**Introduction**

Choosing the right file format is a critical decision in big data applications as it directly affects storage efficiency, query performance, and compatibility with analytical tools. Among the most widely used file formats are **Avro**, **Parquet**, and **ORC**. Each of these formats has its strengths, weaknesses, and ideal use cases.

In this tutorial, we will explore these formats in detail, presenting their features, advantages, disadvantages, and practical usage scenarios. We will also include example code for handling each format in Apache Hive using HiveQL.

To perform the Hive QL code covered in this tutorial you can create a Zeppelin note and use **hive** interpreter (**%hive**).

%hive

show databases;

**Data Preparation**

In this section we will prepare the initial data to be used in the next sections of this tutorial. This will be our initial data to import and convert into other file formats.

**Input dataset**

The input dataset has four columns:

* user id
* movie id
* rating
* unix timestamp.  (the actual format is not compatible with Hive)

Let’s start by creating the destination directory on HDFS and upload the input dataset to it.

%sh

# Create hdfs directory and upload the input file to it

hdfs dfs -mkdir -p /tutorials/files/hive/ratings/csv

hdfs dfs -put /home/training/Data/ml-latest/ratings1M.csv /tutorials/files/hive/ratings/csv

**Initial Hive Table**

Create the initial Hive table to read the ratings data file.

%sh

# Create hdfs directory and upload the input file to it

hdfs dfs -mkdir -p /tutorials/files/hive/ratings/csv

hdfs dfs -put /home/training/Data/ml-latest/ratings1M.csv /tutorials/files/hive/ratings/csv

%hive

-- Create the tutorials database

create database if not exists tutorials;

%hive

-- Create the ratings table

create external table if not exists tutorials.ratings\_csv (

userid int,

movieid int,

rating float,

timest bigint

)

Row Format Delimited

fields terminated by ','

Stored as Textfile

Location '/tutorials/files/hive/ratings/csv/';

%hive

-- check the data

select \* from tutorials.ratings\_csv limit 10;

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-csv-table.png)

**Compute Table Statistics**

Now the Hive table is ready let’s compute its statistics.

%hive

-- Compute the table Statistics

Analyze table tutorials.ratings\_csv compute statistics;

Now we can show some of the table properties.

%hive

SHOW TBLPROPERTIES tutorials.ratings\_csv('totalSize');

%hive

SHOW TBLPROPERTIES tutorials.ratings\_csv('numRows');

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-csv-table-properties-1.png)

**Avro Format**

**Presentation**

Apache Avro is a row-based file format designed for data serialization. It provides efficient storage and is often used in streaming data pipelines and data serialization for remote procedure calls (RPCs).

**Advantages**

* Compact and fast serialization format.
* Schema evolution support, allowing changes to the schema without breaking compatibility.
* Well-suited for streaming and row-level operations.
* Language-neutral format, making it interoperable across platforms.

**Disadvantages**

* Not optimized for complex analytical queries.
* Less efficient for columnar data analysis compared to Parquet or ORC.

**Usage Scenario**

Avro is ideal for streaming use cases where data serialization and deserialization are required, such as Kafka producers and consumers or real-time data pipelines.

**Avro Schema**

Avro purely relies on a schema. When Avro data is read, the schema that is used for writing it is always present. This permits each datum to be written with no per-value overheads, making the serialization small and faster. This method also facilitates using it with dynamic scripting languages, since the data is together with its schema, it is fully self-describing.

When Avro data is stored in a file, its schema is also stored with it, so that files may be processed later by any application. If the application reading the data expects a different schema this can be easily resolved, since both schema’s are present.

When Avro is used in RPC, the client and server exchange schema’s in the connection handshake. (This can be optimized so that, for most calls, no schema’s are transmitted.) Since both client and server have the other’s full schema, correspondence between same named fields, missing fields, extra fields, etc. can all be easily resolved.

Avro schema are defined with JSON . This facilitates implementation in languages that already have JSON libraries. Using Avro, we can convert unstructured and semi-structured data into properly structured data using its schema.

You can say that Avro format is actually a combination of a JSON data structure and a schema for validation purposes. So before we create an Avro file which usually has an extension .avro, we will be creating its schema having usually the .avsc extension.

The following figure shows the structure of an AVRO container.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/08/apache-parquet-apache-avro-adatform-tum-png-favpng-jnW7qFbxwZTdMZNxdhUeNpAdw.png)

{"namespace": "tutorial",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "age", "type": ["int", "null"]},

{"name": "gender", "type": ["string", "null"]}

]

}

The JSON structure above represents an AVRO schema structure example.

* We will be mentioning a **namespace** first. It is nothing but a string. Usually, it follows the same format which is used by the Java packaging naming convention that is the reverse of your domain name but not necessary.
* After that we mention the type of your schema, here it is of **record**type. There are other types too like **enum**,**arrays**etc.
* After that, we mention the name of the schema which is **User**here.
* The next is **field**item which could be one or more. It has mandatory fields like **name**and **type**and optional fields like **doc**and **alias**. The **doc**field is used to document your field while **alias**is used to give the field a name other than the one mentioned in **name**.

Usually, the schema is saved in a file with the extension **.avsc**.

**HiveQL Code Example**

In this example we will create a new table and populate it with the data from the initial table. Hive will infer the table schema as well the Avro schema so we do not need to provide it explicitly.

**Note:** *By default output compression is disabled in Hive. We will enabled prior running the query.*

%hive

-- Enable Output Compression and set the compression codec to Snappy.

set hive.exec.compress.output=true;

set avro.output.codec=snappy;

-- Create a new table in Avro format and populate the table with data from the initial table

create table if not exists tutorials.ratings\_avro

stored as avro

Location '/tutorials/files/hive/ratings/avro/'

as select \* from tutorials.ratings\_csv;

We compute the table statistics and show ‘totalsize‘ and ‘numRows’ properties.

%hive

-- Compute the table Statistics

Analyze table tutorials.ratings\_avro compute statistics;

%hive

SHOW TBLPROPERTIES tutorials.ratings\_avro('totalSize');

%hive

SHOW TBLPROPERTIES tutorials.ratings\_avro('numRows');

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-avro-table-properties.png)

**Avro Tools**

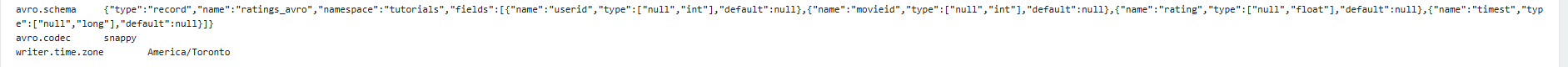
Apache Avro provides a command-line utility called **avro-tools** for various operations on Avro files, such as inspecting, validating, and manipulating Avro data. One of its key features is the ability to extract the schema from an Avro container file using the getmeta command.

To use **getmeta**and extract the schema, run the following command:

%sh

## Show the Avro container meta data

hadoop jar avro-tools getmeta /tutorials/files/hive/ratings/avro/000000\_0

[](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-avro-schema.png)

The **getmeta**command display the avro schema as well the compression codec.

**Parquet Format**

**What is Parquet?**

[Apache Parquet](https://parquet.apache.org/) is a free and open-source data format for data storage and retrieval. It’s also a product of the Hadoop project, but it differs from Avro in very important ways.

Avro’s language support differs from Parquet. The core Parquet project only releases [Java jars,](https://parquet.apache.org/blog/2022/05/26/1.12.3/) but, C, C++, and python support is available via the [Arrow](https://arrow.apache.org/docs/cpp/parquet.html) project. There is also a [Python library](https://pypi.org/project/parquet/) for reading Parquet files, and you can process them with [Pandas](https://pandas.pydata.org/).

The biggest difference between Avro and Parquet is that Parquet is a**column-oriented** data format, meaning Parquet stores data by column instead of row. This makes Parquet a good choice when you only need to access specific fields. It also makes reading Parquet files very fast in search situations.

This difference also means that Parquet is not a good choice for network messages, since streaming data and column formats don’t work well together. While it’s possible to use a column-oriented format for streaming data, it often eliminates many of the performance benefits.

Parquet’s schema support is like Avro’s. It supports primitive types like boolean, int, long, and string, and offers robust support for complex and user-defined data types. But schema evolution is expensive with column data, since changes require reprocessing entire data sets, instead of record-by-record in row-oriented data.

The following figure shows the structure of an Parquet container.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/08/parquet.png)

**What Is Parquet Used For?**

Parquet is ideal for batch processing and analytical queries where only a subset of columns is required, such as data warehouses or Spark SQL queries. Applications that need rapid access to specific fields in a large dataset use **Parquet**. The format works remarkably well for read-intensive applications and low latency data storage and retrieval.

When you want to aggregate a few columns in a large data set, Parquet is your best bet. Writing files in Parquet is more compute-intensive than Avro, but querying is faster.

**Parquet Schema**

Apache Parquet schema format is actually a JSON structure. As per the Avro format, schema is need when you create and read a parquet file. So before we create a Parquet file which usually has an extension .parquet, we will be creating its schema (usually having the same extension as Avro schema). The following file is a sample Parquet schema:

message emp\_schema {

optional int32 EmpID;

optional binary LName (UTF8);

optional binary FName (UTF8);

optional double salary;

optional int32 age;

}

**Compression**

Parquet’s columnar storage allows for better compression, as the data in a column is usually more homogeneous. It supports various compression codecs like Snappy, Gzip, and LZO.

Avro, while supporting similar codecs, doesn’t achieve the same level of compression due to its row-based nature.

**HiveQL Code Example**

In this example we will create a new table and populate it with the data from the initial table. Hive will infer the table schema as well the Parquet schema so we do not need to provide it explicitly.

**Note:** *By default output compression is disabled in Hive. We will enabled prior running the query.*

%hive

-- Enable Output Compression and set the compression codec to Snappy.

set hive.exec.compress.output=true;

-- Create a new table in Parquet format and populate the table with data from the initial table

create table if not exists tutorials.ratings\_parquet

stored as parquet

Location '/tutorials/files/hive/ratings/parquet/'

TBLPROPERTIES ('parquet.compression'='SNAPPY')

as select \* from tutorials.ratings\_csv;

We compute the table statistics and show ‘totalsize‘ and ‘numRows’ properties.

%hive

-- Compute the table Statistics

Analyze table tutorials.ratings\_parquet compute statistics;

%hive

SHOW TBLPROPERTIES tutorials.ratings\_avro('totalSize');

%hive

SHOW TBLPROPERTIES tutorials.ratings\_avro('numRows');

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-parquet-table-properties.png)

**Parquet Tools**

Apache Parquet provides a command-line utility called **parquet-tools** for inspecting and manipulating Parquet files. One of its useful features is the ability to extract metadata from a Parquet file using the **meta** command. This metadata includes schema information, column types, and file properties.

To use the **meta**command and extract the schema, run the following command:

%sh

## Show the Parquet container meta data

hadoop jar parquet-tools meta /tutorials/files/hive/ratings/parquet/000000\_0

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-parquet-schema-1.png)

The **getmeta**command display the parquet schema as well the compression codec.

**ORC (Optimized Row Columnar)**

**What is ORC?**

ORC, short for Optimized Row Columnar, is a file format optimized for analytical workloads, primarily within the context of the Apache Hive data warehousing platform. Its features and advantages make it an excellent choice for scenarios where analytical querying and efficient storage are critical. Here are the key characteristics and strengths of ORC:

1. **Columnar Storage:** ORC stores data column-wise rather than row-wise, a fundamental departure from traditional row-based storage formats. This columnar storage design has several advantages for analytical workloads, as it allows for highly efficient selective column retrieval and aggregation.
2. **Compression Efficiency**: ORC offers robust compression capabilities specifically tailored for columnar storage. It supports compression codecs like Zlib, Snappy, and LZO, optimizing storage requirements while maintaining query performance.
3. **Predicate Pushdown**: ORC supports predicate pushdown, a query optimization technique that filters data at the storage level before retrieving it. This feature significantly enhances query performance by reducing I/O overhead. Predicate pushdown in ORC is particularly beneficial for analytical queries on large datasets.
4. **Optimized for Hive**: While ORC can be used independently, it is most commonly associated with the Apache Hive data warehousing platform. It has been fine-tuned to work seamlessly with Hive, providing additional query performance benefits when using Hive for analytics.

**ORC Container Structure**

The ORC file is organized into a container format with multiple layers to optimize both storage and retrieval. Each ORC file consists of the following components:

* **Stripe**: The main data storage unit. Each stripe contains rows of data broken into column chunks. Stripes also include an index for faster lookups and a footer containing metadata.
* **File Footer**: Contains metadata about all the stripes in the file, including the column data types, compression types, and offsets.
* **Postscript**: Holds information about the overall file structure, such as compression algorithm and file length.

The following figure shows the structure of an ORC container.

[A diagram of a data flow

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/0_J3f3fsUGgg1j7LXM.png)

**ORC Schema**

ORC uses a rich schema to store structured data. It supports:

* **Primitive Types**: Such as int, float, double, boolean, string, and binary.
* **Complex Types**: Such as struct, list, map, and union.

Each column is stored separately, allowing optimized access to specific columns without reading the entire dataset.

**Differences between ORC and Parquet**

* Parquet is more widely adopted and supported by the community than ORC. It has more libraries and tools to read and write Parquet files, such as Apache Arrow, Apache Parquet C++, Apache Parquet Python, etc.
* ORC has better compression rates than Parquet, especially for numeric data. It also has lightweight indexes stored within the file, which can improve read performance by skipping irrelevant rows or stripes.
* ORC works very well with ACID transactions in Hive, which provide features like update, delete, and merge. Parquet does not support ACID transactions natively, but it can work with Delta Lake or Apache Hudi to enable them.
* Parquet is more optimized for analytical workloads and complex queries, while ORC is more suitable for write-heavy workloads and transactional processing.

**Compression**

ORC’s columnar storage allows for better compression, as the data in a column is usually more homogeneous. It supports various compression codecs like Snappy and ZLib which is the default compression codec in Hive.

**Disadvantages**

* Less widely supported than Parquet.
* Write performance is slower compared to Avro.

**Usage Scenario**

ORC is best suited for Hadoop-based data lakes and Hive-based analytical workloads where query performance and compression are critical.

**HiveQL Code Example**

We will create a new table and populate it with the data from the initial table. Hive will infer the table schema as well the ORC schema so we do not need to provide it explicitly.

**Note:** *By default output compression is disabled in Hive. We will enabled prior running the query. By enabling compression, Hive will use the deflate compression codec by default.*

%hive

-- Enable Output Compression, by default Hive will use the deflate compression codec.

set hive.exec.compress.output=true;

-- Create a new table in ORC format and populate the table with data from the initial table

create table if not exists tutorials.ratings\_orc

stored as orc

Location '/tutorials/files/hive/ratings/orc/'

as select \* from tutorials.ratings\_csv;

We compute the table statistics and show ‘totalsize‘ and ‘numRows’ properties.

%hive

-- Compute the table Statistics

​Analyze table tutorials.ratings\_orc compute statistics;

%hive

SHOW TBLPROPERTIES tutorials.ratings\_avro('totalSize');

%hive

SHOW TBLPROPERTIES tutorials.ratings\_orc('numRows');

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-orc-table-properties.png)

**ORC Tools**

Apache ORC provides a command-line utility called **orc-tools** for inspecting and working with ORC files. One of its essential commands is **meta**, which allows users to extract metadata, including schema details, compression information, and stripe details.

To use the **meta**command and extract the schema, run the following command:

%sh

hadoop jar orc-tools meta /tutorials/files/hive/ratings/orc/000000\_0

[A screenshot of a computer program

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/01/fileformat-ratings-orc-schema.png)

The **meta**command extract the schema and show the compression codec as well.

**Comparison Table**

| **Feature** | **Avro** | **Parquet** | **ORC** |
| --- | --- | --- | --- |
| **File Type** | Row-based | Columnar | Columnar |
| **Best For** | Streaming, Serialization | Analytical Queries | Hadoop and Hive Queries |
| **Compression** | Moderate | High | Very High |
| **Schema Evolution** | Excellent | Limited | Limited |
| **Write Efficiency** | High | Moderate | Moderate |
| **Read Efficiency** | Moderate | High | Very High |

**Summary**

Choosing the right file format depends on the specific requirements of your application. Avro is ideal for streaming and serialization tasks, Parquet excels in analytical and batch processing workloads, and ORC is optimized for Hadoop-based ecosystems. By understanding the strengths and weaknesses of each format, you can make informed decisions to optimize your data storage and processing pipelines.

Open Zeppelin Note

[Choosing the Right Big Data File Format](http://localhost:19995/#/notebook/2KKCF6V9X)