DATABRICKS NOTES BY PRADEEP

1. Overview Of Databricks and Ecosystem

Subject: Apache Spark

Topics: #spark #databricks #basics

Let's understand basics of Databricks and why do even need it on first place

The Problem

Nowadays, most organizations are adapting to Data Driven Decision Making (DDDM) approach for decision-making. They are spending millions to interpret the data that is ubiquitously available around the organization sphere.

There are many solutions available for big data processing; out of all these solutions, Apache Spark (released in May 2014) became very popular in a short period of time. It is very fast, flexible, and developer-friendly, which were the reasons for its popularity.

Organizations started configuring their own on-premise dedicated clusters to process their big data workloads using Apache Spark. But the main difficulties with this setup was infra scalability, maintenance, and security.

As we know, managing organizational data never comes easy because data grows rapidly and scales frequently. And data centers were facing challenges with regard to maintenance, security as well as scalability.

In recent years, organizations started moving to cloud, as cloud technology became popular and more reliable. Cloud technology gives rise to many SaaS products pertaining to big data.

The Solution

Among all these products, Databricks is the industry leading cloud data platform for big data processing and machine learning.

Databricks is an Apache Spark based cloud unified analytics platform that can perform large-scale big data processing and machine learning workloads with ease. Databricks was started in the year 2013 by the original creators of Apache Spark

Databricks provides a user-friendly programmable and interactive notebook environment where Data Engineers/Analysts, Machine Learning Engineers, and Data Scientists can work collaboratively as the platform supports R, Python, SQL, and Scala languages.

Databricks notebooks are so flexible that they can have code, documentation/text, data visualization in real time, and so on at one place. These notebooks can also be version controlled. All the code will be executed on Databricks spark clusters

What can we do with Databricks?
As a Data Engineer: We can build data lakes/data warehouses and perform data processing at scale. For example, Data Engineers read data from adls in databricks, apply transformation logic with a notebook, and output as new file to adls.
As a Data Analyst: We can perform analytics and visualization for business decisions over the data at scale. For example, data analysts can create visualization reports on the top of a data lake prepared by the data engineers.
As a Data Scientist/ML Engineer: We can build machine learning/deep learning pipelines and process petabytes of data for predictive analytics. For example, data scientists can build machine learning models to predict business insights, and so on.
Databricks is integrated with AWS, Azure, and GCP cloud providers, so it can
be used within these cloud providers as a service (or) it is also available as a
standalone cloud solution.

. Databricks Architecture**
Subject: Apache Spark
Topics: #spark #databricks #architecture

Databricks is designed with flexibility in mind, supporting multiple cloud vendors such as AWS, Azure, and GCP. Its underlying operations are abstracted to allow customers to focus on Data Engineering and Data

Science tasks.

Databricks operates with two main components:

1. **Control Plane**

- **Web Application:** Provides an interface for managing interactive notebooks, cluster management, jobs, and queries for Databricks SQL.

- **Notebooks:** Web-based interface containing executable code, visualizations, and narrative text, used for ETLs, machine learning, etc.

- **Cluster Management:** Handles provisioning and maintaining clusters based on user-defined parameters like worker nodes, memory, and Spark runtime version. Also supports auto-scaling.

- **Jobs:** Scheduled or immediate tasks to run a notebook on a Databricks cluster, providing a non-interactive way to execute code or visualizations.

- **Queries:** Databricks SQL is a web-based query editor for executing SQL queries and visualizing data.

2. **Data/Compute Plane**

- Resides in the client's cloud (compatible with AWS, Azure, and GCP).

- **Classic Compute Plane:** Computation resources are in the client's AWS account, used for notebooks, jobs, and Databricks SQL warehouses.

In Simple Terms:

- The **Control Plane** includes the backend services managed by Databricks in your account.

- The **Compute Plane** is where data processing occurs.

Here's a formatted version of your notes on Databricks Lakehouse Architecture:

3.. Databricks Lakehouse Architecture**

Subject: Apache Spark

Topics: #spark #databricks #architecture #lakehouse

A data lakehouse is a modern, open data management architecture that integrates the flexibility, cost-efficiency, and scalability of data lakes with the data management and ACID transactions of data warehouses. This architecture enables business intelligence (BI) and machine learning (ML) on all data types.

Databricks Lakehouse Platform offers an architecture that combines the strengths of data lakes and data warehouses. This combination provides high-quality, reliable data at a low cost and supports both structured and unstructured data.

Data lakehouses are powered by a new, open system design, which implements data structures and management features similar to those in a data warehouse directly on the low-cost storage typical of data lakes.

Databricks Lakehouse Platform Architecture Components:

1. **Workspace**
- The core of the Databricks Lakehouse Platform, serving as a collaborative environment for data teams. In the workspace, users can create notebooks, schedule jobs, manage clusters, and perform other essential data workflow tasks.

2. **Runtime**
- **Databricks Runtime** is the execution engine optimized for data workloads. Built on Apache Spark, it includes enhancements and optimizations to deliver high performance for both big data and machine learning tasks.

3. **Cloud Services**
- **Databricks Cloud Services** provide the underlying infrastructure for the Lakehouse Platform. They manage cluster operations, job scheduling, security, and more, ensuring scalability and robustness for handling large data workloads.

Delta Lake

Subject: Apache Spark

Topics: #spark #databricks #architecture #deltalake

Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads. Integrated with Databricks, Delta Lake provides a more reliable and performant data management solution, addressing many of the limitations of traditional data lakes.

Basics of Delta Lake

Delta Lake sits on top of your existing data lake, providing a transactional storage layer. Fully compatible with Apache Spark, Delta Lake supports efficient batch and streaming data processing. It stores data in **Parquet files**, a columnar storage format optimized for fast data processing.

Key Features of Delta Lake:

1. **ACID Transactions:** Ensures atomicity, consistency, isolation, and durability, allowing concurrent reads and writes and preventing data corruption.
2. **Scalable Metadata Handling:** Handles petabytes of data and billions of files without performance degradation.
3. **Schema Enforcement & Evolution:** Enforces schema on write to ensure data quality while allowing schema changes without breaking pipelines.
4. **Time Travel (Data Versioning):** Maintains a history of transactions, enabling access to previous data versions for auditing, rollbacks, and reproducing experiments.
5. **Unified Batch and Streaming:** Supports both batch and streaming data as sources and sinks for easier pipeline building.

Delta Lake Transaction Logs

The **Delta Lake transaction log** (DeltaLog) is an ordered record of every transaction ever performed on a Delta table. It serves as a **single source of truth**, ensuring accurate data views and guaranteeing atomicity for all operations (INSERT, UPDATE, DELETE).

How the Transaction Log Works:

- Each transaction is broken down into discrete steps (commits):
- **Add file:** Adds a data file.

- **Remove file:** Removes a data file.
- **Update metadata:** Changes table schema, name, or partitioning.
- **Set transaction:** Marks a micro-batch as committed.
- **Change protocol:** Updates to new software protocol features.
- **Commit info:** Records the operation, its origin, and timestamp.

Delta Lake Transaction Log at the File Level:
When a table is created, its transaction log is saved in the `_delta_log` subdirectory, with each commit recorded as a JSON file. The first commit is `000000.json`, followed by `000001.json`, `000002.json`, and so on.

Basic Delta Table Operations:

```sql
-- Create table
CREATE TABLE employees
(id INT, name STRING, salary DOUBLE);

-- Insert data
INSERT INTO employees
VALUES
(1, "Adam", 3500.0),
(2, "Sarah", 4020.5),
(3, "John", 2999.3),
(4, "Thomas", 4000.3),
(5, "Anna", 2500.0),
(6, "Kim", 6200.3);

-- Select data
SELECT * FROM employees;

-- Describe table details
```

```
DESCRIBE DETAIL employees;

-- Update data
UPDATE employees
SET salary = salary + 100
WHERE name LIKE "A%";

-- View table history
DESCRIBE HISTORY employees;

-- Check Delta log
%fs ls 'dbfs:/user/hive/warehouse/employees/_delta_log';
%fs head 'dbfs:/user/hive/warehouse/employees/_delta_log/00000000000000000003.json';
```

For a deep dive into Delta Lake, check out this blog post:
[https://www.databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html]
(https://www.databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html)

---

5. Databricks Notebook & File System (DBFS)
Subject: Apache Spark
Topics: #spark #databricks #dbfs

The **Databricks File System (DBFS)** is a distributed file system installed on Databricks clusters. DBFS leverages cloud storage to provide a scalable, secure, and managed data storage environment that integrates seamlessly with Databricks' analytics and AI capabilities.

DBFS allows users to store data files, notebooks, libraries, and other artifacts, making them accessible across various Databricks workspaces and clusters. It simplifies file management in a distributed environment by providing a familiar file system interface on top of the underlying cloud storage.

DBFS supports operations such as **reading**, **writing**, **moving**, and **deleting** files and directories. One of the primary tools for interacting with DBFS is **dbutils**, a collection of utilities available within Databricks notebooks for performing various tasks, including file system operations.

How to Interact with DBFS Using dbutils

dbutils provides several commands for interacting with DBFS, making it easy to manage files directly from a Databricks notebook. Below are examples of common file operations using `dbutils`.

Accessing DBFS

You can access DBFS using the `dbutils.fs` module, which provides functions similar to those of a local file system.

Listing Files and Directories

To list files and directories, use the `ls` command.

```python
# List the contents of the root directory of DBFS
dbutils.fs.ls("/")
```

```
# List the contents of a specific directory
dbutils.fs.ls('/databricks-datasets')

# Storing and displaying the list of files
files = dbutils.fs.ls('/databricks-datasets')
print(files)
display(files)
```

Creating Directories

To create a new directory in DBFS, use the `mkdirs` command.

```python
# Create a new directory in DBFS
dbutils.fs.mkdirs("/my-new-directory")
```

Uploading and Downloading Files

Files can be uploaded to DBFS using the Databricks UI, or moved programmatically using the `cp` command.
For downloading files to your local system, the **Databricks CLI** can be used.

```python
# Copy a file from one location in DBFS to another
dbutils.fs.cp("/my-source-directory/my-file.txt", "/my-destination-directory/my-file.txt")
```

Reading and Writing Files

To read and write files in DBFS, you can use **Spark** or any library that supports **Hadoop File System (HDFS)** paths.

Example of Writing a Text File:

```python
# Writing to a file in DBFS
textData = ["Hello, Databricks!", "This is a text file in DBFS."]
sparkContext.parallelize(textData).saveAsTextFile("/my-directory/my-text-file.txt")
```

Example of Reading a Text File:

```python
# Reading from a file in DBFS
textFileRDD = sparkContext.textFile("/my-directory/my-text-file.txt")
for line in textFileRDD.collect():
print(line)
```

Deleting Files and Directories

To delete a file or directory, use the `rm` command. If you need to remove a directory and all its contents, use the recursive option.

```python
```

```
# Delete a file
dbutils.fs.rm("/my-directory/my-text-file.txt")

# Delete a directory recursively
dbutils.fs.rm("/my-directory", recurse=True)
```

---

32. Delta Lake

Subject: Apache Spark

Topics: #spark #databricks #architecture #deltalake

Delta Lake is an open-source storage layer that brings **ACID transactions** to **Apache Spark** and big data workloads. Integrated with **Databricks**, it provides a more reliable and performant way to manage data for analytics and AI, solving many challenges that traditional data lakes face in modern data architectures.

Basics of Delta Lake

Delta Lake sits on top of existing data lakes and provides a **transactional storage layer**. It is fully compatible with **Apache Spark**, enabling efficient batch and streaming data processing. Delta Lake uses **Parquet files**, a columnar storage format optimized for fast data processing and analysis.

Key Features of Delta Lake:

1. **ACID Transactions**

Delta Lake ensures atomicity, consistency, isolation, and durability, allowing concurrent reads and writes, reducing the risk of data corruption.

2. **Scalable Metadata Handling**

Delta Lake handles **petabytes of data** and **billions of files** efficiently without performance decline.

3. **Schema Enforcement and Evolution**

It enforces schema-on-write to ensure data quality while allowing schema evolution without breaking pipelines.

4. **Time Travel (Data Versioning)**

Delta Lake stores historical versions of data, enabling **auditing**, **rollbacks**, and **experiment reproduction**.

5. **Unified Batch and Streaming Sources/Sinks**

Delta Lake serves as both a source and sink for **batch** and **streaming data**, making complex data pipelines easier to manage.

Delta Lake Transaction Logs

The **Delta Lake transaction log** (DeltaLog) tracks every change made to a Delta Lake table.

Transaction Log Use:

- **Single Source of Truth**

The transaction log ensures that all readers and writers access the correct view of the data at any time.

Atomicity in Delta Lake:

Delta Lake uses the transaction log to guarantee atomicity in operations like **INSERT** or **UPDATE**—either the operation completes fully, or not at all.

How the Transaction Log Works

Delta Lake breaks down operations into **atomic commits**:

- **Add file**: Adds a data file.
- **Remove file**: Removes a data file.
- **Update metadata**: Updates table metadata (e.g., schema, partitioning).
- **Set transaction**: Records a streaming job's commit.
- **Change protocol**: Enables new features by upgrading the transaction log protocol.
- **Commit info**: Contains details about the commit (operation, source, and time).

Each commit is recorded in the **_delta_log** subdirectory as ordered **JSON files**.

Working with Delta Tables:

Create Table:

```sql
CREATE TABLE employees
(id INT, name STRING, salary DOUBLE);
```

Insert Data:

```sql
INSERT INTO employees VALUES (1, "Adam", 3500.0), (2, "Sarah", 4020.5);
INSERT INTO employees VALUES (3, "John", 2999.3), (4, "Thomas", 4000.3);
INSERT INTO employees VALUES (5, "Anna", 2500.0);
INSERT INTO employees VALUES (6, "Kim", 6200.3);
```

Select Data:

```sql
SELECT * FROM employees;
```

Update Data:

```sql
UPDATE employees
SET salary = salary + 100
WHERE name LIKE "A%";
```

Check Table History:

```sql
DESCRIBE HISTORY employees;
```

Inspect Delta Log:

```sql
%fs ls 'dbfs:/user/hive/warehouse/employees/_delta_log';
%fs head 'dbfs:/user/hive/warehouse/employees/_delta_log/00000000000000000003.json';
```

For a more in-depth understanding, check out this [blog post on Delta Lake](https://www.databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html).

33. Advanced Delta Lake (Time Travel, Optimize, Vacuum)

Subject: Apache Spark

Topics: #spark #databricks #deltalake

Time Travel

Delta Lake offers a powerful feature called **time travel**, which allows you to access and revert to previous versions of data stored in Delta tables. This is useful for:

- **Version Control**
- **Auditing**
- **Tracking Changes Over Time**
- **Restoring Data after Mistakes (e.g., accidental DELETE or UPDATE statements)**

Syntax for Time Travel:

- Using a **timestamp**:

```sql
SELECT * FROM my_table TIMESTAMP AS OF "2019-01-01"
```

- Using a **version number**:

```sql
SELECT * FROM my_table VERSION AS OF 36
-- or
SELECT * FROM my_table@v36
```

Restoring a Table:

- Restore table to a **specific timestamp**:

```sql
RESTORE TABLE my_table TO TIMESTAMP AS OF '2019-01-01';
```

- Restore table to a **specific version**:

```sql
RESTORE TABLE my_table TO VERSION AS OF 36;
```

Example:

- Check **table history**:

```sql
DESCRIBE HISTORY employees;
```

- Select data from a **specific version**:

```sql
SELECT * FROM employees VERSION AS OF 1;
-- or
SELECT * FROM employees@v1;
```

- Delete table and **restore it**:

```sql
DELETE FROM employees;
RESTORE TABLE employees TO VERSION AS OF 5;
SELECT * FROM employees;
```

Compacting Small Files

In **Delta Lake**, you may end up with many small files, which can slow down queries. **Compaction** combines these files into larger ones for better performance.

Z-Ordering:

- **Z-Ordering** is a technique that organizes data files based on column values, which improves query performance.

```sql
OPTIMIZE employees ZORDER BY id;
```

- Check **table details** and **history**:

```sql
DESCRIBE DETAIL employees;
DESCRIBE HISTORY employees;
```

The VACUUM Command

The **VACUUM** command removes old data files that are no longer needed, helping to free up storage and maintain a clean data environment. It is essential to be cautious with **retention periods** when using VACUUM to avoid deleting files necessary for time travel.

- Basic **VACUUM** command:

```sql
VACUUM my_table;
```

Important Notes:

- The **default retention period** is usually 7 days.
- When VACUUM is used, **time travel is no longer possible** for the deleted versions.

Example:

1. Check the **file system** after vacuuming:

```sql
%fs ls 'dbfs:/user/hive/warehouse/employees';
```

2. Set **retention to 0 hours**:

```sql
VACUUM employees RETAIN 0 HOURS;
```

3. If you encounter an error, disable the retention check:

```sql
SET spark.databricks.delta.retentionDurationCheck.enabled = false;
VACUUM employees RETAIN 0 HOURS;
```

4. After vacuuming, trying to **select an older version**:

```sql
SELECT * FROM employees@v1;
-- Error? This version has been removed because it was vacuumed.
```

34. Databases and Tables on Databricks

Subject: Apache Spark

Topics: #spark #databricks #databases #tables

Databases in Databricks

A **database** in Databricks is a logical grouping of tables, similar to a **schema** in traditional databases like SQL or Snowflake. It contains tables, views, and functions, which helps organize data into logical groups for easier management, especially with large datasets or multiple projects.

- **Create a Database**:

```sql
CREATE DATABASE IF NOT EXISTS my_database;
```

- Databricks uses the **Hive metastore** to store metadata for databases, tables, and views.
- **Hive metastore** = repository for table metadata.

Tables in Databricks

Tables in Databricks are structured collections of data, similar to relational databases, where data is stored in rows and columns. There are two types of tables in Databricks:

1. **Managed Tables**:
- Databricks manages **both the data and the metadata**.
- If you drop a managed table, **both the data and metadata** are deleted from DBFS.

```sql
CREATE TABLE my_database.my_managed_table (
id INT,
data STRING
) USING DELTA;
```

2. **Unmanaged (External) Tables**:
- Databricks manages only the **metadata**, while the **data remains stored externally** in a specified

location.

- If you drop an unmanaged table, **only the metadata** is deleted, and the data remains in storage.

```sql
CREATE TABLE my_database.my_unmanaged_table (
id INT,
data STRING
) USING DELTA
LOCATION '/mnt/my-external-storage/my-table-data';
```

The LOCATION Keyword

- **LOCATION** is used in SQL commands to specify where the data for a table is stored in the underlying file system.
- For **managed tables**, if no LOCATION is specified, Databricks stores the data in a default directory.
- For **unmanaged tables**, the LOCATION is defined, and the data is stored externally.

Hands-On Examples

1. Creating and Managing Tables:

1. **Create a managed table**:
```sql
CREATE TABLE managed_default (width INT, length INT, height INT);
INSERT INTO managed_default VALUES (3, 2, 1);
```

2. **Describe table information**:

```sql
DESCRIBE EXTENDED managed_default;
```

3. **Create an external table**:

```sql
CREATE TABLE external_default (width INT, length INT, height INT)
LOCATION 'dbfs:/mnt/demo/external_default';
INSERT INTO external_default VALUES (3, 2, 1);
```

4. **Drop the tables**:

- Drop the **managed table**:

```sql
DROP TABLE managed_default;
```

- Drop the **external table**:

```sql
DROP TABLE external_default;
```

5. **Check the file system**:

- For **managed table files** (check after dropping):

```sql
%fs ls 'dbfs:/mnt/demo/managed_default';
```

- For **external table files**:

```sql
%fs ls 'dbfs:/mnt/demo/external_default';
```

```

---

#### **2. Working with Schemas**:

1. **Create a new schema**:
```sql
CREATE SCHEMA new_default;
DESCRIBE DATABASE EXTENDED new_default;
```

2. **Switch to a new schema**:
```sql
USE new_default;
```

3. **Create a table in the new schema**:
- **Managed table**:
```sql
CREATE TABLE managed_new_default (width INT, length INT, height INT);
INSERT INTO managed_new_default VALUES (3, 2, 1);
```

- **External table**:
```sql
CREATE TABLE external_new_default (width INT, length INT, height INT)
LOCATION 'dbfs:/mnt/demo/external_new_default';
INSERT INTO external_new_default VALUES (3, 2, 1);
```

4. **Drop the tables**:
```

- Drop **managed** and **external tables**:

```sql
DROP TABLE managed_new_default;

DROP TABLE external_new_default;
```

5. **Check file system for tables**:

- **Managed table**:

```sql
%fs ls 'dbfs:/user/hive/warehouse/new_default.db/managed_new_default';
```

- **External table**:

```sql
%fs ls 'dbfs:/mnt/demo/external_new_default';
```

---

3. Create Custom Schema

1. **Create a schema with a custom location**:

```sql
CREATE SCHEMA custom LOCATION 'dbfs:/Shared/schemas/custom.db';

USE custom;
```

35. Views in Databricks

Subject: Apache Spark

Topics: #spark #databricks #views

In Databricks, views function similarly to tables in that they save a predefined query that you can reference

as if it were a table.

However, views do not store the data themselves but instead execute the saved query upon each reference.

There are three main types of views in Databricks:

simple (or basic) views

temporary views

global temporary views.

Each serves different scopes and lifetimes.

Simple (Basic) Views

A simple view is a saved SQL query that you can reference like a table in your SQL statements. It's persistent and stored in the metastore, meaning it remains available across different sessions and clusters until you drop the view.

syntax

```
-- Create a simple view
CREATE VIEW my_database.my_simple_view AS
SELECT id, data
FROM my_database.my_table
WHERE filter_column = 'value';

-- Use the simple view in a query
SELECT * FROM my_database.my_simple_view;
```

Temporary Views

Temporary views are session-scoped and will not be available once the session terminates. They are useful for sharing data within a session without affecting other users or sessions.

syntax

```
-- Create a temporary view
CREATE TEMP VIEW my_temp_view AS
SELECT id, data
FROM my_database.my_table
WHERE filter_column = 'value';

-- Use the temporary view in a query
SELECT * FROM my_temp_view;

-- The temporary view is no longer available after the session ends
```

Global Temporary Views

Global temporary views are similar to temporary views, but they are tied to a system preserved temporary database called global_temp. This makes them accessible across multiple sessions and clusters, but they are still temporary and will be dropped when the Spark application terminates.

```
-- Create a global temporary view
CREATE GLOBAL TEMP VIEW my_global_temp_view AS
SELECT id, data
FROM my_database.my_table
WHERE filter_column = 'value';

-- Use the global temporary view in a query (must use the global_temp database)
SELECT * FROM global_temp.my_global_temp_view;

-- The global temporary view is available to all sessions until the Spark application ends
```

Example

Simple View

```
CREATE TABLE IF NOT EXISTS smartphones
(id INT, name STRING, brand STRING, year INT);

INSERT INTO smartphones
VALUES (1, 'iPhone 14', 'Apple', 2022),
```

```
(2, 'iPhone 13', 'Apple', 2021),
(3, 'iPhone 6', 'Apple', 2014),
(4, 'iPad Air', 'Apple', 2013),
(5, 'Galaxy S22', 'Samsung', 2022),
(6, 'Galaxy Z Fold', 'Samsung', 2022),
(7, 'Galaxy S9', 'Samsung', 2016),
(8, '12 Pro', 'Xiaomi', 2022),
(9, 'Redmi 11T Pro', 'Xiaomi', 2022),
(10, 'Redmi Note 11', 'Xiaomi', 2021)

SHOW TABLES



CREATE VIEW view_apple_phones
AS SELECT *
FROM smartphones
WHERE brand = 'Apple';

SELECT * FROM view_apple_phones;



SHOW TABLES;
Temp View
CREATE TEMP VIEW temp_view_phones_brands
AS SELECT DISTINCT brand
FROM smartphones;

SELECT * FROM temp_view_phones_brands;

SHOW TABLES;
Global View
CREATE GLOBAL TEMP VIEW global_temp_view_latest_phones
```

```
AS SELECT * FROM smartphones
WHERE year > 2020
ORDER BY year DESC;

SELECT * FROM global_temp.global_temp_view_latest_phones;

-- you can query in diffrent notebook

SHOW TABLES;

SHOW TABLES IN global_temp;
```

36. Delta Tables (Working with Files)

Subject: Apache Spark

Topics: #spark #databricks #deltatables #code

Overview of Delta Tables

Delta Tables in Apache Spark are an advanced version of traditional tables, leveraging Delta Lake's capabilities to provide ACID transactions, scalable metadata handling, and the ability to handle both batch and streaming data efficiently. They are built on top of Parquet files but offer additional features for data management.

Creating Delta Tables Using Spark

1. Writing Delta Tables

You can create a Delta table by writing a Spark DataFrame in Delta format. This involves saving the DataFrame to a specified location in Delta format.

Example:

```python
from pyspark.sql import SparkSession

# Start Spark session
spark = SparkSession.builder.appName("DeltaTableExample").getOrCreate()

# Create data DataFrame
data = spark.range(0, 5)

# Write the data DataFrame to /delta location in Delta format
data.write.format("delta").save("/delta")
```

Partitioning Data:

Partitioning helps improve performance by organizing data into smaller, more manageable pieces based on a specific column.

```python
# Write the Spark DataFrame to Delta table partitioned by date
data.withColumn("date", spark.current_date()).write.partitionBy("date").format("delta").save("/delta")
```

Modes of Writing:

- **Append**: Adds new data to an existing Delta table.
- **Overwrite**: Replaces the existing data with new data.

```python
# Append new data to your Delta table
data.write.format("delta").mode("append").save("/delta")

# Overwrite your Delta table
data.write.format("delta").mode("overwrite").save("/delta")
```

---

2. Reading Delta Tables

Delta tables can be read into Spark DataFrames using the Delta format.

Example:

```python
# Read the Delta table into a DataFrame
df = spark.read.format("delta").load("/delta")

# Show the DataFrame
df.show()
```

You can also use SQL to read from Delta tables.

```sql
SELECT * FROM delta.`/delta`
```

Creating Delta Tables from Existing Datasets

Example with Credit Card Fraud Dataset:

1. **Create a Temporary View:**

```sql
CREATE DATABASE credit_card;
USE credit_card;

CREATE OR REPLACE TEMP VIEW credit_card_temp_view AS
SELECT *
FROM parquet.`dbfs:/databricks-datasets/credit-card-fraud/data/part-00000-tid-8989911650787988880-9c1caa7b-283d-47c4-9be1-aa61587b3675-0-c000.snappy.parquet`;
```

2. **Create a Managed Delta Table from the View:**

```sql
CREATE TABLE credit_card_managed
USING DELTA
AS SELECT * FROM credit_card_temp_view;
```

Example with Flight Data:

1. **Create a Temporary View from CSV:**

```sql
CREATE DATABASE flight;
USE flight;
```

```
CREATE OR REPLACE TEMP VIEW temp_flights_view AS
SELECT *
FROM csv.`dbfs:/databricks-datasets/flights/departuredelays.csv`;
```

2. **Create a Delta Table Using the Temporary View:**

```sql
CREATE TABLE flight_data
USING DELTA
AS SELECT * FROM temp_flights_view;
```

Working with Delta Tables in Spark

Example of Creating a Delta Table from a CSV File:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when

# Start Spark session
spark = SparkSession.builder.appName("FlightDataDeltaLake").getOrCreate()

# Load CSV file into DataFrame
df = spark.read.format("csv").option("header", "true").load("dbfs:/databricks-
datasets/flights/departuredelays.csv")

# Define the path for the Delta table
deltaTablePathManaged = "/delta/flights_delay_managed"
```

```
# Write the DataFrame to a Delta table (managed)
df.write.format("delta").mode("overwrite").save(deltaTablePathManaged)

# Read the data from the Delta table
flightDataDF = spark.read.format("delta").load(deltaTablePathManaged)

# Display the schema
flightDataDF.printSchema()

# Filter Flights Delayed by More than 60 Minutes
delayedFlightsDF = flightDataDF.filter("delay > 60")

# Add a New Column for Significant Delays
enhancedFlightsDF = delayedFlightsDF.withColumn("significant_delay", when(col("delay") > 120,
"Yes").otherwise("No"))

# Aggregate: Average Delay by Origin Airport
avgDelayByOriginDF = enhancedFlightsDF.groupBy("origin").avg("delay").withColumnRenamed("avg(delay)",
"average_delay")

# Show the result of the transformations
avgDelayByOriginDF.show()

# Save the Transformed Data
transformedDeltaTablePath = "/delta/transformed_flights_delay"
avgDelayByOriginDF.write.format("delta").mode("overwrite").save(transformedDeltaTablePath)
```

Summary

Delta Tables provide advanced features for data management in Spark, including ACID transactions, schema evolution, and efficient data handling. They can be created from various data sources and formats and support multiple modes of writing and reading data. The use of Delta Tables enhances data reliability and performance in both batch and streaming contexts.

37. Medallion Architecture Complete Guide

Subject: Apache Spark

Topics: #spark #databricks #architecture #lakehouse

Overview of Medallion Architecture

The Medallion Architecture is a data design pattern used in data lakehouses to manage and organize data efficiently. It involves three key layers—Bronze, Silver, and Gold—each serving a specific purpose in the data processing pipeline. This architecture helps in improving data quality and structuring data in a way that supports both batch and streaming data flows.

Medallion Architecture Layers

1. Bronze Layer (Raw Data)

- **Purpose**: Acts as the landing zone for raw data. No transformations or business rules are applied at this stage.
- **Data Storage**: Data is stored in its original format—CSV, JSON, Parquet, or AVRO.
- **Example**: If you're ingesting CSV data, it remains in CSV format in the Bronze layer.

Example Code:

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("MedallionArchitecture").getOrCreate()

# Define the path for the Bronze layer
bronzePath = "/mnt/delta/customers/bronze/"

# Read raw data into DataFrame
bronzeDF = spark.read.format("csv") \
.option("header", "true") \
.option("inferSchema", "true") \
.load("dbfs:/databricks-datasets/retail-org/customers/customers.csv")

# Write raw data to Bronze layer in Delta format
bronzeDF.write.mode("overwrite").format("delta").save(bronzePath)
```

2. Silver Layer (Cleansed and Normalized Data)

- **Purpose**: Cleanse, normalize, and enhance data. This includes transforming data into standard formats, handling missing values, and deduplicating.
- **Data Storage**: Data is stored in a cleaned and normalized format.

Example Code:

```python
from pyspark.sql.functions import col, initcap, trim, upper, when

# Define the path for the Silver layer
silverPath = "/mnt/delta/customers/silver/"
```

```
# Load data from Bronze layer
silverDF = spark.read.format("delta").load(bronzePath) \
.withColumn("customer_name", initcap(trim(col("customer_name")))) \
.withColumn("state", upper(trim(col("state")))) \
.withColumn("city", initcap(trim(col("city")))) \
.fillna({"tax_code": "UNKNOWN"}) \
.dropDuplicates(["customer_id"])

# Write cleansed data to Silver layer in Delta format
silverDF.write.mode("overwrite").format("delta").save(silverPath)
```

3. Gold Layer (Business-Level Aggregates)

- **Purpose**: Create business-level aggregates and perform final transformations. This layer typically includes detailed analysis and aggregation.
- **Data Storage**: The data is aggregated and transformed to provide business insights.

Example Code:

```python
from pyspark.sql.functions import count, collect_list, desc
from pyspark.sql.window import Window

# Define the path for the Gold layer
goldPath = "/mnt/delta/customers/gold/"

# Load data from Silver layer
goldDF = spark.read.format("delta").load(silverPath) \
.groupBy("state", "customer_age_group") \
.agg(count("*").alias("total_customers"),
collect_list("customer_name").alias("sample_customers")) \
```

```
.orderBy("state", "total_customers")

# Write aggregated data to Gold layer in Delta format
goldDF.write.mode("overwrite").format("delta").save(goldPath)
```

SQL for Analytics

Once the Gold layer is created, you can perform various analytical queries using SQL.

Example SQL Queries:

```sql
CREATE DATABASE customer;
USE customer;

-- Create a Delta table from the Gold layer
CREATE TABLE customer_gold
AS SELECT *
FROM delta.`/mnt/delta/customers/gold/`;

-- Query to get all records from the Gold table
SELECT * FROM customer_gold;

-- Query to find top 10 states with the most customers
SELECT state, total_customers
FROM customer_gold
ORDER BY total_customers DESC
LIMIT 10;
```

```
-- Query to get total customers by state
SELECT state, SUM(total_customers) AS customers_in_state
FROM customer_gold
GROUP BY state
ORDER BY customers_in_state DESC;

-- Query to find states with less than 50 customers
SELECT state, total_customers
FROM customer_gold
WHERE total_customers < 50
ORDER BY total_customers;
```

Summary

The Medallion Architecture provides a structured approach to data processing in a lakehouse environment. By organizing data into Bronze, Silver, and Gold layers, it facilitates better data management, quality control, and analytics. This architecture supports both batch and streaming data flows and enhances the usability and reliability of data across an organization.

38. In-Depth Parquet Files

Subject: Apache Spark

Topics: #spark #databricks #architecture #fileformat

Overview of Apache Parquet

Apache Parquet is a widely-used columnar storage file format known for its efficiency and performance in handling big data. It is open-source and designed to be compatible with various Hadoop data processing frameworks.

Key Characteristics of Parquet:

- **Column-Oriented Storage**: Unlike row-based formats like CSV, Parquet stores data in columns. This means that all values for a particular column are stored together, rather than all values for a particular row. This layout is beneficial for analytical workloads, as it allows for more efficient data processing and retrieval.

- **Columnar Compression**: Parquet's column-oriented format facilitates better compression because similar data types and values are stored together. Different encoding schemes can be applied to different data types, enhancing compression efficiency.

- **Self-Describing Format**: Parquet files contain metadata about the data schema, column statistics, and row groups, which helps with data skipping and querying efficiency. This metadata includes information such as min/max values and number of values, making Parquet a self-describing format.

- **Row Groups and Column Chunks**: Parquet files are organized into row groups, which contain column chunks. Each column chunk is further divided into pages that store the actual column data. This organization allows for efficient reading and processing of specific columns, improving performance.

Example Parquet File Structure:

1. **Row Groups**: Each row group contains data for a set of rows and is divided into column chunks.
2. **Column Chunks**: Each column chunk contains data for a particular column within a row group.
3. **Pages**: Each column chunk is divided into pages that store the actual data.

Advantages of Parquet Files

1. High Performance

- **Efficient Data Retrieval**: Since Parquet is column-oriented, it can read only the necessary columns for a given query, reducing the amount of data read and improving query performance.
- **Enhanced Compression**: Columnar storage enables better compression algorithms as similar values are stored together. This reduces storage requirements and speeds up data processing.

2. Cost-Effective

- **Reduced Storage Costs**: Better compression and encoding reduce the amount of disk space needed to store the data, which translates to lower storage costs.
- **Efficient Data Transfer**: Smaller file sizes lead to reduced data transfer times and costs, making Parquet an economical choice for large-scale data operations.

3. Interoperability

- **Wide Support**: Parquet has been widely adopted across various big data tools and frameworks (e.g., Apache Hadoop, Apache Spark, Apache Drill, etc.), ensuring compatibility and ease of integration.
- **Standardization**: Its long-standing popularity and broad adoption mean that it is a well-supported and standardized format, making it a reliable choice for data storage and processing.

Example Usage in Apache Spark

Writing Data to Parquet Format:

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("ParquetExample").getOrCreate()

# Create a DataFrame
data = spark.range(0, 10)
```

```
# Write DataFrame to Parquet format
data.write.format("parquet").save("/path/to/parquet/data")
```

Reading Data from Parquet Format:

```python
# Read the data from Parquet format
df = spark.read.format("parquet").load("/path/to/parquet/data")

# Show the DataFrame
df.show()
```

Performance Considerations:

- **Column Pruning**: Use column pruning to read only the necessary columns from Parquet files, which can significantly speed up queries.
- **Partitioning**: For large datasets, partitioning data in Parquet files can improve performance by reducing the amount of data scanned for each query.

Resources

- [Apache Parquet Documentation](https://parquet.apache.org/docs/)
- [Apache Spark Documentation on Parquet](https://spark.apache.org/docs/latest/sql-data-sources-parquet.html)

By understanding and leveraging the advantages of Parquet files, organizations can enhance the efficiency

and performance of their big data processing and analytical workflows.