Apache CarbonData

Ver 1.0

Documentation

The Apache Software Foundation **2017-01-20**

Table of Contents

1 Quick Start

Quick Start

This tutorial provides a quick introduction to using CarbonData.

1.1 Getting started with Apache CarbonData

1.2 Installation

- Download a released package of Spark 1.6.2 or 2.1.0.
- Download and install Apache Thrift 0.9.3, make sure Thrift is added to system path.
- Download Apache CarbonData code and build it. Please visit Building CarbonData And IDE Configuration for more information.

1.3 Prerequisites

- 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
```

1.4 Interactive Analysis with Spark Shell

1.5 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.

1.5.1.1 Basics

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

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

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()
```

1.5.1.2 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(s"load data inpath
'${new java.io.File("../carbondata/sample.csv").getCanonicalPath}'
into table test_table")
```

1.Query Data from a Table

```
scala>spark.sql("select * from test_table").show

scala>spark.sql("select city, avg(age),
sum(age) from test_table group by city").show
```

1.6 Interactive Analysis with Spark Shell

1.7 Version 1.6

1.7.1.1 Basics

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

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

NOTE: In this shell, SparkContext is readily available as sc.

- In order to execute the Queries we need to import CarbonContext:

```
import org.apache.spark.sql.CarbonContext
```

- Create an instance of CarbonContext in the following manner :

```
val cc = new CarbonContext(sc)
```

NOTE: By default store location is pointed to “../carbon.store”, user can provide own store location to CarbonContext like new CarbonContext(sc, storeLocation).

1.7.1.2 Executing Queries

1.Creating a Table

```
scala>cc.sql("create table if not exists test_table
(id string, name string, city string, age Int) STORED BY 'carbondata'")
```

To see the table created :

```
scala>cc.sql("show tables").show
```

1.Loading Data to a Table

```
scala>cc.sql(s"load data inpath
'${new java.io.File("../carbondata/sample.csv").getCanonicalPath}'
into table test_table")
```

1.Query Data from a Table

```
scala>cc.sql("select * from test_table").show
scala>cc.sql("select city, avg(age), sum(age)
from test_table group by city").show
```

1.8 Building CarbonData

To get started, get CarbonData from the downloads section on the http://carbondata.incubator.apache.org. CarbonData uses Hadoop's client libraries for HDFS and YARN and Spark's libraries. Downloads are pre-packaged for a handful of popular Spark versions.

If you'd like to build CarbonData from source, visit Building CarbonData And IDE Configuration.

2 User Guide

User Guide

Welcome to Apache CarbonData. Apache CarbonData(incubating) is a new big data file format for faster interactive query using advanced columnar storage, index, compression and encoding techniques to improve computing efficiency, which helps in speeding up queries by an order of magnitude faster over PetaBytes of data. This user guide provides a detailed description about the CarbonData and its features.

Let’s get started !

3 Overview

Overview

This tutorial provides a detailed overview about :

- Introduction
- Features

3.1 Introduction

CarbonData is a fully indexed columnar and Hadoop native data-store for processing heavy analytical workloads and detailed queries on big data. CarbonData allows faster interactive query using advanced columnar storage, index, compression and encoding techniques to improve computing efficiency, which helps in speeding up queries by an order of magnitude faster over PetaBytes of data.

In customer benchmarks, CarbonData has proven to manage Petabyte of data running on extraordinarily low-cost hardware and answers queries around 10 times faster than the current open source solutions (column-oriented SQL on Hadoop data-stores).

Some of the salient features of CarbonData are :

- Low-Latency for various types of data access patterns like Sequential, Random and OLAP.
- Fast query on fast data.
- Space efficiency.
- General format available on Hadoop-ecosystem.

3.2 Features

CarbonData file format is a columnar store in HDFS. It has many features that a modern columnar format has, such as splittable, compression schema, complex data type etc and CarbonData has following unique features:

- Unique Data Organization: Though CarbonData stores data in Columnar format, it differs from traditional Columnar formats as the columns in each row-group(Data Block) is sorted independent of the other columns. Though this arrangement requires CarbonData to store the row-number mapping against each column value, it makes it possible to use binary search for faster filtering and since the values are sorted, same/similar values come together which yields better compression and offsets the storage overhead required by the row number mapping.
- Advanced Push Down Optimizations: CarbonData pushes as much of query processing as possible close to the data to minimize the amount of data being read, processed, converted and transmitted/shuffled. Using projections and filters it reads only the required columns form the store and also reads only the rows that match the filter conditions provided in the query.
- Multi Level Indexing: CarbonData uses multiple indices at various levels to enable faster search and speed up query processing.
- Global Multi Dimensional Keys(MDK) based B+Tree Index for all non- measure columns: Aids in quickly locating the row groups(Data Blocks) that contain the data matching search/filter criteria.
- Min-Max Index for all columns: Aids in quickly locating the row groups(Data Blocks) that contain the data matching search/filter criteria.
- Data Block level Inverted Index for all columns: Aids in quickly locating the rows that contain the data matching search/filter criteria within a row group(Data Blocks).
- Dictionary Encoding: Most databases and big data SQL data stores employ columnar encoding to achieve data compression by storing small integers numbers (surrogate value) instead of

full string values. However, almost all existing databases and data stores divide the data into row groups containing anywhere from few thousand to a million rows and employ dictionary encoding only within each row group. Hence, the same column value can have different surrogate values in different row groups. So, while reading the data, conversion from surrogate value to actual value needs to be done immediately after the data is read from the disk. But CarbonData employs global surrogate key which means that a common dictionary is maintained for the full store on one machine/node. So CarbonData can perform all the query processing work such as grouping/aggregation, sorting etc on light weight surrogate values. The conversion from surrogate to actual values needs to be done only on the final result. This procedure improves performance on two aspects. Conversion from surrogate values to actual values is done only for the final result rows which are much less than the actual rows read from the store. All query processing and computation such as grouping/aggregation, sorting, and so on is done on lightweight surrogate values which requires less memory and CPU time compared to actual values.

- Deep Spark Integration: It has built-in spark integration for Spark 1.5, 1.6 and interfaces for Spark SQL, DataFrame API and query optimization. It supports bulk data ingestion and allows saving of spark dataframes as CarbonData files.
- Update Delete Support: It supports batch updates like daily update scenarios for OLAP and Base +Delta file based design.
- Store data along with index: Significantly accelerates query performance and reduces the I/O scans and CPU resources, when there are filters in the query. CarbonData index consists of multiple levels of indices. A processing framework can leverage this index to reduce the task it needs to schedule and process. It can also do skip scan in more finer grain units (called blocklet) in task side scanning instead of scanning the whole file.
- Operable encoded data: It supports efficient compression and global encoding schemes and can query on compressed/encoded data. The data can be converted just before returning the results to the users, which is “late materialized”.
- Column group: Allows multiple columns to form a column group that would be stored as row format. This reduces the row reconstruction cost at query time.
- Support for various use cases with one single Data format: Examples are interactive OLAP-style query, Sequential Access (big scan) and Random Access (narrow scan).

4 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 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.

Each blocklet in the file is further divided into chunks of data called data chunks. Each data chunk is organized either in columnar format or row format, and stores the data of either a single column or a set of columns. All blocklets in a file contain the same number and type of data chunks.

Each data chunk contains multiple groups of data called as pages. There are three types of pages.

- Data Page: Contains the encoded data of a column/group 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.

5 Data Types

Data Types

5.1.1.1 CarbonData supports the following data types:

- Numeric Types
- SMALLINT
- INT/INTEGER
- BIGINT
- DOUBLE
- DECIMAL
- Date/Time Types
- TIMESTAMP
- String Types
- STRING
- Complex Types
  - arrays: ARRAY `<data_type>`
  - structs: STRUCT `<col_name : data_type COMMENT col_comment, ...>`

6 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

6.1 Installing and Configuring CarbonData on Standalone Spark Cluster

6.1.1 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.

6.1.2 Procedure

- Build the CarbonData project and get the assembly jar from “./assembly/target/scala-2.10/carbondata_xxx.jar” and put in the `"<SPARK_HOME>/carbonlib"` folder.

  NOTE: Create the carbonlib folder if it does not exists inside `"<SPARK_HOME>"` path.
- 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)
- Copy the carbon.properties.template to `"<SPARK_HOME>/conf/carbon.properties"` folder from “./conf/” of CarbonData repository.
- Copy the “carbonplugins” folder to `"<SPARK_HOME>/carbonlib"` folder from “./processing/” folder of CarbonData repository.

  NOTE: carbonplugins will contain .kettle folder.
- In Spark node, configure the properties mentioned in the following table in `"<SPARK_HOME>/conf/spark-defaults.conf"` file.

Property	Value	Description
carbon.kettle.home	$SPARK_HOME /carbonlib/ carbonplugins	Path that will be used by CarbonData internally to create graph for loading the data
spark.driver.extraJavaOptions	-Dcarbon.properties.filepath= $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.filepath= $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.

- Add the following properties in "<SPARK_HOME>/conf/" carbon.properties:

Property	Required	Description	Example	Remark
carbon.storelocation	NO	Location where data CarbonData will create the store and write the data in its own format.	hdfs://HOSTNAME:PORT/Opt/CarbonStore	Propose to set HDFS directory
carbon.kettle.home	YES	Path that will be used by CarbonData internally to create graph for loading the data.	$SPARK_HOME/carbonlib/carbonplugins	

- 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 , DDL Operations on CarbonData

6.2 Installing and Configuring CarbonData on “Spark on YARN” Cluster

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

6.2.1 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.

6.2.2 Procedure

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

- Build the CarbonData project and get the assembly jar from “./assembly/target/scala-2.10/carbondata_xxx.jar” and put in the "<SPARK_HOME>/carbonlib" folder.

  NOTE: Create the carbonlib folder if it does not exists inside "<SPARK_HOME>" path.
- Copy “carbonplugins” folder to "<SPARK_HOME>/carbonlib" folder from “./processing/” folder of CarbonData repository. carbonplugins will contain .kettle folder.
- Copy the “carbon.properties.template” to "<SPARK_HOME>/conf/carbon.properties" folder from conf folder of CarbonData repository.
- Modify the parameters in “spark-default.conf” located in the "<SPARK_HOME>/conf"

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.	"<YOUR_SPARK_HOME_PATH>"/conf/carbon.properties
spark.yarn.dist.archives	Comma-separated list of archives to be extracted into the working directory of each executor.	"<YOUR_SPARK_HOME_PATH>"/carbonlib/carbondata_xxx.jar
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="<YOUR_SP conf/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	"<YOUR_SPARK_HOME_PATH>"/carbonlib/carbonlib/carbondata_xxx.jar
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.	"<YOUR_SPARK_HOME_PATH>"/carbonlib/carbonlib/carbondata_xxx.jar
spark.driver.extraJavaOptions	A string of extra JVM options to pass to the driver. For instance, GC settings or other logging.	-Dcarbon.properties.filepath="<YOUR_SP conf/carbon.properties
carbon.kettle.home	Path that will be used by CarbonData internally to create graph for loading the data.	"<YOUR_SPARK_HOME_PATH>"/carbonlib/carbonplugins

- 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.	hdfs://HOSTNAME:PORT/Opt/CarbonStore	Propose to set HDFS directory
carbon.kettle.home	YES	Path that will be used by CarbonData internally to create graph for loading the data.	$SPARK_HOME/carbonlib/carbonplugins	

- 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 , DDL Operations on CarbonData

6.3 Query Execution Using CarbonData Thrift Server

6.3.1 Starting CarbonData Thrift Server

a. cd <SPARK_HOME>

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

```
./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>
```

Parameter	Description	Example
CARBON_ASSEMBLY_JAR	CarbonData assembly jar name present in the "<SPARK_HOME>"/carbonlib/ folder.	carbondata_2.10-0.1.0-incubating-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.	hdfs//<host_name>:54310/user/hive/warehouse/carbon.store

6.3.2 Examples

- Start with default memory and executors

```
./bin/spark-submit --conf spark.sql.hive.thriftServer.singleSession=true
--class org.apache.carbondata.spark.thriftserver.CarbonThriftServer
$SPARK_HOME/carbonlib
/carbondata_2.10-0.1.0-incubating-SNAPSHOT-shade-hadoop2.7.2.jar
hdfs://hacluster/user/hive/warehouse/carbon.store
```

- Start with Fixed executors and resources

```
./bin/spark-submit --conf spark.sql.hive.thriftServer.singleSession=true
--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_2.10-0.1.0-incubating-SNAPSHOT-shade-hadoop2.7.2.jar
hdfs://hacluster/user/hive/warehouse/carbon.store
```

6.3.3 Connecting to CarbonData Thrift Server Using Beeline

```
cd <SPARK_HOME>
./bin/beeline jdbc:hive2://<thrftserver_host>:port

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

7 Configuring CarbonData

Configuring CarbonData

This tutorial guides you through the advanced configurations of CarbonData :

- System Configuration
- Performance Configuration
- Miscellaneous Configuration
- Spark Configuration

7.1 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	/user/hive/warehouse/carbon.store	Location where CarbonData will create the store, and write the data in its own format. NOTE: Store location should be in HDFS.
carbon.ddl.base.hdfs.url	hdfs://hacluster/opt/data	This property is used to configure the HDFS relative path from the HDFS base path, configured in fs.defaultFS. 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	/opt/Carbon/Spark/badrecords	Path where the bad records are stored.
carbon.kettle.home	$SPARK_HOME/carbonlib/carbonplugins	Path used by CarbonData internally to create graph for loading the data.

carbon.data.file.version	2		If this parameter value is set to 1, then CarbonData will support the data load which is in old format. If the value is set to 2, then CarbonData will support the data load of new format only. NOTE: The file format created before DataSight Spark V100R002C30 is considered as old format.

7.2 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.sort.file.buffer.size	20	File read buffer size used during sorting. This value is expressed in MB.	Min=1 and Max=100
carbon.graph.rowset.size	100000	Rowset size exchanged between data load graph steps.	Min=500 and Max=1000000
carbon.number.of.cores.whi	6	Number of cores to be used while loading data.	
carbon.sort.size	500000	Record count to sort and write intermediate files to temp.	
carbon.enableXXHash	true	Algorithm for hashmap for hashkey calculation.	
carbon.number.of.cores.blo	7	Number of cores to use for block sort while loading data.	
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.ca	-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.update.persist.enabl	true	Enabling this parameter considers persistent data. Enabling this will reduce the execution time of UPDATE operation.

- **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.	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.

- **Query Configuration**

Parameter	Default Value	Description	Range
carbon.number.of.cores	4	Number of cores to be used while querying.	
carbon.inmemory.record.siz	120000	Number of records to be in memory while querying.	Min=100000 and Max=240000
carbon.enable.quick.filter	false	Improves the performance of filter query.	
no.of.cores.to.load.blocks.in	10	Number of core to load the blocks in driver.	

7.3 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	10485760	File write buffer size used during sorting.
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.sort.intermediate.files.limit	20	Minimum number of intermediate files after which merged sort can be started.
carbon.block.meta.size.reserved.perc	10	Space reserved in percentage for writing block meta data in CarbonData file.
carbon.csv.read.buffersize.byte	1048576	csv reading buffer size.
high.cardinality.value	100000	To identify and apply compression for non-high cardinality columns.
carbon.merge.sort.reader.thread	3	Maximum no of threads used for reading intermediate files for final merging.
carbon.load.metadata.lock.retries	3	Maximum number of retries to get the metadata lock for loading data to table.
carbon.load.metadata.lock.retry.timec	5	Interval between the retries to get the lock.
carbon.tempstore.location	/opt/Carbon/TempStoreLoc	Temporary store location. By default it takes System.getProperty("java.io.tmpdir").
carbon.load.log.counter	500000	Data loading records count logger.

- **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.

- **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.

- **Global Dictionary Configurations**

Parameter	Default Value	Description

high.cardinality.identify.enable	true	If the parameter is true, the high cardinality columns of the dictionary code are automatically recognized and these columns will not be used as global dictionary encoding. If the parameter is false, all dictionary encoding columns are used as dictionary encoding. The high cardinality column must meet the following requirements: value of cardinality > configured value of high.cardinalityEqually, the value of cardinality is higher than the threshold.value of cardinality/ row number x 100 > configured value of high.cardinality.row.count.percentageEqually, the ratio of the cardinality value to data row number is higher than the configured percentage.
high.cardinality.threshold	1000000	Threshold to identify whether high cardinality column.Configuration value formula: Value of cardinality > configured value of high.cardinality. The minimum value is 10000.
high.cardinality.row.count.percentage	80	Percentage to identify whether column cardinality is more than configured percent of total row count.Configuration value formula:Value of cardinality/ row number x 100 > configured value of high.cardinality.row.count.percentageThe value of the parameter must be larger than 0.
carbon.cutOffTimestamp	1970-01-01 05:30:00	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”. NOTE: The CarbonData supports data store up to 68 years from the cut-off time defined. For example, if the cut-off time is 1970-01-01 05:30:00, then the data can be stored up to 2038-01-01 05:30:00.
carbon.timegranularity	SECOND	The property used to set the data granularity level DAY, HOUR, MINUTE, or SECOND.

7.4 Spark Configuration

Spark Configuration Reference in spark-defaults.conf

Parameter	Default Value	Description
spark.driver.memory	1g	Amount of memory to use for the driver process, i.e. where SparkContext is initialized. NOTE: In client mode, this config must not be set through the SparkConf directly in your application, because the driver JVM has already started at that point. Instead, please set this through the –driver-memory command line option or in your default properties file.
spark.executor.memory	1g	Amount of memory to use per executor process.
spark.sql.bigdata.register.analyseRul	org.apache.spark.sql.hive.acl.Carbon	CarbonAccessControlRules need to be set for enabling Access Control.

8 Using CarbonData

Using CarbonData

This tutorial discusses the disciplines related to management of data in Apache CarbonData. Following below each section is a brief introduction to respective disciplines related to data management.

8.1 Data Management

This section shall be dealing with the disciplines related to managing data in the application, focusing on conceptual details related to operations like load data, delete data, update data and Compacting Data.

For complete details refer to Data Management

8.2 Data Definition Language Support

This section deals with the aspects related to creation and modification of the structure of database. It shall discuss in detail about

- Table creation
- Table deletion
- Table description
- Compaction

For complete details refer to DDL Operations on CarbonData

8.3 Data Manipulation Language Support

This section deals with the aspects related to data manipulation in database. It shall discuss in detail about selecting, loading and deleting in a database. This manipulation comprises of

- Loading data into database tables
- Retrieving existing data
- Deleting data from existing tables
- Deleting segments from existing tables
- Updating data in existing tables

For complete details refer to DML Operations on CarbonData

9 Data Management

Data Management

This tutorial is going to introduce you to the conceptual details of data management like:

- Loading Data
- Deleting Data
- Compacting Data
- Updating Data

9.1 Loading Data

- **Scenario**

After creating a table, you can load data to the table using the LOAD DATA command. The loaded data is available for querying. When data load is triggered, the data is encoded in CarbonData format and copied into HDFS CarbonData store path (specified in carbon.properties file) in compressed, multi dimensional columnar format for quick analysis queries. The same command can be used to load new data or to update the existing data. Only one data load can be triggered for one table. The high cardinality columns of the dictionary encoding are automatically recognized and these columns will not be used for dictionary encoding.

- **Procedure**

Data loading is a process that involves execution of multiple steps to read, sort and encode the data in CarbonData store format. Each step is executed on different threads. After data loading process is complete, the status (success/partial success) is updated to CarbonData store metadata. The table below lists the possible load status.

Status	Description
Success	All the data is loaded into table and no bad records found.
Partial Success	Data is loaded into table and bad records are found. Bad records are stored at carbon.badrecords.location.

In case of failure, the error will be logged in error log. Details of loads can be seen with SHOW SEGMENTS command. The show segment command output consists of :

- SegmentSequenceID
- START_TIME OF LOAD
- END_TIME OF LOAD
- LOAD STATUS

The latest load will be displayed first in the output.

Refer to DML operations on CarbonData for load commands.

9.2 Deleting Data

- **Scenario**

If you have loaded wrong data into the table, or too many bad records are present and you want to modify and reload the data, you can delete required data loads. The load can be deleted using the Segment Sequence Id or if the table contains date field then the data can be deleted using the date field. If there are some specific records that need to be deleted based on some filter condition(s) we can delete by records.

- **Procedure**

The loaded data can be deleted in the following ways:

- Delete by Segment ID

  After you get the segment ID of the segment that you want to delete, execute the DELETE command for the selected segment. The status of deleted segment is updated to Marked for delete / Marked for Update.

SegmentSequenceId	Status	Load Start Time	Load End Time
0	Success	2015-11-19 19:14:...	2015-11-19 19:14:...
1	Marked for Update	2015-11-19 19:54:...	2015-11-19 20:08:...
2	Marked for Delete	2015-11-19 20:25:...	2015-11-19 20:49:...

- Delete by Date Field

  If the table contains date field, you can delete the data based on a specific date.
- Delete by Record

  To delete records from CarbonData table based on some filter Condition(s).

  For delete commands refer to DML operations on CarbonData.
- **NOTE**:
  - When the delete segment DML is called, segment will not be deleted physically from the file system. Instead the segment status will be marked as “Marked for Delete”. For the query execution, this deleted segment will be excluded.
  - The deleted segment will be deleted physically during the next load operation and only after the maximum query execution time configured using “max.query.execution.time”. By default it is 60 minutes.
  - If the user wants to force delete the segment physically then he can use CLEAN FILES Command.

Example :

```
CLEAN FILES FOR TABLE table1
```

This DML will physically delete the segment which are “Marked for delete” immediately.

9.3 Compacting Data

- **Scenario**

Frequent data ingestion results in several fragmented CarbonData files in the store directory. Since data is sorted only within each load, the indices perform only within each load. This means that there will be one index for each load and as number of data load increases, the number of indices also increases. As each index works only on one load, the performance of indices is reduced. CarbonData provides provision for compacting the loads. Compaction process combines several segments into one large segment by merge sorting the data from across the segments.

- **Procedure**

There are two types of compaction Minor and Major compaction.

- **Minor Compaction**

  In minor compaction the user can specify how many loads to be merged. Minor compaction triggers for every data load if the parameter carbon.enable.auto.load.merge is set. 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 bigger segment.

- **Major Compaction**

  In Major compaction, many segments can be merged into one big segment. User will specify the compaction size until which segments can be merged. Major compaction is usually done during the off-peak time.

There are number of parameters related to Compaction that can be set in carbon.properties file

Parameter	Default	Application	Description	Valid Values
carbon.compaction.le\	4, 3	Minor	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.	NA
carbon.major.compac\	1024 MB	Major	Major compaction size can be configured using this parameter. Sum of the segments which is below this threshold will be merged.	NA
carbon.numberof.pres	0	Minor/Major	If the user wants to preserve some number of segments from being compacted then he can set this property. Example: carbon.numberof.pres then 2 latest segments will always be excluded from the compaction. No segments will be preserved by default.	0-100

carbon.allowed.compa	0	Minor/Major	Compaction will merge the segments which are loaded within 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.	0-100
carbon.number.of.core	2	Minor/Major	Number of cores which is used to write data during compaction.	0-100

For compaction commands refer to DDL operations on CarbonData

9.4 Updating Data

- **Scenario**

  Sometimes after the data has been ingested into the System, it is required to be updated. Also there may be situations where some specific columns need to be updated on the basis of column expression and optional filter conditions.

- **Procedure**

  To update we need to specify the column expression with an optional filter condition(s).

  For update commands refer to DML operations on CarbonData.

10 DDL

DDL Operations on CarbonData

This tutorial guides you through the data definition language support provided by CarbonData.

10.1 Overview

The following DDL operations are supported in CarbonData :

- CREATE TABLE
- SHOW TABLE
- DROP TABLE
- COMPACTION

10.2 CREATE TABLE

This command can be used to create a CarbonData table by specifying the list of fields along with the table properties.

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
                [(col_name data_type , ...)]
STORED BY 'carbondata'
[TBLPROPERTIES (property_name=property_value, ...)]
// All Carbon's additional table options will go into properties
```

10.2.1 Parameter Description

Parameter	Description	Optional
db *name* | *Name of the database. Database name should consist of alphanumeric characters and underscore(*) special character.	Yes	
field *list* | *Comma separated List of fields with data type. The field names should consist of alphanumeric characters and underscore(*) special character.	No	
table *name* | *The name of the table in Database. Table Name should consist of alphanumeric characters and underscore(*) special character.	No	
STORED BY	“org.apache.carbondata.format”, identifies and creates a CarbonData table.	No
TBLPROPERTIES	List of CarbonData table properties.	

10.2.2 Usage Guidelines

Following are the guidelines for using table properties.

- **Dictionary Encoding Configuration**

  Dictionary encoding is enabled by default for all String columns, and disabled for non-String columns. You can include and exclude columns for dictionary encoding.

```
TBLPROPERTIES ("DICTIONARY_EXCLUDE"="column1, column2")
TBLPROPERTIES ("DICTIONARY_INCLUDE"="column1, column2")
```

Here, DICTIONARY_EXCLUDE will exclude dictionary creation. This is applicable for high-cardinality columns and is an optional parameter. DICTIONARY_INCLUDE will generate dictionary for the columns specified in the list.

- **Row/Column Format Configuration**

  Column groups with more than one column are stored in row format, instead of columnar format. By default, each column is a separate column group.

```
TBLPROPERTIES ("COLUMN_GROUPS"="(column1,column3),
(Column4,Column5,Column6)")
```

- **Table Block Size Configuration**

  The block size of table files can be defined using the property TABLE_BLOCKSIZE. It accepts only integer values. The default value is 1024 MB and supports a range of 1 MB to 2048 MB. If you do not specify this value in the DDL command , default value is used.

```
TBLPROPERTIES ("TABLE_BLOCKSIZE"="512 MB")
```

Here 512 MB means the block size of this table is 512 MB, you can also set it as 512M or 512.

- **Inverted Index Configuration**

  Inverted index is very useful to improve compression ratio and query speed, especially for those low-cardinality columns who are in reward position. By default inverted index is enabled. The user can disable the inverted index creation for some columns.

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

No inverted index shall be generated for the columns specified in NO_INVERTED_INDEX. This property is applicable on columns with high-cardinality and is an optional parameter.

NOTE:

- By default all columns other than numeric datatype are treated as dimensions and all columns of numeric datatype are treated as measures.
- All dimensions except complex datatype columns are part of multi dimensional key(MDK). This behavior can be overridden by using TBLPROPERTIES. If the user wants to keep any column (except columns of complex datatype) in multi dimensional key then he can keep the columns either in DICTIONARY_EXCLUDE or DICTIONARY_INCLUDE.

10.2.3 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 ('COLUMN_GROUPS'='(productName,productCategory)',
           'DICTIONARY_EXCLUDE'='productName',
           'DICTIONARY_INCLUDE'='productNumber',
           'NO_INVERTED_INDEX'='productBatch')
```

10.3 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];`

10.3.1 Parameter Description

Parameter	Description	Optional
IN db_Name	Name of the database. Required only if tables of this specific database are to be listed.	Yes

10.3.2 Example:

```
SHOW TABLES IN ProductSchema;
```

10.4 DROP TABLE

This command is used to delete an existing table.

```
DROP TABLE [IF EXISTS] [db_name.]table_name;
```

10.4.1 Parameter Description

Parameter	Description	Optional

db_Name	Name of the database. If not specified, current database will be selected.	YES
table_name	Name of the table to be deleted.	NO

10.4.2 Example:

```
DROP TABLE IF EXISTS productSchema.productSalesTable;
```

10.5 COMPACTION

This command merges the specified number of segments into one segment. This enhances the query performance of the table.

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

To get details about Compaction refer to Data Management

10.5.1 Parameter Description

Parameter	Description	Optional
db_name	Database name, if it is not specified then it uses current database.	YES
table_name	The name of the table in provided database.	NO

10.5.2 Syntax

- **Minor Compaction**

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

- **Major Compaction**

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

11 DML

DML Operations on CarbonData

This tutorial guides you through the data manipulation language support provided by CarbonData.

11.1 Overview

The following DML operations are supported in CarbonData :

- LOAD DATA
- INSERT DATA INTO A CARBONDATA TABLE
- SHOW SEGMENTS
- DELETE SEGMENT BY ID
- DELETE SEGMENT BY DATE
- UPDATE CARBONDATA TABLE
- DELETE RECORDS FROM CARBONDATA TABLE

11.2 LOAD DATA

This command loads the user data in raw format to the CarbonData specific data format store, this allows CarbonData to provide good performance while querying the data. Please visit Data Management for more details on LOAD.

11.2.1 Syntax

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

OPTIONS are not mandatory for data loading process. Inside OPTIONS user can provide either of any options like DELIMITER, QUOTECHAR, ESCAPECHAR, MULTILINE as per requirement.

NOTE: The path shall be canonical path.

11.2.2 Parameter Description

Parameter	Description	Optional
folder_path	Path of raw csv data folder or file.	NO
db_name	Database name, if it is not specified then it uses the current database.	YES
table_name	The name of the table in provided database.	NO
OPTIONS	Extra options provided to Load	YES

11.2.3 Usage Guidelines

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='#')
```

- **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 on CSV.

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

- **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:** Date format for specified column.

```
OPTIONS('DATEFORMAT'='column1:dateFormat1, column2:dateFormat2')
```

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

11.2.4 Example:

```
LOAD DATA local inpath '/opt/rawdata/data.csv' INTO table carbontable
options('DELIMITER'=',', 'QUOTECHAR'='"','COMMENTCHAR'='#',
'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'
)
```

11.3 INSERT DATA INTO A 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.

NOTE : The client node where the INSERT command is executing, must be part of the cluster.

11.3.1 Syntax

```
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> } ];
```

11.3.2 Parameter Description

Parameter	Description
CARBON TABLE	The name of the Carbon table in which you want to perform the insert operation.
sourceTableName	The table from which the records are read and inserted into destination CarbonData table.

11.3.3 Usage Guidelines

The following condition must be met for successful insert operation :

- The source table and the CarbonData table must have the same table schema.
- The table must be created.
- Overwrite is not supported for CarbonData table.
- The data type of source and destination table columns should be same, else the data from source table will be treated as bad records and the INSERT command fails.
- 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.

To enable data load or update during insert operation, configure the following property to true.

```
carbon.insert.persist.enable=true
```

By default the above configuration will be false.

NOTE: Enabling this property will reduce the performance.

11.3.4 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 INTO table1 SELECT * FROM table2
where exists (select * from table3
where table2.item1 = table3.item1);
```

The Status Success/Failure shall be captured in the driver log.

11.4 SHOW SEGMENTS

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

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

11.4.1 Parameter Description

Parameter	Description	Optional
db_name	Database name, if it is not specified then it uses the current database.	YES

table_name	The name of the table in provided database.	NO
number_of_segments	Limit the output to this number.	YES

11.4.2 Example:

```
SHOW SEGMENTS FOR TABLE CarbonDatabase.CarbonTable LIMIT 4;
```

11.5 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 dbname.tablename 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 SEGMENT segment_sequence_id1, segments_sequence_id2, ....
FROM TABLE tableName
```

11.5.1 Parameter Description

Parameter	Description	Optional
segment_id	Segment Id of the load.	NO
db_name	Database name, if it is not specified then it uses the current database.	YES
table_name	The name of the table in provided database.	NO

11.5.2 Example:

```
DELETE SEGMENT 0 FROM TABLE CarbonDatabase.CarbonTable;
DELETE SEGMENT 0.1,5,8 FROM TABLE CarbonDatabase.CarbonTable;
```

NOTE: Here 0.1 is compacted segment sequence id.

11.6 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 [schema_name.]table_name
WHERE[DATE_FIELD]BEFORE [DATE_VALUE]
```

11.6.1 Parameter Description

Parameter	Description	Optional
DATE_VALUE	Valid segment load start time value. All the segments before this specified date will be deleted.	NO
db_name	Database name, if it is not specified then it uses the current database.	YES
table_name	The name of the table in provided database.	NO

11.6.2 Example:

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

11.7 Update CarbonData Table

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

11.7.1 Syntax

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

alternatively the following the 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> } ];
```

11.7.2 Parameter Description

Parameter	Description

table_name	The name of the Carbon table in which you want to perform the update operation.
column_name	The destination columns to be updated.
sourceColumn	The source table column values to be updated in destination table.
sourceTable	The table from which the records are updated into destination Carbon table.

11.7.3 Usage Guidelines

The following conditions must be met for successful updation :

- The update command fails if multiple input rows in source table are matched with single row in destination table.
- If the source table generates empty records, the update operation will complete successfully without updating the table.
- If a source table row does not correspond to any of the existing rows in a destination table, the update operation will complete successfully without updating the table.
- In sub-query, if the source table and the target table are same, then the update operation fails.
- If the sub-query used in UPDATE statement contains aggregate method or group by query, then the UPDATE operation fails.

11.7.4 Examples

Update is not supported for queries that contain aggregate or group by.

```
UPDATE t_carbn01 a
SET (a.item_type_code, a.profit) = ( SELECT b.item_type_cd,
sum(b.profit) from t_carbn01b b
WHERE item_type_cd =2 group by item_type_code);
```

Here the Update Operation fails as the query contains aggregate function sum(b.profit) and group by clause in the sub-query.

```
UPDATE carbonTable1 d
SET(d.column3,d.column5 ) = (SELECT s.c33 ,s.c55
FROM sourceTable1 s WHERE d.column1 = s.c11)
WHERE d.column1 = 'china' EXISTS( SELECT * from table3 o where o.c2 > 1);
```

```
UPDATE carbonTable1 d SET (c3) = (SELECT s.c33 from sourceTable1 s
WHERE d.column1 = s.c11)
WHERE exists( select * from iud.other o where o.c2 > 1);
```

```
UPDATE carbonTable1 SET (c2, c5 ) = (c2 + 1, concat(c5 , "y" ));
```

```
UPDATE carbonTable1 d SET (c2, c5 ) = (c2 + 1, "xyx")
WHERE d.column1 = 'india';
```

```
UPDATE carbonTable1 d SET (c2, c5 ) = (c2 + 1, "xyx")
WHERE d.column1 = 'india'
and EXISTS( SELECT * FROM table3 o WHERE o.column2 > 1);
```

The Status Success/Failure shall be captured in the driver log and the client.

11.8 Delete Records from CarbonData Table

This command allows us to delete records from CarbonData table.

11.8.1 Syntax

```
DELETE FROM table_name [WHERE expression];
```

11.8.2 Parameter Description

Parameter	Description
table_name	The name of the Carbon table in which you want to perform the delete.

11.8.3 Examples

```
DELETE FROM columncarbonTable1 d WHERE d.column1  = 'china';
```

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

```
DELETE FROM columncarbonTable1
WHERE column1 IN (SELECT column11 FROM sourceTable2);
```

```
DELETE FROM columncarbonTable1
WHERE column1 IN (SELECT column11 FROM sourceTable2 WHERE
column1 = 'USA');
```

```
DELETE FROM columncarbonTable1 WHERE column2 >= 4
```

The Status Success/Failure shall be captured in the driver log and the client.

12 Useful Tips

Useful Tips

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

- Suggestions to create CarbonData Table
- Configurations For Optimizing CarbonData Performance

12.1 Suggestions to Create CarbonData Table

Recently CarbonData was used to analyze performance of Telecommunication field. 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	Numeric(20,0)	NA	Measure
…	…	NA	Measure
counter_100	Numeric(20,0)	NA	Measure

CarbonData has more than 50 test cases, on the basis of these we have following suggestions to enhance the query performance :

- **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,
...
)STORED BY 'org.apache.carbondata.format'
TBLPROPERTIES ( 'DICTIONARY_EXCLUDE'='MSISDN,..',
'DICTIONARY_INCLUDE'='...');
```

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 as Dime_1 has the lowest cardinality. The create table command can be modified as suggested below :

```
create table carbondata_table(
Dime_1 String,
HOST String,
MSISDN String,
...
)STORED BY 'org.apache.carbondata.format'
TBLPROPERTIES ( 'DICTIONARY_EXCLUDE'='MSISDN,HOST..',
'DICTIONARY_INCLUDE'='Dime_1..');
```

- **Put the Dimension type columns in order of low to high cardinality**

If the columns used to filter are not frequently used, then it is suggested to order all the columns of dimension type in order of low to high cardinality. The create table command can be modified as below :

```
create table carbondata_table(
Dime_1 String,
BEGIN_TIME bigint
HOST String,
MSISDN String,
...
)STORED BY 'org.apache.carbondata.format'
TBLPROPERTIES ( 'DICTIONARY_EXCLUDE'='MSISDN,HOST,IMSI..',
'DICTIONARY_INCLUDE'='Dime_1,END_TIME,BEGIN_TIME..');
```

- **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
HOST String,
MSISDN String,
counter_1 double,
counter_2 double,
...
counter_100 double
)STORED BY 'org.apache.carbondata.format'
TBLPROPERTIES ( 'DICTIONARY_EXCLUDE'='MSISDN,HOST,IMSI',
'DICTIONARY_INCLUDE'='Dime_1,END_TIME,BEGIN_TIME');
```

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 start_time is incremental for each load, it is suggested to put start_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,
...
counter_100 double
)STORED BY 'org.apache.carbondata.format'
TBLPROPERTIES ( 'DICTIONARY_EXCLUDE'='MSISDN,HOST,IMSI',
'DICTIONARY_INCLUDE'='Dime_1,END_TIME,BEGIN_TIME');
```

- **Avoid adding high cardinality columns to dictionary**

If the system has low memory configuration, then it is suggested to exclude high cardinality columns from the dictionary to enhance load performance. Creation of dictionary for high cardinality columns at time of load will degrade load performance due to excessive memory usage.

By default CarbonData determines the cardinality at the first data load and allows for dictionary creation only if the cardinality is less than 1 million.

12.2 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.intermedia	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.core	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.lev	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.partit	spark/con/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.

num-executors/ executor-cores/ executor-memory	spark/con/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.siz	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.

13 Use Cases

CarbonData Use Cases

This tutorial discusses about the problems that CarbonData addresses. It shall take you through the identified top use cases of CarbonData.

13.1 Introduction

For big data interactive analysis scenarios, many customers expect sub-second response to query TB-PB level data on general hardware clusters with just a few nodes.

In the current big data ecosystem, there are few columnar storage formats such as ORC and Parquet that are designed for SQL on Big Data. Apache Hive's ORC format is a columnar storage format with basic indexing capability. However, ORC cannot meet the sub-second query response expectation on TB level data, as it performs only stride level dictionary encoding and all analytical operations such as filtering and aggregation is done on the actual data. Apache Parquet is a columnar storage format that can improve performance in comparison to ORC due to its more efficient storage organization. Though Parquet can provide query response on TB level data in a few seconds, it is still far from the sub-second expectation of interactive analysis users. Cloudera Kudu can effectively solve some query performance issues, but kudu is not hadoop native, can't seamlessly integrate historic HDFS data into new kudu system.

However, CarbonData uses specially engineered optimizations targeted to improve performance of analytical queries which can include filters, aggregation and distinct counts, the required data to be stored in an indexed, well organized, read-optimized format, CarbonData's query performance can achieve sub-second response.

13.2 Motivation: Single Format to provide Low Latency Response for all Use Cases

The main motivation behind CarbonData is to provide a single storage format for all the usecases of querying big data on Hadoop. Thus CarbonData is able to cover all use-cases into a single storage format.

13.3 Use Cases

13.3.1 Sequential Access

- Supports queries that select only a few columns with a group by clause but do not contain any filters. This results in full scan over the complete store for the selected columns.

Scenario

- ETL jobs
- Log Analysis

13.3.2 Random Access

- Supports Point Query. These are queries used from operational applications and usually select all or most of the columns and involves a large number of filters which reduce the result to a small size. Such queries generally do not involve any aggregation or group by clause.

- Row-key query(like HBase)
- Narrow Scan
- Requires second/sub-second level low latency

Scenario

- Operational Query
- User Profiling

13.3.3 Olap Style Query

- Supports Interactive data analysis for any dimensions. These are queries which are typically fired from Interactive Analysis tools. Such queries often select a few columns and involves filters and group by on a column or a grouping expression. It also supports queries that :
  - Involves aggregation/join
  - Roll-up,Drill-down,Slicing and Dicing
  - Low-latency ad-hoc query

Scenario

- Dash-board reporting
- Fraud & Ad-hoc Analysis

14 Troubleshooting

Troubleshooting

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

14.1.1 General Prevention and Best Practices

- When trying to create a table with a single numeric column, table creation fails: One column that can be considered as dimension is mandatory for table creation.
- “Files locked for updation” when same table is accessed from two or more instances: Remove metastore_db from the examples folder.

15 FAQs

FAQs

- **Auto Compaction not Working**

  The Property carbon.enable.auto.load.merge in carbon.properties need to be set to true.
- **Getting Abstract method error**

  You need to specify the spark version while using Maven to build project.
- **Getting NotImplementedException for subquery using IN and EXISTS**

  Subquery with in and exists not supported in CarbonData.
- **Getting Exceptions on creating a view**

  View not supported in CarbonData.
- **How to verify if ColumnGroups have been created as desired.**

  Try using desc table query.
- **Did anyone try to run CarbonData on windows? Is it supported on Windows?**

  We may provide support for windows in future. You are welcome to contribute if you want to add the support :)