HIVE
Apache Hive

Plain Text Data Storage vs Byte Data Storage

Storing data as plain text and storing data as bytes are two different ways of representing data, each with their advantages and disadvantages.

Plain Text Data Storage:

- In plain text data storage, data is stored as readable characters.
- For example, consider the number 12345. In plain text, it's stored as the characters '1', '2', '3', '4', and '5'.
- Each character typically uses 1 byte of memory (in ASCII), or 2 bytes (in UTF-16), so this number would use 5 to 10 bytes of memory.
- The advantage of plain text storage is that it's human-readable and easy to interpret without any conversion.
- The disadvantage is that it's not space-efficient. Larger numbers or data types other than integers (like floating-point numbers) will use more space.

Byte (Binary) Data Storage:

- In byte (or binary) data storage, data is stored as binary values, not as readable characters. Each byte consists of 8 bits, and each bit can be either 0 or 1.
- Using our previous example, the number 12345 can be represented in binary format as 11000000111001, which is 14 bits or 2 bytes (with 6 unused bits). In a more memory-optimized format, it could use only the necessary 14 bits.
- The advantage of binary storage is that it's very space-efficient. Each type of data (integer, float, etc.) has a standard size, regardless of its value.
- The disadvantage is that binary data is not human-readable. You need to know the type of data and how it's encoded to convert it back to a readable format.

SerDe in Hive

- SerDe stands for Serializer/Deserializer.
- It's a crucial component of Hive used for IO operations, specifically for reading and writing data.
- It helps Hive to read data in custom formats and translate it into a format Hive can process (deserialization) and vice versa (serialization).

Role of SerDe in Hive:

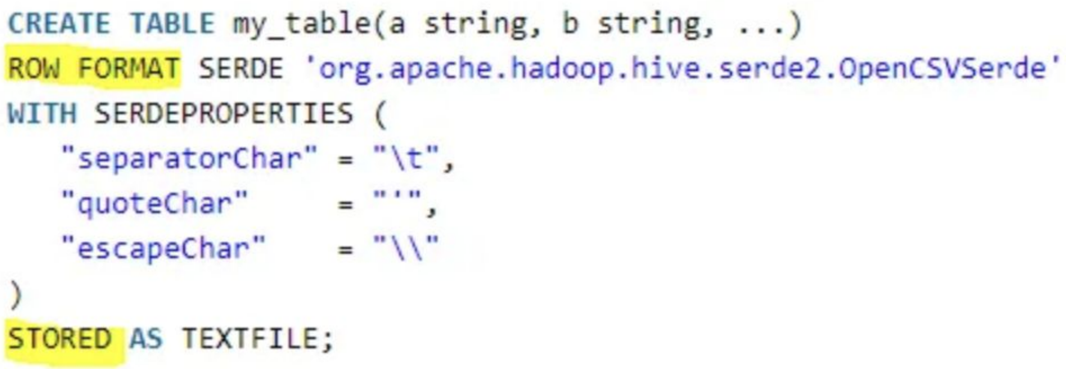
- When reading data from a table (input to Hive), deserialization is performed by the SerDe.
- When writing data to a table (output from Hive), serialization is performed by the SerDe.

Serialization - Process of converting an object in memory into bytes that can be stored in a file or transmitted over a network.

Deserialization - Process of converting the bytes back into an object in memory.

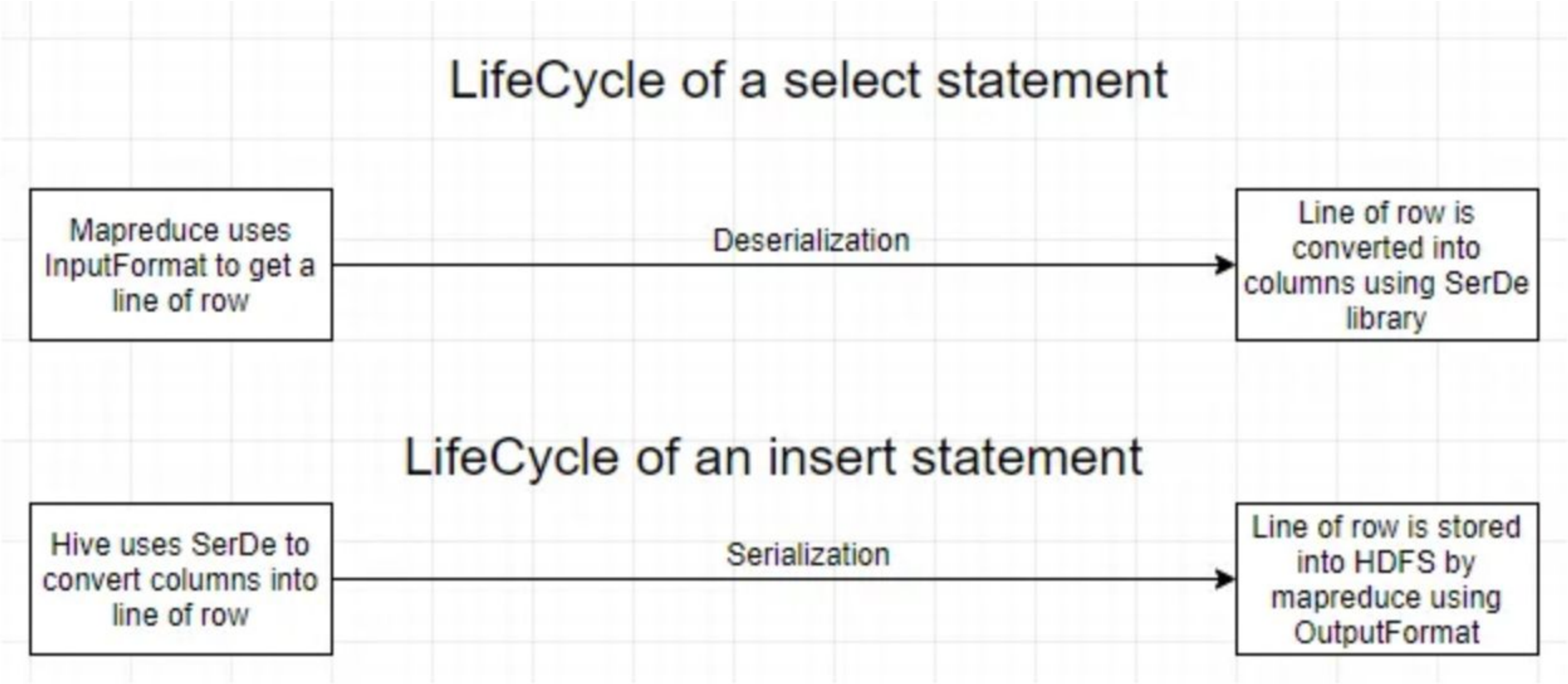
“A select statement creates deserialized data(columns) that is understood by Hive. An insert statement creates serialized data(files) that can be stored into an external storage like HDFS”.

Hive Row format and Map-reduce Input/Output format



```
CREATE TABLE my_table(a string, b string, ...)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
   "separatorChar" = "\t",
   "quoteChar"     = "'",
   "escapeChar"    = "\\"
)
STORED AS TEXTFILE;
```

In any table definition, there are two important sections. The **“Row Format”** describes the **libraries** used to convert a given row into columns. The **“Stored as”** describes the **InputFormat** and **OutputFormat** libraries used by map-reduce to read and write to HDFS files.



LifeCycle of a select statement

Mapreduce uses InputFormat to get a line of row

Deserialization

Line of row is converted into columns using SerDe library

LifeCycle of an insert statement

Hive uses SerDe to convert columns into line of row

Serialization

Line of row is stored into HDFS by mapreduce using OutputFormat

File formats in Hive with SerDe Library

TextFile:

- This is the default file format.
- Each line in the text file is a record. Hive uses the ***LazySimpleSerDe*** for serialization and deserialization.
- It's easy to use but doesn't provide good compression or the ability to skip over not-needed columns during read.

JSON:

- Hive supports reading and writing JSON format data.
- Note that JSON files typically do not have a splittable structure, which can affect performance as only one mapper can read the data when not splittable.
- Default SerDe is ***org.apache.hive.hcatalog.data.JsonSerDe***

CSV (Comma Separated Values):

- CSV is a simple, human-readable file format used to store tabular data. Columns are separated by commas, and rows by new lines.
- CSV files can be read and written in Hive using the ***org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe*** or ***org.apache.hadoop.hive.serde2.OpenCSVSerde*** for advanced CSV parsing.
- CSV lacks a splittable structure, which may affect performance due to limited parallel processing.

ORCFile (Optimized Row Columnar File):

- Introduced by Hortonworks, ORC is a highly efficient way to store Hive data.
- It provides efficient compression and encoding schemes with enhanced performance to handle complex data types.
- Default SerDe is **org.apache.hadoop.hive.ql.io.orc.OrcSerde**

Parquet:

- Columnar storage format, available to any project in the Hadoop ecosystem.
- It's designed to bring efficient columnar storage of data compared to row-based like CSV or TSV files.
- Default SerDe is ***org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe***

Avro:

- It's a row-oriented format that is highly splittable.
- It also supports schema evolution - you can have Avro data files where each file has a different schema but all are part of the same table.
- Default SerDe is ***org.apache.hadoop.hive.serde2.avro.AvroSerDe***

ORC File Format

ORC (Optimized Row Columnar):

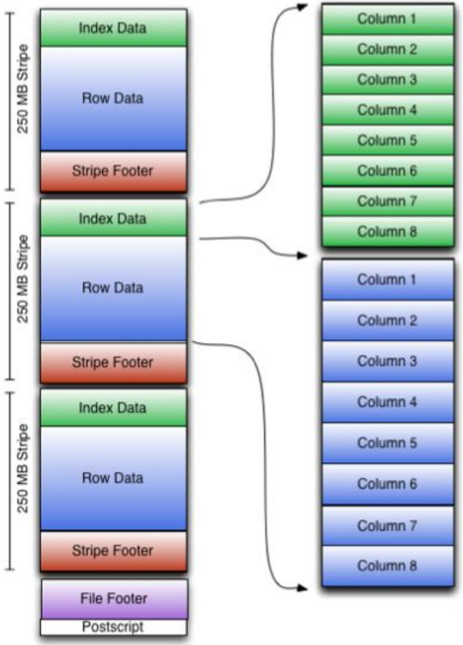
- Built for the Hive query engine, ORC is a **columnar storage** format that allows Hive to read, write, and process data faster.
- It allows for **efficient compression**, which saves storage space, and adds improvements in the speed of data retrieval, making it suitable for performing high-speed queries.
- It stores **collections of rows**, not individual rows.
- Each file consists of **row index**, **column statistics**, and **stripes** (a row of data consisting of several rows) that contain the column data.
- Supports complex types: Structs, Lists, Maps, and Unions.
- Also supports advanced features like **bloom filters** and indexing. A bloom filter is a data structure that can identify whether an element might be present in a set, or is definitely not present. In the context of ORC, bloom filters can help skip unnecessary reads when performing a lookup on a particular column value.

ORC File Structure

An ORC file contains groups of row data called **stripes**, along with auxiliary information in a **file footer**. At the end of the file a **postscript** holds compression parameters and the size of the compressed footer.

The default stripe size is **250** MB. Large stripe sizes enable large, efficient reads from HDFS.

The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum.



Strip Structure

As shown in the diagram, each stripe in an ORC file holds **index data**, **row data**, and a **stripe footer**.

ORC organizes data for each column into streams, the **stripe footer** records the location and size of these streams within the stripe. This allows the reader to locate and access specific parts of the stripe efficiently.

Row data is used in table scans.

Index data includes min and max values for each column and the row positions within each column. Row index entries provide offsets that enable seeking to the right compression block and byte within a decompressed block. Note that ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

Parquet File Format

Parquet:

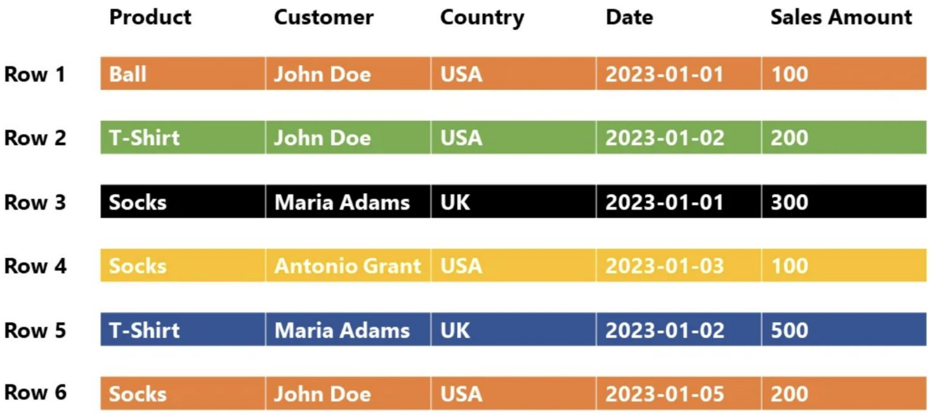
- It is columnar in nature and designed to bring efficient columnar storage of data.
- Provides efficient data compression and encoding schemes with enhanced performance to handle complex data in comparison to row-based files like CSV.
- Schema evolution is handled in the file metadata allowing compatible schema evolution.
- It supports all data types, including nested ones, and integrates well with flat data, semi-structured data, and nested data sources.

Parquet is considered a de-facto standard for storing data nowadays:

- **Data compression** – by applying various encoding and compression algorithms, Parquet file provides reduced memory consumption
- **Columnar storage** – this is of paramount importance in analytic workloads, where fast data read operation is the key requirement. But, more on that later in the article…
- **Language agnostic** – as already mentioned previously, developers may use different programming languages to manipulate the data in the Parquet file
- **Open-source format** – meaning, you are not locked with a specific vendor
- Support for complex data types

Parquet File Format

In traditional, row-based storage, the data is stored as a sequence of rows. Something like this:



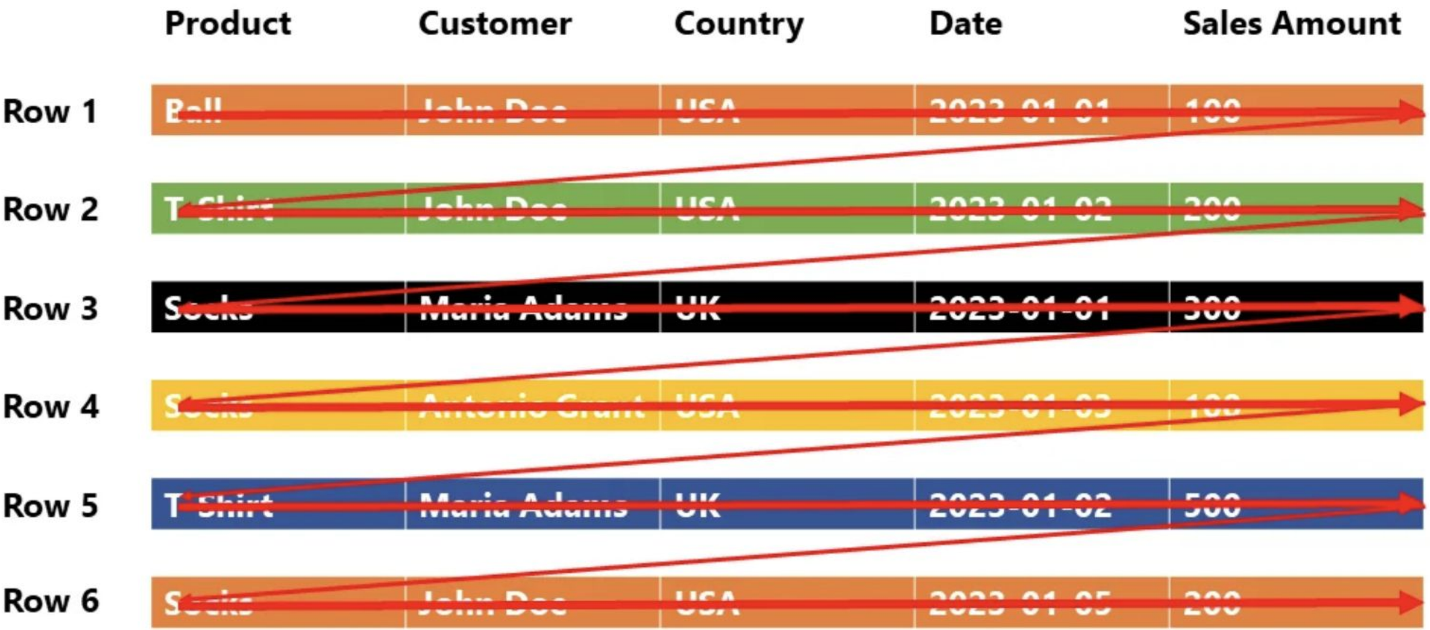
	Product	Customer	Country	Date	Sales Amount
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

Now, when we are talking about OLAP scenarios, some of the common questions that your users may ask are:

- How many balls did we sell?
- How many users from the USA bought T-Shirt?
- What is the total amount spent by customer Maria Adams?
- How many sales did we have on January 2nd?

To be able to answer any of these questions, the engine must scan each and every row from the beginning to the very end! So, to answer the question: how many users from the USA bought T-Shirt, the engine has to do something like this:

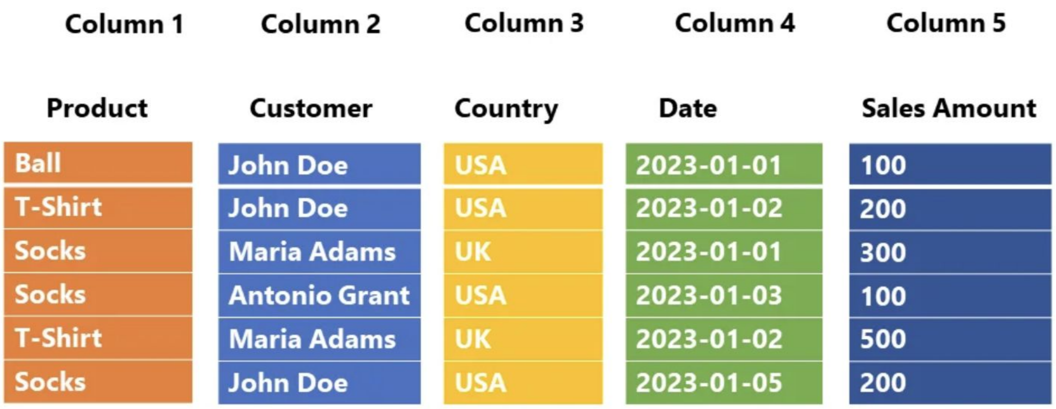
Parquet File Format



	Product	Customer	Country	Date	Sales Amount
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

Parquet File Format

Let’s now examine how the column store works. As you may assume, the approach is 180 degrees different:



Column 1	Column 2	Column 3	Column 4	Column 5
Product	Customer	Country	Date	Sales Amount
Ball	John Doe	USA	2023-01-01	100
T-Shirt	John Doe	USA	2023-01-02	200
Socks	Maria Adams	UK	2023-01-01	300
Socks	Antonio Grant	USA	2023-01-03	100
T-Shirt	Maria Adams	UK	2023-01-02	500
Socks	John Doe	USA	2023-01-05	200

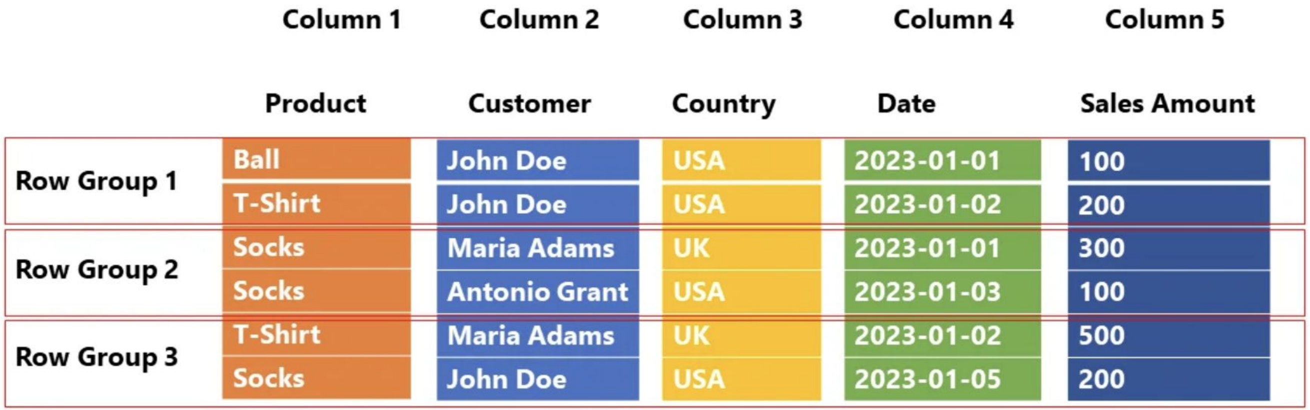
In this case, each column is a separate entity – meaning, each column is physically separated from other columns! Going back to our previous business question: the engine can now scan only those columns that are needed by the query (Product and country), while skipping scanning the unnecessary columns. And, in most cases, this should improve the performance of the analytical queries.

Ok, that’s nice, but the column store existed before Parquet and it still exists outside of Parquet as well. So, what is so special about the Parquet format?

Parquet File Format

Parquet is a columnar format that stores the data in row groups!
Wait, what?! Wasn’t it enough complicated even before this? Don’t worry, it’s much easier than it sounds:)

Let’s go back to our previous example and depict how Parquet will store this same chunk of data:



	Column 1	Column 2	Column 3	Column 4	Column 5
	Product	Customer	Country	Date	Sales Amount
Row Group 1	Ball	John Doe	USA	2023-01-01	100
	T-Shirt	John Doe	USA	2023-01-02	200
Row Group 2	Socks	Maria Adams	UK	2023-01-01	300
	Socks	Antonio Grant	USA	2023-01-03	100
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02	500
	Socks	John Doe	USA	2023-01-05	200

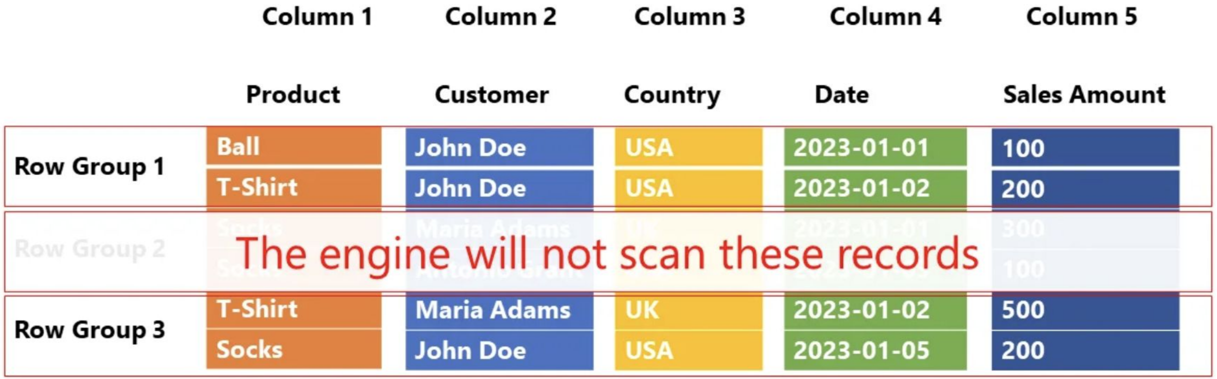
Parquet File Format

Let’s stop for a moment and explain the illustration above, as this is exactly the structure of the Parquet file (some additional things were intentionally omitted, but we will come soon to explain that as well). Columns are still stored as separate units, but Parquet introduces additional structures, called Row group.

Why is this additional structure super important?

You’ll need to wait for an answer for a bit:). In OLAP scenarios, we are mainly concerned with two concepts: projection and predicate(s). Projection refers to a SELECT statement in SQL language – which columns are needed by the query. Back to our previous example, we need only the Product and Country columns, so the engine can skip scanning the remaining ones.

Predicate(s) refer to the WHERE clause in SQL language – which rows satisfy criteria defined in the query. In our case, we are interested in T-Shirts only, so the engine can completely skip scanning Row group 2, where all the values in the Product column equal socks!

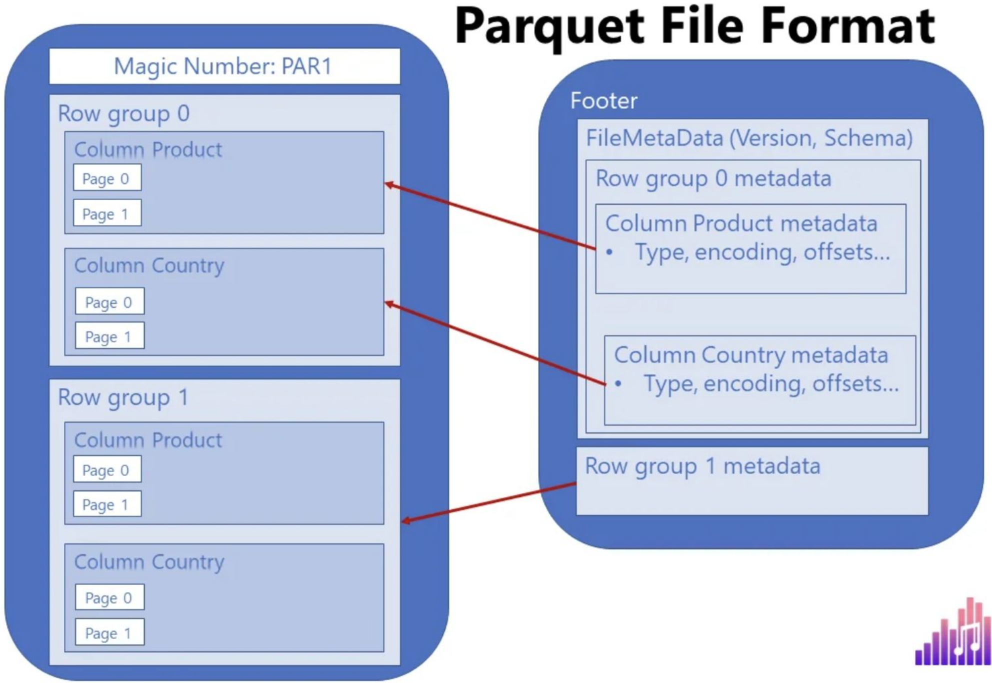


	Column 1	Column 2	Column 3	Column 4	Column 5
	Product	Customer	Country	Date	Sales Amount
Row Group 1	Ball	John Doe	USA	2023-01-01	100
	T-Shirt	John Doe	USA	2023-01-02	200
Row Group 2	Socks	Maria Adams	UK	2023-01-01	300
	Socks	Antonio Grant	[illegible]	[illegible]	100
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02	500
	Socks	John Doe	USA	2023-01-05	200

The engine will not scan these records

Parquet File Format

This means, every Parquet file contains “data about data” – information such as minimum and maximum values in the specific column within the certain row group. Furthermore, every Parquet file contains a footer, which keeps the information about the format version, schema information, column metadata, and so on.

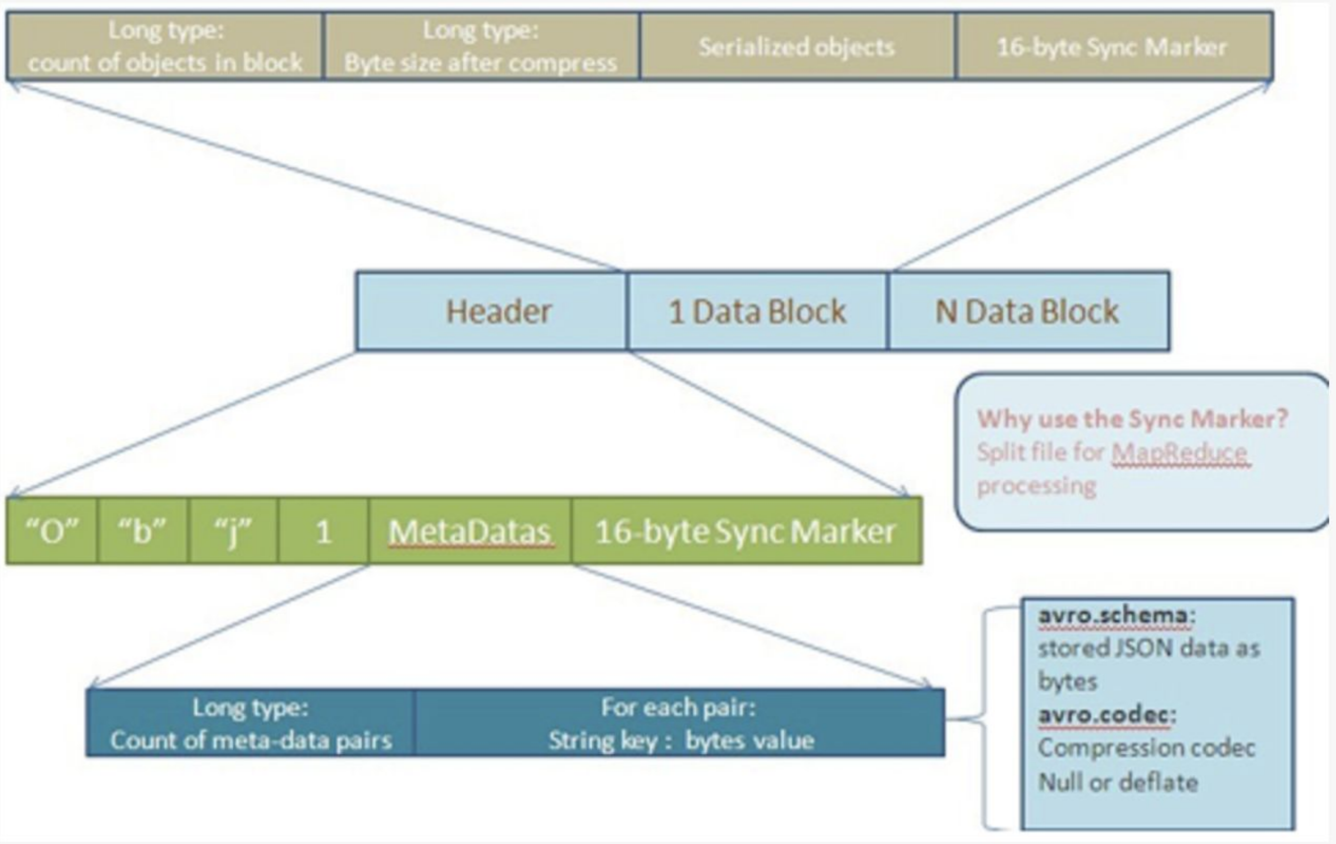


Avro File Format

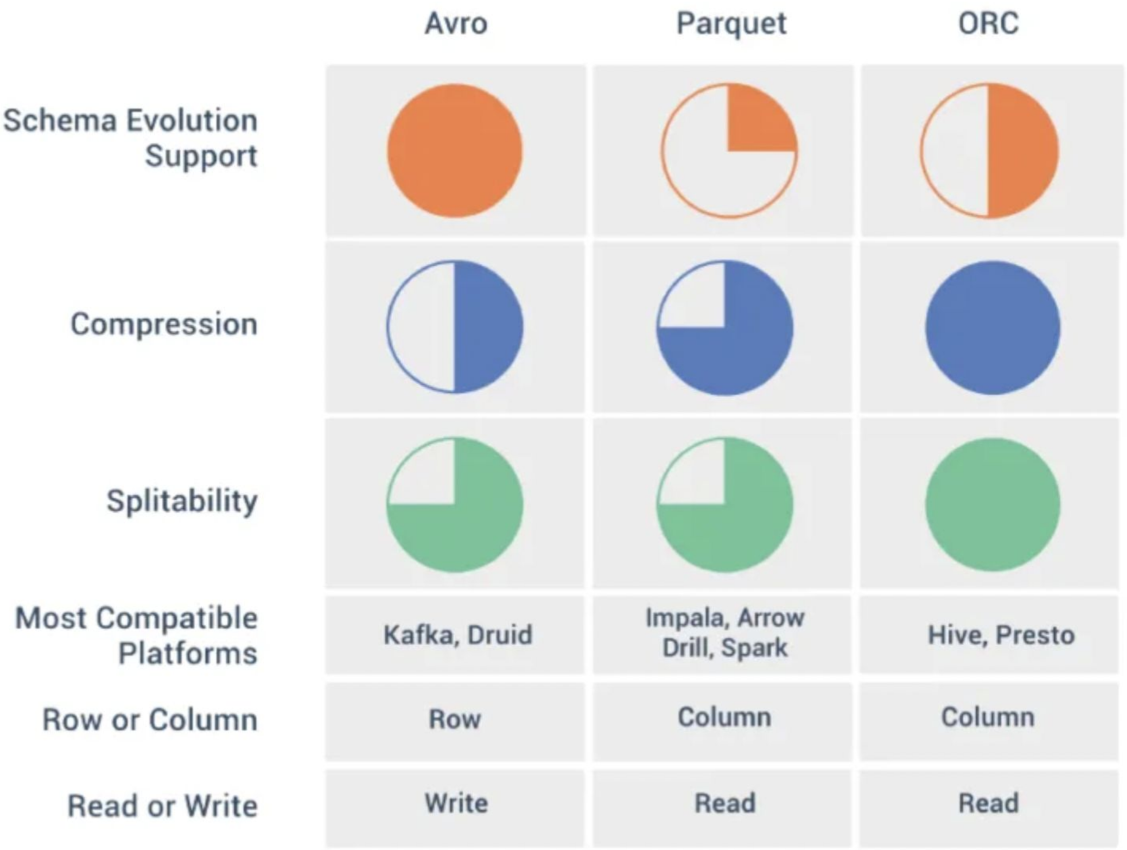
Avro:

- **Schema-Based:** Avro uses a schema to define the structure of the data. The schema is written in JSON and is included in the serialized data, allowing data to be self-describing and ensuring that the reader can understand the data structure without external information.
- **Compact and Fast**: Avro data is serialized in a compact binary format, which makes it highly efficient in terms of both storage and transmission.
- **Compression**: Avro supports various compression codes such as Snappy, Deflate, Bzip2, Xz
- **Schema Evolution**: Avro supports schema evolution, allowing the schema to change over time without breaking compatibility with old data. This is particularly useful in big data environments where data structures might evolve.
- **Rich Data Structures**: Avro supports complex data types, including nested records, arrays, maps, and unions, allowing for flexible and powerful data modelling.
- **Interoperability**: Avro is designed to work seamlessly with other big data tools and frameworks, especially within the Hadoop ecosystem, such as Apache Hive, Apache Pig, and Apache Spark.
- **Language Agnostic**: Avro has libraries for many programming languages, including Java, C, C++, Python, and more, enabling cross-language data exchange.

Avro File Format



ORC vs Parquet vs AVRO



	Avro	Parquet	ORC
Schema Evolution Support	●	◔	◑
Compression	◑	◕	●
Splitability	◕	◕	●
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

How to decide which file format to choose?

- **Columnar vs Row-based:** Columnar storage like Parquet and ORC is efficient for read-heavy workloads and is especially effective for queries that only access a small subset of total columns, as it allows skipping over non-relevant data quickly. Row-based storage like Avro is typically better for write-heavy workloads and for queries that access many or all columns of a table, as all of the data in a row is located next to each other.

- **Schema Evolution:** If your data schema may change over time, Avro is a solid choice because of its support for schema evolution. Avro stores the schema and the data together, allowing you to add or remove fields over time. Parquet and ORC also support schema evolution, but with some limitations compared to Avro.

- **Compression:** Parquet and ORC, being columnar file formats, allow for better compression and improved query performance as data of the same type is stored together. Avro also supports compression but being a row-based format, it might not be as efficient as Parquet or ORC.

- **Splittability:** When compressed, splittable file formats can still be divided into smaller parts and processed in parallel. Parquet, ORC, and Avro are all splittable, even when compressed.

- **Complex Types and Nested Data:** If your data includes complex nested structures, then Parquet is a good choice because it provides efficient encoding and compression of nested data.