Practical Work 4 in Cloud Computing

Gustavo Freitas Cunha

Federal University of Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1 Task 1: Data Modeling

1.1 Schema and Location of Each Table

The Data Lake is organized into two zones: **Silver** and **Gold**. Each table and its schema are detailed below:

Table 1: Silver Zone Tables

Table Name	**Schema**
`song_info`	`name`: STRING, `track_uri`: STRING, `duration`: LONG, `album_uri`: STRING, `artist_uri`: STRING
`album_info`	`name`: STRING, `album_uri`: STRING, `track_uri`: STRING, `artist_uri`: STRING
`artist_info`	`name`: STRING, `artist_uri`: STRING
`playlist_info`	`name`: STRING, `playlist_id`: LONG, `description`: STRING, `collaborative`: BOOLEAN
`playlist_tracks`	`playlist_id`: LONG, `position`: INTEGER, `track_uri`: STRING, `artist_uri`: STRING, `album_uri`: STRING, `duration`: LONG

Table 2: Gold Zone Tables

Table Name	**Schema**
`gold_playlist_info`	`playlist_id`: LONG, `number_of_tracks`: INTEGER, `number_of_artists`: INTEGER, `number_of_albums`: INTEGER, `total_duration`: LONG, `name`: STRING, `description`: STRING, `collaborative`: BOOLEAN
`gold_playlist_tracks`	`playlist_id`: LONG, `position`: INTEGER, `song_name`: STRING, `album_name`: STRING, `artist_name`: STRING

1.2 Data Transformations Across Layers

The transformations applied to the data across the Silver and Gold layers are as follows:

- **Raw to Silver Transformations:**
  - Data is ingested from JSON files (`playlists_v1.json` and `tracks_v1.json`).
  - Redundant data is removed using `distinct()` where necessary (e.g., `album_info`, `artist_info`).

    - Columns are selected and renamed to create normalized tables: `song_info`, `album_info`, `artist_info`, `playlist_info`, and `playlist_tracks`.

- **Silver to Gold Transformations:**
    - `gold_playlist_info`:
        * Aggregations are performed on `playlist_tracks` to compute metrics like `number_of_tracks`, `number_of_artists`, `number_of_albums`, and `total_duration`.
        * The aggregated data is joined with `playlist_info` to include playlist metadata.
    - `gold_playlist_tracks`:
        * Tracks are enriched by joining `playlist_tracks` with `song_info`, `album_info`, and `artist_info` to include track, album, and artist names.
        * Only relevant columns are selected for the final table.

1.3 Performance Comparison Between JSON and Parquet

To evaluate the performance of JSON and Parquet formats, the `gold_playlist_info` table was loaded and counted using both formats. The measured load times are as follows:

- **JSON**: 0.41 seconds
- **Parquet**: 0.26 seconds

The results demonstrate that the Parquet format is faster for data loading. This improvement can be attributed to Parquet's columnar storage format, which optimizes both storage space and query performance compared to JSON.

2 Task 2: Data Pipeline

During this task, my primary difficulty was the incompatibility between Data Lake and Parquet operations. I needed to make some research for implementing the manipulation and updating of the tables in both layers.

After implementing the initial code for this task, I encountered significant slowness due to the combination of several expensive operations in PySpark, such as joins, unions, and multiple writes to disk. When applied repeatedly without proper optimizations, these operations led to high execution times. To improve performance, I applied several optimization strategies. First, I cached the DataFrames to prevent repeated data reads and reduce the execution time of subsequent operations. Additionally, I replaced the union operation with an approach using the `coalesce` function, allowing new data to replace old data without the need for duplication. Another key improvement was reducing the number of writes, consolidating them at the end of the process, which resulted in a significant reduction in I/O overhead. I also performed a conditional update specifically for playlist ID 11992, avoiding the need for additional joins in this case. These changes resulted in more efficient execution and reduced processing time.

Additionally, I have found some obstacles with columns data types when I was merging the new dataset and I had to make some explicit data casts.

Also, I catch myself wondering how I could verify that the updates have been reflected on the tables in the Data Lake. To solve that, I implemented an additional code comparing the `v1` and `v2` datasets and asserting whether the updates have been successfully applied.

3 Task 3: New Data Pipeline Using Delta Format

In Task 3, I implemented the data lake construction and updates using the Delta Lake format, and compared its performance and storage usage to the previous implementation using Parquet in Task 2. The main differences between the two implementations are primarily related to the time taken for operations and the storage space consumed.

For Task 2, the total time for constructing and updating the data lake in Parquet was 49.33 seconds, with a total storage size of 203.28 MB. This includes the Parquet loading time of 0.26 seconds for the initial dataset. In comparison, the implementation in Task 3, which used the Delta Lake format, took 35.86 seconds for all operations, and the total storage size was 117.40 MB.

The time comparison between Task 2 and Task 3 shows that the Delta Lake implementation is faster. Specifically, the combined time for Task 2 ($T1 + T2 = 0.26 + 49.33 = 49.59$ seconds) is slightly higher than the time for Task 3, which took 35.86 seconds. This indicates that, in this case, Delta Lake provides a more efficient way of handling data operations.

Regarding storage, the Delta Lake format is more space-efficient, as the total size of the data lake in Delta format (117.40 MB) is significantly smaller compared to the Parquet version (203.28 MB). This reduction in storage usage can be attributed to Delta Lake's built-in features, such as its support for incremental updates and efficient handling of data versions.

Overall, while both formats serve their purpose in a data lake architecture, Delta Lake outperforms Parquet in both execution time and storage efficiency in this particular implementation.