Working with HDFS File Formats for Hive and user defined SerDe

ANJANEYULU GUNTI (366244)
Cognizant Technology Solutions

AIM:

This White Paper explains, how the hive Query Engine works and it explains when to use Different file formats for storing the HDFS data for optimized way of using space and query performance when using in Hive. This article also explains reading and writing the custom file format by implementing the custom in-out File Format's and Custom SerDe implementation. This article also lists common best practices for selecting the data file format.

Contents

1. Hive Introduction

Apache Hive is a data warehouse infrastructure built on top of Hadoop that facilitates querying, analyzing and managing large datasets residing in distributed storage. Hive is designed to enable easy data summarization. Hive provides a language called HiveQL which allows users to query and is similar to SQL.

Like SQL, HiveQL handles structured and semi structured data. By default Hive has derby database to store the Meta data. We can configure Hive for data and MySQL database for its Meta data. HiveQL is also very extensible. It supports user defined column transformation (UDF) and aggregation (UDAF) functions for demanding data processing.

Hive also allows users to embed custom map-reduce scripts written in any language using a simple row-based streaming interface, traditional map reduce programs, when it is inconvenient or inefficient to execute the logic in HiveQL. Hive can easily integrated with other data technologies by using Hive JDBC connection.

1.1 Hive Query Execution Process

The Fig: 1 shows the basic **Hadoop Hive architecture.** Primarily The diagram represents CLI (Command Line Interface), JDBC/ODBC and Web GUI (Web Graphical User Interface).This represents when user uses CLI (Hive Terminal) then CLI directly connects to Hive Drivers, When User uses JDBC/ODBC (JDBC Program) at that time by using API (Thrift Server) and is connected to Hive driver and when the user uses Web GUI (Ambari server) it is directly connected to Hive Driver.

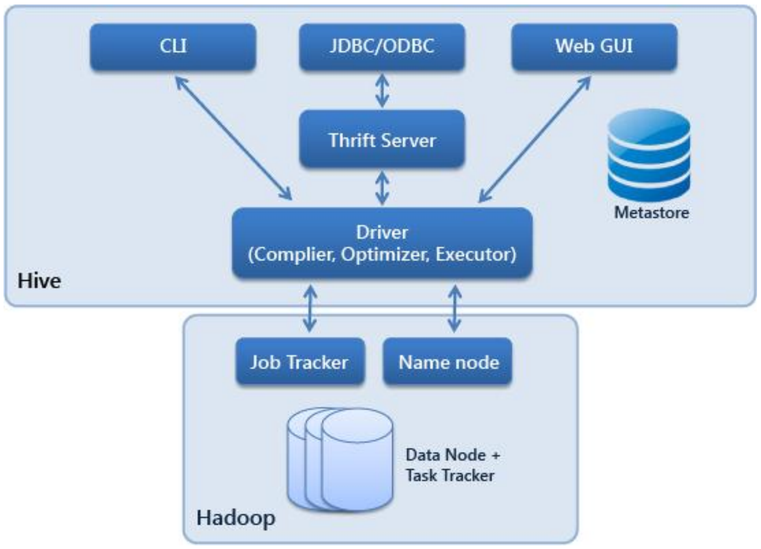

Fig. 1: Hive System Architecture

As Fig 2: shows, the HiveQL statement is submitted via the CLI, the web UI or an external client using the thrift, Odbc or Jdbc interfaces. The driver first passes the query to the compiler where it goes through the typical parse, type check and semantic analysis phases, using the metadata stored in the Metastore.

The compiler generates a logical plan that is then optimized through a simple rule based optimizer. Finally an optimized plan in the form of a DAG of map-reduce tasks and HDFS tasks is generated. The execution engine then executes these tasks in the order of their dependencies, using Hadoop.

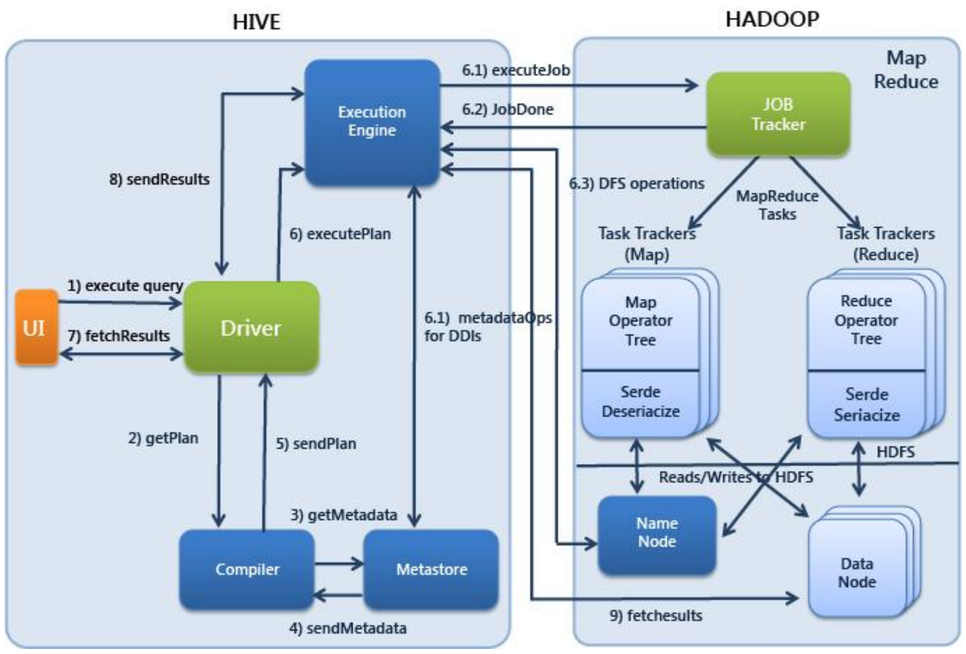

Fig 2: Hive Query Execution in Hadoop Hive Architecture

For simple query, only mappers run. The Input Output format is responsible for managing an input split and reading the data from HDFS. Next, the data flows into a layer called SerDe (Serializer Deserializer). In this case data as byte stream gets converted to a structured format by the deserializer part of the SerdDe.

For aggregate queries, the Map Reduce jobs will also include reducers. In this case, the serializer of the SerDe converts structured data to byte stream which gets handed over to the Input Output format which writes it to the HDFS.

A typical Hive query will involve one or more Map Reduce jobs and full file scan of the data, unless the table has partitions.

Warning

Hive is a batch processing system and hive jobs takes much latency to execute the quires comparing to other databases like RDBMS. The RDBMS databases can supports only GBs of data but in Hive we can execute more than TBs of data. Hive aims to provide acceptable (but not optimal) latency for interactive data browsing, queries over small data sets or test queries. Hive is not designed for online transaction supports and does not offer real-time queries and row level updates. It is best used for batch jobs over large sets of immutable data (like web logs).

2 Hive Input File Formats:

Input formats are playing very important role in Hive performance. Hive works equally well on Thrift, control delimited, or custom specialized data formats. Hive uses Files available in the HDFS or any other storage (FTP) to store data.

Hive does not verify whether the data that are loading matches the schema for the table or not. However, it verifies if the file format matches the table definition or not.
Choosing the optimal file format in Hadoop is one of the most essential drivers of functionality and performance for big data processing and query**.**

2.1 File Format

A file format in Hadoop specifies how records are stored in a file. A file format is the way in which information is stored or encoded in a computer file.

In Hive it refers to how records are stored inside the file. As we are dealing with structured data, each record has to be its own structure. How records are encoded in a file defines a file format. These file formats mainly varies between data encoding, compression rate, usage of space and disk I/O.

2.2 Key considerations while selecting the file formats:

Hadoop can store and process structured and unstructured data like video, text, etc. This is a significant advantage for Hadoop. There are many uses of Hadoop with structured or flexibly structured data. Data with flexible structure can have fields added, changed or removed over time and even vary amongst concurrently ingested records.

Then, why do file formats matter so much?"

2.2.1 What processing and query tools are using?

The popular tools may not support all popular file formats. The Avro files can be readable and writable only with the Avro readers and writers. The Cloudera distribution Impala currently does not support ORC file format. The Horton works Hive-Stinger does not support the Parquet file format. We need to double check before make any final decisions on file format.

2.2.2 If data structure change over time?

If our data structure changes over time, then we need to select the file formats enable flexible and evolving schema.

2.2.3 Is the file format is splittable?

Hadoop stores and processes data in blocks that must be able to begin reading data at any point within a file in order to take fullest advantage of Hadoop’s distributed processing. For example, CSV files are splittable since, we can start reading at any line in the file and the data will still make sense, but the XML file is not splittable since XML has an opening tag at the beginning and a closing tag at the end. We cannot start processing at any point in the middle of those tags.

2.2.4 Is file format accepts block compression?

Hadoop stores large files by splitting them into blocks, its best if the blocks can be independently compressed. If a file format does not support block compression then, if file is compressed and it becomes rendered non-splittable. When processed, the decompressor must begin reading the file at its beginning in order to obtain any block within the file. For a large file with many blocks, this could generate a substantial performance penalty.

2.2.5 How big are the files?

Large files in Hadoop consume a lot of disk -- especially when considering standard 3x replication. So, there is an economic incentive to compress data. I.e. store more data per byte of disk. There is also a performance incentive as disk IO is expensive.

2.2.6 Concerned about processing or query performance?

A columnar, compressed file format like Parquet or ORC may optimize partial and full read performance, but they do so at the expense of write performance. Conversely, uncompressed CSV files are fast to write but due to the lack of compression and column-orientation are slow for reads.

When we have multiple use cases for Hadoop data, then we will use multiple file formats. It is quite common to store the same (or very similar) data in multiple file formats to support varied processing, query and extract requirements. Primary choices of Input Format are Text, Sequence File, RC File, and ORC.

Data is eventually stored in files. There are some specific file formats which Hive can handle such as:

- TEXTFILE
- SEQUENCEFILE
- RCFILE
- ORCFILE
- PARAQUET FILE
- CUSTOM FILE

2.3 TEXTFILE

TEXTFILE format is quite common and often used for exchanging data between Hadoop and external systems. They are readable and ubiquitously parsable. They come in handy when doing a dump from a database or bulk loading data from Hadoop into an analytic database.

In Hive if we define a table as TEXTFILE it can load data of form CSV (Comma Separated Values), delimited by Tabs, Spaces and JSON data. This means fields in each record should be separated by comma or space or tab or it may be JSON (Java Script Object Notation) data.
By default if we use TEXTFILE format then each line is considered as a record.

We can create a TEXTFILE format in Hive as follows:
create table table_name (schema of the table) row format delimited by ',' stored as TEXTFILE.

At the end we need to specify the type of file format. If we do not specify anything it will consider the file format as TEXTFILE format.
The TEXTFILE input and TEXTFILE output format is present in the Hadoop package as shown below:

```
org.apache.hadoop.mapred.TextInputFormat
org.apache.hadoop.mapred.TextOutputFormat
```

2.3.1 Creating TEXTFILE

Creating the table with comma delimited TEXTFILE format.

```
CREATE TABLE Table_TEXT (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
```

We can check the schema of created table using:

```
describe Table_TEXT;
```

Inserting the data into Table_Text from local text file.

```
LOAD DATA LOCAL INPATH '/Users/data/comment.txt' OVERWRITE INTO TABLE
Table_Text;
```

2.3.2 The challenges of using text formats.

The first catch is that dealing with row and field delimiters with text is painstaking. It regularly takes a bit of data that can make for nasty surprises. The input delimiter, no matter how unlikely a character may be, some input source will contain it and break the file format. Never rely solely on a character or byte combination to be unlikely enough to not show up inside big data.

Text files do not support block compression, thus compressing a Text file in Hadoop often comes at a significant read performance cost.

The Text/CSV files in Hadoop are never include header or footer lines in it. That means there is no metadata stored with the Text file. We must know how the file was written in order to make use of it. Also, the file structure is dependent on field order; new fields can only be appended at the end of records while existing fields can never be deleted. As such, Text files have limited support for schema evolution.

Text is a verbose means of storing data and it forces Hive to scan the whole data for every query. Moreover, if the data can contain the delimiters and requires escaping, or is complex like JSON or logs, then a SerDe (Serializer/Deserializer extensions) is essential to evaluate the data. They are freely available for CSV, TSV, JSON, and RegEx, for example, but cost performance since they parse the input after reading it.

2.3.3 POSSIBLE SOLUTION:

Out of the box Hive merely supports a trivial text file format as an input, which does not allow for row or field delimiters inside the fields or rows, i.e. escaping them as supported by most systems reading CSV. A possible option to make it still work is to employ a pre-processing step using sed or tr (*shell commands), i.e., to replace unwanted characters before reading it with Hive.

Some performance enhancement can be obtained by partitioning the data in a directory structure, e.g. by year, month, and days for logs, or by region and countries for geographic data. It enables Hive to only query and read requested parts of the data.

Using compressed files as data files. Hive also ships with a wide variety of compression algorithms. Hive can transparently read and write Gzip, Bzip, LZO, or Snappy compressed files, trading space and IO for computing time. Snappy is a good choice since it is a lightweight compression that will balance CPU and IO needs. Importantly, Hadoop mappers reading it can split it into smaller chunks as needed.

2.4 SEQUENCEFILE

Sequence files store data in a binary format with a similar structure to CSV, which is a binary storage format for key value pairs. It has the benefit of being more compact than text and fits well the map-reduce output format. Sequence files can be compressed on value, or block level, to improve its IO profile further.

When Hive converts queries to MapReduce jobs, it decides on the appropriate key-value pairs to be used for a given record. Sequence files are in binary format which are able to split and the main use of these files is to club two or more smaller files and make them as a one sequence file.

In Hive we can create a sequence file by specifying STORED AS SEQUENCEFILE in the end of a CREATE TABLE statement. There are three types of sequence files:

- Uncompressed key/value records.
- Record compressed key/value records – only 'values' are compressed here
- Block compressed key/value records – both keys and values are collected in 'blocks' separately and compressed. The size of the 'block' is configurable.

Hive has its own SEQUENCEFILE reader and SEQUENCEFILE writer for reading and writing through sequence files. In Hive we can create a sequence file format as follows: create table table_name (schema of the table) row format delimited by \t,' | stored as SEQUENCEFILE. Hive uses the SEQUENCEFILE input and output formats from the following packages:

```
org.apache.hadoop.mapred.SequenceFileInputFormat
org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
```

2.4.1 Creating SEQUENCEFILE

Here we are creating a table with name Text_Sequence and the schema of the table is as specified above and the data inside my input file is delimited by tab space. At the end the file format is specified as SEQUENCEFILE format.

```
CREATE TABLE Table_TEXT (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS SEQUENCEFILE;
```

We can check the schema of created table using:

```
describe Text_sequence;
```

Now to load data into this table is somewhat different from loading into the table created using TEXTFILE format. We need to insert the data from another table because this SEQUENCEFILE format is binary format. It compresses the data and then stores it into the table. If we want to load directly as in TEXTFILE format that is not possible because we cannot insert the compressed files into tables. So to load the data into SEQUENCEFILE we need to use the following approach: The below code snippet will create the LzoCodec compressed file from Table_Text data.

```
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=com.hadoop.compression.lzo.LzoCodec;
`
INSERT OVERWRITE TABLE Table_sequence
SELECT * FROM Table_Text;
```

We have already created table by name Table_Text which is of TEXTFILE format and then we are writing the contents of the Table_Text table into Table_sequence table.

Thus we have successfully loaded data into the SEQUENCEFILE.

2.4.2 The challenges of using Sequence Files:

Sequence files do not store metadata with the data so the only schema evolution option is appending new fields.

Unfortunately, sequence files are not an optimal solution for Hive since it saves a complete row as a single binary value. Consequently, Hive has to read a full row and decompress it even if only one column is being requested.

2.5 RCFILE

RCFILE stands of Record Columnar File which is another type of binary file format which offers high compression rate on the top of the rows.
RCFILE is used when we want to perform operations on multiple rows at a time.

The goal of the format development was (1) fast data loading, (2) fast query processing, (3) highly efficient storage space utilization, and (4) strong adaptively to highly dynamic workload patterns.

As per Fig 3 shows: The RCFile splits data horizontally into row groups. For example, rows 1 to 100 are stored in one group and rows 101 to 200 in the next and so on. One or several groups are stored in a HDFS file. The RCFile saves the row group data in a columnar format. So instead of storing row one then row two, it stores column one across all rows then column two across all rows and so on.

The benefit of this data organization is that Hadoop's parallelism still applies since the row groups in different files are distributed redundantly across the cluster and processable at the same time. Subsequently, each processing node reads only the columns relevant to a query from a file and skips irrelevant ones. Additionally, compression on a column base is more efficient. It can take advantage of similarity of the data in a column.

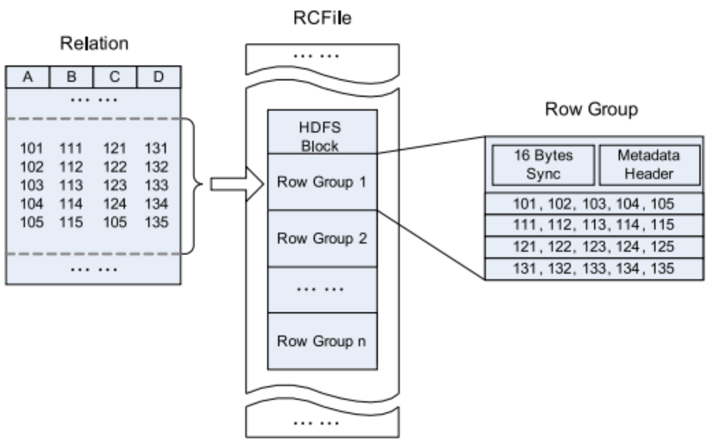

Fig 3: Hive RCFile format

Storing the intermediate tables as (compressed) RCFile reduces the IO and storage requirements significantly over text, sequence file, and row formats in general. Querying tables stored in RCFile format allows Hive to skip large parts of the data and get the results faster and cheaper.

This column oriented storage is very useful while performing analytics. It is easy to perform analytics when we "hive' a column oriented storage type.
Facebook uses RCFILE as its default file format for storing of data in their data warehouse as they perform different types of analytics using Hive.

In Hive we can create a RCFILE format as follows:
create table table_name (schema of the table) row format delimited by ',' stored as RCFILE
Hive has its own RCFILE Input format and RCFILE output format in its default package:

```
org.apache.hadoop.hive.ql.io.RCFileInputFormat
org.apache.hadoop.hive.ql.io.RCFileOutputFormat
```

2.5.1 Creating RCFILE

```
CREATE TABLE Table_RCFILE (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS RCFILE;
```

Here we are creating a table with name Table_rcfile and the schema of the table is as specified above. The data inside the input file is delimited by tab space. At the end the file format is specified as RCFILE format. We can check the schema of created table using:

```
describe Table_RCFILE;
```

We cannot load data into RCFILE directly. First we need to load data into another table and then we need to overwrite it into our newly created RCFILE as shown below: The below code snippet will create the LzoCodec compressed file from Table_Text data.

```
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=com.hadoop.compression.lzo.LzoCodec;

INSERT OVERWRITE TABLE Table_RCFILE
SELECT * FROM Table_Text;
```

2.5.2 The challenges of using RC files:

The current SerDes for RC files in Hive and other tools do not support schema evolution. In order to add a column to data, then data must rewrite every pre-existing RC file.

The RC files are good for querying, writing an RC file requires more memory and computation than non-columnar file formats. They are generally slower to write.

2.6 ORCFILE

The Horton works- Stinger initiative heads the ORC file format development to replace the RCFile. ORC stands for Optimized Row Columnar which means it can store data in an optimized way than the other file formats. ORC reduces the size of the original data up to 75%. As a result the speed of data processing also increases. ORC shows better performance than Text, Sequence and RC file formats.

As Fig 4: Shows, the ORC stores collections of rows in one file and within the collection the row data is stored in a columnar format. This allows parallel processing of row collections across a cluster. Each file with the columnar layout is optimized for compression and skipping of data/columns to reduce read and decompression load.

ORC goes beyond RCFile and uses specific encoders for different column data types to improve compression further, e.g. variable length compression on integers.

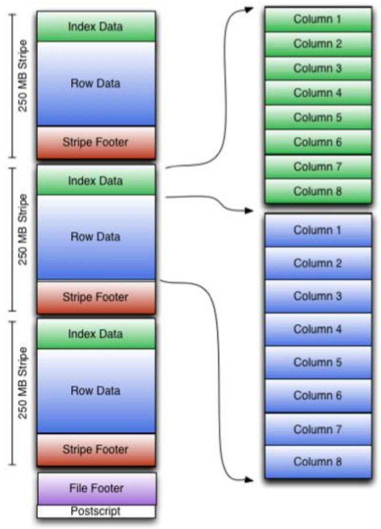

Fig 4: Hive ORCFile format

ORC introduces a lightweight indexing that enables skipping of complete blocks of rows that do not match a query. It comes with basic statistics — min, max, sum, and count — on columns. Lastly, a larger block size of 256 MB by default optimizes for large sequential reads on HDFS for more throughput and fewer files to reduce load on the namenode.

Hive has its own ORCFILE Input format and ORCFILE output format in its default package:

```
org.apache.hadoop.hive.ql.io.ORCFileInputFormat
org.apache.hadoop.hive.ql.io.ORCFileOutputFormat
```

In Hive we can create a RCFILE format as follows:

2.6.1 Creating ORCFILE

```
 CREATE TABLE Table_ORCFILE (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS ORCFILE;
```

Here we are creating a table with name Table_orcfile and the schema of the table is as specified above. The data inside the input file is delimited by tab space.At the end the file format is specified as ORCFILE format. We can check the schema of created table using:

```
describe Table_orcfile;
```

We cannot load data into ORCFILE directly. First we need to load data into another table and then we need to overwrite it into our newly created ORCFILE.

```
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=com.hadoop.compression.lzo.LzoCodec;

INSERT OVERWRITE TABLE Table_ORCFILE
SELECT * FROM Table_Text;
```

We have already created table by name Table_Text which is of TEXTFILE format and then we need to write the contents of the Table_Text table into Table_orcfile table.

2.6.2 The challenges of using ORC files:

The current SerDes for ORC files in Hive and other tools do not support schema evolution. In order to add a column to data, then data must rewrite every pre-existing ORC file.

The ORC files compress to be the smallest of all file formats in Hadoop. It is worthwhile to note that, at the time of this writing an RC file requires more memory and computation than non-columnar file formats. They are generally slower to write.

Cloudera Impala does not support ORC files.

2.7 PARQUETFILE

Parquet Files are yet another columnar file format that originated from Hadoop creator Doug Cutting's Trevni project.

Like RC and ORC, Parquet enjoys compression and query performance benefits and it is generally slower to write than non-columnar file formats. However, unlike RC and ORC files Parquet serdes support limited schema evolution. In Parquet, new columns can be added at the end of the structure. At present, Hive and Impala are able to query newly added columns, but other tools in the ecosystem such as Hadoop Pig may face challenges.

For more information about FARQUETFILE from below link:

https://parquet.apache.org/documentation/latest/

Parquet is supported by Cloudera and optimized for Cloudera Impala. Native Parquet support is rapidly being added for the rest of the Hadoop ecosystem.

Hive has its own ORCFILE Input format and ORCFILE output format in its default package:

```
parquet.hive.DeprecatedParquetInputFormat
parquet.hive.DeprecatedParquetOutputFormat
```

In Hive we can create a RCFILE format as follows:

2.7.1 Creating Parquet Table

```
CREATE TABLE Table_Parquet (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS PARQUET;
```

Here we are creating a table with name Table_Parquet and the schema of the table is as specified above. The data inside the input file is delimited by tab space. At the end the file format is specified as PARQUET format. we can check the schema of created table using:

```
describe Table_Parquet;
```

We cannot load data into PARQUET directly. First we need to load data into another table and then we need to overwrite it into our newly created PARQUET.

```
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=com.hadoop.compression.lzo.LzoCodec;

INSERT OVERWRITE TABLE Table_Parquet
SELECT * FROM Table_Text;
```

We have already created table by name Table_Text which is of TEXTFILE format and then we need to write the contents of the Table_Text table into Table_Parquet table.

Parquet files support with Hive. But it is very important that Parquet column names are lowercase. If Parquet file contains mixed case column names, Hive will not be able to read the column and will return queries on the column with null values and not log any errors. Unlike Hive, Impala handles mixed case column names.

2.8 Custom File format

We may have different kinds of Hadoop files, each file may have its own format. The file may have multiple new lines and tabs within a field that are escaped by a backslash (\\n and \\t).

For ex: JSON files…

if we have an input file, where one record spans across multiple lines and convert those tabs to spaces when Hive is going to try to do a split on the tabs and want to read it as it is then we should first write a custom input format to read in 1 record.

Hive uses to process records and map them to column data types in Hive tables is called SerDe, which is short for SerializerDeserializer.

SerDe will deserialize a row of data from the bytes in the file to objects used internally by Hive to operate on that row of data. When performing an INSERT or CTAS, the table's SerDe will serialize Hive's internal representation of a row of data into the bytes that are written to the output file.

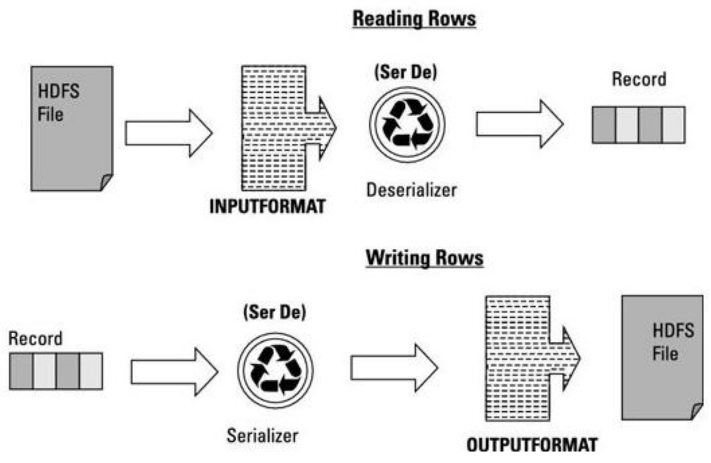

Fig 5: Hive Reading and Writing the Records

The fig: 5, showing how SerDes are leveraged and it will help us to understand how Hive keeps file formats separate from record formats. SerDe is a short name for "Serializer and Deserializer.Hive uses SerDe (and FileFormat) to read and write table rows.

Fig 5: clearly inferring the difference between the FileFormat and RowFormat. We need to have custom fileformat (input/output) when we want to read a record in a different way than usual (where records are separated by normal delimiter), then we need to have customize SerDe. When we want to interpret the read records in a custom way.

Let's take an example of commonly used format JSON:

Scenario 1: Let's say we have an input json file where one line contains one json record. So, now we just needs Custom Serde to interpret the read record in a way we want. No need of custom in/out format as 1 line will be 1 record.

Scenario 2: Now if we have an input file where our one json record spans across multiple lines and we want to read it as it is then we should first write a custom input format to read in 1 json record and then this read json record will go to Custom SerDe.

2.8.1 Custom Serde to interpret the Row:

We can get the anatomy of the input processing using SerDe from this location:

https://cwiki.apache.org/confluence/display/Hive/SerDe

In most of the cases, we want to write a Deserializer instead of a SerDe, because we just want to read the custom data format instead of writing to it.

For SerDe implementation, the first thing that will be required is the row deserializer/serializer (CustomRowSerDe). This java class will be in charge of mapping our row structure into Hive's row structure.

The CustomRowSerDe should be able to mirror the row class. This CustomRowSerDe will maintain all the table columns datatype's properties into Hive's ObjectInspector interface. For each of our types it'll find and return the equivalent type in the Hive API.

For ex: column datatype into Hive's ObjectInspector

- id int -> JavaIntObjectInspector
- string description -> JavaStringObjectInspector
- byte[] payload -> JavaBinaryObjectInspector
- class ExampleCustomRow -> StructObjectInspector

Once we have a way of mapping our rows into hadoop rows, Then reading and writing of the data from HDFS file. This is done via de Input and Output format APIs. The SerDe class will be used in Mapper and Reducer classes to serialize and deserialize the data.

The customRowSerDe class should implements the SerDe interface.

```
public class CustomRowSerDe implements SerDe {
```

The SerDe Interface having the interface methods, those to be implemented in the implementation classes. Those methods are used for initialize, serialize and desterilize the row data.

- **initialize ()** method is called only once and gathers some commonly-used pieces of information from the table properties, such as the column names and types. Using the type info of the row, we can instantiate an ObjectInspector for the row (ObjectInspectors are Hive objects that are used to describe and examine complex type hierarchies.)

```
@Override
 public void initialize(Configuration conf, Properties tbl)
     throws SerDeException {
   // Get a list of the table's column names.
   String colNamesStr = tbl.getProperty(Constants.LIST_COLUMNS);
   colNames = Arrays.asList(colNamesStr.split(","));

   // Get a list of TypeInfos for the columns. This list lines up with
   // the list of column names.
   String colTypesStr = tbl.getProperty(Constants.LIST_COLUMN_TYPES);
   List&lt;TypeInfo&gt; colTypes =
       TypeInfoUtils.getTypeInfosFromTypeString(colTypesStr);

   rowTypeInfo =
       (StructTypeInfo) TypeInfoFactory.getStructTypeInfo(colNames, colTypes);
   rowOI =
       TypeInfoUtils.getStandardJavaObjectInspectorFromTypeInfo(rowTypeInfo);
```

```
  }
```

- **serialize ()** method takes a Java object representing a row of data, and converts that object into a serialized representation of the row. The serialized class is determined by the return type ofgetSerializedClass().

```
 @Override
  public Writable serialize(Object obj, ObjectInspector oi)
      throws SerDeException {
    // Take the object and transform it into a serialized representation
    return new Text();
  }
```

- **deserialize()** method is opposite of serialize(). The deserialize() method takes a CustomFileRow string, and converts it into a Java object that Hive can process.

```
 **
   * This method does the work of deserializing a record into Java objects
   * that Hive can work with via the ObjectInspector interface.
   */
  @Override
  public Object deserialize(Writable blob) throws SerDeException {
    row.clear();
    // Do work to turn the fields in the blob into a set of row fields
    return row;
  }
  // row = new ArrayList&lt;
```

Some of the SerDes provided with Hive as well as third-party SerDe's, those we can get it from internet are useful. Examples are :

- **LazySimpleSerDe:** The default SerDe that's used with the TEXTFILE format.
- **ColumnarSerDe:** Used with the RCFILE format.
- **RegexSerDe:** The regular expression SerDe, used for Hive to enable the parsing of text files, RegexSerDe can form a powerful approach for building structured data in Hive tables from unstructured blogs, semi-structured log files, e-mails, tweets, and other data from social media. Regular expressions allow us to extract meaningful information (an e-mail address, for example) with HiveQL from an unstructured or semi-structured text document incompatible with traditional SQL and RDBMSs.
- **HBaseSerDe:** Included with Hive to enables it to integrate with HBase. we can store Hive tables in HBase by leveraging this SerDe.
- **JSONSerDe:** A third-party SerDe for reading and writing JSON data records with Hive. we can quickly find (via Google and GitHub) two JSON SerDes by searching online for the phrase json serde for hive.
- **AvroSerDe:** Included with Hive so that we can read and write Avro data in Hive tables.

2.8.2 Using the SerDe

Tables can be configured to process data using a SerDe by specifying the SerDe to use at table creation time, or through the use of an ALTER TABLE statement. For example:

```
CREATE TABLE Table_CustomSerDe (
column1 STRING,
column2 STRING,
column3 INT,
column4 INT
) ROW FORMAT SERDE
  'org.hive.CustomSerde'
  STORED AS TEXTFILE
```

2.8.3 Custom SerDe and CustomFileFormat

The custom SerSe is developed same as explained previous chapter. The Value class or the custom record class wil be maintained as common class between the CustomRecordReader class and CustomSerDe class for identifying the record in serialize and deserialize time.

If we want to use the custom file format for regular hadoop jobs, we have to implement InputFormat interface.

InputFormat describes the input-specification for a Map-Reduce job. The FileInputFormat class is the abstract class for implementing the InputFormat interface. Now, any in/out file FileFormat class for Hadoop should extends the FileInputFormat class.

FileInputFormat is the abstract class for all file-based InputFormats. This provides a generic implementation of getSplits(JobConf, int).

Some common subclasses of FileInputFormat are FixedLengthInputFormat, KeyValueText InputFormat, MultiFileInputFormat, NLineInputFormat, SequenceFileInput Format, TextInputFormat …etc classes. The customInputFileFormat can extends any of these classes based on the Custom Input file type.

For ex: The CustomDataFileInputFormat is extending the TextInputFormat for reading the Text file.

```
class CustomDataFileInputFormat extends TextInputFormat {

        @Override
        public RecordReader<LongWritable, Text> getRecordReader(InputSplit inputSplit,
JobConf job, Reporter reporter) throws IOException {
                return new CustomDataFileRecordReader((FileSplit)inputSplit, job);
        }
}
```

The FileInputFormat will give the logical data splits and RecordReader interface implementation class object.

The Record reader class can read the inputsplits data. For that we need to create the custom record reader.

```
  public class CustomDataFileRecordReader implements RecordReader<LongWritable, Text>
{
```

The CustomRecordReader will parse the File row and return the out to the CustomSarDe.We can create the jar file with all these java classes.

we will use this jar to create the Hive table and load the data in that table.

```
  ADD JAR /tmp/hive-serdes-1.0-SNAPSHOT.jar

  CREATE TABLE Table_CustomSerDe (
  column1 STRING,
  column2 STRING,
  column3 INT,
  column4 INT
  ) ROW FORMAT SERDE
    'org.hive.CustomSerde'
    STORED AS INPUTFORMAT
    'org.apache.hadoop.mapred.TextInputFormat'
    OUTPUTFORMAT
    'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
```

3 Hive data FileFormat Best Practices:

We can use the above discussed file formats depending on data.
For example,

- If data is delimited by some parameters then we can use TEXTFILE format.
- CSV files are excellent if we are going to extract data from Hadoop to bulk load into a database.
- If data is in small files whose size is less than the block size then we can use SEQUENCEFILE format.
- If we want to perform analytics on data and we want to store data efficiently for that then we can use RCFILE format.
- If we want to store data in an optimized way which lessens storage and increases performance then we can use ORCFILE format.
- The tables with very large number of columns and tend to use specific columns frequently, RC file format would be a good choice.
- If we are storing intermediate data between MapReduce jobs, then Sequence files are preferred.
- If query performance against the data is most important, ORC (HortonWorks/Hive) or Parquet (Cloudera/Impala) are optimal
- Avro is great if the schema is going to change over time, but query performance will be slower than ORC or Parquet.

4 Conclusion

Hive provides a solution to perform business intelligence of huge data on top of mature Hadoop map-reduce platform. The SQL-like HiveQL cuts off the learning curve compared with low-level map-reduce programs.

File Input format play very important role in Hive performance. The best suitable format for the business use-case will lead to high performance and optimized space HDFS space usage.

The SerDe interface is extremely powerful for operations on data with a complex schema. By utilizing SerDes, any dataset can be queried through Hive.

5 References:

https://cwiki.apache.org/confluence/display/Hive/LanguageManual

http://blog.cloudera.com/

http://www.congiu.com/a-json-readwrite-serde-for-hive/

http://stackoverflow.com/questions

https://github.com/rcongiu/Hive-JSON-Serde

http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html

http://hive.apache.org/javadocs/r0.10.0/api/org/apache/hadoop/hive/ql/io/RCFile.html

https://parquet.apache.org/documentation/latest/