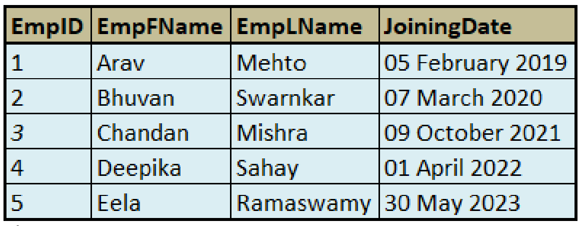
Big Datafile Format theory

25 May 2024 06:15

Sample data

Let’s take the sample data which is gonna use as a base for the study here. Consider I have employee table with 5 records for 4 fields. The same is shown in the figure 1.



EmpID	EmpFName	EmpLName	JoiningDate
1	Arav	Mehto	05 February 2019
2	Bhuvan	Swarnkar	07 March 2020
3	Chandan	Mishra	09 October 2021
4	Deepika	Sahay	01 April 2022
5	Eela	Ramaswamy	30 May 2023

Figure 1: Sample Data

Storage mechanisms

Primarily there are only two ways how the data stores in the files. These are:

1. Row oriented storage

   This storage type supported data formats are the one’s that stores data row-wise to the computer disk.

2. Column oriented storage

   This storage type supported data formats are the one’s that stores data column-wise to the computer disk.

Row Oriented Storage

The best example of explaing the row-oriented data is the CSV format. The data gets persisted in the CSV format as a list of rows. As in above example, consider rows as the block and that get’s stored as a single unit to the storage. Something shown in the below diagram.

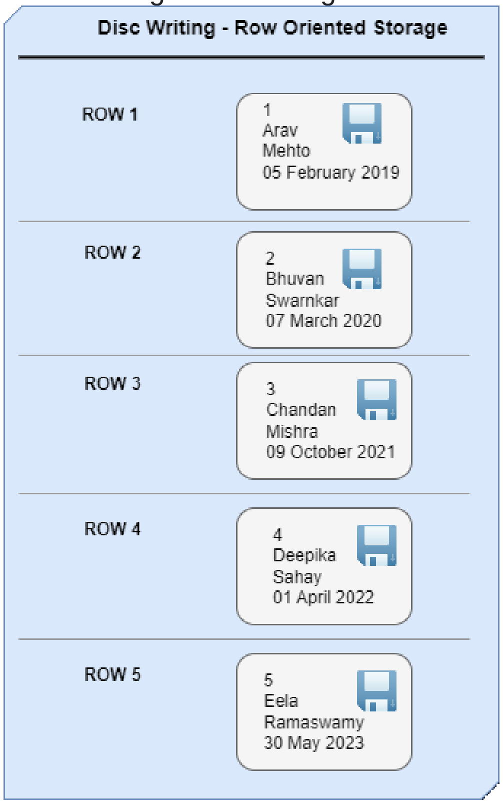

Disc Writing - Row Oriented Storage
ROW 1
1 Arav Mehto 05 February 2019
ROW 2
2 Bhuvan Swarnkar 07 March 2020
ROW 3
3 Chandan Mishra 09 October 2021
ROW 4
4 Deepika Sahay 01 April 2022
ROW 5
5 Eela Ramaswamy 30 May 2023

Figure 2: CSV Storage Format

Column Oriented Storage

The best example of explaing the column-oriented data is the Parquet format. The data gets persisted in the CSV format as a list of columns. As in sample data, consider one column as the block and that get’s stored as a single unit to the storage. Something shown in the diagram below.

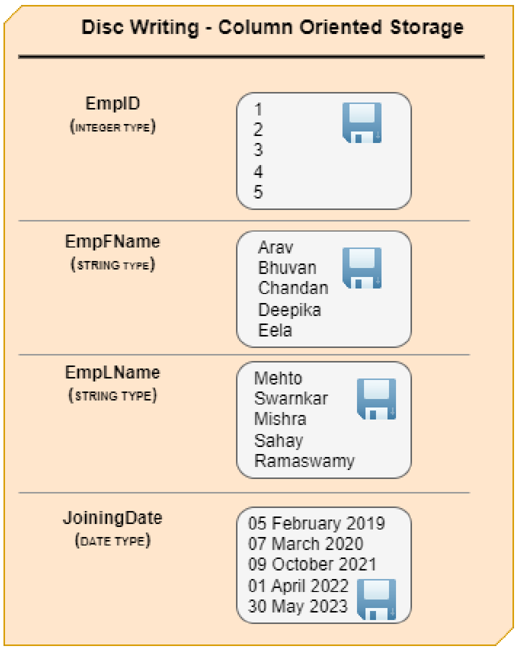


Figure 3: Parquet Storage Format

Comparison (Row vs Col)

If the dataset is small or medium then it doesn't matter what is the row oriented or column oriented. But the way data is growing and now we're talking in Petabytes and Zetabytes of data. Thus, it is critical to decide what data format should you be considered for your project.

Feature	Row-Based	Column-Based
Performance	Slower to query	Faster to query
Size (Memory consumption)	Larger	Smaller
Compatibility (with variety of applications)	Widely Compatible	Less compatible
Suitable usecase	General purpose (OLTP)	Analytics & ML applications (OLAP)

Figure 4: Feature comparison

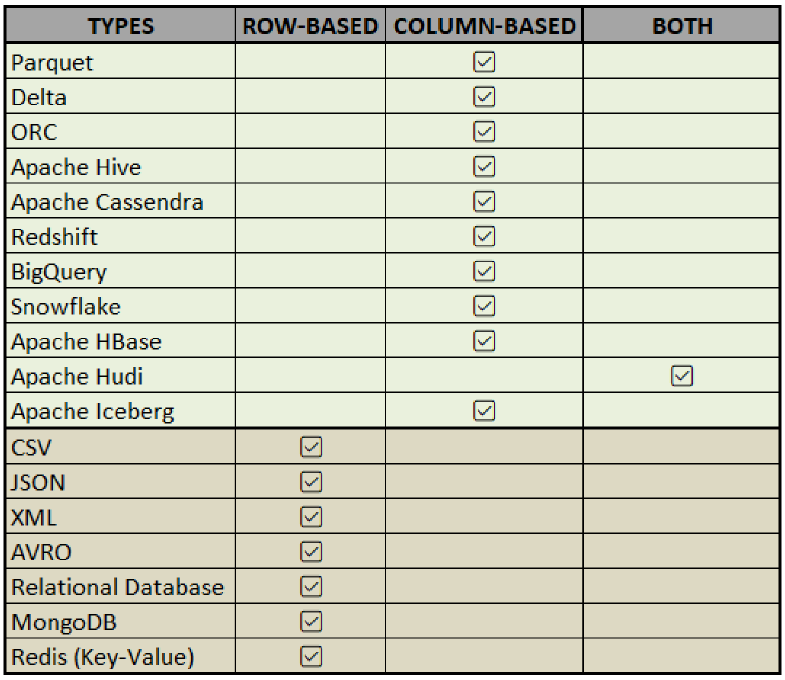
Why to choose Column as a first choice in today's era of big data?

Row-based files (like CSV) are usually larger in size as compare to Column-based files (like Parquet). This is because the columnar storage is the column compressed storage where compression depends on their data types (string, datetime, integer, etc). Which is not the case with row-based storage as it stores entire row which many times store on the basis of string types (in case of CSV, for instance).

Secondly, the important upperhand that column-based files like Parquet have over row-based is the speed (performance). The columnar data is really fast to scan and extract the aggregates. This is because the query runs only to one column and not on all.

Check popular ones

There are many data formats, cloud storages, and backend databases (relational and NoSQL). Lets look here in the table that explains about the category it belongs to based on the way data gets stored in the disc (row or column).

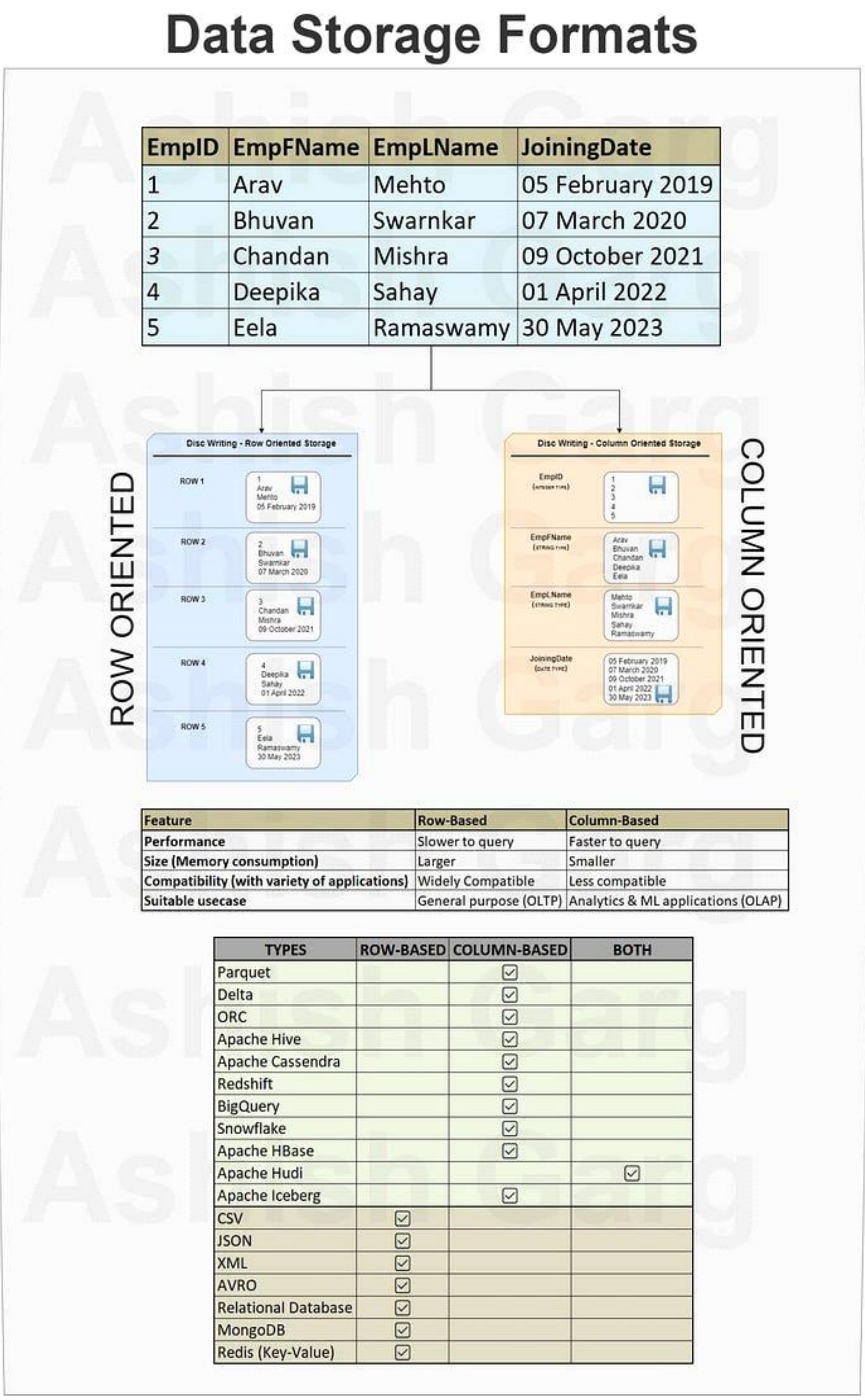


TYPES	ROW-BASED	COLUMN-BASED	BOTH
Parquet		☑	
Delta		☑	
ORC		☑	
Apache Hive		☑	
Apache Cassendra		☑	
Redshift		☑	
BigQuery		☑	
Snowflake		☑	
Apache HBase		☑	
Apache Hudi			☑
Apache Iceberg		☑	
CSV	☑		
JSON	☑		
XML	☑		
AVRO	☑		
Relational Database	☑		
MongoDB	☑		
Redis (Key-Value)	☑		

Figure 5: Popular data stores mapping to row-based/column-based

Single Page View

The diagram here shows the single page view for the entire concepts I covered above. Hope this helps you to understand well.



Data Storage Formats

EmpID	EmpFName	EmpLName	JoiningDate
1	Arav	Mehto	05 February 2019
2	Bhuvan	Swarnkar	07 March 2020
3	Chandan	Mishra	09 October 2021
4	Deepika	Sahay	01 April 2022
5	Eela	Ramaswamy	30 May 2023

ROW ORIENTED
Disc Writing - Row Oriented Storage
ROW 1
1 Arav Mehto 05 February 2019
ROW 2
2 Bhuvan Swarnkar 07 March 2020
ROW 3
3 Chandan Mishra 09 October 2021
ROW 4
4 Deepika Sahay 01 April 2022
ROW 5
5 Eela Ramaswamy 30 May 2023
COLUMN ORIENTED
Disc Writing - Column Oriented Storage
EmpID (INTEGER TYPE)
1 2 3 4 5
EmpFName (STRING TYPE)
Arav Bhuvan Chandan Deepika Eela
EmpLName (STRING TYPE)
Mehto Swarnkar Mishra Sahay Ramaswamy
JoiningDate (DATE TYPE)
05 February 2019 07 March 2020 09 October 2021 01 April 2022 30 May 2023

Feature	Row-Based	Column-Based
Performance	Slower to query	Faster to query
Size (Memory consumption)	Larger	Smaller
Compatibility (with variety of applications)	Widely Compatible	Less compatible
Suitable usecase	General purpose (OLTP)	Analytics & ML applications (OLAP)

TYPES	ROW-BASED	COLUMN-BASED	BOTH
Parquet		✓	
Delta		✓	
ORC		✓	
Apache Hive		✓	
Apache Cassendra		✓	
Redshift		✓	
BigQuery		✓	
Snowflake		✓	
Apache HBase		✓	
Apache Hudi			✓
Apache Iceberg		✓	
CSV	✓		
JSON	✓		
XML	✓		
AVRO	✓		
Relational Database	✓		
MongoDB	✓		
Redis (Key-Value)	✓		

we will look at the most common file formats used to store the data in big data world. Whether we are building a data lake, a warehouse or a mart, we often deal with variety of data which are of different format going out or coming from different data sources. So, we will look at what are those file formats, example data for each, their pros and cons. To start with, below are the most common types of the file formats:

1. CSV (Comma-Separated Values)
2. JSON (JavaScript Object Notation)
3. Avro
4. Parquet

5. ORC (Optimized Row Columnar)

CSV:

CSV stands for “Comma-Separated Values”, which is a plain-text file format used for storing tabular data. In a CSV file, each row represents a record, and each column within the row represents a field of that record. The values in each field are separated by commas.

Here is an example of a simple CSV file:

```
Name, Age, Gender
John, 25, Male
Alice, 30, Female
Bob, 28, Male
```

In this example, the first row represents the column headers or field names. The second, third, and fourth rows represent records, with each column separated by commas.

Advantages:

- Easy to read and write using many different programming languages and tools.
- Widely supported by many different applications and systems.
- Compact and lightweight, with minimal overhead for storage and transmission.

Disadvantages:

- Limited support for complex data types, such as nested structures or arrays.
- Limited support for metadata or schema information.
- Inefficient for querying or processing large data sets.

JSON:

JSON (JavaScript Object Notation) is a lightweight file format used for storing and exchanging data. It is often used in web applications and APIs as a way to transmit data between a server and a client. It is also a preferred option to transmit data through messaging services such as Kafka. In a JSON file, data is represented as key-value pairs, and the file itself is structured as a collection of objects and arrays.

Here is an example of the same data as the CSV file example, formatted in JSON:

```
[
 {
 "Name": "John",
 "Age": 25,
 "Gender": "Male"
 },
 {
 "Name": "Alice",
 "Age": 30,
 "Gender": "Female"
 },
 {
 "Name": "Bob",
 "Age": 28,
 "Gender": "Male"
 }
]
```

In this example, the data is represented as an array of three objects, where each object represents a record. Each object contains key-value pairs, where the keys are the field names and the values are the data values.

Advantages:

- Like CSV, this is also easy to read and write using many different programming languages and tools.
- Supports complex data types, such as nested structures or arrays.
- Supports metadata or schema information using JSON Schema or other formats.

Disadvantages:

- Can be verbose and inefficient for large data sets.
- Can be less efficient for querying or processing compared to columnar formats.
- Limited support for compression or data encoding.

Avro:

Avro is a binary data serialization format that is used for transmitting data between systems. It is used in big data applications because of its efficient data encoding and compression, and its support for schema evolution. Here is an example of the same data, formatted in Avro:

```
{
 "type": "array",
 "items": {
 "type": "record",
 "name": "Person",
 "fields": [
 {"name": "Name", "type": "string"},
 {"name": "Age", "type": "int"},
 {"name": "Gender", "type": "string"}
 ]
 }
}
{
 "Name": "John",
 "Age": 25,
 "Gender": "Male"
}
{
 "Name": "Alice",
 "Age": 30,
 "Gender": "Female"
}
{
 "Name": "Bob",
 "Age": 28,
 "Gender": "Male"
}
```

In this example, the data is represented as an Avro schema, followed by the actual data records. The schema specifies the data type and structure of the data, including the field names and types. The data records themselves are encoded in a binary format that is compact and efficient for transmission and storage.

Advantages:

- Efficient data encoding and compression, with compact binary data representation.
- Supports schema evolution, allowing for changes to the data structure over time.
- Supports metadata or schema information using Avro Schema.

Disadvantages:

- Can be less human-readable compared to text-based formats like CSV or JSON.
- Limited support for querying or processing compared to columnar formats.
- Requires more overhead for encoding and decoding compared to text-based formats.

Parquet:

Parquet is a columnar storage format for big data applications that is designed for efficient data storage, compression, and processing. It is an open-source file format that is widely used in the Hadoop ecosystem, and it is supported by many big data processing frameworks, including Apache Spark, Apache Hive, and Apache Impala.

Here is an example of the same data, formatted in Parquet:

```
Name | Age | Gender
------+-----+-------
John | 25 | Male
Alice | 30 | Female
Bob | 28 | Male
```

In this example, the data is stored in a columnar format, where each column represents a field of the data. This allows for efficient data compression and processing, as well as fast querying and analysis of specific fields. And parquet schema for this file is as below:

```
message Example {
 required binary Name;
 required int32 Age;
 required binary Gender;
}
```

Which contains three required fields: “Name” (a binary string), “Age” (a 32-bit integer), and “Gender” (a binary string). The Parquet format is flexible and supports many different data types, including strings, integers, floating-point numbers, timestamps, and more.

Advantages:

- Efficient columnar storage and compression, with fast querying and processing of specific fields.
- Supports complex data types, such as nested structures or arrays.
- Supports metadata or schema information using Parquet Schema.

Disadvantages:

- Can be less efficient for writing compared to row-based formats like CSV or ORC.
- Can be less human-readable compared to text-based formats like CSV or JSON.
- Requires more overhead for encoding and decoding compared to text-based formats.

ORC:

ORC (Optimised Row Columnar) is a columnar storage format for big data applications that is similar to Parquet. Like Parquet, ORC is designed for efficient data storage, compression, and processing, and it is widely used in the Hadoop ecosystem and supported by many big data processing frameworks, including Apache Hive and Apache Pig.
Here is an example of the same data as the CSV, JSON, and Parquet examples, formatted in ORC:

```
Name | Age | Gender
------+-----+-------
John | 25 | Male
Alice | 30 | Female
Bob | 28 | Male
```

In this example, the data is stored in a columnar format, similar to Parquet. However, ORC also supports the storage of row data, which can be more efficient for certain types of queries and processing. ORC files also support advanced features such as predicate pushdown and column statistics, which can improve query performance and reduce data processing overhead.
Here is an example ORC schema for the file in the previous example:

```
struct<Name:string,Age:int,Gender:string>
```

In this schema, the data is represented as a struct type that contains three fields: “Name” (a string), “Age” (an integer), and “Gender” (a string). The ORC format is flexible and supports many different data types, including arrays, maps, and complex nested structures.
ORC schemas are used to define the structure of the data in an ORC file, and they are often used to optimize queries and analysis by specifying which columns and fields are relevant to a given query.

Advantages:

- Efficient row and columnar storage and compression, with fast querying and processing of specific fields.
- Supports complex data types, such as nested structures or arrays.
- Supports advanced features such as predicate pushdown and column statistics.

Disadvantages:

- Can be less efficient for writing compared to row-based formats like CSV or Avro.
- Requires more overhead for encoding and decoding compared to text-based formats.

- Less widely used and supported compared to formats like CSV, JSON, or Parquet.

Pqrquest in details

Parquet file format in a nutshell!

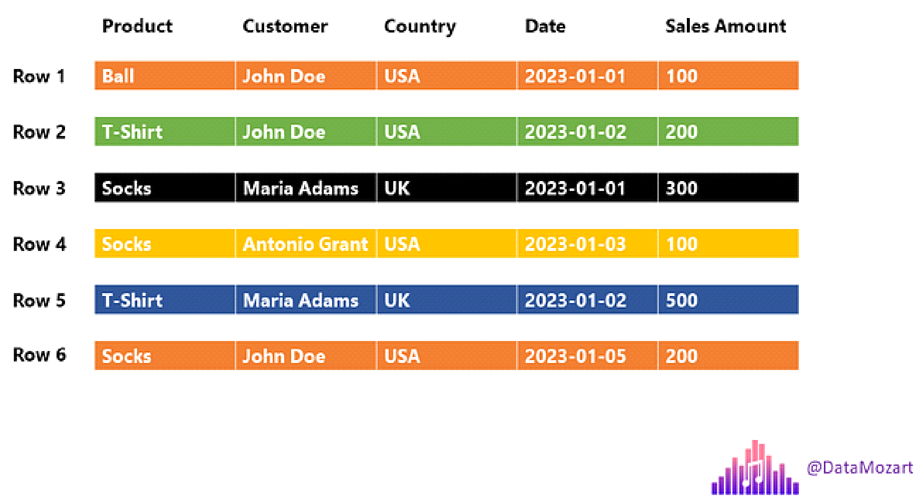
Before I show you ins and outs of the Parquet file format, there are (at least) five main reasons why Parquet is considered a de-facto standard for storing data nowadays:

- ***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***

Row-store vs Column-store

We've already mentioned that Parquet is a column-based storage format. However, to understand the benefits of using the Parquet file format, we first need to draw the line between the row-based and column-based ways of storing the data.

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

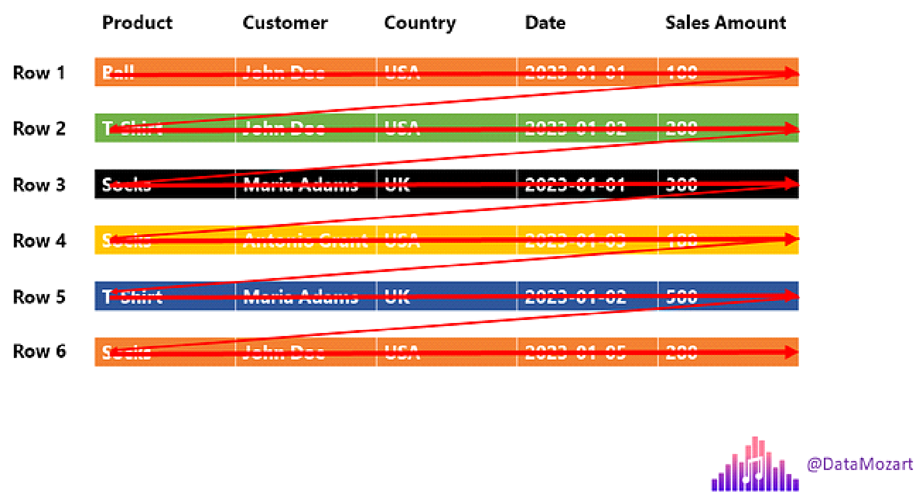
@DataMozart

Image by author

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:



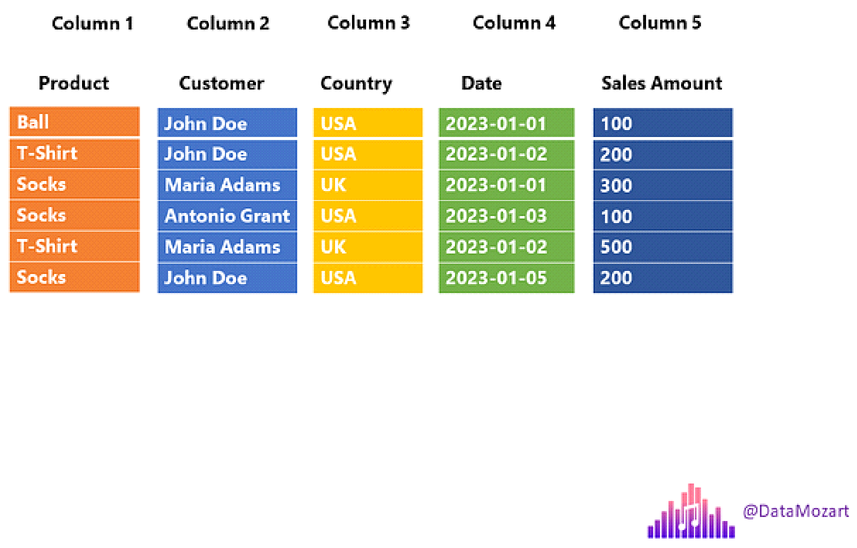
	Product	Customer	Country	Date	Sales Amount
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

Image by author

Essentially, we just need the information from two columns: Product (T-Shirts) and Country (USA), but the engine will scan all five columns. This is not the most efficient solution — I think we can agree on that…

Column store

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

@DataMozart

Image by author

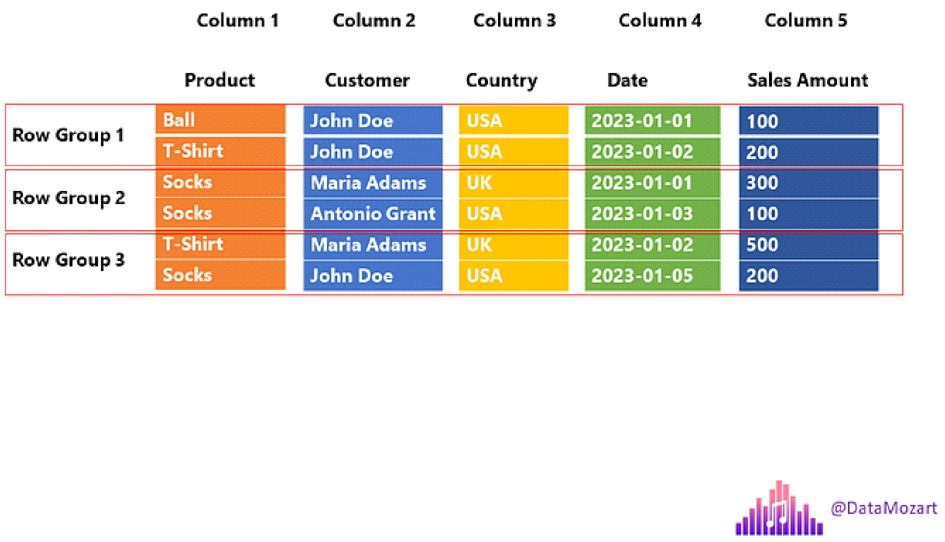
In this case, each column is a separate entity — meaning, each column is physically separated from other columns. Going back to our previous business question: the engine can now scan only those columns that are needed by the query (Product and country), while *skipping scanning* the unnecessary columns. 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 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

@DataMozart

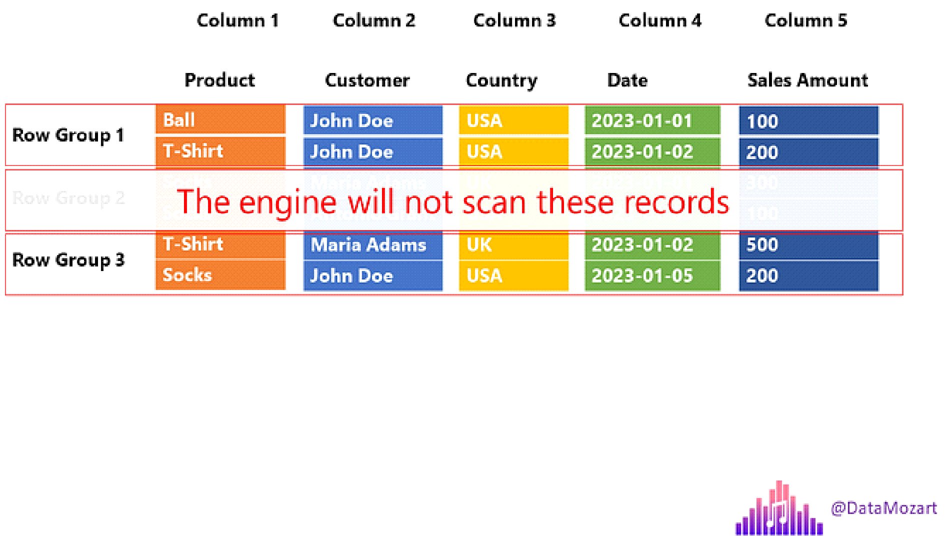
Image by author

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	The engine will not scan these records				300
					100
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02	500
	Socks	John Doe	USA	2023-01-05	200

@DataMozart

Image by author

Let's quickly stop here, as I want you to realize the difference between various types of storage in terms of the work that needs to be performed by the engine:

- Row store — the engine needs to scan all 5 columns and all 6 rows
- Column store — the engine needs to scan 2 columns and all 6 rows
- Column store with row groups — the engine needs to scan 2 columns and 4 rows

Obviously, this is an oversimplified example, with only 6 rows and 5 columns, where you will definitely not see any difference in performance between these three storage options. However, in real life, when you're dealing with much larger amounts of data, the difference becomes more evident.

Now, the fair question would be: how Parquet “knows” which row group to skip/scan?

Parquet file contains metadata

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. You can find more details about Parquet metadata types here.

Important: In order to optimize the performance and eliminate unnecessary data structures (row groups and columns), the engine first needs to “get familiar” with the data, so it first reads the metadata. It’s not a slow operation, but it still requires a certain amount of time. Therefore, if you’re querying the data from multiple small Parquet files, query performance can degrade, because the engine will have to read metadata from each file. So, you should be better off merging multiple smaller files into one bigger file (but still not too big:)…

I hear you, I hear you: Nikola, what is “small” and what is “big”? Unfortunately, there is no single “golden” number here, but for example, Microsoft Azure Synapse Analytics recommends that the individual Parquet file should be at least a few hundred MBs in size.

What else is in there?

Here is a simplified, high-level, illustration of the Parquet file format:

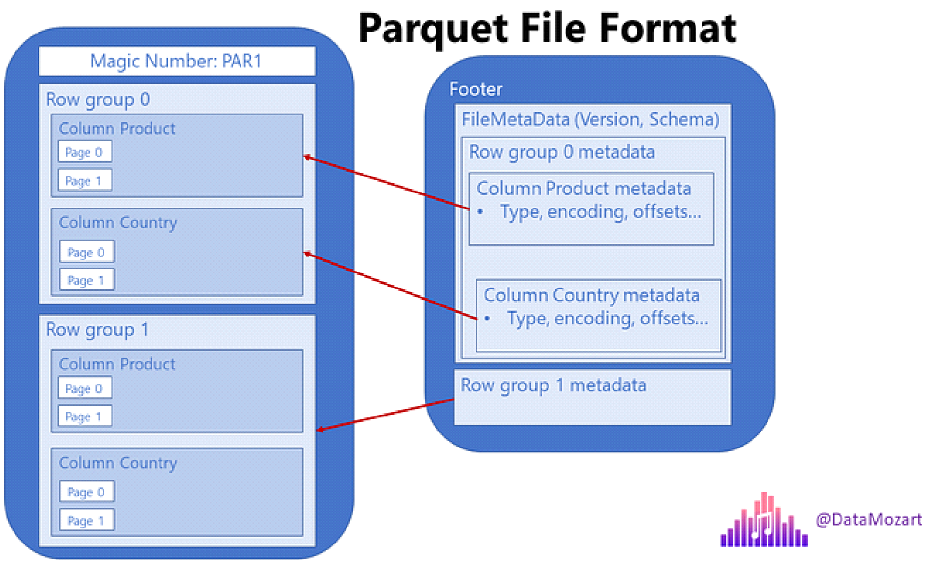


Image by author

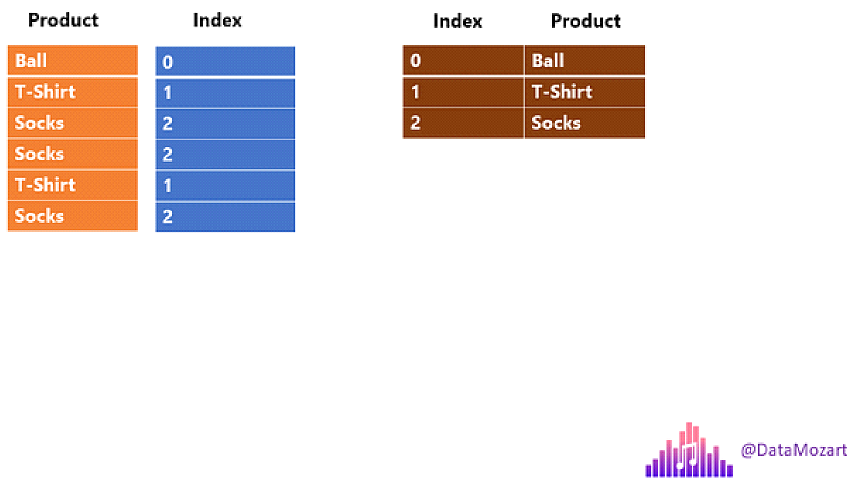
Can it be better than this? Yes, with data compression

Ok, we’ve explained how skipping the scan of the unnecessary data structures (row groups and columns) may benefit your queries and increase the overall performance. But, it’s not only about that — remember when I told you at the very beginning that one of the main advantages of the Parquet format is the reduced memory footprint of the file? This is achieved by applying various compression algorithms.

I’ve already written about various data compression types in Power BI (and the Tabular model in general) here, so maybe it’s a good idea to start by reading this article.

There are two main encoding types that enable Parquet to compress the data and achieve astonishing savings in space:

- *Dictionary encoding* — Parquet creates a dictionary of the distinct values in the column, and afterward replaces “real” values with index values from the dictionary. Going back to our example, this process looks something like this:



Product	Index
Ball	0
T-Shirt	1
Socks	2
Socks	2
T-Shirt	1
Socks	2

Index	Product
0	Ball
1	T-Shirt
2	Socks

@DataMozart

Image by author

You might think: why this overhead, when product names are quite short, right? Ok, but now imagine that you store the detailed description of the product, such as: “Long arm T-Shirt with application on the neck”. And, now imagine that you have this product sold million times…Yeah, instead of having million times repeating value “Long arm…bla bla”, the Parquet will store only the Index value (integer instead of text).

- *Run-Length-Encoding with Bit-Packing* — when your data contains many repeating values, Run-Length-Encoding (RLE) algorithm may bring additional memory savings.

Can it be better than THIS?! Yes, with the Delta Lake file format

Ok, what the heck is now a Delta Lake format?! This is the article about the Parquet, right?

So, to put it in plain English: Delta Lake is nothing else but the Parquet format “on steroids”. When I say “steroids”, the main one is the versioning of Parquet files. It also stores a transaction log, to enable keeping track of all changes applied to the Parquet file. This is also known as ACID-compliant transactions.

Since it supports not only ACID transactions, but also supports time travel (rollbacks, audit trails, etc.) and DML (Data Manipulation Language) statements, such as INSERT, UPDATE and DELETE, you won’t be wrong if you think of the Delta Lake as a “data warehouse on the data lake” (who said: Lakehouse:))