Parquet File Format Demystified

Published by **Arulraj Gopal** on September 13, 2025

Many people think Parquet is a columnar format. That is not entirely true. Parquet is actually a *hybrid* format—it blends the strengths of both row-based and columnar storage.

This has become the go-to file format for data engineering.

So, what makes it special? Why is it so widely adopted in modern data platforms? Let's break it down in this article.

Parquet format capabilities

1. Supports column pruning – reads only required columns.
2. Parallel reads – even single file can be read with multithreading.
3. Efficient encoding – repetitive values stored once.
4. Metadata driven – scans only relevant blocks and skips unnecessary data
5. Self-Describing format – No need for external schema definitions
6. Supports nested data structures – natively support array & maps
7. Supports multiple compression codecs – Snappy, Gzip, ZSTD, LZO, etc.

With these capabilities, Parquet is highly optimized for analytical workloads.

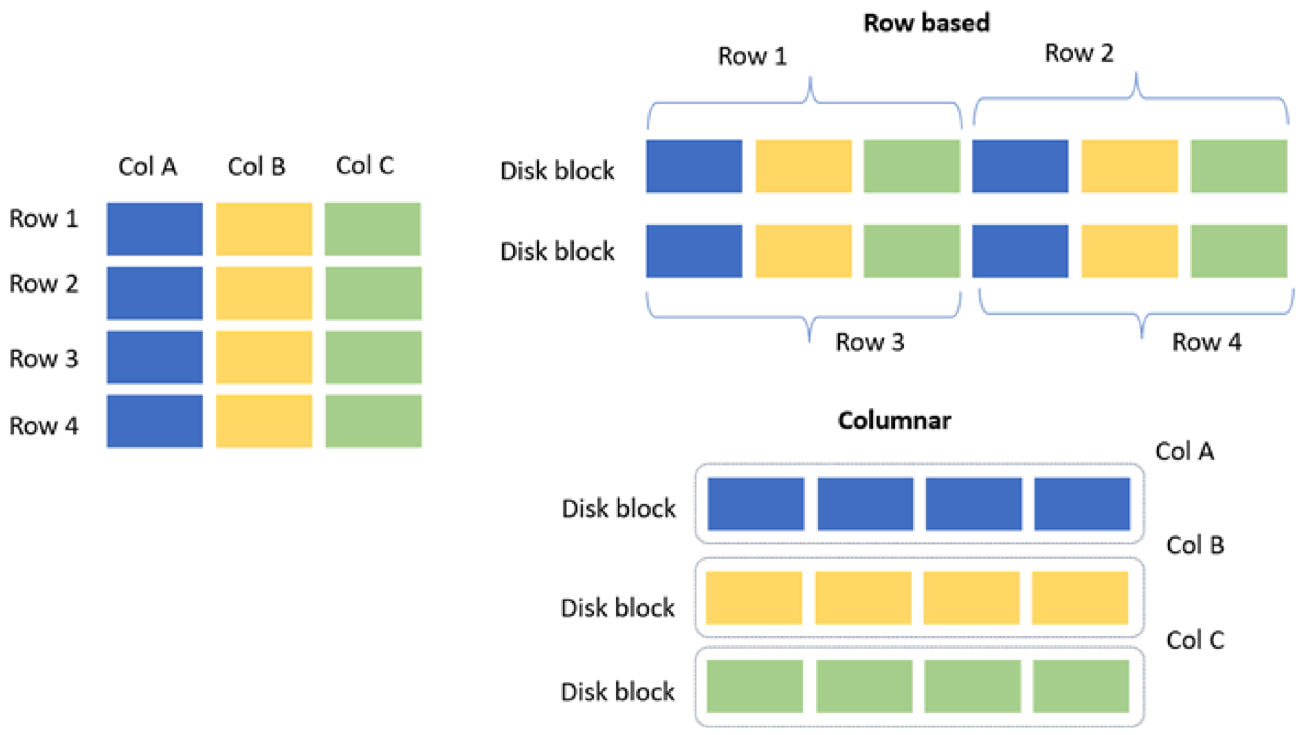
Row based, columnar and parquet structure

Before jumping into Parquet, let's first understand the difference between row-based and columnar formats.

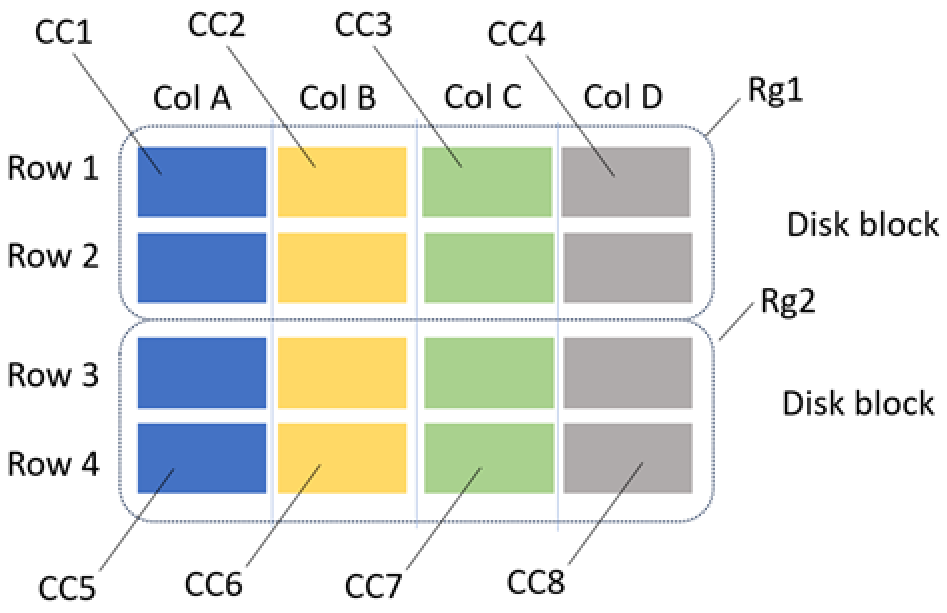
- In row-based systems, data is stored in disk blocks row by row.
- In columnar systems, data is stored column by column within disk blocks.

Both approaches have their own pros and cons. For **OLAP systems**, columnar storage is usually the better choice. For **OLTP systems**, row-based storage is preferred, since in most cases all columns of a row need to be read together.

The below image illustrates this difference more clearly.



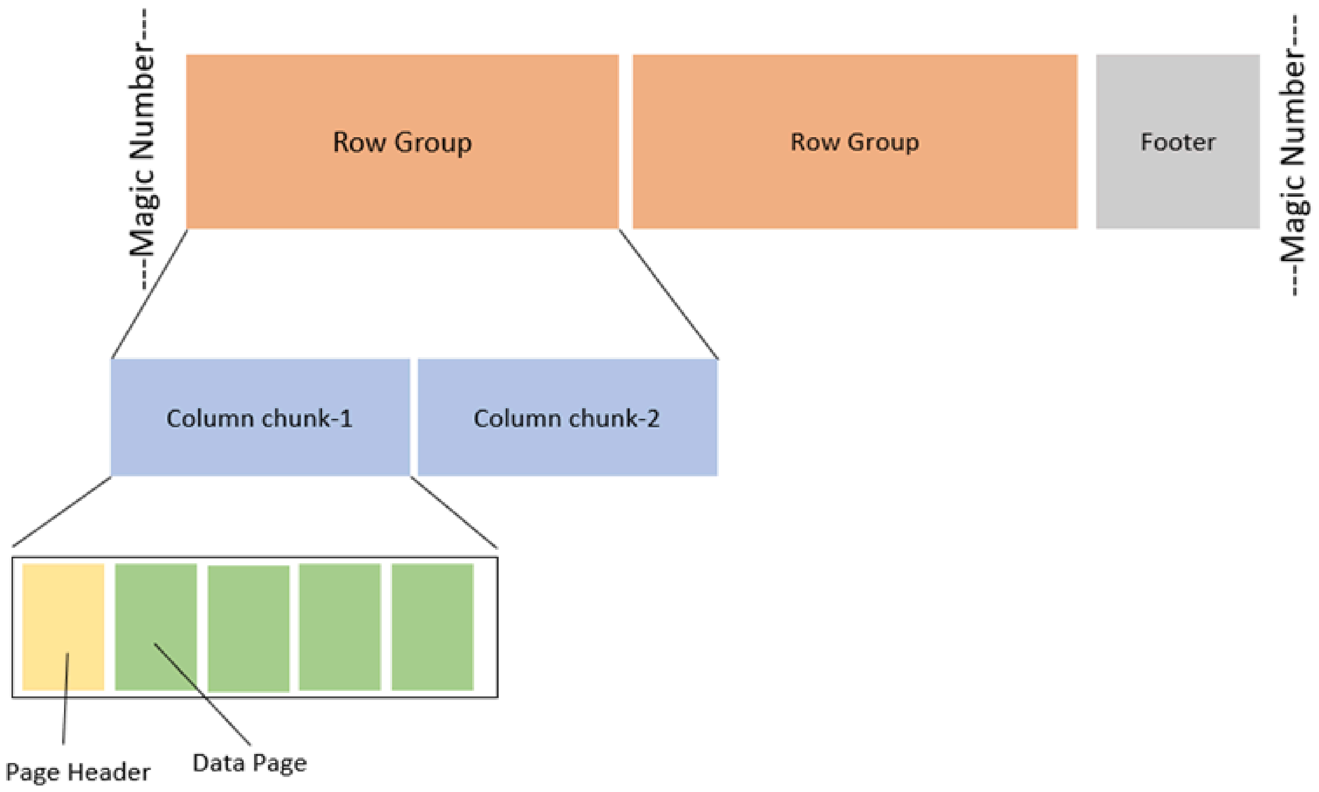
Parquet data structure as hybrid. As shown in the image below, a row is divided into Row Groups (RGs), and each row group is further divided into Column Chunks. Data is then written to storage row group by row group into disk blocks.



Note: In real scenarios, a row or column may not fully fit into a single disk block. In such cases, the remaining portion spills over into the next disk block. The illustration here is simplified for clarity.

Architecture of Parquet

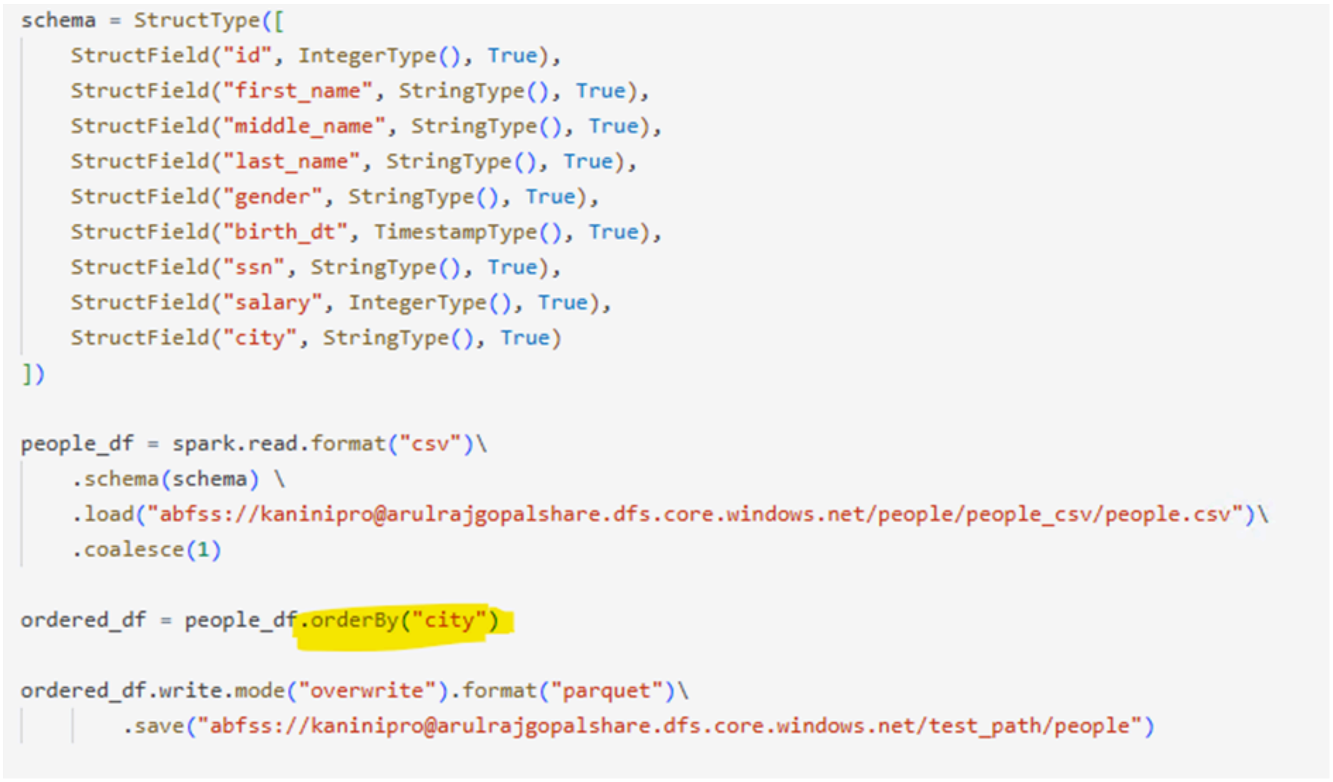
How parquet is stores data in hybrid and efficiently reading performing.



1. Row group: A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. A row group consists of a column chunk for each column in the dataset.
2. Column chunk: A chunk of the data for a particular column. They live in a particular row group and are guaranteed to be contiguous in the file.
3. Page: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding). There can be multiple page types which are interleaved in a column chunk.
4. Footer: contains file metadata
5. Page header metadata contains information about encoding. So, the reader can use this for decoding the data.
6. This magic number indicates that the file is in parquet format.

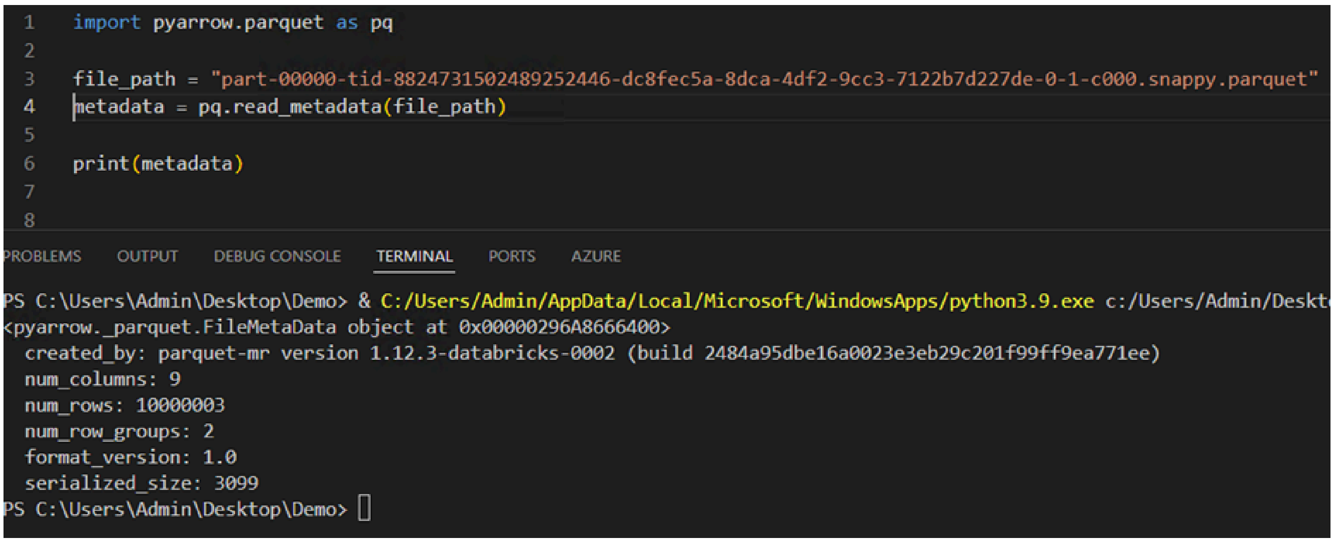
Parquet Slice and dice analysis

A Parquet file is written using the Spark code shown below. The key point to note is that an ORDER BY “city” column was applied on the DataFrame. As a result, while storing the data, each row group contains ordered city values instead of random distribution.



```
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("first_name", StringType(), True),
    StructField("middle_name", StringType(), True),
    StructField("last_name", StringType(), True),
    StructField("gender", StringType(), True),
    StructField("birth_dt", TimestampType(), True),
    StructField("ssn", StringType(), True),
    StructField("salary", IntegerType(), True),
    StructField("city", StringType(), True)
])

people_df = spark.read.format("csv")\
    .schema(schema) \
    .load("abfss://kaninipro@arulrajgopalshare.dfs.core.windows.net/people/people_csv/people.csv")\
    .coalesce(1)

ordered_df = people_df.orderBy("city")

ordered_df.write.mode("overwrite").format("parquet")\
        .save("abfss://kaninipro@arulrajgopalshare.dfs.core.windows.net/test_path/people")
```

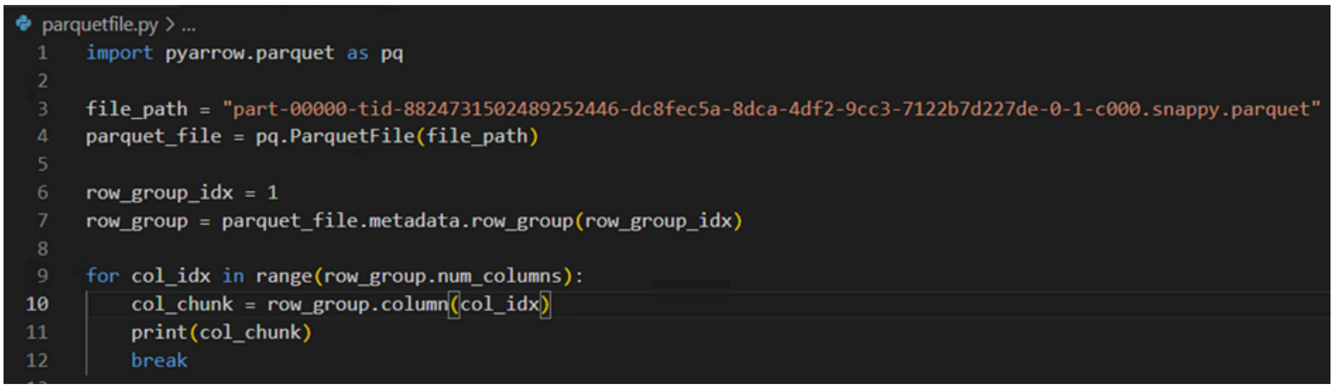
For analysis, I downloaded the file locally and inspected its metadata using pyarrow. From this, it is clear that the file contains two row groups.



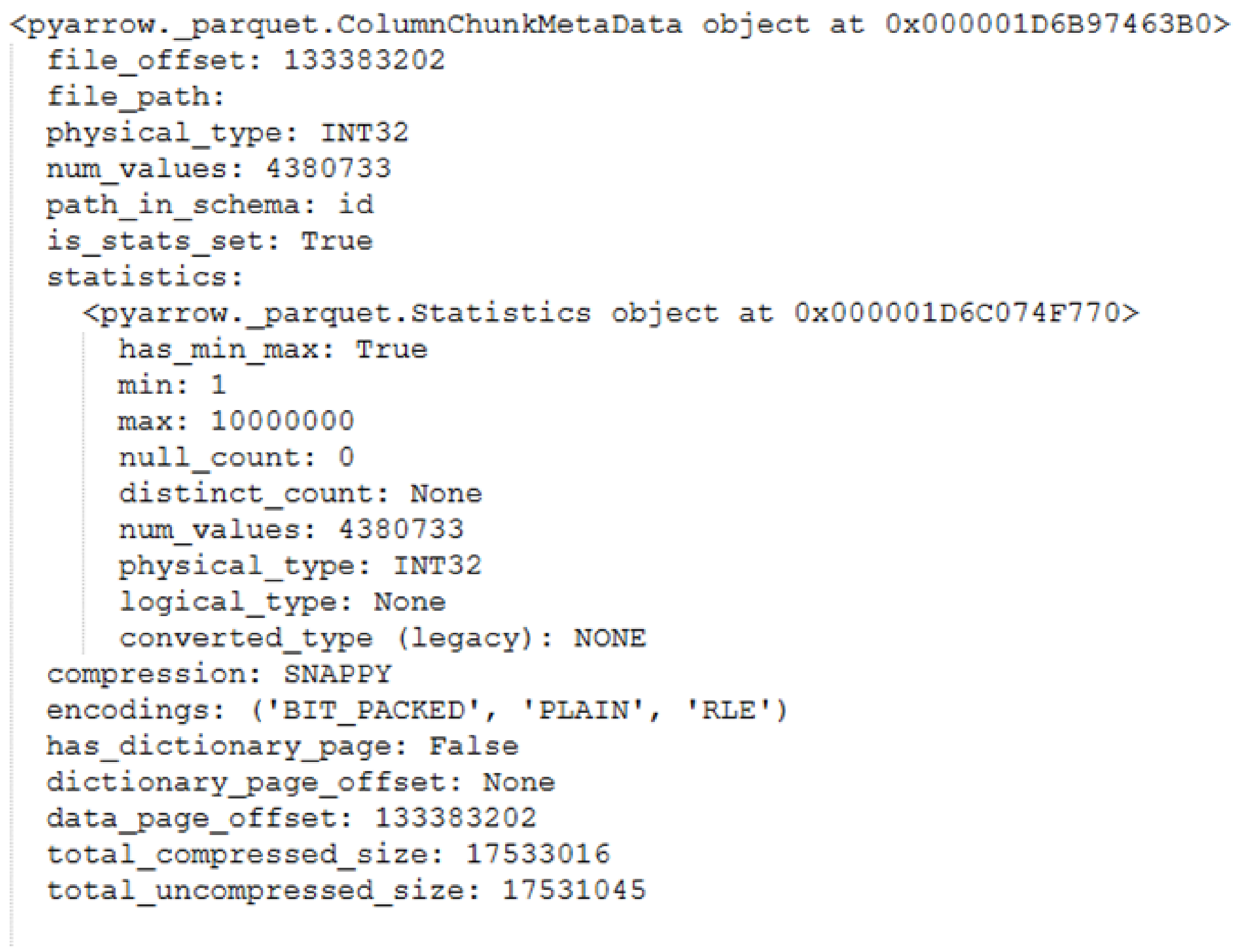
```python
import pyarrow.parquet as pq

file_path = "part-00000-tid-8824731502489252446-dc8fec5a-8dca-4df2-9cc3-7122b7d227de-0-1-c000.snappy.parquet"
metadata = pq.read_metadata(file_path)

print(metadata)
```

```
PS C:\Users\Admin\Desktop\Demo> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.9.exe c:/Users/Admin/Deskt
<pyarrow._parquet.FileMetaData object at 0x00000296A8666400>
  created_by: parquet-mr version 1.12.3-databricks-0002 (build 2484a95dbe16a0023e3eb29c201f99ff9ea771ee)
  num_columns: 9
  num_rows: 10000003
  num_row_groups: 2
  format_version: 1.0
  serialized_size: 3099
PS C:\Users\Admin\Desktop\Demo>
```

While inspecting one of the column chunks, the details revealed a lot of useful information, including the data type, minimum, and maximum values. Using these min and max statistics, data scans can be performed more efficiently by skipping irrelevant blocks.



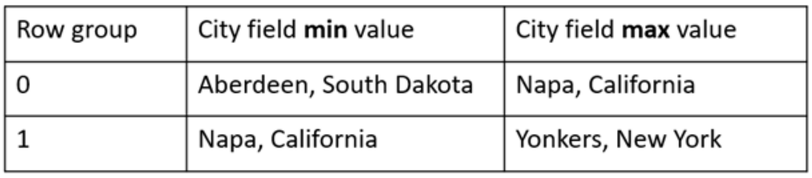
```python
import pyarrow.parquet as pq

file_path = "part-00000-tid-8824731502489252446-dc8fec5a-8dca-4df2-9cc3-7122b7d227de-0-1-c000.snappy.parquet"
parquet_file = pq.ParquetFile(file_path)

row_group_idx = 1
row_group = parquet_file.metadata.row_group(row_group_idx)

for col_idx in range(row_group.num_columns):
    col_chunk = row_group.column(col_idx)
    print(col_chunk)
    break
```



```
<pyarrow._parquet.ColumnChunkMetaData object at 0x000001D6B97463B0>
  file_offset: 133383202
  file_path:
  physical_type: INT32
  num_values: 4380733
  path_in_schema: id
  is_stats_set: True
  statistics:
    <pyarrow._parquet.Statistics object at 0x000001D6C074F770>
      has_min_max: True
      min: 1
      max: 10000000
      null_count: 0
      distinct_count: None
      num_values: 4380733
      physical_type: INT32
      logical_type: None
      converted_type (legacy): NONE
  compression: SNAPPY
  encodings: ('BIT_PACKED', 'PLAIN', 'RLE')
  has_dictionary_page: False
  dictionary_page_offset: None
  data_page_offset: 133383202
  total_compressed_size: 17533016
  total_uncompressed_size: 17531045
```

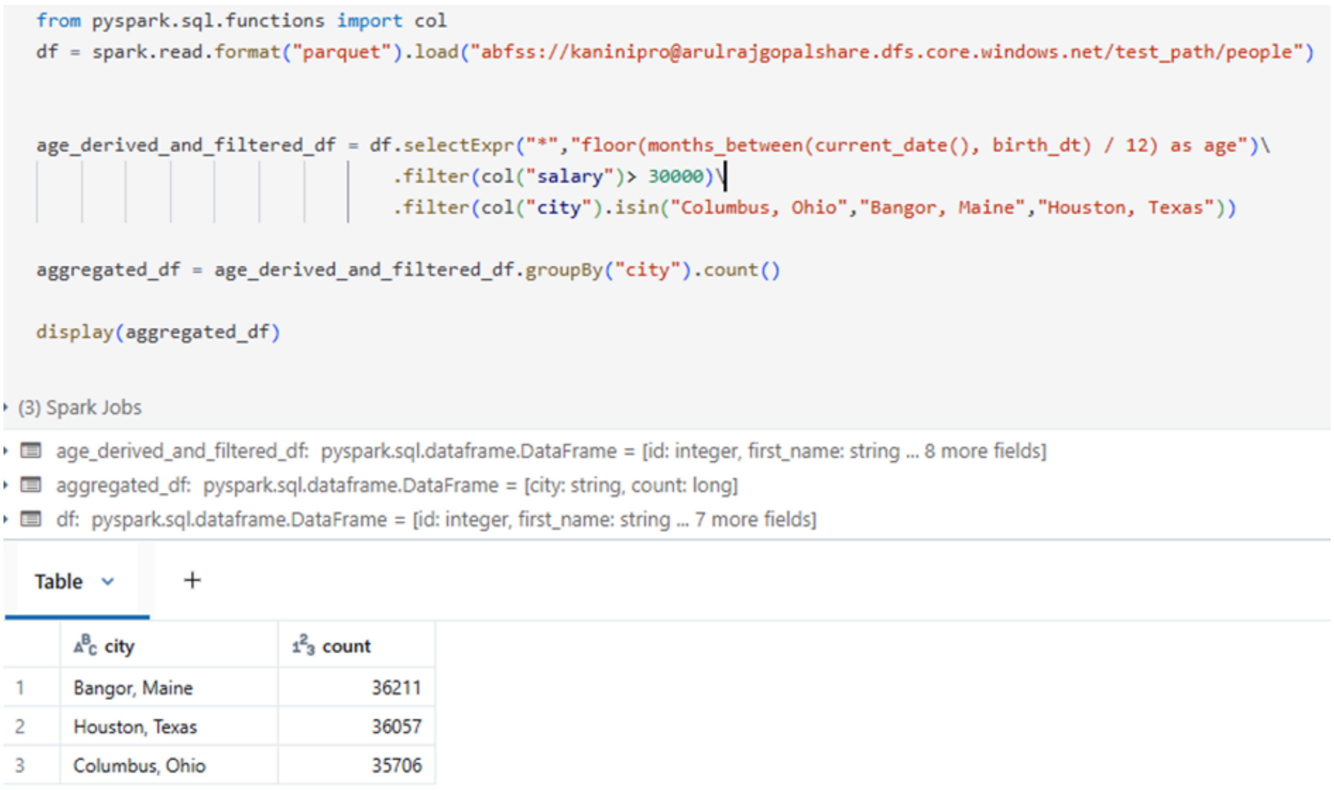
Demo on limiting data scan

The table below illustrates how the data is laid out for the **"city"** column which take from previous query. It clearly shows that while writing to Parquet, the **ORDER BY city** was applied, resulting in cities being sorted and divided into two row groups alphabetically.



Row group	City field **min** value	City field **max** value
0	Aberdeen, South Dakota	Napa, California
1	Napa, California	Yonkers, New York

The PySpark code below calculates the count of people in a few listed cities whose salary is greater than 30K. For this query, only the salary and city columns are required. Since all the listed cities fall within row group 0 , Spark can skip scanning row group 1. Let's see the spark execution metrics that how it performed the data scan effectively.



```
from pyspark.sql.functions import col
df = spark.read.format("parquet").load("abfss://kaninipro@arulrajgopalshare.dfs.core.windows.net/test_path/people")


age_derived_and_filtered_df = df.selectExpr("*","floor(months_between(current_date(), birth_dt) / 12) as age")\
                                     .filter(col("salary")> 30000)\
                                     .filter(col("city").isin("Columbus, Ohio","Bangor, Maine","Houston, Texas"))

aggregated_df = age_derived_and_filtered_df.groupBy("city").count()

display(aggregated_df)
```

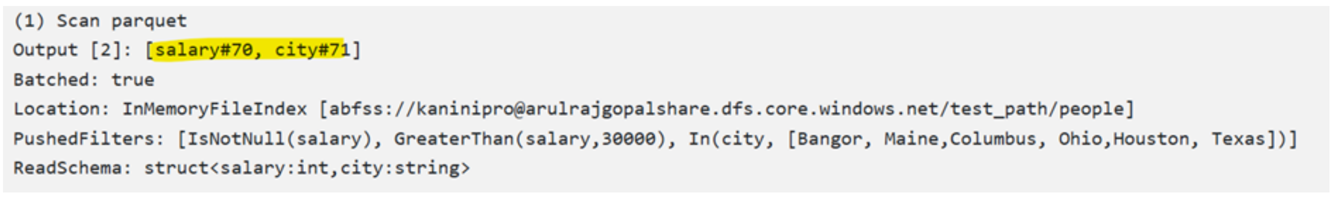
(3) Spark Jobs

- age_derived_and_filtered_df: pyspark.sql.dataframe.DataFrame = [id: integer, first_name: string ... 8 more fields]
- aggregated_df: pyspark.sql.dataframe.DataFrame = [city: string, count: long]
- df: pyspark.sql.dataframe.DataFrame = [id: integer, first_name: string ... 7 more fields]

Table

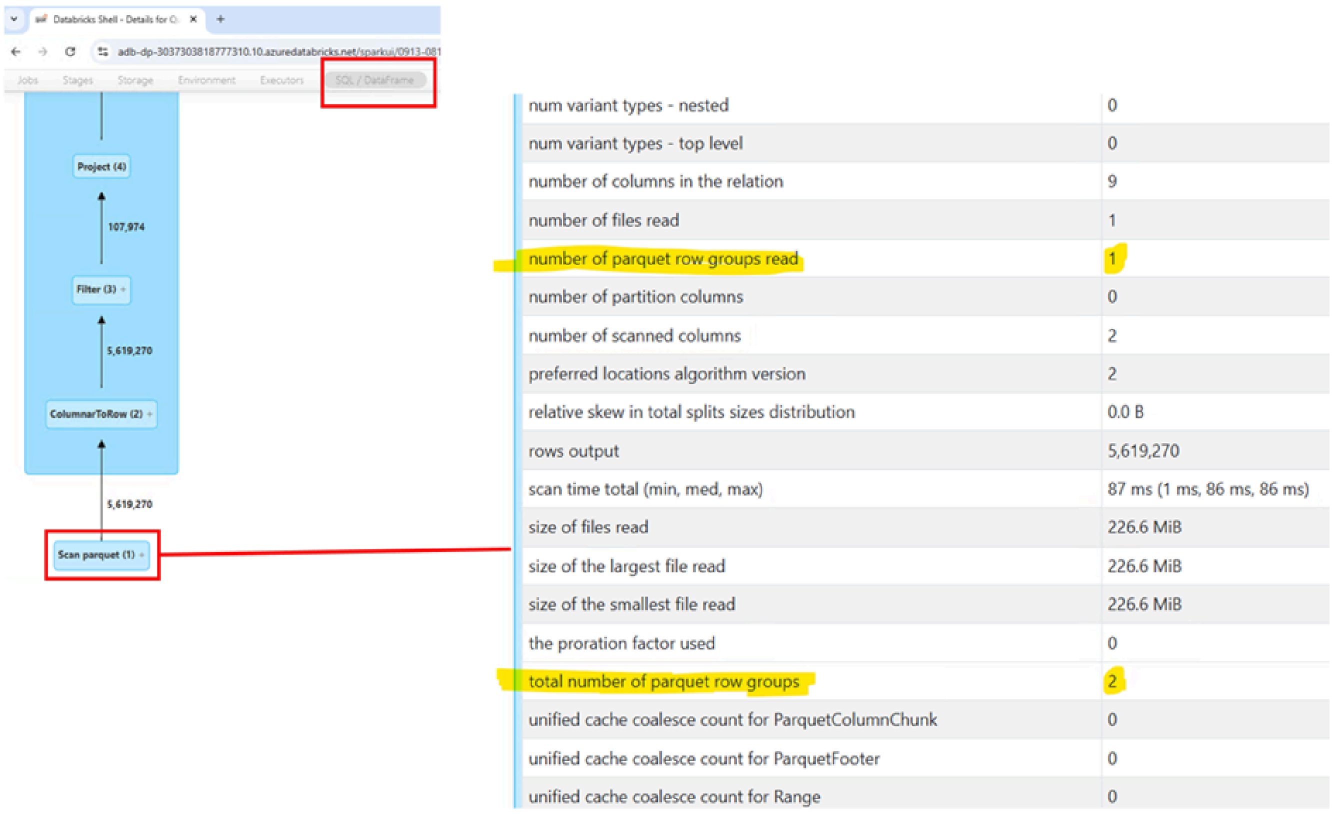
	city	count
1	Bangor, Maine	36211
2	Houston, Texas	36057
3	Columbus, Ohio	35706

Spark SQL plan



```
(1) Scan parquet
Output [2]: [salary#70, city#71]
Batched: true
Location: InMemoryFileIndex [abfss://kaninipro@arulrajgopalshare.dfs.core.windows.net/test_path/people]
PushedFilters: [IsNotNull(salary), GreaterThan(salary,30000), In(city, [Bangor, Maine,Columbus, Ohio,Houston, Texas])]
ReadSchema: struct<salary:int,city:string>
```

Spark scan metrics from spark UI



num variant types - nested	0
num variant types - top level	0
number of columns in the relation	9
number of files read	1
number of parquet row groups read	1
number of partition columns	0
number of scanned columns	2
preferred locations algorithm version	2
relative skew in total splits sizes distribution	0.0 B
rows output	5,619,270
scan time total (min, med, max)	87 ms (1 ms, 86 ms, 86 ms)
size of files read	226.6 MiB
size of the largest file read	226.6 MiB
size of the smallest file read	226.6 MiB
the proration factor used	0
total number of parquet row groups	2
unified cache coalesce count for ParquetColumnChunk	0
unified cache coalesce count for ParquetFooter	0
unified cache coalesce count for Range	0

From the Spark plan, it is clear that only the required two columns were read. The scan metrics further confirm that out of the two row groups, only one was scanned while the other was skipped as unnecessary. In other words, for this query, Spark processed only about 10% of the data to produce the result.

Capabilities like this are what make the Parquet format stand out.

Conclusion

Parquet's hybrid design combines the best of row-based and columnar storage, making it both flexible and efficient. Its metadata-driven optimizations enable selective scans, pruning, and compression that save time and resources.
This is why Parquet has become the backbone of modern analytical data platforms.

Leave a comment

Write a comment...

Log in or provide your name and email to leave a comment.

Email (Address never made public)

Name

Website (Optional)

Email me new posts

Instantly Daily Weekly

Email me new comments

Save my name, email, and website in this browser for the next time I comment.

Comment

← Previous: Delta Table: Under the Hood

Let's connect

LinkedIn

Mail

Recent posts

Parquet File Format Demystified

Delta Table: Under the Hood

Databricks System Tables

Demystifying Apache Spark: Jobs, Stages, and Tasks

How Spark Saves Time & Cost by Being Lazy

Spark Performance Pitfalls: The Hidden Cost of Misplaced Unions

“***I constantly see people rise in life who are not the smartest, sometimes not even the most diligent, but they are learning machines***” – Charlie Munger