Plans and DataFrame API in Spark

Database Administration
Lab Guide 7

2022/2023

Implement SQL queries with DataFrame operators and analyze their plans. Execute queries over data stored in a distributed file system.

Steps

1. Follow the steps in the appendix to deploy the cluster and to store the data files in Google's Cloud Storage.
2. Using DataFrame SQL operators (check the appendix for more information), implement the following queries:
   - The main list of people (title.principals) involved in the title with `tconst = "tt3896198"`, returning `nconst`, `primaryName` (name.basics), `category`, and `characters` (place the `where` operator after the `join`);
   - The top 10 people with more titles, returning `nconst, primaryName, count`.
3. Execute the above queries and analyze their plans. This can be done in one of two ways: by accessing the history server at `http://localhost:18080`, selecting the respective application, choosing the SQL/DataFrame tab, and finally clicking the query description[1]; or by using `explain(mode="extended")` as the last operation in the query.
4. Using the DataFrame function `write.json(`*`path`*`)`, store the JSON output of the two queries in Google's Cloud Storage.

Questions

1. Using the `explain(mode="extended")` operator in the first query, compare the *Analyzed Logical Plan* with the *Optimized Logical Plan*. What is the main difference between the two? Why did the optimizer decide to change the initial plan?
2. Analyze the physical plan of the first query. What filters were pushed down to the source? What are the advantages behind predicate pushdown for remote data?
3. Analyze the physical plan of the second query. Explain the following sequence of operators: *HashAggregate* → *Exchange* → *HashAggregate*.
4. What is the interest in decoupling compute from storage, as well as storing data objects in distributed storage systems such as Google's Cloud Storage or Amazon's S3?

Learning Outcomes Get familiarized with the DataFrame SQL API. Understand plans generated by Spark. Get familiarized with distributed data storage and understand its importance in data processing tasks.

[1] Make sure the query is actually executed, by using, e.g., the `show()` operator.

Spark cluster with history server and GCS connector

1. Download and extract the new supplementary files;
2. Deploy the cluster with Docker Compose:
   `docker-compose -p spark up -d --scale spark-worker=3`
3. Start the history server:
   `docker exec spark_spark_1 start-history-server.sh`

Cloud Storage instructions

IMPORTANT: Delete unused data after completing the guide to avoid unwanted costs.

1. Create a new bucket:
   (a) (On the left menu) Select Cloud Storage → Buckets → Create;
   (b) Name the bucket;
   (c) Choose where to store your data → Region → us-east1;
   (d) Select Create.
2. Create a service account to access the bucket and generate a credential key:
   (a) (On the left menu) Select IAM and admin → Service accounts → Create service account;
   (b) Enter the account name → Create service account;
   (c) Select *Storage Object Admin* as the role → Continue;
   (d) Select Done.
   (e) Select the new account → Manage Keys → Add Key → Create new key → JSON;
   (f) Add the downloaded file to the `app` folder with the name `credentials.json`.
3. Add the Parquet files to the bucket, by clicking Upload Files or by dragging the files directly in the bucket's page (Cloud Storage → Buckets → *Bucket name*).

Main DataFrame SQL operators

- `df1.`**`join`**`(df2, df1["a"] == df2["b"])` – joins `df1` and `df2` using columns `a` and `b`;
- `df.`**`where`**`(`*`expr`*`)` – filters `df1` with some expression. Expression examples:
  `df["a"] == "abc"`
  `df["b"] > x`, where x is a variable
  `even(df["b"])`, where $even(x)$ is a function that returns true if x is an even number
- `df.`**`select`**`(df["a"], df["b"] * 10, df["c"])` – projects `a`, `b` times 10, and `c`;
- `df.`**`groupBy`**`(df["a"], df["b"]]).`**`agg`**`(`f`(df["c"])` – groups by columns `a` and `b` and executes the aggregation function f over column `c`. Examples of aggregation functions:[2] `count`, `avg`, `max`, `min`, `sum`;
- `df.`**`sort`**`(df["a"], ascending=False)` – sorts by column `a`, from top to bottom;
- `df.`**`limit`**`(`n`)` – returns the n first rows;

Note that string expressions may also be used. For example, the two expressions are equivalent:
`df.where(df["a"] == "abc")` and `df.where("a = 'abc'")`.

[2]The following code must be added: `from pyspark.sql.functions import` f, where f is the aggregation function to import.