Spark integration

Integration of Spark with different metastores / catalogs and lakehouses

JERIN SAM

Table of Contents

[Spark, Lakehouse, Catalog and Object Storage 2](#_Toc189238438)

[Spark 2](#_Toc189238439)

[Key features 2](#_Toc189238440)

[Lakehouse 2](#_Toc189238441)

[Key features 2](#_Toc189238442)

[Catalog 3](#_Toc189238443)

[Key features 3](#_Toc189238444)

[Object Storage 3](#_Toc189238445)

[Key features 4](#_Toc189238446)

[Integration Architecture 5](#_Toc189238447)

[Wite Operations 5](#_Toc189238448)

[Read Operations 5](#_Toc189238449)

[Integrations 5](#_Toc189238450)

[Spark and Delta Lake Integration 5](#_Toc189238451)

[Services: 5](#_Toc189238452)

[Installation and Configuration 5](#_Toc189238453)

[Spark Session 9](#_Toc189238454)

[Config Testing Jupyter Notebook 11](#_Toc189238455)

[Spark, Delta Lake and Minio Integration 11](#_Toc189238456)

[Services: 11](#_Toc189238457)

[Installation and Configuration 11](#_Toc189238458)

[Spark Session 14](#_Toc189238459)

[Config Testing Jupyter Notebook 16](#_Toc189238460)

[Spark, Delta Lake and Unity Catalog Integration 17](#_Toc189238461)

[Services: 17](#_Toc189238462)

[Installation and Configuration 17](#_Toc189238463)

[Spark Session 21](#_Toc189238464)

[Config Testing Jupyter Notebook 24](#_Toc189238465)

[Spark, Delta Lake, Unity Catalog and Minio Integration 24](#_Toc189238466)

[Services: 24](#_Toc189238467)

[Installation and Configuration 24](#_Toc189238468)

[Integration Issue 27](#_Toc189238469)

[Config Testing Jupyter Notebook 29](#_Toc189238470)

[Spark, Iceberg and Minio Integration 30](#_Toc189238471)

[Services: 30](#_Toc189238472)

[Installation and Configuration 30](#_Toc189238473)

[Spark Session 30](#_Toc189238474)

[Config Testing Jupyter Notebook 35](#_Toc189238475)

# Spark, Lakehouse, Catalog and Object Storage

## 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

## Object Storage

***Object storage*** is a data storage architecture that manages data as objects, rather than in a hierarchical file system or as blocks within sectors.

Each object consists of:

* **Data** – The actual content being stored (e.g., images, videos, documents).
* **Metadata** – Descriptive information about the object (e.g., size, creation date, permissions).
* **Unique Identifier** – A globally unique ID that enables retrieval without a traditional file system.

Object storage is commonly used in cloud environments and is designed for scalability, durability, and accessibility over the internet.

|  |  |  |  |
| --- | --- | --- | --- |
| **Storage Type** | **How Data is Stored** | **How Data is Retrieved** | **Use Case** |
| **Object Storage** | Data is stored as self-contained **objects** (data + metadata + unique identifier) in a **flat namespace**. No folders or directories. | Objects are accessed using a unique identifier (e.g., via REST API). | Cloud storage, backups, big data, AI/ML, and web applications. |
| **File Storage** | Data is stored in a **hierarchical file system** with folders and directories (like in Windows or Linux file systems). | Files are accessed using paths (/folder1/folder2/file.txt). | Shared storage for applications, document management, and network file shares. |
| **Block Storage** | Data is broken into fixed-size **blocks** and stored on disks (like traditional hard drives or SSDs). | The operating system manages blocks and uses them to build file systems. | Databases, high-performance applications, and virtual machines. |

***Note:*** *MinIO supports hierarchical naming, but it does not use a hierarchical file system like traditional file storage*

### Key features

* **Scalability**
  + Can scale horizontally to store petabytes or exabytes of data across multiple distributed systems.
  + Designed for unstructured data like images, videos, and logs.
* **Durability and Redundancy**
  + Data is replicated across multiple locations (e.g., different data centers) to prevent data loss.
  + Uses erasure coding and replication for fault tolerance.
* **Metadata-Rich Storage**
  + Allows customizable metadata (key-value pairs) that can be used for indexing, searching, and managing objects.
  + Enhances data categorization and retrieval.
* **Global Accessibility**
  + Accessible via APIs (e.g., Amazon S3 API, Azure Blob Storage API, Google Cloud Storage API).
  + Supports HTTP/S-based access, making it ideal for cloud applications.
* **Immutable Data Storage**
  + Objects are immutable (cannot be modified after creation).
  + New versions can be created to support versioning and data retention policies.
* **Security and Access Control**
  + Provides encryption (both at rest and in transit).
  + Integrates with Identity and Access Management (IAM) for user authentication and access control.
* **Cost-Effective**
  + Uses tiered storage (e.g., hot, warm, and cold storage) to optimize costs based on access frequency.
  + No need for complex storage management as in block or file storage.
* **Easy Integration with Data Processing Tools**
  + Works well with Big Data (e.g., Hadoop, Spark), AI/ML workloads, and Data Lakes.
  + Can be used with event-driven architectures (e.g., serverless computing, IoT pipelines).

**Open-Source Object Store services are –**

* **MinIO** – Self-hosted S3-compatible object storage
* **Ceph** – Open-source distributed storage

# 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*

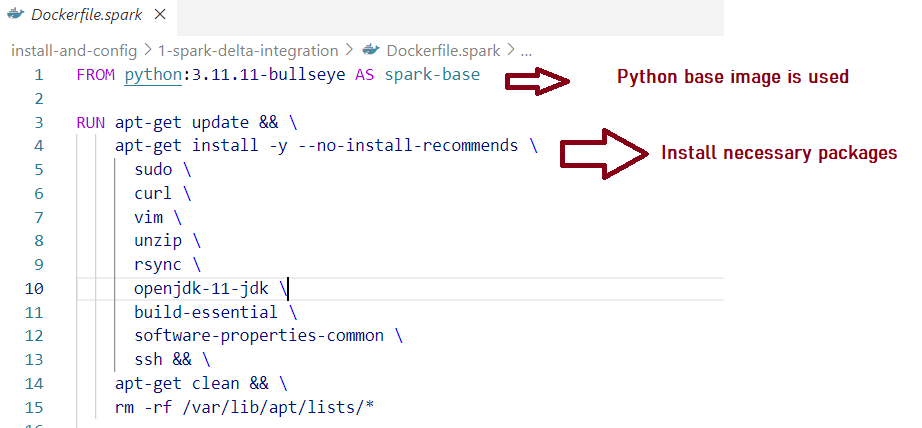
Below snippets shows details of some important scripts present in docker, entrypoint or notebook –

* ***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

***spark-defaults.conf – In above snippet***

Spark defaults file is updated with required details, when spark session is created then it will be used by default.

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

In docker container volume section, all the logs generated from entrypoint scripts, data and Spark notebooks are synced between Container and Host Machine.

***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 present in entrypoint.sh file–

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")**

1. **appName("jupyter-pyspark-session") 🡪** Names the Spark application
2. **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

In docker container volume section, all the logs generated from entrypoint-minio scripts, data and Spark notebooks are synced between Container and Host Machine.

### 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

### Services:

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

### Installation and Configuration

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

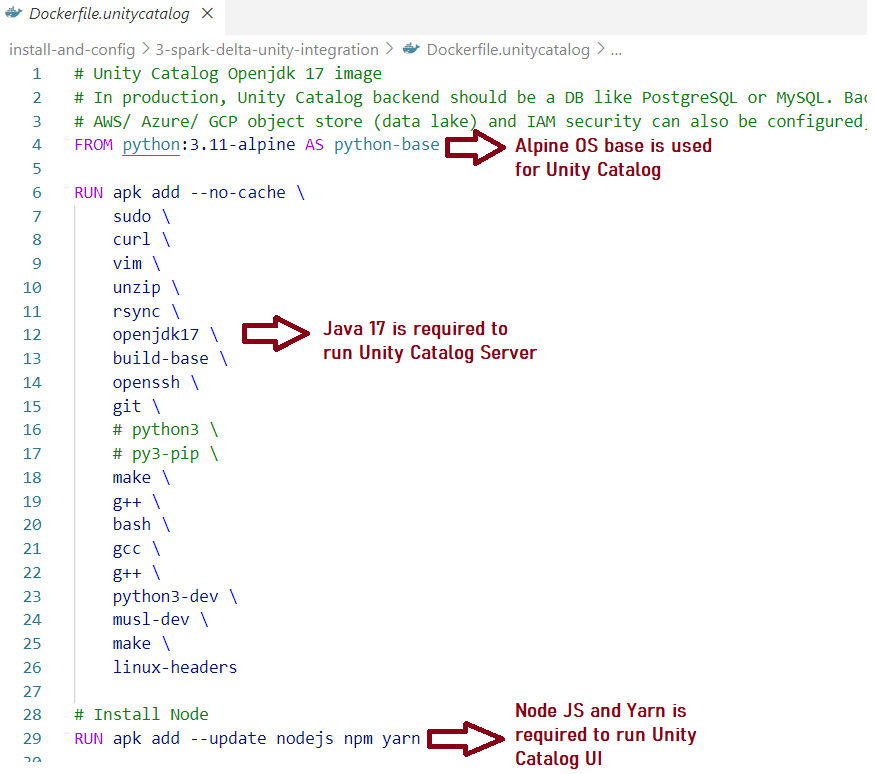
Docker is used to setup the spark, jupyter and Uniity Catalog services. These files need to be accessed in the following order for better understanding–

* + *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***



A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

In production, Unity Catalog backend should be a DB like PostgreSQL or MySQL. Backend database can be configured using ***hibernate.properties*** file present at ***/unitycatalog/etc/config*** folder.

AWS/ Azure/ GCP object store (data lake) and IAM security can also be configured, which can be done using ***server.properties*** file present at ***/unitycatalog/etc/config*** folder.

1. ***entrypoint-unitycatalog.sh***

A screenshot of a computer

Description automatically generated

A close-up of a computer code

Description automatically generated

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 which is same as mentioned in “Spark and Delta Integration” section*

A screenshot of a computer

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, 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 and Unity Catalog

Snippet of Spark Session Script present in entrypoint.sh file –

A close-up of a computer code

Description automatically generated

1. ***-- packages***

* **Purpose**
  + Includes the required libraries for Delta Lake and Unity Catalog.
* **Details**
  + "***io.delta:delta-spark\_2.12:3.2.1***" → Adds Delta Lake support for Spark (Scala 2.12).
  + "***io.unitycatalog:unitycatalog-spark\_2.12:0.2.0***" → Enables Unity Catalog support in Spark.
* **Why It’s Needed**
  + Delta Lake provides ACID transactions and time travel for tables.
  + Unity Catalog helps centrally manage permissions, metadata, and governance.
* **Key Features It Enables**
  + Delta table support (MERGE, DELETE, UPDATE, time travel).
  + Unity Catalog for data governance and access control.
  + Cross-workspace table access in Databricks (if used in Databricks).

1. ***--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"***

* **Purpose**
  + Extends Spark SQL to support Delta Lake commands.
* **Details**
  + Enable Delta-specific SQL functions in Spark.
* **Why It’s Needed**
  + Spark doesn’t natively support Delta tables without this extension.
* **Key Features It Enables**
  + MERGE INTO, DELETE, UPDATE in SQL.
  + Schema enforcement and evolution.
  + Transaction log handling.

1. ***--conf "spark.sql.catalog.spark\_catalog=io.unitycatalog.spark.UCSingleCatalog"***

* **Purpose**
  + Configures Spark's default catalog to use Unity Catalog instead of the default Spark catalog.
* **Details**
  + Uses UCSingleCatalog, which is Unity Catalog’s default catalog implementation.
* **Why It’s Needed**
  + Unity Catalog requires a specific catalog implementation to function.
* **Key Features It Enables**
  + Default catalog becomes Unity Catalog.
  + Allows managing permissions and governance at the catalog level.

1. ***--conf "spark.sql.catalog.unity=io.unitycatalog.spark.UCSingleCatalog"***

* **Purpose**
  + Defines/ Creates a named catalog (unity) that uses Unity Catalog.
* **Details**
  + "unity" is an alias for the Unity Catalog backend.
* **Why It’s Needed**
  + Allows explicit Unity Catalog references (SELECT \* FROM unity.schema.table).
* **Key Features It Enables**
  + Multiple catalog support (e.g., default Spark catalog + Unity Catalog).
  + Enables federated governance across catalogs.

1. ***--conf "spark.sql.catalog.unity.uri=http://unity-catalog:8080"***

* **Purpose**
  + Specifies the Unity Catalog service URL for governance and metadata retrieval.
* **Details**
  + "http://unity-catalog:8080" is the Unity Catalog API endpoint.
* **Why It’s Needed**
  + Unity Catalog runs as a separate service, and Spark needs to communicate with it.
* **Key Features It Enables**
  + Connects Spark to the Unity Catalog governance service.
  + Allows remote metadata management and access control.

1. ***--conf "spark.sql.catalog.unity.token="***

* **Purpose**
  + Defines an authentication token to connect to Unity Catalog.
* **Details**
  + This is empty (""), meaning authentication is either disabled or expected to be passed manually.
* **Why It’s Needed**
  + Unity Catalog requires authentication to enforce permissions.
* **Key Features It Enables**
  + Enables secure authentication for Unity Catalog.
  + Allows fine-grained access control at the table level.

1. ***--conf "spark.sql.defaultCatalog=unity"***

* **Purpose**
  + Sets Unity Catalog as the default catalog for SQL queries.
* **Details**
  + Without this, Spark defaults to spark\_catalog.
* **Why It’s Needed**
  + Ensures all queries use Unity Catalog tables by default.
* **Key Features It Enables**
  + Users don’t need to prefix queries with unity (SELECT \* FROM schema.table instead of SELECT \* FROM unity.schema.table).
  + Makes Unity Catalog the primary catalog for governance.

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

### 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***

## Spark, Delta Lake, Unity Catalog and Minio Integration

### Services:

* Spark Master
* Spark Worker
* Jupyter Lab
* Unity Catalog
* Minio – S3 Compatible Object Store

### Installation and Configuration

* **Installation and Config Folder** ***- /install-and-config/*** ***4-issue-spark-delta-unity-minio-integration***

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

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

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

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

1. ***spark-defaults.conf***

Spark defaults file is updated with Minio details, when spark session is created then it will be used by default.

A screenshot of a computer

Description automatically generated

1. ***Dockerfile.unitycatalog***

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. ***entrypoint-unitycatalog.sh***

**Note**: *Unlike in "Spark, Delta Lake, and Unity Catalog integration," where PySpark is used to create a default Spark session and start Jupyter Lab, in this integration, only Jupyter Lab is started, and the Spark session is explicitly created in a Jupyter Notebook*

A screen shot of a computer

Description automatically generated

1. ***Additional Env Variables in Dockerfile.unitycatalog and Dockerfile.spark***

The *AWS\_REGION, AWS\_ACCESS\_KEY\_ID, and AWS\_SECRET\_ACCESS\_KEY* environment variables are utilized by AWS bundle JARs to access MinIO. These variables must be set in the container from which Spark scripts are executed and submitted for processing.

In this setup, Spark Master, Workers, and Jupyter Lab are responsible for executing Spark scripts. The container includes all necessary JARs, including Unity Catalog and AWS S3 SDK JARs.

Since MinIO is AWS S3-compatible and accessed using AWS S3 SDK JARs, the corresponding AWS S3 environment variables are used. Unity Catalog interacts with MinIO through the AWS S3 bundle JAR, which retrieves region and credential information from the environment variables within the Docker container.

A close-up of a person holding an object

Description automatically generated

### Integration Issue

In Unity Catalog to use object store like AWS/ Azure/ GCP, Unity Catalog’s ***server.properties*** file present at ***/unitycatalog/etc/config*** has to be configured.

**Snippet of *server.properties* file**

A screenshot of a computer program

Description automatically generated

Since Minio is AWS S3 compatible storage, so AWS parameters can be used, however Unity Catalog does not provide any parameter for AWS S3 endpoint, it uses default endpoints to communicate with AWS storage therefore Minio endpoint can not be added in the “server.properties” file.

A screenshot of a computer program

Description automatically generated

***Spark Session used to test the integration –***

Following spark session is used to test the integration.

***Spark Notebook Location*** - */spark-working-folder/spark\_apps/4-issue-unity-calatog-and-minio-config.ipynb*



If any table is created with S3 path details then Table gets successfully created in Minio and gets registered in Unity Catalog, however the storage location is incorrectly registered with local file system directory name as prefix in minio S3 location.

A screenshot of a computer code

Description automatically generated

While inserting the data Spark throws “***table not found***” error due to incorrect storage location present in Unity Catalog.

A screenshot of a computer

Description automatically generated

If any table is created without specifying the table location, then Unity Catalog will try to use default location mentioned in the Spark Configuration, and since the location is not accessible by Unity Catalog the create statement will fail with “” error

A screenshot of a computer

Description automatically generated

### 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/4-issue-unity-calatog-and-minio-config.ipynb***

## Spark, Iceberg 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/6-spark-iceberg-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 and Minio’s Dockerfile, entrypoint, requirement and Spark Config files are same/ similar to the details mentioned in “Spark, Delta Lake and Minio integration” Section*

### 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/6-iceberg-minio-config.ipynb***

A close-up of a computer screen

Description automatically generated

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

***Iceberg Configuration in Spark Session –***

1. ***spark.jars.packages***

* **Purpose**
  + Loads the necessary JAR dependencies for Iceberg, Hive, and AWS S3 (MinIO).
* **Details**
  + ***"org.apache.iceberg:iceberg-spark-runtime-3.5\_2.12:1.7.1"*** 🡪 Iceberg support for Spark 3.5 (Scala 2.12).
  + ***"org.apache.hive:hive-exec:3.1.3"*** 🡪 Hive execution engine for metadata management.
  + ***"org.apache.hadoop:hadoop-aws:3.3.4"*** 🡪 Hadoop AWS connector for S3/MinIO access.
* **Why It’s Needed**
  + Iceberg requires a runtime library to function in Spark.
  + Hive execution is needed for metadata and schema management.
  + Hadoop AWS library allows Spark to interact with S3-compatible storage.
* **Key Features It Enables**
  + Iceberg table format support in Spark.
  + Hive Metastore integration for schema evolution.
  + S3/MinIO connectivity for storing Iceberg data.

1. ***spark.sql.extensions="org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"***

* **Purpose**
  + Enables Iceberg-specific SQL commands in Spark.
* **Details**
  + Extends Spark with Iceberg SQL capabilities.
* **Why It’s Needed**
  + Spark does not natively support Iceberg table commands like MERGE, REPLACE, and ROLLBACK.
* **Key Features It Enables**
  + MERGE INTO, COPY ON WRITE, and TIME TRAVEL.
  + Iceberg-specific optimizations like hidden partitioning.

1. ***spark.sql.catalog.spark\_catalog="org.apache.iceberg.spark.SparkSessionCatalog”***

* **Purpose**
  + Defines Iceberg as the default catalog in Spark.
* **Details**
  + SparkSessionCatalog makes Iceberg the primary catalog for table operations.
* **Why It’s Needed**
  + Without this, Spark will use its default catalog, which does not support Iceberg.
* **Key Features It Enables**
  + Allows Spark to directly interact with Iceberg tables.
  + Enables querying Iceberg tables without specifying a custom catalog.

1. ***spark.sql.catalog.local = "org.apache.iceberg.spark.SparkCatalog"***

* **Purpose**
  + Defines an alternative catalog (local) that also uses Iceberg.
* **Details**
  + local is an additional catalog that can store tables in Hadoop/S3.
* **Why It’s Needed**
  + Allows users to store tables in different locations (e.g., MinIO instead of Hive Metastore).
* **Key Features It Enables**
  + Supports multiple catalogs (e.g., spark\_catalog and local).
  + Enables custom storage options for different datasets.

1. ***spark.sql.catalog.spark\_catalog.type = "hive"***

* **Purpose**
  + Configures the default Iceberg catalog to use Hive Metastore.
* **Details**
  + Uses Hive Metastore to store table metadata.
* **Why It’s Needed**
  + Hive Metastore allows centralized metadata management for Iceberg tables.
* **Key Features It Enables**
  + Iceberg metadata persistence.
  + Enables cross-session table access.

1. ***spark.sql.catalog.local.type = "hadoop"***

* **Purpose**
  + Configures the local catalog to store metadata directly in a Hadoop-compatible storage (MinIO/S3).
* **Details**
  + Instead of using Hive Metastore, this catalog stores metadata in S3.
* **Why It’s Needed**
  + Useful for setups without Hive Metastore.
* **Key Features It Enables**
  + Iceberg metadata storage in S3 (MinIO).
  + Simplifies deployment by removing the need for Hive Metastore.

1. ***MinIO (S3) Configurations***

These settings allow Spark to read and write data to MinIO (an S3-compatible object store).

* **Purpose**
  + Configures Spark to interact with MinIO via Hadoop AWS connectors.
* **Details**
  + ***spark.hadoop.fs.s3a.endpoint = "http://minio:9000"*** 🡪 MinIO service URL.
  + ***spark.hadoop.fs.s3a.access.key = "root"*** 🡪 MinIO access key.
  + ***spark.hadoop.fs.s3a.secret.key = "jerinminioserver"*** 🡪MinIO secret key.
  + ***spark.hadoop.fs.s3a.path.style.access = "true"*** 🡪 Enables path-style URLs.
  + ***spark.hadoop.fs.s3a.impl = "org.apache.hadoop.fs.s3a.S3AFileSystem"*** 🡪 Uses Hadoop’s S3A connector.
* **Why It’s Needed**
  + Spark needs authentication and endpoint details to interact with MinIO/S3.
* **Key Features It Enables**
  + S3-compatible storage in Spark.

1. ***spark.sql.catalog.local.warehouse = "s3a://iceberg-bucket/delta-lake/data"***

* **Purpose**
  + Defines the Iceberg table storage location in MinIO.
* **Details**
  + ***s3a://iceberg-bucket/delta-lake/data*** 🡪 All tables under local catalog are stored here.
* **Why It’s Needed**
  + Specifies where Iceberg tables physically store their data.
* **Key Features It Enables**
  + Iceberg table data is stored in MinIO instead of HDFS.
  + Supports scalable cloud storage.

1. ***spark.sql.defaultCatalog = "local"***

* **Purpose**
  + Sets the default catalog to local instead of spark\_catalog.
* **Details**
  + All queries use local catalog by default.
* **Why It’s Needed**
  + Ensures Iceberg tables are created in the MinIO-backed catalog.
* **Key Features It Enables**
  + Users don’t need to specify local. prefix in SQL queries.

1. ***Hive Metastore Configurations***

These settings disable schema verification in Hive Metastore.

* **Purpose**
  + Avoids issues when using Hive Metastore with Iceberg.
* **Details**
  + ***spark.hadoop.datanucleus.autoCreateSchema = "true"*** 🡪 Automatically creates missing schemas.
  + ***spark.hadoop.datanucleus.fixedDatastore = "false"*** 🡪 Allows dynamic schema changes.
  + ***hive.metastore.schema.verification = "false"*** 🡪 Disables strict schema checking.
  + ***hive.metastore.schema.verification.record.version = "false"*** 🡪 Ignores version mismatches.
* **Why It’s Needed**
  + Iceberg and Hive Metastore handle schema differently, so these settings prevent conflicts.
* **Key Features It Enables**
  + Allows Iceberg tables to work without strict schema enforcement.
  + Prevents Hive Metastore errors related to schema changes.

### 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/6-iceberg-minio-config.ipynb***

## Spark, Iceberg, Nessie and Minio Integration

### Services:

* Spark Master
* Spark Worker
* Jupyter Lab
* Nessie
* Minio

### Installation and Configuration

* **Installation and Config Folder** ***- /install-and-config/7-spark-iceberg-nessie-minio-integration***

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

* + *Dockerfile.spark*
  + *Dockerfile.minio*
  + *Dockerfile.pysparkjupyter*
  + *requirement.txt and spark-config*
  + *entrypoint.sh*
  + *entrypoint-pysparkjupyter.sh*
  + *entrypoint-minio.sh*
  + *docker-compose.yml* ***– Nessie is directly created in Docker Compose file***

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

Below snippets shows details of some important scripts present in Nessie and Pyspark Jupyter docker and its entrypoint files.

1. ***Dockerfile.pysparkjupyter***

This is similar to Spark Dockerfile, it is created to have a separate docker container to run jupyter lab server.

Details for Spark Docker container can be found in “Spark and Delta Lake integration”.

A screenshot of a computer program

Description automatically generated

1. ***entrypoint-pysparkjupyter.sh***

In this entrypoint file, script to start jupyter lab server exists.

A close-up of a computer screen

Description automatically generated

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

***Note****: Below snippet shows only Nessie section from docker-compose file but it also container other services like Spark Master, Worker, History server, Minio and jupyter which is same as mentioned in “Spark and Delta Integration” , “Spark, Delta and Minio Integration” sections*

A screenshot of a computer

Description automatically generated

***Note****:*

* 1. *Starting Nessie using Dockerfile and entrypoint file was causing issue to start the Nessie server, therefore its used directly in the docker-compose file.*
  2. *It is important to add the network details in all the services (mention in above snippet) present in the docker compose file so that all server are connect through same network.*

1. ***Additional Env Variables in Dockerfile.pysparkjupyter and Dockerfile.spark***

The *AWS\_REGION, AWS\_ACCESS\_KEY\_ID, and AWS\_SECRET\_ACCESS\_KEY* environment variables are utilized by AWS bundle JARs to access MinIO. These variables must be set in the container from which Spark scripts are executed and submitted for processing.

In this setup, Spark Master, Workers, and Jupyter Lab are responsible for executing Spark scripts. The container includes all necessary JARs, including Unity Catalog and AWS S3 SDK JARs.

Since MinIO is AWS S3-compatible and accessed using AWS S3 SDK JARs, the corresponding AWS S3 environment variables are used. Unity Catalog interacts with MinIO through the AWS S3 bundle JAR, which retrieves region and credential information from the environment variables within the Docker container.

A close-up of a person holding an object

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/7-nessie-minio-config.ipynb***

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

***<Add Config Details>***

### 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/7-nessie-minio-config.ipynb***