Spark integration

Integration of Spark with different metastores / catalogs and lakehouses

JERIN SAM

Table of Contents

[Spark, Lakehouse and Catalog 2](#_Toc188985382)

[Spark 2](#_Toc188985383)

[Key features 2](#_Toc188985384)

[Lakehouse 2](#_Toc188985385)

[Key features 2](#_Toc188985386)

[Catalog 3](#_Toc188985387)

[Key features 3](#_Toc188985388)

[Integration Architecture 4](#_Toc188985389)

[Wite Operations 4](#_Toc188985390)

[Read Operations 5](#_Toc188985391)

[Integrations 5](#_Toc188985392)

[Spark and Delta Lake Integration 5](#_Toc188985393)

[Services: 5](#_Toc188985394)

[Installation and Configuration 5](#_Toc188985395)

[Spark Session 9](#_Toc188985396)

# Spark, Lakehouse and Catalog

## Spark

Apache Spark is an open-source, distributed data processing framework designed for large-scale data analytics and processing.

### Key features

* **Unified Engine:** Supports batch processing, stream processing, machine learning, and graph processing.
* **In-Memory Processing**: Processes data in memory, making it much faster than traditional disk-based processing.
* **Distributed Computing**: Divides tasks across multiple nodes in a cluster for parallel execution.
* **Support for Multiple Languages**: Compatible with Python, Java, Scala, and R.
* **Extensive Libraries**: Includes built-in libraries such as Spark SQL, Spark Streaming, MLlib (for machine learning), and GraphX (for graph processing).

## Lakehouse

Lakehouse is a modern data architecture that combines the best features of a data lake and a data warehouse. It is designed to manage large volumes of structured and unstructured data while enabling analytics and reporting at scale.

### Key features

* **Unified Storage:** Stores raw and processed data in a Data Lake.
* **Support for Transactions**: Ensures ACID (Atomicity, Consistency, Isolation, Durability) compliance for reliable data operations.
* **Schema Enforcement**: Supports schema management and governance, similar to a data warehouse.
* **Data Lake Flexibility**: Retains the ability to handle unstructured and semi-structured data.
* **High Performance**: Optimized for querying and analytical workloads.

**Open-Source Lakehouse techs are –**

1. Delta Lake
2. Iceberg
3. Hudi

## Catalog

A **Catalog** is a service that manages metadata about datasets. It acts as a central repository that stores information about datasets, schemas, and other metadata.

### Key features

* **Metadata Management**: Stores information about tables, columns, data types, and more.
* **Data Governance**: Helps enforce security, access control, and compliance policies.
* **Integration**: Facilitates integration with processing engines like Spark or SQL engines like Trino.

**Open-Source Catalog services are –**

1. Unity Catalog – Primarily supports Delta Table but can provide interoperability for Apache Iceberg as well using UniForm feature.
2. Nessie – Supports Iceberg and Delta Table, provides git like branching feature for data.
3. Hive Metastore – Supports Iceberg and Delta Table.
4. Apache Polaris – Supports Iceberg.

**Following Catalogs support interoperability with all the 3 major lakehouses –**

1. Apache XTable
2. Unity Catalog

# Integration Architecture

Integration architecture of Spark communicating with Catalog and Data Lake can be seen below.

A diagram of a catalog

Description automatically generated

## Wite Operations

When a ***Write Operation*** is executed in Spark, following processes are executed –

1. ***Interaction with the Catalog***

* **Metadata Registration:**
  + Spark registers the table's metadata (e.g., schema, location, partitioning information) in the configured catalog (e.g., Hive Metastore, Apache Nessie, Unity Catalog).
  + This metadata is stored in the catalog for table discovery, governance, and querying.
* **Storage Path:**
  + The table's storage location (data lake path) is usually specified in the CREATE TABLE statement or determined by the catalog's configuration.

1. ***Interaction with the Data Lake***

* **Data Location**
  + Spark does not store the actual data in the catalog. Instead, the data is written directly to the data lake.
* **Write Operation**
  + When data is inserted into the table (e.g., using INSERT INTO), Spark writes the data directly to the specified path in the data lake.
  + For *managed tables, the catalog manages the location*. For *external tables, the location is user-defined, and the catalog only stores the metadata.*

## Read Operations

When a ***Read Operation*** is executed in Spark, following processes are executed –

1. ***Metadata Retrieval from Catalog***

* Spark queries the catalog/metastore to fetch metadata about the table, such as:
  + **Schema**: Column names and data types.
  + **Storage Location**: The path in the data lake (e.g., s3://bucket/path or /hdfs/path).
  + **Partitioning Information**: Details of partition columns and their locations.
  + **File Format**: The format of the files (e.g., Parquet, ORC, Delta, etc.).
* The catalog/metastore does not contain the data itself, only the metadata required to locate and interpret the data.

1. ***Direct Data Access from Data Lake***
   * Once Spark retrieves the storage location from the catalog/metastore, it bypasses the catalog/metastore for data reading.
   * Spark communicates directly with the underlying data lake (e.g., HDFS, S3, ADLS, GCS) to access and process the data

# Integrations

## Spark and Delta Lake Integration

### Services:

* Spark Master
* Spark Worker
* Jupyter Lab

### Installation and Configuration

* **Installation and Config Folder** ***- /install-and-config/1-spark-delta-integration***

Docker is used to setup the spark and jupyter services. These files need to be accessed in the following order for better understanding–

* + *Dockerfile.spark*
  + *requirement.txt and spark-config*
  + *entrypoint.sh*
  + *docker-compose.yml*

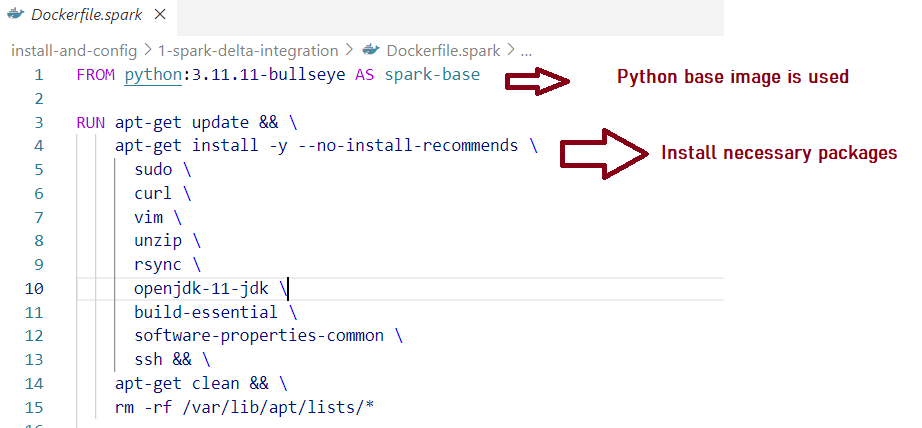
Below snippets shows details of some important scripts present in docker, entrypoint or notebook –

* ***Dockerfile.spark***

In this Dockerfile, Spark is configured with necessary details like Environment Variables, required python packages like jupyter and entrypoint file which needs to execute at the start of docker container.

The same Dockerfile will be used to setup *Spark Master and Worker docker containers.*

Following snippets from Dockerfile of Spark will cover only the important part of the Dockerfile scripts.



A screenshot of a computer program

Description automatically generated

A green arrow pointing at a line

Description automatically generated with medium confidence

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A close-up of a sign

Description automatically generated

* ***entrypoint.sh***

Entrypoint file is used to execute the bash or shell scripts whenever the container starts.

In this entrypoint file, based on the parameter following scripts will be executed, these parameters are passed from *docker-compose.yml* file to *entrypoint* script. (**Note: Passing parameter from docker-compose file can be seen in the docker-compose snippet** **present below**) –

1. Parameter = ***master***

* Start spark master by executing spark master script.
* Create a pyspark session, which will start jupyter lab server. Token for the jupyter lab can be found by executing below script in spark master container.

|  |
| --- |
| ***jupyter server list*** |

1. Parameter = ***worker***

* Start spark worker by executing spark worker script.

1. Parameter = ***history***

* Start spark history server by executing spark history server script.

A screenshot of a computer error

Description automatically generated

A screenshot of a computer

Description automatically generated

* ***Docker-compose.yml***

***Docker Compose*** is used to define and manage multi-container Docker applications.

This helps to orchestrates multiple containers as part of a single application.

Instead of managing each container individually, manage all of them with one command.

A screenshot of a computer screen

Description automatically generated

***Note: Other Services are also created in the similar method shown in the above snippet.***

### Spark Session

Spark Session can be created using following methods –

1. Jupyter Notebook.
2. Executing Pyspark command with necessary configs (as declared in entrypoint file).

In this integration, the PySpark command is used to initialize a Spark session. Since the Dockerfile specifies Jupyter in the PySpark environment variables, executing the PySpark command automatically starts the Jupyter server and creates the Spark session. This session can then be directly accessed in Jupyter Notebook using the ***spark*** variable.

#### Details of Spark Session using Delta Lake

Snippet of Spark Session Script –

A close-up of a computer code

Description automatically generated

In the above script, following configs enable Delta Table capability in Spark.

* **appName("jupyter-pyspark-session").master("spark://spark-master:7077")**
* **appName("jupyter-pyspark-session") 🡪** Names the Spark application
* **master("spark://spark-master:7077")** **🡪** Specifies the Spark master node (running in a distributed Spark cluster).
* **--packages** ***io.delta:delta-spark\_2.12:3.2.0***
  + ***Purpose***:

Adds the Delta Lake library to Spark session.

* + ***Details***:
    - io.delta:delta-spark\_2.12: Refers to the Delta Lake connector library compatible with Scala 2.12.
    - 3.2.0: Specifies the version of the Delta Lake library.
  + ***Why It’s Needed***:
    - Spark doesn’t natively support Delta Lake; the library is required to enable Delta-specific features.
    - Provides functionality for reading, writing, and managing Delta tables.
  + **Key Features**:
    - Integration with Spark DataFrames and SQL APIs.
    - Delta operations like MERGE, UPDATE, DELETE, and time travel.
* **--conf "*spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension*"**
  + ***Purpose***:
    - Activates Delta Lake SQL extensions in the Spark session.
  + ***Details***:
    - DeltaSparkSessionExtension: Adds Delta Lake-specific SQL features to the Spark SQL engine.
  + ***Why It’s Needed***:
    - Spark’s native SQL engine doesn’t support Delta-specific commands or features.
    - This extension enhances SQL to support:
      * Time travel queries: Query data at a specific version or timestamp.
      * ACID transactions for data integrity.
      * Schema enforcement and evolution during write operations.
  + **Key Features**:
    - Delta-specific SQL commands:
      * Example*: SELECT \* FROM delta./path/to/delta-table VERSION AS OF 0.*
      * Example: MERGE INTO, UPDATE, DELETE.
* **--conf "*spark.sql.catalog.spark\_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"*** 
  + ***Purpose***:
    - Configures the Spark session to use ***DeltaCatalog*** for table management.
  + ***Details***:
    - ***spark\_catalog***: Spark's default catalog for table metadata management.
    - ***DeltaCatalog***: A specialized catalog that manages Delta tables.
  + ***Why It’s Needed***:
    - Spark’s default catalog doesn’t understand Delta table metadata or advanced features like ACID transactions and versioning.
    - Replacing the default catalog ensures Delta tables are handled correctly.
  + **Key Features**:
    - Seamless creation and management of Delta tables.
    - Metadata management for versioning and schema evolution.

### Config Testing Jupyter Notebook

To test the above spark session and integrations, use jupyter notebook present in below location

**Spark Notebook** - ***/spark-working-folder/spark\_apps/1-delta-lake-config.ipynb***

## Spark, Delta Lake and Minio Integration

### Services:

* Spark Master
* Spark Worker
* Jupyter Lab
* Minio – Minio is the S3 Compatible Object Storage

### Installation and Configuration

* **Installation and Config Folder** ***- /install-and-config/*** ***2-spark-delta-minio-integration***

Docker is used to setup the spark, jupyter and Minio services. These files need to be accessed in the following order for better understanding–

* + *Dockerfile.spark*
  + *Dockerfile.minio*
  + *requirement.txt and spark-config*
  + *entrypoint.sh*
  + *entrypoint-minio.sh*
  + *docker-compose.yml*

***Note****: Spark Dockerfile, entrypoint, requirement and Spark Config files are same/ similar to the details mentioned in “Spark and Delta Lake integration” Section*

Below snippets shows details of some important scripts present in Minio docker and its entrypoint.

1. ***Dockerfile.minio***

A screenshot of a computer program

Description automatically generated

1. ***entrypoint.sh***

In this entrypoint file, based on the parameter following scripts will be executed –

1. Parameter = ***master***

* Start spark master by executing spark master script.
* Start Jupyter lab server. Token for the jupyter lab can be found by executing below script in spark master container.

|  |
| --- |
| ***jupyter server list*** |

1. Parameter = ***worker***

* Start spark worker by executing spark worker script.

1. Parameter = ***history***

* Start spark history server by executing spark history server script.

***Note****: Script to start Spark worker and history server is also controlled using this entrypoint file. Below snippet shows only spark master but file also container spark worker and history server.*

A screen shot of a computer

Description automatically generated

1. ***entrypoint-minio.sh***

A screenshot of a computer

Description automatically generated

1. ***docker-compose.yml***

***Note****: Below snippet shows only minio section from docker-compose file but it also container other services like Spark Master, Worker and History server*

A close-up of a white background

Description automatically generated

### Spark Session

Spark Session can be created using following methods –

1. Jupyter Notebook.
2. Executing Pyspark command with necessary configs (as declared in entrypoint file).

In this integration, Spark session is initialized in the jupyter notebook itself. Notebook is present at ***/spark-working-folder/spark\_apps/2-delta-lake-minio-config.ipynb.***

A computer screen shot of a computer code

Description automatically generated

***Note****: Delta configurations are same to the details mentioned in Spark session of “Spark and Delta Lake integration” Section.*

***Configuration for Minio***

* **spark.jars.packages**
  + ***Purpose***:

Adds Hadoop S3 and Delta Lake library (jars) to Spark session.

* + ***Details***:
    - ***org.apache.hadoop:hadoop-aws:3.3.4****:* Adds AWS S3 support via Hadoop's s3a connector.
  + ***Why It’s Needed***:
    - Spark doesn’t natively support **S3 storage**. These dependencies enable that functionality.
    - The hadoop-aws library is necessary for Spark to interact with **MinIO (S3-compatible storage)**.
  + ***Key Features:***
    - Enable storage interaction with S3/MinIO.
* **spark.hadoop.fs.s3a.endpoint="http://minio:9000"** 
  + ***Purpose***:

Specifies the S3-compatible endpoint for object storage (MinIO in this case).

* + ***Details***:
    - The s3a endpoint allows Spark to interact with MinIO, which acts as an S3-compatible storage system.
  + ***Why It’s Needed***:
    - Spark needs an endpoint to connect to MinIO.
  + ***Key Features:***
    - Read/write data from MinIO using ***s3a://*** paths.
    - Enables MinIO as a data lake storage backend for Delta Lake.
* **spark.hadoop.fs.s3a.access.key="root"** 
  + ***Purpose***:

Defines the *access key* for authenticating Spark to MinIO

* + ***Details***:
    - Acts like an AWS *Access Key ID*, but for MinIO.
    - *root* is a user credential for MinIO.
  + ***Why It’s Needed***:
    - MinIO requires authentication to allow read/write access.
  + ***Key Features:***
    - Grant access to MinIO storage from Spark.
* **spark.hadoop.fs.s3a.secret.key="jerinminioserver"** 
  + ***Purpose***:
    - Defines the *secret key* for authenticating Spark to MinIO.
  + ***Details***:
    - Works like an AWS *Secret Access Key* for Minio.
    - *jerinminioserver* is the MinIO user password.
  + ***Why It’s Needed***:
    - MinIO requires authentication to allow read/write access.
  + ***Key Features:***
    - Grant access to MinIO storage from Spark.
* **spark.hadoop.fs.s3a.path.style.access="true"** 
  + ***Purpose***:
    - Enables *path-style access* for MinIO instead of *virtual-hosted access*.
  + ***Details***:
    - *true*: Uses path-style (http://minio:9000/*bucket-name*).
    - *false*: Uses virtual-hosted style (http://*bucket-name*.minio:9000).
  + ***Why It’s Needed***:
    - MinIO uses path-style URLs by default.
    - Some S3-compatible systems (including MinIO) don’t support virtual-hosted access.
  + ***Key Features:***
    - Ensures Spark correctly formats MinIO storage URLs.
* **spark.hadoop.fs.s3a.impl="org.apache.hadoop.fs.s3a.S3AFileSystem"** 
  + ***Purpose***:
    - Specifies the *S3A implementation* for Hadoop’s file system.
  + ***Details***:
    - *org.apache.hadoop.fs.s3a.S3AFileSystem:* The Hadoop class that enables Spark to communicate with S3-like storage.
  + ***Why It’s Needed***:
    - Without this, Spark wouldn’t recognize *s3a://* paths.
  + ***Key Features:***
    - Enables MinIO/S3 storage support in Spark.
* **spark.sql.warehouse.dir="s3a://delta-bucket/delta-lake/data"** 
  + ***Purpose***:
    - Defines the *default location* for Spark-managed tables.
  + ***Details***:
    - Specifies that Spark’s warehouse directory is stored in MinIO (s3a://delta-bucket/delta-lake/data).
  + ***Why It’s Needed***:
    - Ensures that Spark stores all tables in MinIO rather than the local file system.
  + ***Key Features:***
    - Allows Delta Lake tables to be stored in MinIO.
    - Ensures Spark persists table metadata in a central storage.

### Config Testing Jupyter Notebook

To test the above spark session and integrations, use jupyter notebook present in below location

**Spark Notebook** - ***/spark-working-folder/spark\_apps/2-delta-lake-minio-config.ipynb***

## Spark, Delta Lake and Unity Catalog Integration <WIP>

### Services:

* Spark Master
* Spark Worker
* Jupyter Lab
* Unity Catalog

### Installation and Configuration

* **Installation and Config Folder** ***- /install-and-config/3-spark-delta-unity-integration***

Docker is used to setup the spark, jupyter and Uniity Catalog services. These files need to be accessed in the following order for better understanding–

* + *Dockerfile.spark*
  + *Dockerfile.unitycatalog*
  + *requirement.txt and spark-config*
  + *entrypoint.sh*
  + *entrypoint-unitycatalog.sh*
  + *docker-compose.yml*

***Note****: Spark Dockerfile, entrypoint, requirement and Spark Config files are same/ similar to the details mentioned in “Spark and Delta Lake integration” Section*

Below snippets shows details of some important scripts present in Unity Catalog docker and its entrypoint.

1. ***Dockerfile.unitycatalog***
2. ***entrypoint-unitycatalog.sh***

1. ***docker-compose.yml***

***Note****: Below snippet shows only unity catalog section from docker-compose file but it also container other services like Spark Master, Worker and History server*

### Spark Session

Spark Session can be created using following methods –

1. Jupyter Notebook.
2. Executing Pyspark command with necessary configs (as declared in entrypoint file).

In this integration, Spark session is initialized in the jupyter notebook itself. Notebook is present at ***/spark-working-folder/spark\_apps/3-unity-calatog-config.ipynb.***

***Note****: Spark and Delta configurations are same to the details mentioned in Spark session of “Spark and Delta Lake integration” Section.*

***Configuration for Unity Catalog***

### Config Testing Jupyter Notebook

To test the above spark session and integrations, use jupyter notebook present in below location

**Spark Notebook** - ***/spark-working-folder/spark\_apps/3-unity-calatog-config.ipynb***