the pre-aggregate table spark.sql(s""" | CREATE DATAMAP agg_sales | ON TABLE sales | USING "preaggregate" | AS | SELECT country, sum(quantity), avg(price) | FROM sales | GROUP BY country """.stripMargin)

import spark.implicits._ import org.apache.spark.sql.SaveMode import scala.util.Random

// Load data to the main table, it will also // trigger immediate load to pre-aggregate table. // These two loading operation is carried out in a // transactional manner, meaning that the whole // operation will fail if one of the loading fails val r = new Random() spark.sparkContext.parallelize(1 to 10) .map(x => ("ID." + r.nextInt(100000), "country" + x % 8, x % 50, x % 60)) .toDF("user_id", "country", "quantity", "price") .write .format("carbondata") .option("tableName", "sales") .option("compress", "true") .mode(SaveMode.Append) .save()

spark.sql(s""" |SELECT country, sum(quantity), avg(price) | from sales GROUP BY country """.stripMargin).show

spark.stop ```

12.1.1.1 DataMap Management

DataMap can be created using following DDL `CREATE DATAMAP [IF NOT EXISTS] datamap_name ON TABLE main_table USING "datamap_provider" DMPROPERTIES ('key'='value', ...) AS SELECT statement` The string followed by USING is called DataMap Provider, in this version CarbonData supports two kinds of DataMap: 1. preaggregate, for pre-aggregate table. Pre-Aggregate table supports two values for DMPROPERTIES. a. ‘path’ is used to specify the store location of the datamap.(‘path’=‘/location/’). b. ‘partitioning’ when set to false enables user to disable partitioning of the datamap. Default value is true for this property. 2. timeseries, for timeseries roll-up table. Please refer to Timeseries DataMap

DataMap can be dropped using following DDL `DROP DATAMAP [IF EXISTS] datamap_name ON TABLE main_table` To show all DataMaps created, use: `SHOW DATAMAP ON TABLE main_table` It will show all DataMaps created on main table.

12.2 Preaggregate DataMap Introduction

Pre-aggregate tables are created as DataMaps and managed as tables internally by CarbonData. User can create as many pre-aggregate datamaps required to improve query performance, provided the storage requirements and loading speeds are acceptable.

Once pre-aggregate datamaps are created, CarbonData’s SparkSQL optimizer extension supports to select the most efficient pre-aggregate datamap and rewrite the SQL to query against the selected datamap instead of the main table. Since the data size of pre-aggregate datamap is smaller, user queries are much faster. In our previous experience, we have seen 5X to 100X times faster in production SQLs.

For instance, main table called **sales** which is defined as

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

User can create pre-aggregate tables using the Create DataMap DDL

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

12.2.1.1 Functions supported in pre-aggregate table

Function	Rollup supported
SUM	Yes
AVG	Yes
MAX	Yes
MIN	Yes
COUNT	Yes

12.2.1.2 How pre-aggregate tables are selected

When a user query is submitted, during query planning phase, CarbonData will collect all matched pre-aggregate tables as candidates according to Relational Algebra transformation rules. Then, the best pre-aggregate table for this query will be selected among the candidates based on cost. For simplicity, current cost estimation is based on the data size of the pre-aggregate table. (We assume that query will be faster on smaller table)

For the main table **sales** and pre-aggregate table **agg_sales** created above, following queries ``` SELECT country, sex, sum(quantity), avg(price) from sales GROUP BY country, sex

SELECT sex, sum(quantity) from sales GROUP BY sex

SELECT avg(price), country from sales GROUP BY country ```

will be transformed by CarbonData's query planner to query against pre-aggregate table **agg_sales** instead of the main table **sales**

However, for following queries ``` SELECT user_id, country, sex, sum(quantity), avg(price) from sales GROUP BY user_id, country, sex

SELECT sex, avg(quantity) from sales GROUP BY sex

SELECT country, max(price) from sales GROUP BY country ```

will query against main table **sales** only, because it does not satisfy pre-aggregate table selection logic.

12.3 Loading data

For existing table with loaded data, data load to pre-aggregate table will be triggered by the CREATE DATAMAP statement when user creates the pre-aggregate table. For incremental loads after aggregates tables are created, loading data to main table triggers the load to pre-aggregate tables once main table loading is complete.

These loads are transactional meaning that data on main table and pre-aggregate tables are only visible to the user after all tables are loaded successfully, if one of these loads fails, new data are not visible in all tables as if the load operation is not happened.

12.4 Querying data

As a technique for query acceleration, Pre-aggregate tables cannot be queried directly. Queries are to be made on main table. While doing query planning, internally CarbonData will check associated pre-aggregate tables with the main table, and do query plan transformation accordingly.

User can verify whether a query can leverage pre-aggregate table or not by executing `EXPLAIN` command, which will show the transformed logical plan, and thus user can check whether pre-aggregate table is selected.

12.5 Compacting pre-aggregate tables

Running Compaction command (`ALTER TABLE COMPACT`) on main table will **not automatically** compact the pre-aggregate tables created on the main table. User need to run Compaction command separately on each pre-aggregate table to compact them.

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 tables. To further improve the query performance, compaction on pre-aggregate tables can be triggered to merge the segments and files in the pre-aggregate tables.

12.6 Data Management with pre-aggregate tables

In current implementation, data consistence need to be maintained for both main table and pre-aggregate tables. Once there is pre-aggregate table created on the main table, following command on the main table is not supported: 1. Data management command: `UPDATE/DELETE/DELETE SEGMENT`. 2. Schema management command: `ALTER TABLE DROP COLUMN`, `ALTER TABLE`

`CHANGE DATATYPE`, `ALTER TABLE RENAME`. Note that adding a new column is supported, and for dropping columns and change datatype command, CarbonData will check whether it will impact the pre-aggregate table, if not, the operation is allowed, otherwise operation will be rejected by throwing exception.
3. Partition management command: `ALTER TABLE ADD/DROP PARTITION` 4. Complex Datatypes for preaggregate is not supported.

However, there is still way to support these operations on main table, in current CarbonData release, user can do as following: 1. Remove the pre-aggregate table by `DROP DATAMAP` command 2. Carry out the data management operation on main table 3. Create the pre-aggregate table again by `CREATE DATAMAP` command Basically, user can manually trigger the operation by re-building the datamap.

13 CarbonData Timeseries DataMap

CarbonData Timeseries DataMap

13.1 Timeseries DataMap Introduction (Alpha Feature)

Timeseries DataMap a pre-aggregate table implementation based on ‘pre-aggregate’ DataMap. Difference is that Timeseries DataMap has built-in understanding of time hierarchy and levels: year, month, day, hour, minute, so that it supports automatic roll-up in time dimension for query.

The data loading, querying, compaction command and its behavior is the same as preaggregate DataMap. Please refer to Pre-aggregate DataMap for more information.

To use this datamap, user can create multiple timeseries datamap on the main table which has a *event_time* column, one datamap for one time granularity. Then Carbondata can do automatic roll-up for queries on the main table.

For example, below statement effectively create multiple pre-aggregate tables on main table called **timeseries**

```
CREATE DATAMAP agg_year
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'year_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), sum(
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_month
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'month_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), sum(
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_day
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'day_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), sum(
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_sales_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(
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_minute
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'minute_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), sum(
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_minute
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'minute_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), sum(
 avg(price) FROM sales GROUP BY order_time, country, sex
```

For querying timeseries data, Carbondata has builtin support for following time related UDF to enable automatically roll-up to the desired aggregation level `timeseries(timeseries column name, 'aggregation level')` `SELECT timeseries(order_time, 'hour'), sum(quantity) FROM sales GROUP BY timeseries(order_time, 'hour')`

It is **not necessary** to create pre-aggregate tables for each granularity unless required for query. Carbondata can roll-up the data and fetch it.

For Example: For main table **sales** , if following timeseries datamaps were created for day level and hour level pre-aggregate

```
CREATE DATAMAP agg_day
ON TABLE sales
USING "timeseries"
DMPROPERTIES (
  'event_time'='order_time',
  'day_granularity'='1',
) AS
SELECT order_time, country, sex, sum(quantity), max(quantity), count(user_id), su
 avg(price) FROM sales GROUP BY order_time, country, sex

CREATE DATAMAP agg_sales_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), su
 avg(price) FROM sales GROUP BY order_time, country, sex
```

Queries like below will be rolled-up and hit the timeseries datamaps ``` Select timeseries(order_time, ‘month’), sum(quantity) from sales group by timeseries(order_time, ‘month’)

Select timeseries(order_time, ‘year’), sum(quantity) from sales group by timeseries(order_time, ‘year’) ```

NOTE (**RESTRICTION**): * Only value of 1 is supported for hierarchy levels. Other hierarchy levels will be supported in the future CarbonData release. * timeseries datamap for the desired levels needs to be created one after the other * timeseries datamaps created for each level needs to be dropped separately

13.2 Compacting timeseries datamp

Refer to Compaction section in preaggregation datamap. Same applies to timeseries datamap.

13.3 Data Management on timeseries datamap

Refer to Data Management section in preaggregation datamap. Same applies to timeseries datamap.

14 FAQs

FAQs

14.1 What are Bad Records?

Records that fail to get loaded into the CarbonData due to data type incompatibility or are empty or have incompatible format are classified as Bad Records.

14.2 Where are Bad Records Stored in CarbonData?

The bad records are stored at the location set in carbon.badRecords.location in carbon.properties file. By default **carbon.badRecords.location** specifies the following location `/opt/Carbon/Spark/badrecords`.

14.3 How to enable Bad Record Logging?

While loading data we can specify the approach to handle Bad Records. In order to analyse the cause of the Bad Records the parameter `BAD_RECORDS_LOGGER_ENABLE` must be set to value `TRUE`. There are multiple approaches to handle Bad Records which can be specified by the parameter `BAD_RECORDS_ACTION`.

- To pad the incorrect values of the csv rows with NULL value and load the data in CarbonData, set the following in the query : `'BAD_RECORDS_ACTION'='FORCE'`
- To write the Bad Records without padding incorrect values with NULL in the raw csv (set in the parameter **carbon.badRecords.location**), set the following in the query : `'BAD_RECORDS_ACTION'='REDIRECT'`

14.4 How to ignore the Bad Records?

To ignore the Bad Records from getting stored in the raw csv, we need to set the following in the query : `'BAD_RECORDS_ACTION'='IGNORE'`

14.5 How to specify store location while creating carbon session?

The store location specified while creating carbon session is used by the CarbonData to store the meta data like the schema, dictionary files, dictionary meta data and sort indexes.

Try creating `carbonsession` with `storepath` specified in the following manner :

```
val carbon = SparkSession.builder().config(sc.getConf)
             .getOrCreateCarbonSession(<store_path>)
```

Example:

```
val carbon = SparkSession.builder().config(sc.getConf)
             .getOrCreateCarbonSession("hdfs://localhost:9000/carbon/store")
```

14.6 What is Carbon Lock Type?

The Apache CarbonData acquires lock on the files to prevent concurrent operation from modifying the same files. The lock can be of the following types depending on the storage location, for HDFS we specify it to be of type HDFSLOCK. By default it is set to type LOCALLOCK. The property carbon.lock.type configuration specifies the type of lock to be acquired during concurrent operations on table. This property can be set with the following values : **- LOCALLOCK** : This Lock is created on local file system as file. This lock is useful when only one spark driver (thrift server) runs on a machine and no other CarbonData spark application is launched concurrently. **- HDFSLOCK** : This Lock is created on HDFS file system as file. This lock is useful when multiple CarbonData spark applications are launched and no ZooKeeper is running on cluster and the HDFS supports, file based locking.

14.7 How to resolve Abstract Method Error?

In order to build CarbonData project it is necessary to specify the spark profile. The spark profile sets the Spark Version. You need to specify the `spark version` while using Maven to build project.

14.8 How Carbon will behave when execute insert operation in abnormal scenarios?

Carbon support insert operation, you can refer to the syntax mentioned in DML Operations on CarbonData. First, create a source table in spark-sql and load data into this created table.

```
CREATE TABLE source_table(
id String,
name String,
city String)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ",";
```

```
SELECT * FROM source_table;
id  name    city
1   jack    beijing
2   erlu    hangzhou
3   davi    shenzhen
```

Scenario 1 :

Suppose, the column order in carbon table is different from source table, use script "SELECT * FROM carbon table" to query, will get the column order similar as source table, rather than in carbon table's column order as expected.

```
CREATE TABLE IF NOT EXISTS carbon_table(
id String,
city String,
name String)
STORED BY 'carbondata';
```

```
INSERT INTO TABLE carbon_table SELECT * FROM source_table;
```

```
SELECT * FROM carbon_table;
id  city    name
1   jack    beijing
2   erlu    hangzhou
3   davi    shenzhen
```

As result shows, the second column is city in carbon table, but what inside is name, such as jack. This phenomenon is same with insert data into hive table.

If you want to insert data into corresponding column in carbon table, you have to specify the column order same in insert statement.

```
INSERT INTO TABLE carbon_table SELECT id, city, name FROM source_table;
```

Scenario 2 :

Insert operation will be failed when the number of column in carbon table is different from the column specified in select statement. The following insert operation will be failed.

```
INSERT INTO TABLE carbon_table SELECT id, city FROM source_table;
```

Scenario 3 :

When the column type in carbon table is different from the column specified in select statement. The insert operation will still success, but you may get NULL in result, because NULL will be substitute value when conversion type failed.

14.9 Why aggregate query is not fetching data from aggregate table?

Following are the aggregate queries that won't fetch data from aggregate table:

- **Scenario 1** : When SubQuery predicate is present in the query.

Example:

```
create table gdp21(cntry smallint, gdp double, y_year date) stored by 'carbondata';
create datamap ag1 on table gdp21 using 'preaggregate' as select cntry, sum(gdp) fr
select ctry from pop1 where ctry in (select cntry from gdp21 group by cntry);
```

- **Scenario 2** : When aggregate function along with 'in' filter.

Example:

```
create table gdp21(cntry smallint, gdp double, y_year date) stored by 'carbondata';
create datamap ag1 on table gdp21 using 'preaggregate' as select cntry, sum(gdp) fr
select cntry, sum(gdp) from gdp21 where cntry in (select ctry from pop1) group by c
```

- **Scenario 3** : When aggregate function having 'join' with equal filter.

Example:

```
create table gdp21(cntry smallint, gdp double, y_year date) stored by 'carbondata';
create datamap ag1 on table gdp21 using 'preaggregate' as select cntry, sum(gdp) fr
select cntry,sum(gdp) from gdp21,pop1 where cntry=ctry group by cntry;
```

14.10 Why all executors are showing success in Spark UI even after Dataload command failed at Driver side?

Spark executor shows task as failed after the maximum number of retry attempts, but loading the data having bad records and BAD_RECORDS_ACTION (carbon.bad.records.action) is set as "FAIL" will attempt only once but will send the signal to driver as failed instead of throwing the exception to retry, as there is no point to retry if bad record found and BAD_RECORDS_ACTION is set to fail. Hence the Spark executor displays this one attempt as successful but the command has actually failed to execute. Task attempts or executor logs can be checked to observe the failure reason.

14.11 Why different time zone result for select query output when query SDK writer output?

SDK writer is an independent entity, hence SDK writer can generate carbondata files from a non-cluster machine that has different time zones. But at cluster when those files are read, it always takes cluster time-zone. Hence, the value of timestamp and date datatype fields are not original value. If wanted to control timezone of data while writing, then set cluster's time-zone in SDK writer by calling below API. `TimeZone.setDefault(timezoneValue)` **Example:** `cluster timezone is Asia/Shanghai TimeZone.setDefault(TimeZone.getTimeZone("Asia/Shanghai"))`

15 Troubleshooting

Troubleshooting

This tutorial is designed to provide troubleshooting for end users and developers who are building, deploying, and using CarbonData.

15.1 When loading data, gets tablestatus.lock issues:

Symptom `17/11/11 16:48:13 ERROR LocalFileLock: main hdfs:/localhost:9000/carbon/store/default/hdfstable/tablestatus.lock (No such file or directory) java.io.FileNotFoundException: hdfs:/localhost:9000/carbon/store/default/hdfstable/tablestatus.lock (No such file or directory) at java.io.FileOutputStream.open0(Native Method) at java.io.FileOutputStream.open(FileOutputStream.java:270) at java.io.FileOutputStream.<init>(FileOutputStream.java:213) at java.io.FileOutputStream.<init>(FileOutputStream.java:101)`

Possible Cause If you use `<hdfs path>` as store path when creating carbonsession, may get the errors,because the default is LOCALLOCK.

Procedure Before creating carbonsession, sets as below: `import org.apache.carbondata.core.util.CarbonProperties import org.apache.carbondata.core.constants.CarbonCommonConstants CarbonProperties.getInstance().addProperty(CarbonCommonConstants.LOCK_TYPE, "HDFSLOCK")`

15.2 Failed to load thrift libraries

Symptom

Thrift throws following exception :

```
thrift: error while loading shared libraries: libthriftc.so.0: cannot open
shared object file: No such file or directory
```

Possible Cause

The complete path to the directory containing the libraries is not configured correctly.

Procedure

Follow the Apache thrift docs at https://thrift.apache.org/docs/install to install thrift correctly.

15.3 Failed to launch the Spark Shell

Symptom

The shell prompts the following error :

```
org.apache.spark.sql.CarbonContext$$anon$$apache$spark$sql$catalyst
$analysis $OverrideCatalog$_setter_$org$apache$spark$sql$catalyst$analysis
$OverrideCatalog$$overrides_$e
```

Possible Cause

The Spark Version and the selected Spark Profile do not match.

Procedure

1. Ensure your spark version and selected profile for spark are correct.

2. Use the following command :

```
"mvn -Pspark-2.1 -Dspark.version {yourSparkVersion} clean package"
```

Note : Refrain from using “mvn clean package” without specifying the profile.

15.4 Failed to execute load query on cluster.

Symptom

Load query failed with the following exception:

`Dictionary file is locked for updation.`

Possible Cause

The carbon.properties file is not identical in all the nodes of the cluster.

Procedure

Follow the steps to ensure the carbon.properties file is consistent across all the nodes:

1. Copy the carbon.properties file from the master node to all the other nodes in the cluster. For example, you can use ssh to copy this file to all the nodes.
2. For the changes to take effect, restart the Spark cluster.

15.5 Failed to execute insert query on cluster.

Symptom

Load query failed with the following exception:

`Dictionary file is locked for updation.`

Possible Cause

The carbon.properties file is not identical in all the nodes of the cluster.

Procedure

Follow the steps to ensure the carbon.properties file is consistent across all the nodes:

1. Copy the carbon.properties file from the master node to all the other nodes in the cluster. For example, you can use scp to copy this file to all the nodes.
2. For the changes to take effect, restart the Spark cluster.

15.6 Failed to connect to hiveuser with thrift

Symptom

We get the following exception :

`Cannot connect to hiveuser.`

Possible Cause

The external process does not have permission to access.

Procedure

Ensure that the Hiveuser in mysql must allow its access to the external processes.

15.7 Failed to read the metastore db during table creation.

Symptom

We get the following exception on trying to connect :

```
Cannot read the metastore db
```

Possible Cause

The metastore db is dysfunctional.

Procedure

Remove the metastore db from the carbon.metastore in the Spark Directory.

15.8 Failed to load data on the cluster

Symptom

Data loading fails with the following exception :

```
Data Load failure exception
```

Possible Cause

The following issue can cause the failure :

1. The core-site.xml, hive-site.xml, yarn-site and carbon.properties are not consistent across all nodes of the cluster.
2. Path to hdfs ddl is not configured correctly in the carbon.properties.

Procedure

Follow the steps to ensure the following configuration files are consistent across all the nodes:

1. Copy the core-site.xml, hive-site.xml, yarn-site,carbon.properties files from the master node to all the other nodes in the cluster. For example, you can use scp to copy this file to all the nodes.

   Note : Set the path to hdfs ddl in carbon.properties in the master node.
2. For the changes to take effect, restart the Spark cluster.

15.9 Failed to insert data on the cluster

Symptom

Insertion fails with the following exception :

```
Data Load failure exception
```

Possible Cause

The following issue can cause the failure :

1. The core-site.xml, hive-site.xml, yarn-site and carbon.properties are not consistent across all nodes of the cluster.
2. Path to hdfs ddl is not configured correctly in the carbon.properties.

Procedure

Follow the steps to ensure the following configuration files are consistent across all the nodes:

1. Copy the core-site.xml, hive-site.xml, yarn-site,carbon.properties files from the master node to all the other nodes in the cluster. For example, you can use scp to copy this file to all the nodes.

   Note : Set the path to hdfs ddl in carbon.properties in the master node.

2. For the changes to take effect, restart the Spark cluster.

15.10 Failed to execute Concurrent Operations(Load,Insert,Update) on table by multiple workers.

Symptom

Execution fails with the following exception :

`Table is locked for updation.`

Possible Cause

Concurrency not supported.

Procedure

Worker must wait for the query execution to complete and the table to release the lock for another query execution to succeed.

15.11 Failed to create a table with a single numeric column.

Symptom

Execution fails with the following exception :

`Table creation fails.`

Possible Cause

Behaviour not supported.

Procedure

A single column that can be considered as dimension is mandatory for table creation.

16 Useful Tips

Useful Tips

This tutorial guides you to create CarbonData Tables and optimize performance. The following sections will elaborate on the below topics :

- Suggestions to create CarbonData Table
- Configuration for Optimizing Data Loading performance for Massive Data
- Optimizing Mass Data Loading

16.1 Suggestions to Create CarbonData Table

For example, the results of the analysis for table creation with dimensions ranging from 10 thousand to 10 billion rows and 100 to 300 columns have been summarized below. The following table describes some of the columns from the table used.

- **Table Column Description**

Column Name	Data Type	Cardinality	Attribution
msisdn	String	30 million	Dimension
BEGIN_TIME	BigInt	10 Thousand	Dimension
HOST	String	1 million	Dimension
Dime_1	String	1 Thousand	Dimension
counter_1	Decimal	NA	Measure
counter_2	Numeric(20,0)	NA	Measure
…	…	NA	Measure
counter_100	Decimal	NA	Measure

- **Put the frequently-used column filter in the beginning**

For example, MSISDN filter is used in most of the query then we must put the MSISDN in the first column. The create table command can be modified as suggested below :

```
create table carbondata_table( msisdn String, BEGIN_TIME bigint, HOST String, Dime_1 String, counter_1, Decimal …

)STORED BY 'carbondata'
TBLPROPERTIES ('SORT_COLUMNS'='msisdn, Dime_1')
```

Now the query with MSISDN in the filter will be more efficient.

- **Put the frequently-used columns in the order of low to high cardinality**

If the table in the specified query has multiple columns which are frequently used to filter the results, it is suggested to put the columns in the order of cardinality low to high. This ordering of frequently used columns improves the compression ratio and enhances the performance of queries with filter on these columns.

For example, if MSISDN, HOST and Dime_1 are frequently-used columns, then the column order of table is suggested as Dime_1>HOST>MSISDN, because Dime_1 has the lowest cardinality. The create table command can be modified as suggested below :

```
create table carbondata_table( msisdn String, BEGIN_TIME bigint, HOST String, Dime_1 String, counter_1, Decimal …

  )STORED BY 'carbondata'
  TBLPROPERTIES ('SORT_COLUMNS'='Dime_1, HOST, MSISDN')
```

- **For measure type columns with non high accuracy, replace Numeric(20,0) data type with Double data type**

For columns of measure type, not requiring high accuracy, it is suggested to replace Numeric data type with Double to enhance query performance. The create table command can be modified as below :

```
create table carbondata_table(
  Dime_1 String,
  BEGIN_TIME bigint,
  END_TIME bigint,
  HOST String,
  MSISDN String,
  counter_1 decimal,
  counter_2 double,
  ...
  )STORED BY 'carbondata'
  TBLPROPERTIES ('SORT_COLUMNS'='Dime_1, HOST, MSISDN')
```

The result of performance analysis of test-case shows reduction in query execution time from 15 to 3 seconds, thereby improving performance by nearly 5 times.

- **Columns of incremental character should be re-arranged at the end of dimensions**

Consider the following scenario where data is loaded each day and the begin_time is incremental for each load, it is suggested to put begin_time at the end of dimensions. Incremental values are efficient in using min/max index. The create table command can be modified as below :

```
create table carbondata_table( Dime_1 String, HOST String, MSISDN
String, counter_1 double, counter_2 double, BEGIN_TIME bigint, END_TIME
bigint, ... counter_100 double )STORED BY 'carbondata' TBLPROPERTIES
('SORT_COLUMNS'='Dime_1, HOST, MSISDN')
```

NOTE: + BloomFilter can be created to enhance performance for queries with precise equal/in conditions. You can find more information about it in BloomFilter datamap document.

16.2 Configuration for Optimizing Data Loading performance for Massive Data

CarbonData supports large data load, in this process sorting data while loading consumes a lot of memory and disk IO and this can result sometimes in “Out Of Memory” exception. If you do not have much memory to use, then you may prefer to slow the speed of data loading instead of data load failure. You can configure CarbonData by tuning following properties in carbon.properties file to get a better performance.

Parameter	Default Value	Description/Tuning
carbon.number.of.cores.while.loading	Default: 2.This value should be >= 2	Specifies the number of cores used for data processing during data loading in CarbonData.
carbon.sort.size	Default: 100000. The value should be >= 100.	Threshold to write local file in sort step when loading data
carbon.sort.file.write.buffer.size	Default: 50000.	DataOutputStream buffer.
carbon.number.of.cores.block.sort	Default: 7	If you have huge memory and CPUs, increase it as you will
carbon.merge.sort.reader.thread	Default: 3	Specifies the number of cores used for temp file merging during data loading in CarbonData.
carbon.merge.sort.prefetch	Default: true	You may want set this value to false if you have not enough memory

For example, if there are 10 million records, and i have only 16 cores, 64GB memory, will be loaded to CarbonData table. Using the default configuration always fail in sort step. Modify carbon.properties as suggested below:

```
carbon.number.of.cores.block.sort=1 carbon.merge.sort.reader.thread=1
carbon.sort.size=5000 carbon.sort.file.write.buffer.size=5000
carbon.merge.sort.prefetch=false
```

16.3 Configurations for Optimizing CarbonData Performance

Recently we did some performance POC on CarbonData for Finance and telecommunication Field. It involved detailed queries and aggregation scenarios. After the completion of POC, some of the configurations impacting the performance have been identified and tabulated below :

| Parameter | Location | Used For | Description | Tuning | |
————————————————————————-|——————————————————|——————————————|

| carbon.sort.intermediate.files.limit | spark/carbonlib/carbon.properties | Data loading | During the loading of data, local temp is used to sort the data. This number specifies the minimum number of intermediate files after which the merge sort has to be initiated. | Increasing the parameter to a higher value will improve the load performance. For example, when we increase the value from 20 to 100, it increases the data load performance from 35MB/S to more than 50MB/S. Higher values of this parameter consumes more memory during the load. | | carbon.number.of.cores.while.loading | spark/carbonlib/carbon.properties | Data loading | Specifies the number of cores used for data processing during data loading in CarbonData. | If you have more number of CPUs, then you can increase the number of CPUs, which will increase the performance. For example if we increase the value from 2 to 4 then the CSV reading performance can increase about 1 times | | carbon.compaction.level.threshold | spark/carbonlib/carbon.properties | Data loading and Querying | For minor compaction, specifies the number of segments to be merged in stage 1 and number of compacted segments to be merged in stage 2. | Each CarbonData load will create one segment, if every load is small in size it will generate many small file over a period of time impacting the query performance. Configuring this parameter will merge the small segment to one big segment which will sort the data and improve the performance. For Example in one telecommunication scenario, the performance improves about 2 times after minor compaction. | | spark.sql.shuffle.partitions | spark/conf/spark-defaults.conf | Querying | The number of task started when spark shuffle. | The value can be 1 to 2 times as much as the executor cores. In an aggregation scenario, reducing the number from 200 to 32 reduced the query time from 17 to 9 seconds. | | spark.executor.instances/spark.executor.cores/spark.executor.memory | spark/conf/spark-defaults.conf | Querying | The number of executors, CPU cores, and memory used for CarbonData query. | In the bank scenario, we provide the 4 CPUs cores and 15 GB for each executor which can get good performance. This 2 value does not mean more the better. It needs to be configured properly in case of limited resources. For example, In the bank scenario, it has enough CPU 32 cores each node but less memory 64 GB each node. So we cannot give more CPU but less memory. For example, when 4 cores and 12GB for each executor. It sometimes happens GC during the query which impact the query performance very much from the 3 second to more than 15 seconds. In this scenario need to increase the memory or decrease the CPU cores. | | carbon.detail.batch.size | spark/carbonlib/carbon.properties | Data loading | The buffer size to store records, returned from the block scan. | In limit scenario this parameter is very important. For example your query limit is 1000. But if we set this value to 3000 that means we get 3000 records from scan but spark will only take 1000 rows. So the 2000 remaining are useless. In one Finance test case after we set it to 100, in the limit 1000 scenario the performance increase about 2 times in comparison to if we set this value to 12000. | | carbon.use.local.dir | spark/carbonlib/carbon.properties | Data loading | Whether use YARN local directories for multi-table load disk load balance | If this is set it to true CarbonData will use YARN local directories for multi-table load disk load balance, that will improve the data load performance. | | carbon.use.multiple.temp.dir | spark/carbonlib/carbon.properties | Data loading | Whether to use multiple YARN local directories during table data loading for disk load balance | After enabling 'carbon.use.local.dir', if this is set to true, CarbonData will use all YARN local directories during data load for disk load balance, that will improve the data load

performance. Please enable this property when you encounter disk hotspot problem during data loading. | | carbon.sort.temp.compressor | spark/carbonlib/carbon.properties | Data loading | Specify the name of compressor to compress the intermediate sort temporary files during sort procedure in data loading. | The optional values are ‘SNAPPY’,‘GZIP’,‘BZIP2’,‘LZ4’,‘ZSTD’ and empty. By default, empty means that Carbondata will not compress the sort temp files. This parameter will be useful if you encounter disk bottleneck. | | carbon.load.skewedDataOptimization.enabled | spark/carbonlib/carbon.properties | Data loading | Whether to enable size based block allocation strategy for data loading. | When loading, carbondata will use file size based block allocation strategy for task distribution. It will make sure that all the executors process the same size of data – It’s useful if the size of your input data files varies widely, say 1MB~1GB. | | carbon.load.min.size.enabled | spark/carbonlib/carbon.properties | Data loading | Whether to enable node minumun input data size allocation strategy for data loading.| When loading, carbondata will use node minumun input data size allocation strategy for task distribution. It will make sure the node load the minimum amount of data – It’s useful if the size of your input data files very small, say 1MB~256MB,Avoid generating a large number of small files. |

Note: If your CarbonData instance is provided only for query, you may specify the property ‘spark.speculation=true’ which is in conf directory of spark.