databricks

Guide

Amazon EMR to Databricks Migration Guide

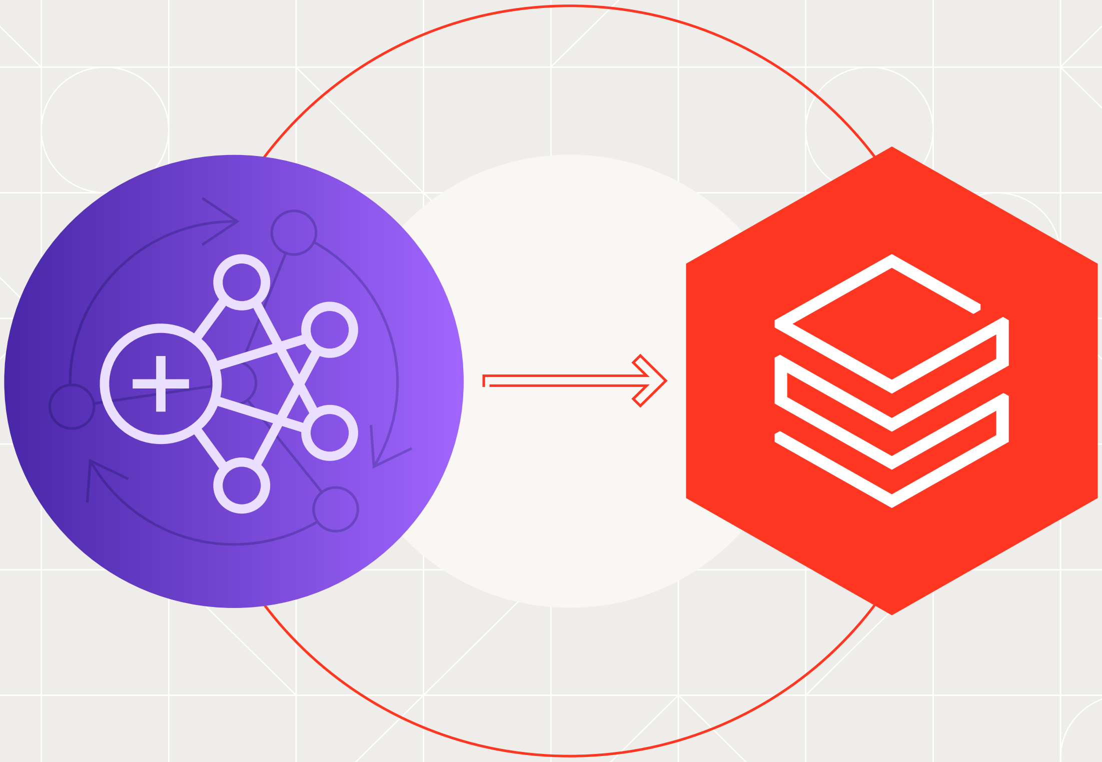


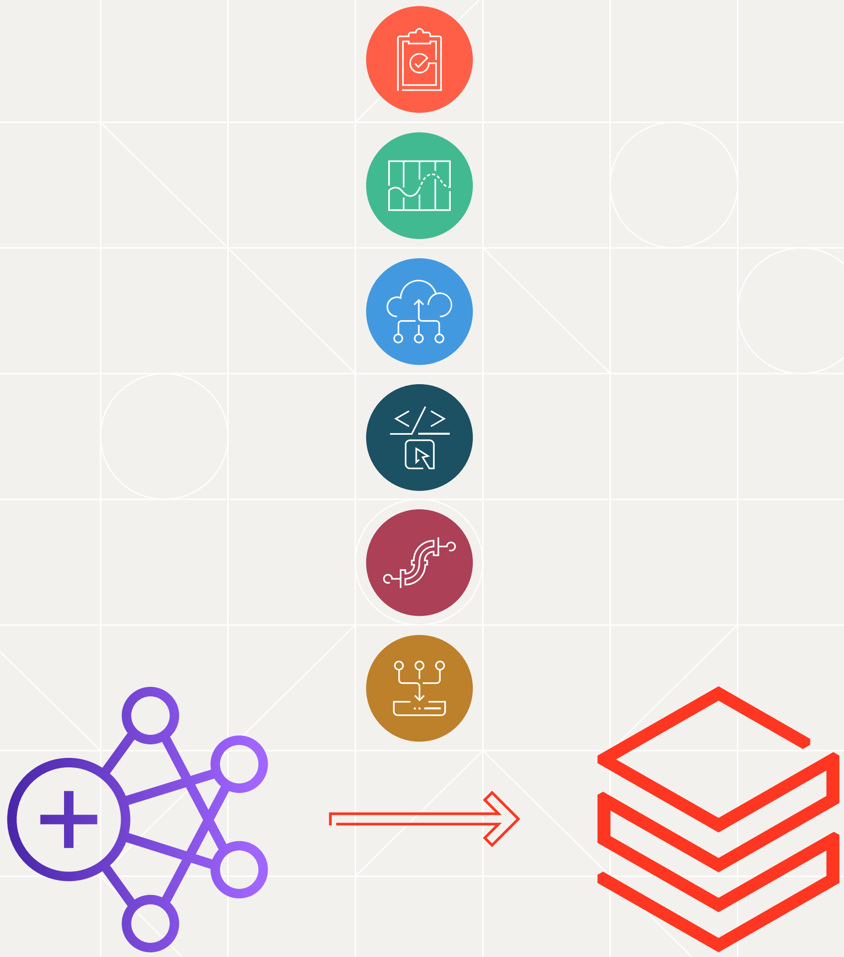
Table of Contents

Preface

The purpose of this document is to provide an overview of the process of migrating workloads from Amazon EMR to Databricks. The goal is to lay out foundational differences, common patterns in migrating data/code, best practices, tooling options, and more from Databricks' collective experience.

Building data pipelines in AWS can be difficult because it requires stitching together multiple services to create end-to-end workflows. AWS offers a wide range of services, each with its own set of features and capabilities, which can make it challenging to choose the right services and configure them to work together seamlessly. This can result in a complex and fragmented architecture that is difficult to manage and maintain.

Databricks provides a comprehensive and integrated solution for managing data pipelines, with high performance, scalability, security, collaboration and integration features that make it the best place to run your data pipelines.



Migration Strategy

When migrating from Amazon EMR to Databricks, it is crucial to plan and execute the process carefully to ensure a successful outcome. By adopting a structured approach, it is possible to minimize risks and increase the chances of success. The migration process can take different routes depending on various factors such as the:

- Current architecture state: dependency on other AWS services, use of third-party tools and open source technologies
- Workload types (batch, streaming, ETL, etc.)
- Business criticality of use cases
- Migration goals (cost reduction, cutover deadlines, user change management, etc.)
- Migration teams' skill sets

The technical execution strategy employed in the migration process is influenced by several workload-specific factors, such as:

- Workload dependency (integrated vs. isolated pipelines)
- Shared vs. isolated clusters
- Current architectural limitations
- Road map backlog and new business requirements
- Migration strategy (lift and shift, refactor and re-architect)
- Job orchestration requirements
- Access to migration tools and migration effort

Based on these factors, one can choose to undertake a bulk migration or a phased migration. A phased migration involves executing the migration in stages, considering dependencies or interconnections. We recommend adopting a phased migration approach to mitigate risks and show progress early in the process. From a high-level strategy perspective, here is the step-by-step plan:

Data ingest
Review all the existing data ingest pipelines in the EMR-based environment, understanding all the data sources and the type of ingest, as well as push vs. pull, and categorize data based on their source.

Data cataloging
If using a catalog tool, leverage it to find out about the nature and lineage of all the data sources. Do a lineage mapping to see how data from different sources are the feed. Data source dependencies are a critical portion of migration when it comes to Silver and Gold layers.

Schema definitions
Review the existing metastores and determine the schema, file formats, ingest frequency and additional metadata that may or not be published.

Orchestration/scheduled job
Understanding how EMR jobs are scheduled and orchestrated enables efficient planning, replication of workflows and optimization of resources.

Data staging and data copies
Determine how and where the intermediate stages of data are being managed. Assume there will be:

- Reading and writing multiple overlapping and/or entirely redundant data sets to and from object storage
- Copying fully redundant sets of data for the sake of access for analytics

Interactive users
Get a list of all the interactive users of EMR, their frameworks and tools they use, and the security setup for their access.

DevOp scripts
Get a list of DevOps tools for any purpose, CI/CD, ad hoc job scheduling, source control syncs and other DevOps scripts.

Pipeline flow and technologies
Get a list of the flows, tech stack, frameworks, integrations and dependencies in the pipeline. Get a prioritization as to which of these are critical to the data operations and which are open to deprecating/relegating. This validation cannot be emphasized enough, as it is most often the differentiator between lift-and-shift and a complete re-architecture.

S3 data is readily available for ingest in Databricks, and easily optimized with Delta Lake; EMR HDFS data may be eliminated altogether (as these are often redundant), and any EMR HDFS data that is purposeful will need to be migrated to Delta on S3 and redesigned to be ingested minimally and efficiently (e.g., via Kafka, AWS DMS).

Bronze, Silver and Gold
Plan building these layers based on the requirements and best practices for Delta Lake.

Schedule of migration
For each pipeline, what are the criteria for eventually turning a production pipeline into Databricks and off of EMR? For example, in some critical cases, you may want to run two concurrent pipelines and run data and process validation concurrently for a duration of time.

Security and data access control
In all likelihood, security has to be looked at from the ground up. Find out the list of all users, service accounts and admin accounts. Determine the functions and responsibilities of each role and if there are additional roles defined and how they would map to Databricks. Learn the existing AWS access policies and IAM roles. Understand the authentication and authorization requirements.

Monitoring, alerting and notifications
Understand the tools being used for monitoring and alerting (native AWS tools like CloudWatch, SNS or SQS, or third-party tools such as Datadog, PagerDuty, Slack).

Third-party tools and libraries
Get an inventory of third-party tools running and libraries installed in the EMR environment and determine if they need to migrate, and then come up with a plan using features like init scripts (bootstrapping), docker images generation, library install and generation via CI/CD pipeline.

Infrastructure configuration
Document AWS infrastructure configuration before migration. It is essential for risk mitigation, compliance, resource planning, dependency mapping, disaster recovery, effective communication and troubleshooting. It provides a solid foundation for a successful and well-executed migration.

In the next few sections, we will dive into the migration process, focusing on the approach.

Overview of the Migration Process

Typically, data and ETL migrations from legacy on-prem technologies to cloud are complex and lengthy engagements — whereas migrations from EMR are relatively easy.

Data engineering teams can reduce costs by simplifying and reducing steps. This is typically done by handling batch and streaming data sources along with schema enforcement and evolution within a single workflow.

Running Apache Spark™ workloads on the Databricks Lakehouse Platform means you benefit from Photon — a fast C++, vectorized execution engine for Spark and SQL workloads that runs behind Spark's existing programming interfaces. Photon provides query performance at low cost while leveraging AWS Graviton.

The migration process typically consists of the following steps, but can vary depending on customer situation and needs:

- Migration Discovery and Assessment
- Architecture and Feature Mapping
- Data Migration
- Code Migration
- Data Pipeline Migration
- Downstream Tools Integration

In addition to migrating technical artifacts, a common activity that spans the entire migration process is change management, which involves user enablement and adoption. This will start with creating a few champions at the beginning and scaling out to more developers and consumers. Databricks Academy is a good place to get started with some self-learning. For larger engagement, it is recommended to add Delivery Solution Architect (DSA) resources in the package. DSAs can help by providing a customized enablement and demo sessions to speed up Databricks adoption.

In general, migrating from one platform to another platform can be complex, which is why it is recommended to consider using experts, at least for the initial pipelines. The Databricks Professional Services team has experience, skills, automation and access to expert partners in helping customers reduce risks and successfully migrate from Amazon EMR to Databricks. The Databricks Brickbuilder Solution for migrations has preferred partners who have demonstrated a unique ability in migrating EMR workloads to the lakehouse successfully.

Phase 1: Migration Discovery and Assessment

Before migrating any data or workloads, one or more migration assessments should be conducted to collect the relevant use-case information. Some of the important questions are listed below.

ARCHITECTURAL DISCOVERY — EMR

IMPORTANT

First understand the business objective and SLAs

Use cases

- Services and 3rd party tools involved
- Streaming, batch or hybrid

Cluster sizing

- Number and size per day/month
- Instance types per cluster
- On-demand/spot usage
- Graviton2 usage
- EMR versions

Frameworks and applications

- Number and type (Spark, Hive, MR, HBase, etc.)
- Automated vs. interactive

Users

- Number of users
- Concurrency requirements
- Job isolation

Data

- Data sources
- Data volume and file types
- Storage options used

Orchestration

- Step
- Airflow

Downstream applications

- Integrate with applications/ processes
- Data visualization and actionable insights

Security and governance

- Authentication
- Authorization
- Metadata management

Databricks strongly recommends using automation tools such as **EMR Profiler** to expedite the process of gathering migration-related information and to better understand your current EMR setup. (To obtain a copy of the profiler, please submit a request through your Databricks representative.)

The EMR Profiler uses the EMR APIs to access and manage various logs related to existing EMR clusters. The specific logs you can access through the API include:

1 **Cluster Logs**
These provide information about the overall cluster execution, including the cluster's state changes, steps executed and job progress. The cluster logs include the following:

- **Cluster Application Logs:** Generated by applications running on the cluster, such as Hadoop, Hive, Spark, etc.
- **Cluster Bootstrap Actions Logs:** Related to the bootstrap actions executed during cluster launch
- **Cluster Instance Logs:** Specific to each instance in the cluster, such as YARN container logs and system logs
- **Cluster Step Logs:** Generated by individual steps executed on the cluster

2 **Instance State Change Logs**
These capture the state changes of the instances within the cluster. They provide information about when instances are added or terminated, and any related events.

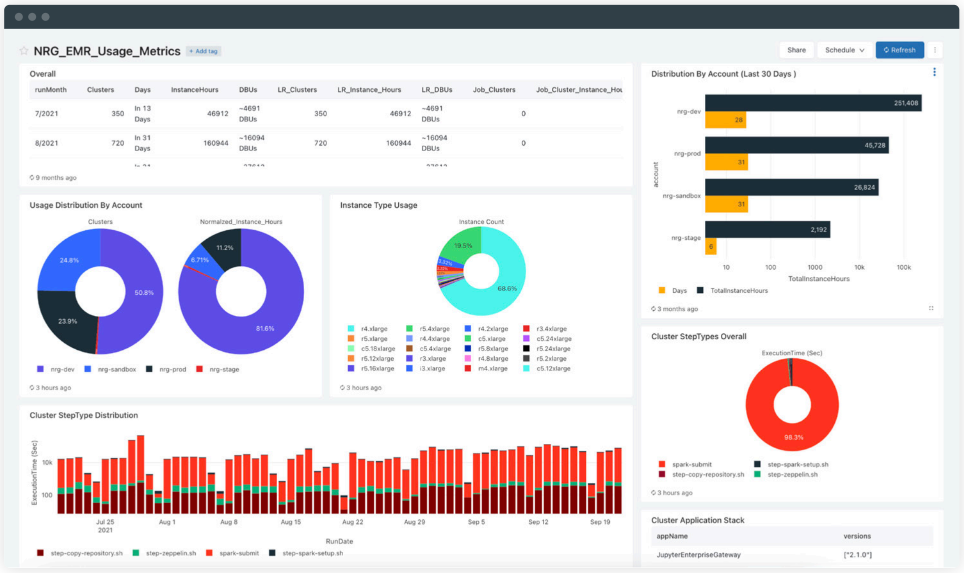
3 **Step Logs**
These contain detailed information about the execution of individual steps within the EMR cluster. They include step-level progress, input/output details and error logs.

4 **Application Logs**
Application-specific logs generated by tools and frameworks used within the EMR cluster, such as Hadoop, Hive, Spark, etc. These logs provide insights into the behavior and performance of the applications.

To access these logs using the EMR API, you can use various methods provided by the API, such as DescribeCluster, ListSteps, DescribeStep, etc. These methods allow you to retrieve detailed information about the cluster, steps and their associated logs.

It's important to note that to access the logs through the API, you need the appropriate permissions and configuration in your EMR setup. Additionally, you may need to enable logging configurations and specify the desired log locations during cluster creation or configuration.

DATABRICKS MIGRATIONS — EMR PROFILER



Overall

runMonth	Clusters	Days	InstanceHours	DBUs	LR_Clusters	LR_Instance_Hours	LR_DBUs	Job_Clusters	Job_Cluster_Instance_Hou
7/2021	350	In 13 Days	46912	~4691 DBUs	350	46912	~4691 DBUs	0	
8/2021	720	In 31 Days	160944	~16094 DBUs	720	160944	~16094 DBUs	0	

9 months ago

Cluster Application Stack

appName	versions
JupyterEnterpriseGateway	["2.1.0"]

Figure 1: Sample output from example profiler result

WORKLOAD INSIGHTS

- Monthly breakdown of overall usage (AWS instance hours)
- Usage distribution by AWS account and region
- Usage distribution by job clusters vs. long-running clusters
- Workload type (Spark, Hive, Flink, etc.) metrics
- AWS instance type metrics
- Application stack (Hive, Spark, Hue, Zeppelin, etc.) information with versions
- Usage distribution by EMR runtime versions

Phase 2: Architecture and Feature Mapping

It is important to note that Databricks clusters are fundamentally different from EMR clusters. The biggest difference is in terms of cluster managers. EMR makes use of YARN, while Databricks makes use of its own in-house cluster manager.

INTERACTIVE AND AUTOMATED JOBS

A key difference between EMR and Databricks is that on Databricks there is the concept of automated vs. interactive.

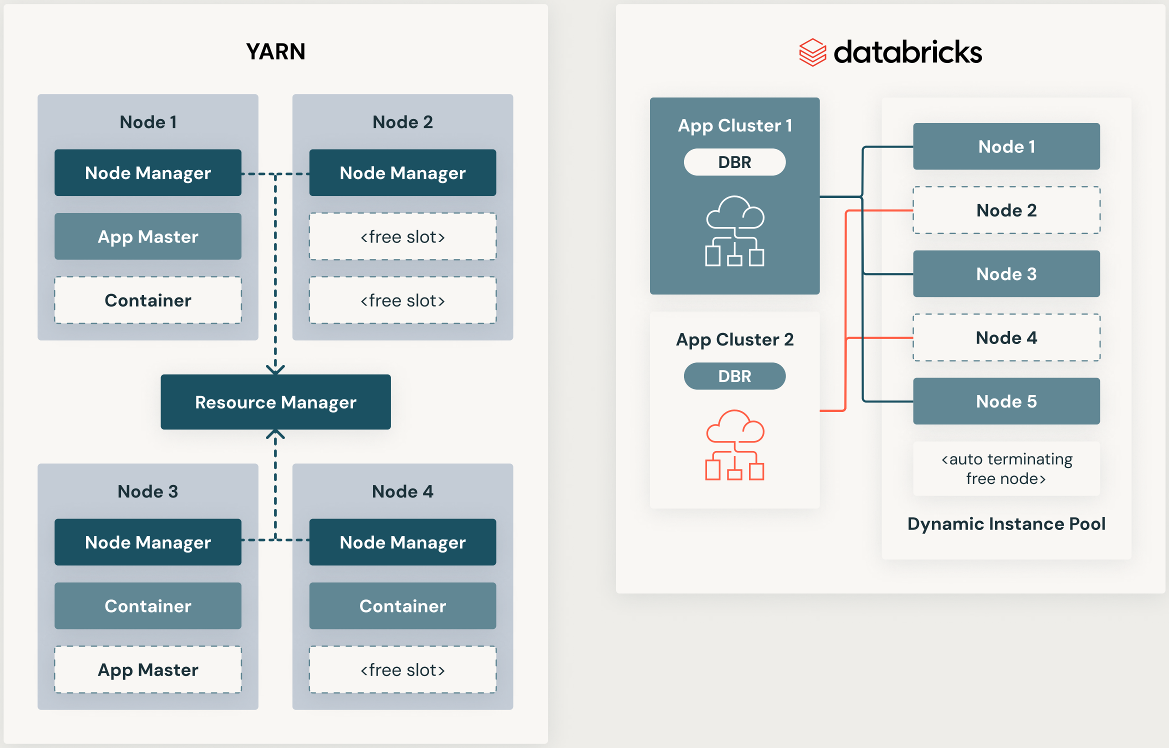
Jobs can run as a set of commands in a notebook or as an automated job. Databricks makes a distinction between all-purpose clusters and job clusters. It is recommended to use all-purpose clusters to analyze data collaboratively using interactive notebooks. You use job clusters to run fast and robust automated jobs. Job clusters are cheaper than all-purpose clusters. More details can be found on the AWS Databricks pricing web page.

CLUSTER MANAGERS

Key differences between **YARN** and **Databricks Cluster Manager**:

- YARN can have multiple driver processes on one cluster, whereas Databricks always has exactly one.
  - **Consequence 1:** In Databricks, multiple Spark jobs running on one cluster share the same driver instead of spinning up a separate driver process that would cost additional CPU/memory
  - **Consequence 2:** Because multiple Spark jobs on one Databricks cluster share one driver process, there will also only be one Spark UI and job log per cluster, which consolidates all the job information
- YARN has multiple executors per EC2 instance, whereas Databricks has just one executor per EC2 instance (but still multiple per cluster).
  - **Consequence 1:** Using one executor per EC2 instance also means that if the instance has a lot of memory, the JVM memory heap is bigger than what you are used to on YARN. This can be beneficial since every JVM has some memory overhead, so fewer JVM processes might make more efficient use of the overall EC2 memory that is available.
  - **Consequence 2:** Because CPU cores can utilize a shared memory pool, executors are more resistant (but not immune) to disk spill in case of skewed data

Because of the conceptual and practical differences between how Databricks clusters and EMR clusters are organized, it is recommended to take a slightly different approach when scheduling jobs in a Databricks environment.



AUTOSCALING CAPABILITIES

In EMR you might have configured your cluster to use automatic scaling. This allows you to scale out/in based on different CloudWatch metrics. It allows very fine-grained control, but with that comes the need for specific expertise, which not every user of a cluster might have. In Databricks, on the other hand, an out-of-the-box autoscaling algorithm can be used, which works well for most workloads. You only have to specify the minimum and maximum number of executors in the cluster, and they will automatically be added/removed based on the load. Since Databricks clusters do not use YARN or HDFS, they do not have to wait for those services to gracefully shut down. Therefore, adding and removing instances can be much more efficient and flexible.

If you want to have more fine-grained control on scaling Databricks clusters, it is also possible to use CloudWatch metrics to trigger cluster resizing events through the Databricks REST API.

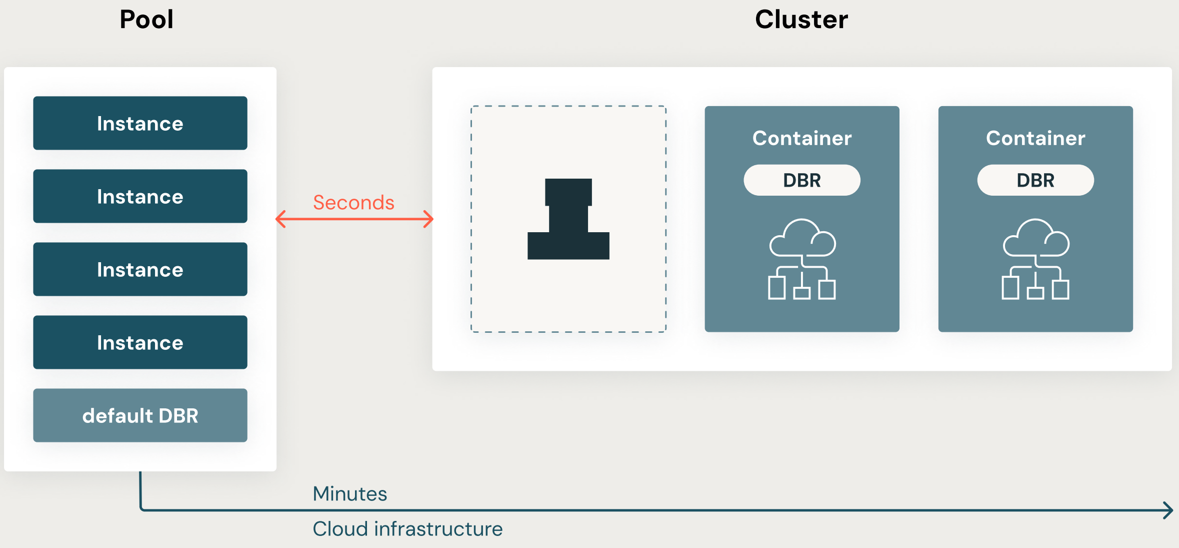
DATABRICKS POOLS

Databricks pools are a set of idle, ready-to-use instances. When cluster nodes are created using the idle instances, cluster start and autoscaling times are reduced. If the pool has no idle instances, the pool expands by allocating a new instance from the instance provider in order to accommodate the cluster's request. When a cluster releases an instance, it returns to the pool and is free for another cluster to use. Only clusters attached to a pool can use that pool's idle instances. You can specify a different pool for the driver node and worker nodes, or use the same pool for both. Databricks does not charge DBUs while instances are idle in the pool. Instance provider billing does apply.

You can manage pools using the UI or the Instance Pools CLI, or by calling the Instance Pools API.

KEEP MACHINES WARM FOR FAST JOBS, FAST AUTOSCALING AND RESOURCE OPTIMIZATION

- Only infrastructure costs (no DBUs)
- Can act as cache so no cost overnight (if no demand)
- Accelerates both interactive and automated
- Default DBR, etc., for fast start



FLEET INSTANCE TYPE

A fleet instance type is a variable instance type that automatically resolves to the best available instance type of the same size.

For example, if you select the fleet instance type `m-fleet.xlarge`, your node will resolve to whichever `.xlarge`, general-purpose instance type has the best spot capacity and price at that moment. The instance type your cluster resolves to will always have the same memory and number of cores as the fleet instance type you chose.

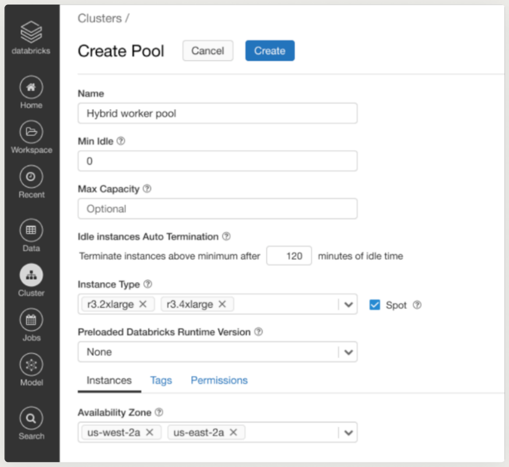
IMPROVED FUNCTIONALITY

Fleet clusters maximize availability in the following ways:

- **Improved Auto-AZ with Spot placement score API:** Fleet instance types use AWS's Spot placement score API to choose the best and most-likely-to-succeed availability zone for your cluster at startup time
- **Auto-AZ available with instance pools:** You can select auto as the availability zone for your pool if the pool uses a fleet instance type
- **High-availability (HA) zone:** Clusters with a fleet instance type for both driver and worker can have their `zone_id` set to `HA`. This allows the cluster's driver and workers to be allocated from any mix of availability zones in the region, wherever capacity is best.

AWS EC2 FLEET SUPPORT

Maximize the cost savings with higher availablility of Spot instances



Name: Hybrid worker pool
Min Idle: 0
Max Capacity: Optional
Idle instances Auto Termination
Terminate instances above minimum after 120 minutes of idle time
Instance Type: r3.2xlarge, r3.4xlarge [✓ Spot]
Preloaded Databricks Runtime Version: None
Availability Zone: us-west-2a, us-east-2a

Before	After
Availability Zone Single AZ	**Availability Zone** Mix and match AZs to get the lowest spot price with higher availability
Instance Family Single instance family	**Instance Family** Mix and match instances for more flexibility on cost vs. availability
Spot Kill Fall Back Immediately fall back to on-demand	**Spot Kill Fall Back** Fetch another spot of a different AZ/instance type for cost saving

CONFIGURING STORAGE AND PERMISSIONS

Let's talk about the differences in interaction with S3 and cluster storage between EMR and Databricks. Specifically, we delve into how intermediate results in consecutive Spark jobs are handled, and how EMR and Databricks deal with the caveats in the S3 consistency model, and finally we discuss how IAM instance profiles and credential pass-through can be used to control access to data objects in S3.

STORAGE

A typical approach to running jobs on EMR is that you use S3 to store input and output data and use local disk for storing shuffle files, while using the cluster-local HDFS to store intermediate results that get reused by consecutive Spark jobs. This has the benefit of not having to keep your cluster up 24/7, and being able to shut it down when no data processing is performed. In Databricks a similar approach is used by decoupling storage from compute. The main difference here is that Databricks clusters do not run HDFS. To make repeated reads more performant, storage-optimized instances like i3 can be used. i3 instance types are Delta caching-enabled by default, and other instance types can also be configured to enable Delta caching. Using this feature, a cluster-local copy of the data will be created on read, so that whenever you (partially) reread the same data, no round trip to S3 is necessary. Note that when you have EMR jobs that make heavy use of HDFS, it is recommended to see if you can chain them together to not have to materialize intermediate results on S3. This will get you the best performance.

S3 COMMIT PROTOCOLS

There are some caveats with the S3 consistency model that both EMR and Databricks solve in their own way. By default S3 is eventually consistent on some of the operations you can perform. The most relevant one is the LIST operation. A Spark job typically writes out multiple output files per job. If you then want to read these files with another Spark job, you might or might not get all the relevant files when you LIST the corresponding S3 location. This can lead to inconsistencies in your data pipeline.

To make sure this does not occur, EMR has implemented EMRFS features like Optimized S3 committer. In Databricks the DBIO Transactional Commit protocol is used to overcome the same S3 caveats, as well as provide a boost in performance over the Hadoop commit protocols.

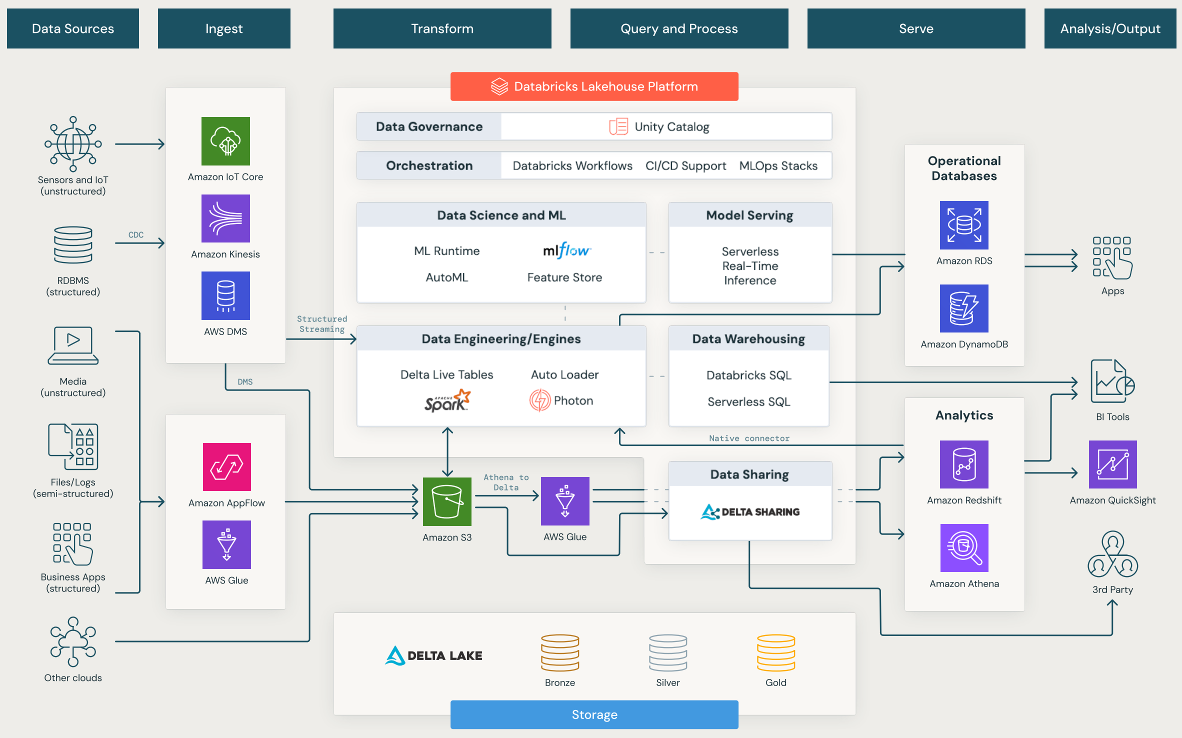
CONNECTING TO S3

On EMR the default IAM service role for the cluster EC2 instances comes with a default managed policy, AmazonElasticMapReduceforEC2Role, which has unrestricted access to S3 resources, in order to make sure setting up a cluster is as easy as possible. It is considered best practice to replace this policy with a more restricted policy in which only certain operations on certain S3 buckets are permitted.

How to connect to AWS S3 from Databricks:
It is recommend to use Unity Catalog external locations to connect to S3.

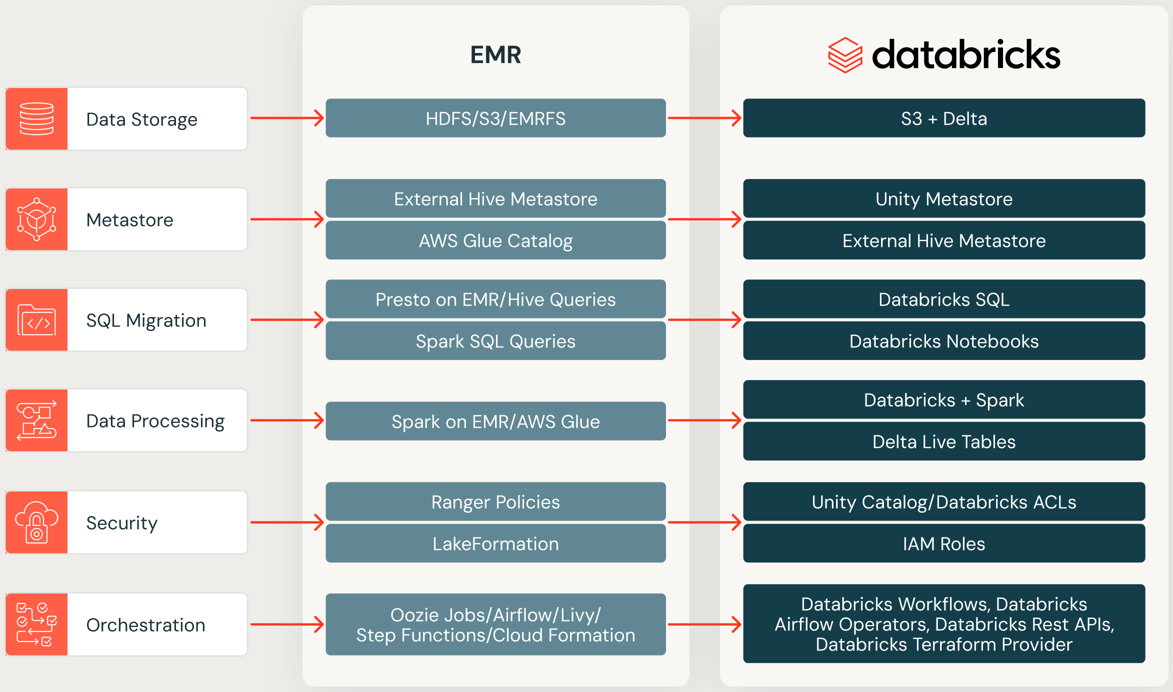
This target reference architecture represents the end state with Databricks. In the following sections, we'll elaborate on the details of each layer.

LAKEHOUSE REFERENCE ARCHITECTURE (AWS)



TECHNOLOGY MAPPING

Component mapping



	EMR	databricks
Data Storage	HDFS/S3/EMRFS	S3 + Delta
Metastore	External Hive Metastore AWS Glue Catalog	Unity Metastore External Hive Metastore
SQL Migration	Presto on EMR/Hive Queries Spark SQL Queries	Databricks SQL Databricks Notebooks
Data Processing	Spark on EMR/AWS Glue	Databricks + Spark Delta Live Tables
Security	Ranger Policies LakeFormation	Unity Catalog/Databricks ACLs IAM Roles
Orchestration	Oozie Jobs/Airflow/Livy/ Step Functions/Cloud Formation	Databricks Workflows, Databricks Airflow Operators, Databricks Rest APIs, Databricks Terraform Provider

Delta Lake

Delta Lake is an open source optimized storage layer that provides the foundation for storing data and tables by extending Parquet data files with file-based transaction logs for ACID transactions and scalable meta data handling.

Unity Catalog

Unity Catalog is a fine-grained governance solution that helps simplify security and governance of data by providing capabilities for secure and standards-compliant data models, data discovery, and built-in auditing and lineage.

Databricks SQL

Databricks SQL provides general compute resources for SQL queries, visualizations and dashboards that are executed on the tables in the lakehouse.

Databricks Notebooks

Databricks Notebooks provides a sophisticated notebook and collaborating editor environment for creating data science and machine learning workflows.

Databricks Runtime (Spark)

Databricks Runtime includes Apache Spark™ but also adds a number of components and updates that improve the usability and performance with Delta Lake, Java, Python and R libraries, Ubuntu and its system libraries, and GPU libraries for GPU enabled clusters.

Delta Live Tables (DLT)

Delta Live Tables a declarative framework for building reliable, maintainable and testable data processing pipelines. DLT manages how data is transformed based on the queries in each step, enforces data quality and provides a view of data lineage.

Databricks Workflows

Databricks Workflows is a fully managed orchestration service integrated with the Databricks platform to help run data engineering, machine learning and DLT workloads. Workflows provides a robust interface to launch job-specific ephemeral clusters that are shut down automatically at the end of the workload.

Databricks also provides rich collection Airflow Operators and REST API endpoints to launch, repair and monitor workflows and other workspace management tasks.

Photon

Photon is the next generation engine on the Databricks Lakehouse Platform that provides extremely fast query performance at low cost — from data ingestion, ETL and streaming to data science and interactive queries — directly on your data lake.

PRICE/PERFORMANCE WITH PHOTON AND GRAVITON2

The Graviton processors are custom designed and optimized by AWS to deliver the best price/performance for cloud workloads running in Amazon EC2. When used with Photon, the high-performance Databricks query engine, Graviton2-based Amazon EC2 instances can deliver up to 3x better price/performance than comparable Amazon EC2 instances for your data lakehouse workloads.

Phase 3: Data Migration

CATALOG CONFIGURATIONS

Using external metastores is a legacy data governance model. Databricks recommends that you upgrade to Unity Catalog. Unity Catalog simplifies security and governance of your data by providing a central place to administer and audit data access across multiple workspaces in your account. Unless there is a critical issue that risks process, adoption of Unity Catalog should be part of EMR migration. Learn more about Unity Catalog.

AWS EXTERNAL HIVE METASTORE INTEGRATION

You can configure Databricks Runtime to use the AWS Glue Data Catalog as its metastore. This can serve as a drop-in replacement for a Hive metastore.

To enable Glue Catalog integration, set the AWS configurations `spark.databricks.hive.metastore.glueCatalog.enabled true`. This configuration is disabled by default.

This page talks at length about configuring Glue Metastore, its limitations and troubleshooting steps.

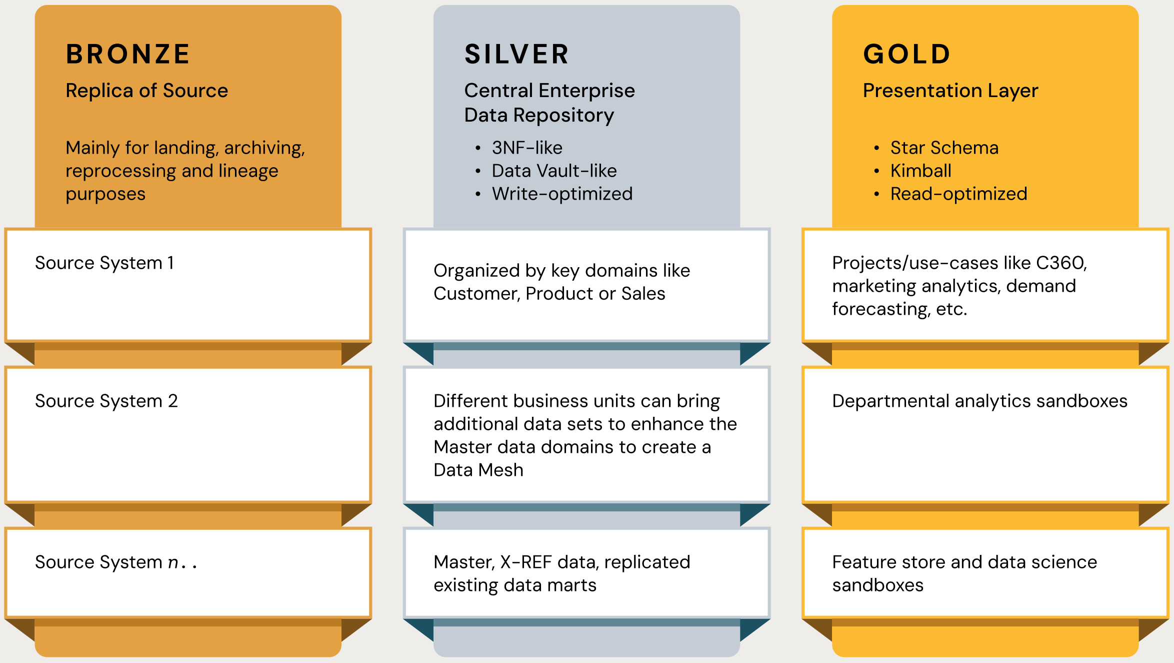
CONFIGURING GLUE CATALOG WITH UC

Configure your cluster just as you do today to access the Glue Metastore. This will be registered as the hive_metastore catalog. The cluster will additionally have access to Unity Catalog. The three-level namespace allows for objects to be selected from both. Alternatively, you could register an external table in the Glue Metastore that points to a Delta Sharing table that was shared with the Glue recipient.

DATA MODELING IN THE LAKEHOUSE

A modern lakehouse serves as a comprehensive enterprise-level data platform. It is very scalable and effective for a wide range of use cases, including ETL, BI, data science and streaming, which may call for various data modeling techniques. See how a typical lakehouse is set up:

DATA LAKEHOUSE ARCHITECTURE


BRONZE
Replica of Source
Mainly for landing, archiving, reprocessing and lineage purposes
Source System 1
Source System 2
Source System n. .
SILVER
Central Enterprise Data Repository
• 3NF-like
• Data Vault-like
• Write-optimized
Organized by key domains like Customer, Product or Sales
Different business units can bring additional data sets to enhance the Master data domains to create a Data Mesh
Master, X-REF data, replicated existing data marts
GOLD
Presentation Layer
• Star Schema
• Kimball
• Read-optimized
Projects/use-cases like C360, marketing analytics, demand forecasting, etc.
Departmental analytics sandboxes
Feature store and data science sandboxes

- **The Bronze layer** uses the data models of source systems. If data is landed in raw formats, it is converted to Delta Lake format within this layer.
- **The Silver layer** for the first time brings the data from different sources together and conforms it to create an enterprise view of the data — commonly using more-normalized, write-optimized data models that are typically 3rd-Normal Form-like or Data Vault-like
- **The Gold layer** is the presentation layer, with more denormalized or flattened data models than the Silver layer, typically using Kimball-style dimensional models or star schemas. The Gold layer also houses departmental and data science sandboxes to enable self-service analytics and data science across the enterprise. Providing these sandboxes and their own separate compute clusters prevents the business teams from creating their own copies of data outside of the Lakehouse.

Different data organization concepts and modeling methodologies may apply to different projects on a lakehouse due to the range of use cases. The Databricks Lakehouse Platform technically supports a wide range of data modeling approaches:

- Data vault
- Dimensional modeling (star schema, Snowflake schema or hybrid)

For more details please read here.

DATA MIGRATION

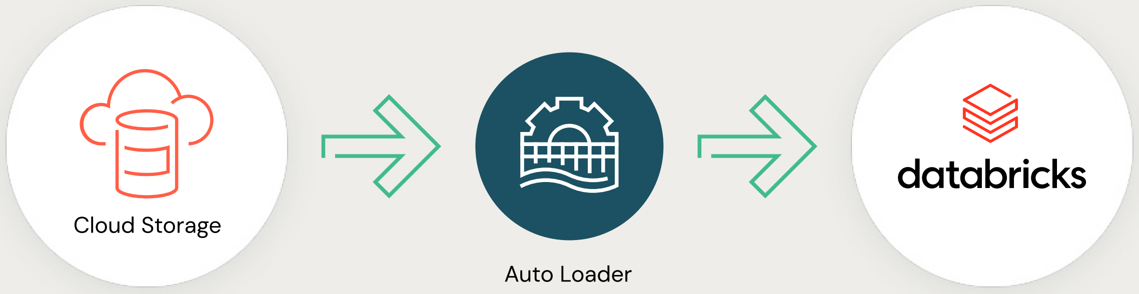
Databricks is optimized for cloud object storage — in this case S3 in Amazon Web Services. In addition to cloud storage, Databricks can also read/write to other storage endpoints, including relational databases (Oracle, SQL Server, Teradata), HDFS, Apache Hive, NoSQL (HBase, Cassandra, Neo4j, MongoDB), in-memory cache (Redis, RocksDB), message bus (Kafka), files (delimited text files, JSON, Parquet, ORC, Avro) and many others. Data sources can be in the cloud or on-premises, but Databricks is optimized for the cloud.

HDFS DATA MIGRATION

To get started with data migration, first look at a dual ingestion strategy. You may already have a defined process to land data into Hadoop. This might be implemented via a third-party ingestion tool or perhaps an in-house built framework. A simple approach could be to fork the target such that data is landed to both HDFS and S3. Getting an initial data feed provides an additional backup location of your data. It will also allow you to unlock new advanced analytics in the cloud with Databricks.

The next step is migration of historical data. This step may take some time, based on the amount of data that exists in HDFS. If possible, try to align data sets with prioritized use cases that need to be migrated away from Hadoop. This will help identify the order in which you'll need to move data to the cloud.

A detailed guide to migrating HDFS data to the cloud can be found here.



EMRFS DATA TO S3 MIGRATION

Migrating data from EMR File System (EMRFS) to Amazon S3 involves following steps to ensure a smooth and efficient transition.

- **Copy data from EMRFS to S3:** There are several methods you can use to copy data from EMRFS to S3. Here are a few common approaches:
  - **AWS CLI:** Use the AWS command-line interface (CLI) aws s3 cp command to copy files from EMRFS to S3. This command allows you to specify source and destination paths, and you can use wildcards to copy multiple files or directories.
  - **S3DistCp:** This is a tool specifically designed for copying large amounts of data between EMRFS and S3. It optimizes the transfer process and provides options for parallelism, data compression and preserving file metadata.
  - **Spark or Hadoop job:** If you have a Spark or Hadoop job running on EMR, you can modify it to read data from EMRFS and write to S3. This approach allows you to customize the migration process and handle any data transformations or filtering during the transfer.
- **Update applications and scripts:** If you have applications or scripts that directly reference EMRFS paths, update them to use the new S3 paths. This includes modifying Spark scripts.
- **Modify configurations:** Update any references to EMRFS or file system-specific settings to reflect the new S3 storage
- **Update workflows and dependencies:** If you have workflows or dependencies that rely on EMRFS data, update them to use the new S3 paths. This includes ETL pipelines, data ingestion processes and downstream analytics applications that consume the migrated data.

Working With Different File Formats

DELTA FORMAT

Delta Lake is the optimized storage layer that provides the foundation for storing data and tables in the Databricks Lakehouse Platform. Delta Lake is open source software that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling. Delta Lake is fully compatible with Apache Spark APIs, and was developed for tight integration with Structured Streaming, allowing you to easily use a single copy of data for both batch and streaming operations and providing incremental processing at scale.

Delta Lake is the default storage format for all operations on Databricks. Unless otherwise specified, all tables on Databricks are Delta tables. Databricks originally developed the Delta Lake protocol and continues to actively contribute to the open source project. Many of the optimizations and products in the Databricks Lakehouse Platform build upon the guarantees provided by Apache Spark and Delta Lake. For information on optimizations on Databricks, see Optimization recommendations on Databricks.

PARQUET AND ICEBERG

At Databricks we always highly recommend that data be converted to the Delta Lake format. The easiest way to make this change is using "Convert to Delta" syntax. This statement converts an existing Parquet table to a Delta table in place. This command lists all the files in the directory, creates a Delta Lake transaction log that tracks these files, and automatically infers the data schema by reading the footers of all Parquet files. The conversion process collects statistics to improve query performance on the converted Delta table. If you provide a table name, the metastore is also updated to reflect that the table is now a Delta table. We can use either the table name or the location directly to convert to Delta.

This command supports converting Iceberg tables whose underlying file format is Parquet. In this case, the converter generates the Delta Lake transaction log based on Iceberg table's native file manifest, schema and partitioning information.

```
CONVERT TO DELTA database_name.table_name; -- only for Parquet tables

CONVERT TO DELTA parquet.'s3://my-bucket/path/to/table'
  PARTIONED BY (date DATE); -- if the table is partitioned

CONVERT TO DELTA iceberg.'s3://my-bucket/path/to/table'; -- uses Iceberg manifest
for metadata
```

UNIFORM

Databricks has introduced a new feature called UniForm — short for Universal Format. Delta Universal Format (UniForm) allows you to read Delta tables with Iceberg reader clients.

UniForm takes advantage of the fact that both Delta Lake and Iceberg consist of Parquet data files and a metadata layer. UniForm automatically generates Iceberg metadata asynchronously, without rewriting data, so that Iceberg clients can read Delta tables as if they were Iceberg tables. A single copy of the data files serves both formats.

Read more about UniForm.

HUDI MIGRATION

Hudi's internal design is different from Delta.

For migration we would have to read Hudi's table as a DataFrame and write it to Delta. Now, Hudi versions after 0.5.2 are incompatible with DBR, and they have relied on low-level Spark APIs, which we have modified for edge features. So, to do the conversion, you will have to install Hudi jar version 0.5.2 or older.

Here is a sample code that could help you to read the Hudi table and write it to Delta in another location.

```
df = spark
.read()
.format("hudi")
.option(DataSourceReadOptions.QUERY_TYPE_OPT_KEY(),
DataSourceReadOptions.QUERY_TYPE_SNAPSHOT_OPT_VAL())
.load(tablePath)
df.write.format("delta").save(deltaTablePath)
```

MIGRATION VALIDATION

Validation is one of the most critical parts of data migration. It requires you to familiarize yourself with the process used to migrate data from EMR to Databricks. Identify the tools, technologies and methods employed during the migration. It also includes determining the validation criteria based on your data requirements and business logic. This may include verifying data completeness, accuracy, consistency and integrity. Generally a testing framework with a script to compare values automatically in both of the platforms is used. Another prerequisite is to select a representative sample of data from both the source (EMR) and the target (Databricks) environments. Choose data sets that cover a range of data types, structures and characteristics. Following are standard test cases that should be validated before cutoff:

- **Data consistency check:** Perform a row-by-row comparison of the sample data between EMR and Databricks. Ensure that the data matches exactly, including all fields, values and formats. Use SQL queries or data comparison tools to facilitate this process.
- **Aggregated metrics validation:** Validate aggregated metrics, such as counts, sums, averages or other calculations, to ensure they match between EMR and Databricks. Compare these metrics at various levels, such as per day, per category or per customer.
- **Schema validation:** Check the schema and structure of the migrated data in Databricks. Ensure that the tables, columns and data types match the expected schema. Validate that any transformations or conversions during the migration process have been correctly applied.
- **Data integrity validation:** Verify the integrity of the data by comparing primary and foreign key relationships. Ensure that referential integrity is maintained between related tables in Databricks.
- **Data quality checks:** Perform data quality checks to identify any anomalies, inconsistencies or missing values in the migrated data. Use data profiling techniques and data quality tools to identify potential issues.
- **Business logic validation:** Validate the data against the defined business rules and logic. Check if the data transformations, calculations or aggregations in Databricks match the expected results based on the business requirements.
- **Performance validation:** Evaluate the performance of the migrated data in Databricks. Compare the query execution times and resource utilization with the performance observed in EMR. Ensure that the data retrieval and processing times are within acceptable limits.

- **Documentation and reporting:** Document the validation process, including the validation criteria, steps performed and the results obtained. Prepare a validation report that outlines the findings, any issues or discrepancies identified, and recommendations for resolution.
- **Iterative validation:** Perform iterative validation on different subsets of data or additional samples to ensure comprehensive coverage. Iterate the validation process until you have validated a sufficient amount of data to gain confidence in the migration.

For more advanced table data and schema comparison, third-party or open source frameworks similar to Datacompy can be considered. By following these steps, you can systematically validate the data migrated from EMR to Databricks, ensuring its accuracy, integrity and alignment with your business requirements.

KEY CONSIDERATIONS

Summarizing the key considerations for best performance in Delta Lake:

- Profile the metastore to get maximum information about existing data
- Use the Delta Lake format as a default and migrate data from other formats for 3 to 4 times better performance
- Migrate data to S3, as Databricks is optimized for cloud object store — this allows you to read/write from any source system
- If required, redesign your data model to serve your use case best instead of using the same data model of your existing architecture
- Validate the data thoroughly before signing off

Phase 4: Code Migration

SPARK

Spark versions — upgrade from 2.x => 3.x

RDDs to DataFrames and data sets

- RDD APIs are supported, but will not be performant compared to DataFrames

Changes to submission (spark-submit): There is no URL to specify and no master to configure

```
spark-submit --class org.apache.spark.template.App --deploy-mode cluster
--master yarn --num-executors 5 --executor-cores 5 --executor-memory 20g
-conf spark.yarn.submit.waitAppCompletion=false s3://myjobsbucket/sparkSubmitCUJ/
spark_template_1_12_SNAPSHOT.jar s3://outputbucket/testOuput
```

Turns into this:

```
[“--class”,”org.apache.spark.template.App”,”s3://myjobsbucket/sparkSubmitCUJ/
max.fisher@databricks.com/sparkSubmitCUJ/spark_template_1_12_SNAPSHOT.jar”,
”s3://myjobsbucket/sparkSubmitCUJ/max.fisher@databricks.com/testOuput/”]
```

Remove hard-coded references:

- Storage locations repointed
- Removal of YARN configurations
- When submitting Spark jobs: make use of Spark Jar tasks rather than Spark Submit

Adapt your existing Apache Spark code for Databricks.

PYSPARK JOB

- Run PySpark code as Jobs
- Remove SparkContext references
- Import code into notebook
- Use the Python wheel task to package and distribute the files required to run your code as Databricks jobs

JAVA/SCALA (Dependency Management for Jars)

When migrating your EMR jobs to Databricks, you need to ensure that the external dependencies used by your Java and Scala code are available in the Databricks environment. Here are two approaches to handle these dependencies:

1. **Upload JAR files to Databricks:** If your EMR jobs rely on custom JAR files or external libraries, you can manually upload these JAR files to Databricks. In the Databricks workspace, you can use the UI or Databricks CLI (command-line interface) to upload the JAR files. Once they're uploaded, you can reference these JAR files in your Databricks notebooks or jobs.
2. **Use Maven/Gradle-based dependency management:** If you manage your dependencies using build tools like Maven or Gradle, you can leverage these tools in Databricks as well. You can include the necessary dependencies in your project's build file (pom.xml for Maven or build.gradle for Gradle) and specify the repositories from which to fetch the dependencies. When you run your code on Databricks, the build tool will automatically fetch and resolve the required dependencies from the specified repositories.

BATCH AND STREAM PROCESSING

There are several reasons why one might choose to use Spark Structured Streaming over Spark Streaming and DStreams:

1. **Higher-level API:** Spark Structured Streaming provides a higher-level API based on DataFrames and data sets, which are more expressive and easier to use compared to the lower-level API provided by Spark Streaming and DStreams. The DataFrame and data set APIs offer a familiar SQL-like programming model and support a wide range of built-in operations for data manipulation and transformation.

2 **Unified batch and streaming processing:** Spark Structured Streaming offers a unified programming model for both batch and streaming processing. It allows developers to write the same code for batch jobs and streaming jobs, making it easier to transition between different processing modes without significant code changes. This unified approach reduces complexity and improves developer productivity.

3 **Continuous processing:** Spark Structured Streaming introduces continuous processing, which enables near real-time data processing with low latency. Unlike the micro-batch processing model used in Spark Streaming and DStreams, continuous processing in Structured Streaming allows data to be processed as it arrives, resulting in lower end-to-end latency and more accurate results.

4 **Fault tolerance:** Structured Streaming provides end-to-end fault tolerance by maintaining the lineage of transformations and ensuring that data and metadata can be recovered in case of failures. It offers exactly-once semantics, meaning that each record is processed exactly once, even in the presence of failures. This guarantees data integrity and consistency, which is crucial for reliable streaming applications.

5 **Integration with the Spark ecosystem:** Spark Structured Streaming seamlessly integrates with the wider Spark ecosystem, including libraries, connectors and tools. It supports a wide range of data sources and sinks, such as files, databases, message queues, and streaming platforms like Apache Kafka. This integration allows users to leverage existing tools and systems and provides a unified data processing platform.

6 **Performance and scalability:** Spark Structured Streaming benefits from the performance optimizations and scalability features of Apache Spark. It leverages the Catalyst optimizer and Tungsten execution engine, which provide significant performance improvements for data processing. Spark's ability to scale horizontally by adding more nodes to the cluster allows Structured Streaming to handle large-scale streaming workloads effectively.

In summary, Spark Structured Streaming offers a higher-level API, unified batch and streaming processing, low latency with continuous processing, fault tolerance, seamless integration with the Spark ecosystem, performance and scalability advantages, and strong community support.

- Migrate Spark Streaming/DStream to Spark Structured Streaming
- Conversion from other streaming frameworks (Apache Flink) to Spark Structured Streaming

MAPREDUCE

- MapReduce to Spark Transformation: Rewrite your MapReduce code to use Apache Spark's APIs, such as RDDs (Resilient Distributed Datasets) or DataFrames
- For MapReduce's Map function, you can use Spark's `map()` or `flatMap()` transformations on RDDs or DataFrames to perform data transformations
- For MapReduce's Reduce function, you can use Spark's `reduceByKey()`, `groupByKey()` or `aggregateByKey()` transformations to perform aggregations or reduce operations
- If your MapReduce job involves custom input/output formats, you might need to adapt them to work with Spark's file input/output operations or leverage Spark's built-in connectors for various data sources

SQOOP

Sqoop is running MapReduce under the hood, and hence is one of the main reasons MapReduce is still deployed. Many customers have moved off Sqoop and started using Spark to read data directly from relational systems. The syntax to read from databases in Spark is very simple, and you have flexibility with how the data is processed and persisted to a target destination.

You can replace the Sqoop calls in Spark code using the JDBC source. This Spark code will reside in a Databricks notebook or be packaged in a code artifact (JAR, python whl, etc.).

See the online documentation. Here is an example call:

```
val jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()
```

Note:
With Databricks, you can leverage Secrets to ensure credentials are not exposed in the code.

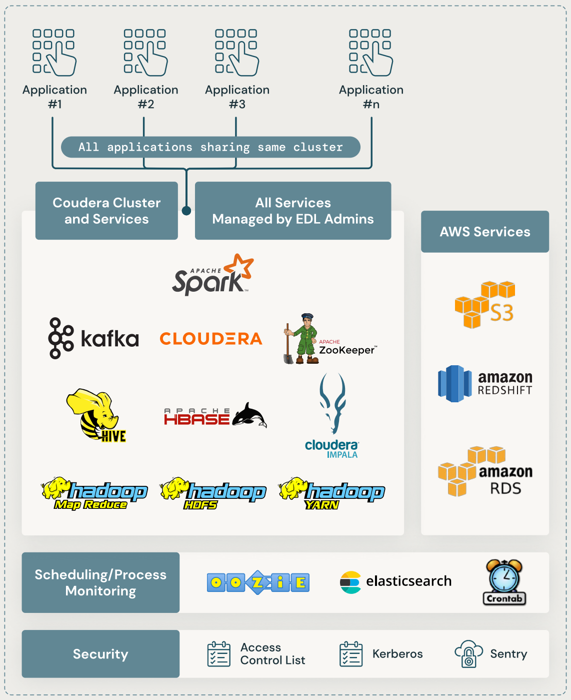
The Spark JDBC source also allows you to specify options similar to Sqoop, such as customized select query, fetch read and batch write sizes, and isolation settings.

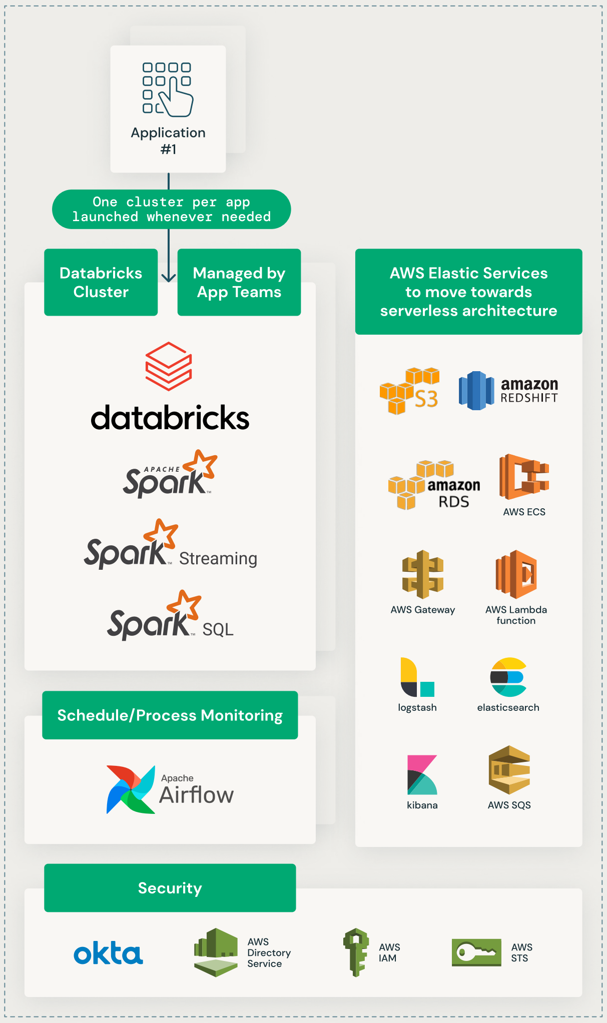
Sqoop provides incremental loads via Sqoop Jobs, and internally it can track a field to determine new data. This field is typically a timestamp or may be an ever-increasing sequence ID. Sqoop will load new data into a target location and will persist the largest value for this field. This value is then used to retrieve new data from the source table. Spark does not support this functionality out of the box. You will need to track a "last modified timestamp" field or sequence ID in code and adjust the SQL query that is used to extract data from the JDBC source.

HIVE/IMPALA, HBASE

- EMR Impala to Spark migration:
  - Column type mapping**:** Depending on the versions of the two products, there are several column types mismatched
  - SQL functions have different naming across the two engines — e.g., 'int_months_between' in Impala is 'month_between' in Spark
- Identify HiveQL code base scripts
- DDL conversion
  - Use Delta Format
  - Convert to USING vs STORED AS
- DML queries should run as is
- Convert scripts to Databricks Notebooks, Delta Live Tables or PySpark scripts
- Hive UDFs can be imported into Databricks SQL

databricks


Application #1
Application #2
Application #3
Application #n
All applications sharing same cluster
Coudera Cluster and Services
All Services Managed by EDL Admins
AWS Services
Spark
kafka
CLOUDERA
ZooKeeper
HIVE
HBASE
cloudera IMPALA
hadoop Map Reduce
hadoop HDFS
hadoop YARN
S3
amazon REDSHIFT
amazon RDS
Scheduling/Process Monitoring
OOZIE
elasticsearch
Crontab
Security
Access Control List
Kerberos
Sentry


Application #1
One cluster per app launched whenever needed
Databricks Cluster
Managed by App Teams
AWS Elastic Services to move towards serverless architecture
databricks
Spark
Spark Streaming
Spark SQL
S3
amazon REDSHIFT
amazon RDS
AWS ECS
AWS Gateway
AWS Lambda function
logstash
elasticsearch
kibana
AWS SQS
Schedule/Process Monitoring
Apache Airflow
Security
okta
AWS Directory Service
AWS IAM
AWS STS

HIVEQL VS. SPARK SQL

Apache Hive is a data warehouse software project that was initially built for the Hadoop ecosystem. Hive can be used on-premises and in the cloud with a variety of storage mediums, including HDFS, Azure cloud storage, Amazon Web Services S3 object storage, and Google Cloud Storage. Hive provides an abstraction layer that represents the data as tables with rows, columns and data types to query and analyze using a SQL interface called HiveQL. Hive uses an in-memory distributed engine called Apache Tez to process the data.

Apache Hive supports transactions (ACID) with Hive LLAP. Transactions guarantee consistent views of the data in an environment in which multiple users/processes are accessing the data at the same time for Create, Read, Update and Delete (CRUD) operations. Databricks offers Delta, which is similar to Hive LLAP in that it provides transaction (ACID) guarantees, but it offers several other benefits to help with performance and reliability when accessing the data. Delta is an open source project.

Spark SQL is Apache Spark's module for interacting with structured data represented as tables with rows, columns and data types. Spark SQL is SQL 2003 compliant and uses Apache Spark as the distributed engine to process the data. In addition to the Spark SQL interface, a DataFrames API can be used to interact with the data using Java, Scala, Python and R.

Spark SQL is similar to HiveQL. Both use ANSI SQL syntax, and the majority of Hive functions will run on Databricks. This includes Hive functions for date/time conversions and parsing, collections, string manipulation, mathematical operations and conditional functions. There are some functions specific to Hive that would need to be converted to the Spark SQL equivalent or that don't exist in Spark SQL on Databricks. You can expect all HiveQL ANSI SQL syntax to work with Spark SQL on Databricks. This includes ANSI SQL aggregate and analytical functions.

Hive is optimized for the Optimized Row Columnar (ORC) file format and also supports Parquet. Databricks is optimized for Parquet and Delta. We always recommend using Delta, which uses open source Parquet as the file format.

Example of a HiveQL table creation using HDFS:

```sql
CREATE EXTERNAL TABLE CUSTOMER_DB.CUSTOMER (USER_ID INT, USER_NAME STRING)
STORED AS PARQUET
LOCATION '/data/customer';
```

Spark SQL on Databricks table creation using object storage:

```
CREATE TABLE CUSTOMER_DB.CUSTOMER (USER_ID INT, USER_NAME STRING) STORED AS PARQUET
LOCATION '/data/customer';
```

You don't need to use the keyword EXTERNAL. Once you specify a location, the table automatically becomes an external table. For the location path, you can use a mount point location, such as `/path_to_my_directory`, or use the cloud service provider's file system client when specifying the location, such as `s3a://my_bucket/path_to_my_files`, if using S3 in AWS.

Or

```
DROP TABLE IF EXISTS CUSTOMER_DB.MY_TABLE;
CREATE TABLE BMATHEW.MY_TABLE (USER_ID INT, USER_NAME STRING)
USING DELTA
LOCATION '/data/customer'
```

Again, you don't need to use the keyword EXTERNAL.

The key difference between Hive vs. Spark SQL on Databricks when creating tables is that Hive syntax uses `stored as` whereas Databricks uses `using`. If you are using Hive LLAP today and migrating to Databricks, then we strongly recommend that you use Delta — `USING DELTA`. Delta provides transactions (ACID), is open source and will improve your data engineering, data science and BI workloads with improved performance, reliability and consistency when accessing the data.

There are also many options that you can set for table configuration. Here are a few common ones Hive users are familiar with that also work with Spark SQL on Databricks.

- `LOCATION` — This is the cloud storage location where the data files will be stored. The default path will always be inside the default root blob storage account at `/user/hive/warehouse/`. Without specifying a location, the table will be created as a managed table — meaning that once you drop the table, all the data files will also be deleted. When you specify a location, the table becomes an unmanaged or external table — meaning that once you drop the table, the data remains in the directory. For the location path, you can use a mount point location, such as `/path_to_my_directory`, or use the cloud service provider's file system client when specifying the location such as `s3a://my_bucket/path_to_my_files`, if using S3 in AWS.

- `PARTITIONED BY` — To partition the table by one or more columns. Always choose partition columns with a lower number of distinct values (low cardinality).
- `CLUSTERED BY` — For columns with a high number of values (high cardinality), bucketing could help with performance.
- `TBL PROPERTIES` — These are additional settings similar to those in Hive that let you specify certain configurations.

Here is an example using the above properties to create a table in Databricks using Parquet:

```
DROP TABLE IF EXISTS BMATHEW.MY_TABLE;
CREATE TABLE BMATHEW.MY_TABLE (
USER_ID INT,
USER_NAME STRING,
TRANSACTION_DATE DATE)
USING PARQUET
PARTITIONED BY (TRANSACTION_DATE)
CLUSTERED BY (USER_ID) SORTED BY (USER_ID) INTO 32 BUCKETS
LOCATION '/tmp/bmathew/test_hive_data'
TBLPROPERTIES ('compression'='snappy', 'owner'='bmathew');
```

It's important to note that bucketing on Databricks is supported only when using Parquet, not Delta

To view the properties of a table including the schema definition:

```
DESCRIBE FORMATTED BMATHEW.MY_TABLE;
```

The Hive style syntax will also work on Databricks:

```
CREATE TABLE my_table STORED AS PARQUET AS (select 1 as user_id);
```

However, we recommend that you don't use the Hive style syntax (i.e., `stored as parquet`). The syntax `using parquet` is specific to Spark SQL, and these tables will always use Databricks optimizations outside of open source for the Spark SQL Catalyst optimizer. By contrast, `stored as parquet` can be used for both Spark and Hive, but not all Databricks-specific Spark SQL optimizations may work as expected. Thus, we recommend using "USING PARQUET" if you don't want to use Delta.

Please refer to the online documentation for more information:

- Databricks databases and tables
- Spark SQL
- Hive compatibility

We recommend that you use Delta to store your data.

MIGRATE SECRETS FROM AWS TO DATABRICKS

Follow these general steps:

- **Identify secrets:** Identify the secrets in AWS that need to be migrated to Databricks. These could include credentials, API keys, tokens, or any other sensitive information stored in AWS services like AWS Secrets Manager or AWS Parameter Store.
- **Evaluate Databricks Secrets Management:** Understand how Databricks manages secrets. Databricks provides a feature called "Secrets" that allows you to securely store and manage secrets within Databricks workspaces.
- **Update applications and notebooks:** Identify the applications, notebooks or scripts that use the AWS secrets. Modify the code to retrieve the secrets from Databricks instead of AWS. In Databricks, you can access secrets programmatically using the Databricks Secrets API.
- **Test and validate:** Test the applications and notebooks to ensure they can successfully retrieve the secrets from Databricks. Validate that the migrated secrets are functioning as expected and that your applications are working properly.
- **Rotate Secrets:** Follow the guidelines set by the customer for rotating the secrets periodically to enhance security. Databricks provides APIs to manage secret rotation programmatically. You can update the secrets in Databricks and then validate your applications and notebooks accordingly.
- **Clean up AWS Secrets:** Once you have migrated the secrets to Databricks and confirmed their functionality, you can delete or disable the corresponding secrets in AWS. Ensure you no longer have any dependencies on the AWS secrets before removing them.
- **Monitor and maintain:** Establish proper monitoring and maintenance processes for secrets in Databricks. Regularly review and update secrets as needed. Monitor the usage and access patterns of secrets to ensure they are adequately protected.

It's worth noting that the specific steps may vary depending on your existing infrastructure, the nature of the secrets and the requirements of your applications. Make sure to carefully plan and test the migration process to ensure a smooth transition and maintain the security of your secrets throughout the process.

Phase 5: Data Pipeline Migration

An end-to-end view of the pipelines from data sources to the consumption layer considering the governance aspects must be thoroughly understood to effectively migrate the workloads. Data pipeline migrations from Hadoop to Databricks consist of several key components: orchestration, source/sink migration, query migration and refactoring.

ORCHESTRATION MIGRATION

An ETL orchestration can refer to orchestrating and scheduling end-to-end pipelines covering data ingestion, data integration, result generation or orchestrating DAGs of a specific workload type like data integration. In EMR the orchestration is typically done using AWS Data Pipeline, AWS Managed Airflow, Apache Airflow, Apache Livy, custom scripts, etc. There are generally two options when migrating these workflows.

1. Use Databricks Workflows to orchestrate the migrated pipelines. In addition, Delta Live Tables can be used for building reliable and efficient data processing pipelines. Using Delta Live Tables provides a standard framework for building both batch and streaming use cases along with critical data engineering features such as automatic data testing, deep pipeline monitoring and recovery. Oozie Jobs in Hadoop get created as Databricks Workflows. It also has out-of-the-box functionality to SCD Type 1 and Type 2 tables.

2. It is also possible to use the external tools for orchestration and repoint these tools from EMR compute to Databricks via the Databricks CLI or REST APIs. It is recommended to use Databricks Workflows for better integration, simplicity and lineage.

3. Here are the steps to migrate Airflow-based DAGs to Databricks Workflows:
   - a. **Evaluate compatibility:** Assess the compatibility of your existing Airflow workflows with Databricks. Check if there are any dependencies or operators that need to be adapted or replaced to work with Databricks.
   - b. **Set up Databricks environment:** Ensure you have a Databricks environment set up and configured with the necessary clusters and authentication mechanisms. This may involve setting up a Databricks workspace and creating clusters.

c **Recreate DAGs as Databricks notebooks:** Convert your Airflow DAGs into Databricks notebooks. Each Airflow task should be translated into a corresponding Databricks notebook. Use Databricks Python or Scala notebook format based on your preferences.

d **Migrate dependencies:** Identify and migrate any dependencies used in your Airflow workflows to Databricks. This includes data sources, libraries and external services. Ensure that the required dependencies are available in the Databricks environment.

e **Update task logic:** Review the logic of each task in your Airflow DAG and update it to work within the Databricks notebook. This may involve rewriting code, adjusting data paths or modifying the execution logic to fit Databricks-specific requirements.

f **Configure Databricks operators:** Replace any Airflow operators specific to other platforms or services with Databricks operators. Databricks provides operators for various operations, such as running notebooks, creating clusters, submitting jobs and managing libraries.

g **Set up connections and credentials:** Configure the necessary connections and credentials in Databricks to access external services or resources required by your workflows. This could include database connections, cloud storage credentials or API keys.

h **Test and validate:** Thoroughly test each Databricks notebook to ensure it functions correctly within the Databricks environment. Verify that the data flows, transformations and dependencies are working as expected.

i **Update scheduling and dependencies:** Set up scheduling and dependencies within Databricks. Use Databricks' built-in scheduling capabilities or integrate with external scheduling tools if necessary. Update any task dependencies to match the new Databricks notebook structure.

j **Monitor and troubleshoot:** Implement monitoring and logging mechanisms within Databricks to track the performance and behavior of your workflows. Set up alerts and notifications to identify and address any issues that may arise during execution.

k **Deploy and execute:** Deploy the migrated Databricks Workflows to your production environment. Execute them and closely monitor the execution to ensure they run smoothly and produce the desired outcomes.

4 There could be scenarios where Airflow is preferred as an orchestration tool. In this case, the existing DAGs need to be refactored to use Databricks Airflow operators. The following operators can be leveraged to retrofit the existing task logic.

- DatabricksCopyIntoOperator
- DatabricksReposCreateOperator
- DatabricksReposDeleteOperator
- DatabricksReposUpdateOperator
- DatabricksRunNowOperator
- DatabricksRunNowDeferrableOperator
- DatabricksSqlOperator
- DatabricksSqlSensor
- DatabricksPartitionSensor
- DatabricksSubmitRunOperator
- DatabricksSubmitRunDeferrableOperator

5 If Apache Livy is used in the existing EMR jobs for submitting the Spark jobs, this will need to be refactored to Databricks REST APIs. Converting Apache Livy code to Databricks REST API calls involves rewriting the Livy-specific code to make appropriate API requests to Databricks. Here are the key steps to convert Apache Livy code to Databricks REST API calls:

a **Authentication:** In Livy, you typically authenticate using the LivyClientBuilder and providing the Livy URL. In Databricks, you need to use the Databricks REST API authentication mechanism. This involves obtaining an access token or using an API key for authentication.

b **Create a new session:** In Livy, you create a session using the LivyClient and CreateSessionRequest classes. In Databricks, you make a POST request to the POST /api/2.0/clusters/create endpoint to create a cluster and obtain the cluster ID.

c **Run code:** In Livy, you submit code for execution using the LivyClient and SubmitStatementRequest classes. In Databricks, you make a POST request to the POST /api/2.0/jobs/run-now endpoint and provide the code as part of the request payload.

d **Retrieve job status:** In Livy, you can get the status of a job using the LivyClient and GetStatementStatusRequest classes. In Databricks, you make a GET request to the GET /api/2.0/jobs/runs/get endpoint and provide the job run ID to retrieve the status.

e **Get job results:** In Livy, you can retrieve job results using the LivyClient and GetStatementRequest classes. In Databricks, you make a GET request to the GET /api/2.0/jobs/runs/get-output endpoint and provide the job run ID to get the output.

f **Close session:** In Livy, you close the session using the LivyClient and DeleteSessionRequest classes. In Databricks, you make a POST request to the POST /api/2.0/clusters/delete endpoint and provide the cluster ID to terminate the cluster.

These are the basic steps involved in converting Livy code to Databricks REST API calls. You'll need to review the specific Livy code you have and rewrite it based on the Databricks REST API documentation. Adapt the code to make the appropriate API requests and handle the responses according to your requirements.

SOURCE/SINK MIGRATION

Databricks supports a wide variety of sources and sinks to read and write data similar to EMR.

One of the most common data sources for EMR workloads is the S3 object storage. Databricks supports connecting to S3 for reading and writing data in various formats including Avro, Parquet, Delta and others.

Standard access patterns with S3

UC EXTERNAL LOCATIONS

The recommended access pattern for reading S3 based data sources is to use Unity Catalog. Unity Catalog allows you to register S3 bucket locations as external locations. Once they're registered, you can use the fully qualified S3 URI to access data secured with Unity Catalog. Because permissions are managed by UC, you do not need to pass any additional credentials or configuration options for authentication.

INSTANCE PROFILES

If Unity Catalog external locations are not a valid option for your Databricks setup, you can access S3 bucket locations by attaching instance profiles to the Databricks cluster. You set up the IAM roles within AWS with the appropriate privileges for S3 access, load these IAM roles into a Databricks workspace and then attach the appropriate instance profile to the cluster at launch. Here is a detailed guide on setting up IAM roles and adding instance profiles to Databricks.

Once an instance profile is added to the Databricks workspace, you can grant users, groups and service principals permission to launch clusters with the instance profile.

AWS KEYS

While not so commonly used or recommended, Databricks supports using AWS keys to access S3 resources. A common pattern is to store the keys in secret scope and then grant users, groups and service principals access to the secret scope as needed. This protects the AWS key while allowing users to access S3 resources.

S3 WITH HADOOP OPTIONS

Databricks also supports configuring the S3A file system using open source Hadoop options. These properties can be configured globally or at per-bucket level.

```
# Global S3 configuration
spark.hadoop.fs.s3a.aws.credentials.provider <aws-credentials-provider-class>
spark.hadoop.fs.s3a.endpoint <aws-endpoint>
spark.hadoop.fs.s3a.server-side-encryption-algorithm SSE-KMS
```

STREAMING SOURCES AND SINKS WITH AWS KINESIS

Databricks supports using AWS Kinesis as a streaming data source and a data sink.

For authentication with Kinesis, you can use the instance profile-based option discussed above for regular ETL workloads. Another possible option is to use the AssumeRole policy if Kinesis is owned by a different AWS account than one that owns the Databricks workspace. Here is a detailed discussion of setting up a cross account Assume Role policy to enable consuming from Kinesis.

A Structured Streaming pipeline that consumes from Kinesis can be started by simply setting the readStream format as "kinesis" and the appropriate configuration parameters discussed here.

The Kinesis source runs Spark jobs in a background thread to periodically prefetch Kinesis data and cache it in the memory of the Spark executors. The streaming query processes the cached data only after each prefetch step completes and makes the data available for processing. Hence, this prefetching step determines a lot of the observed end-to-end latency and throughput. Consider the options such as maxRecordsPerFetch, maxFetchRate, minFetchPeriod and maxFetchDuration to fine-tune the number of records fetched per read to match with the expected throughput required.

WRITING TO KINESIS

Writing to Kinesis can be accomplished using the for-each-batch semantics in Structured Streaming.

REDSHIFT AS SOURCE/SINK

Databricks Runtime includes the Redshift JDBC Driver accessible using the 'redshift' keyword for the Spark data source format option.

Reading and writing to Redshift tables is as straightforward as the following code.

```
# Read data from a query
df = (spark.read
  .format("redshift")
  .option("query", "select x, count(*) <your-table-name> group by x")
  .option("tempdir", "s3a://<bucket>/<directory-path>")
  .option("url", "jdbc:redshift://<database-host-url>")
  .option("user", username)
  .option("password", password)
  .option("forward_spark_s3_credentials", True)
  .load()
)

# After you have applied transformations to the data, you can use
# the data source API to write the data back to another table

# Write back to a table
(df.write
  .format("redshift")
  .option("dbtable", table_name)
  .option("tempdir", "s3a://<bucket>/<directory-path>")
  .option("url", "jdbc:redshift://<database-host-url>")
  .option("user", username)
  .option("password", password)
  .mode("error")
  .save()
)
```

Since query execution from Redshift involves extracting a lot of data to S3 as temp storage, it is advisable to store this data as a Delta table in S3 to take advantage of improved repeated read performance and other Delta optimizations available on S3.

Additional configuration options on authentication mechanisms for Spark Driver to Redshift, Spark to S3, and Redshift to S3 are discussed here.

S3 SELECT

The Databricks S3 connector supports reading only required data from S3 objects for CSV and JSON file formats using the Amazon S3 Select service via 's3select' Spark data source format. Additional details on using this service are here.

OTHER NON-AWS DATA SOURCES

Apart from AWS-specific data sources such as S3 and Redshift, Databricks supports connecting to many other streaming and ETL sources such as Kafka and RDS and other data sources.

TRANSFORM YOUR DATA LAKE INTO A LAKEHOUSE

As part of the EMR to Databricks Lakehouse migration, lakehouse architecture adoption is common, and this entails converting the existing data lake assets into the Delta Lake format.

Here are the typical steps for lakehouse transformation using Delta Lake:

1. **Back up your data:** Before starting the conversion process, it's crucial to create a backup of your historical data lake. This ensures that you have a restore point in case anything goes wrong during the conversion.
2. **Analyze your existing data:** Perform a thorough analysis of your historical data lake to understand its structure, data types, and any existing issues or inconsistencies. Identify any dependencies or applications that rely on the data.
3. **Refactor the existing data lake model to lakehouse medallion architecture governed by Unity Catalog:** Set up new Delta Lake storage locations where the converted data will reside. This can be done on your existing storage system or a new one, depending on your requirements.
4. **Extract data from the historical data lake:** Delta Live Tables pipelines or Databricks Auto Loader jobs are great components for automatic lakehouse ingestion.

5. **Transform and cleanse the data:** During the extraction process, you may encounter data quality issues or inconsistencies. Apply necessary transformations and cleansing techniques to ensure the data is clean and compatible with the Delta Lake schema.

6. **Convert the data to Delta Lake format:** Once the data is cleansed and transformed, load it into the new Delta Lake repository.

7. **Define schema and partitioning:** Define the schema for your Delta Lake tables using Unity Catalog, ensuring that it aligns with the data structure you extracted. Consider partitioning strategies based on the query patterns and performance requirements.

8. **Apply schema enforcement:** Enable schema enforcement in the Delta Lake tables to ensure that only data conforming to the defined schema is written. This helps maintain data consistency and prevents data quality issues.

9. **Validate and test:** Perform comprehensive validation and testing of the converted Delta Lake data. Verify the data integrity, schema correctness, and compatibility with downstream applications or analytics processes.

10. **Migrate applications and processes:** Update or migrate your existing applications, ETL pipelines and analytics processes to consume data from the newly converted Delta Lake. Ensure that they are compatible with the Delta Lake format and take advantage of its capabilities.

11. **Monitor and optimize:** Continuously monitor the performance of your Delta Lake repository and optimize it based on your workload patterns. Leverage Delta Lake's features like Z-Ordering, Data Skipping, and Delta Time Travel to improve query performance. Reach out to your Databricks representative for best practices

QUERY MIGRATION AND REFACTORING

Migrating queries that run as EMR workloads to Databricks should be a relatively straightforward process, given that the underlying Spark APIs are the same between OSS Spark EMR and Databricks. (Please confirm the Spark version bundled with the Databricks Runtime here.)

There are quite a few query optimizations that are available in Databricks Runtime that are not available in EMR. Most of these query optimizations kick in during execution automatically without any changes to the queries themselves.

SKEW JOIN OPTIMIZATION

Databricks Runtime includes a skew join optimization that can be triggered by including skew hints in the queries. If you are on DBR above 7.3 and have Adaptive Query Execution and `spark.sql.adaptive.skewJoin.enabled` this optimization is auto enabled.

Additional details on how to configure skew hints with relation name, skew hints with relation and column name, and additionally with skew values are here.

RANGE JOIN OPTIMIZATION

A range join occurs when two relations/tables are joined using a point in interval overlap condition. The range join optimization support in Databricks Runtime can bring orders of magnitude improvement in query performance with a little bit of manual tuning effort.

Range join optimizations can be applied when:

1. Conditions that can be interpreted as a point in interval or interval overlap range join
2. Values in the range join condition are of numeric type (integer,float,decimal), DATE or TIMESTAMP
3. All join values are of same type

4 The join criteria is an INNER JOIN for point in interval queries or LEFT OUTER JOIN for point value on the left side and RIGHT OUTER JOIN with point values on the right side.

5 Have a 'bin' size parameter — 'bin' is a numeric tuning parameter that splits the range condition into multiple bins of equal size. If the length of the interval is fairly uniform and known, we recommend that you set the bin size to the typical expected length of the value interval. However, if the length of the interval is varying and skewed, a balance must be found to set a bin size that filters the short intervals efficiently, while preventing the long intervals from overlapping too many bins. Additional guidance on choosing the appropriate bin size is here.

A sample range join hint looks like the following; this query specifies 500 as the bin size on the relation 'c', which is involved in interval range join.

```
SELECT /*+ RANGE_JOIN(c, 500) */ *
FROM a
  JOIN b ON (a.b_key = b.id)
  JOIN c ON (a.ts BETWEEN c.start_time AND c.end_time)
```

Other optimizations that automatically trigger when using Databricks Runtime include the following. Please verify that the Databricks Runtime version you are using has these features enabled.

Low shuffle merge: A technique to minimize the amount of data that needs to be shuffled in a merge operation by isolating unmodified rows into a streamlined processing mode rather than with rows that are being modified in the current merge.

Bloom filter indexes: A bloom filter index is a space-efficient data structure that enables data skipping on chosen columns You can set the bloom filter index on a table column at the Spark session level.

Predictive I/O: This is a collection of Databricks optimizations available on the Databricks Runtime with Photon engine. These optimizations use deep learning techniques to:

1. Determine the most efficient access pattern to read the data and only scan the data that is actually used
2. Eliminate decoding of columns and rows that are not required to generate query results
3. Use probabilistic techniques to anticipate the next matching row and only read that data from cloud storage

Predictive I/O also speeds up updates by using deletion vectors. It reduces the frequency of full file rewrites during DELETE, MERGE and UPDATE operations on Delta tables.

MIGRATION VALIDATION

Here are some important steps for migration validation:

1. **Data consistency:** Verify that the data produced by your EMR workflows matches the data generated by the corresponding workflows in Databricks. Compare the outputs of representative jobs or tasks to ensure consistency. This is usually done by comparing the values of important KPIs/columns and row counts. It is a good practice to use automation tools/ frameworks for measuring data consistency.
2. **Performance comparison:** Measure and compare the performance of equivalent jobs or tasks in EMR and Databricks. Monitor metrics such as job execution time, resource utilization and data processing throughput. Identify any significant performance differences and investigate potential optimizations. A combination of appropriate compute resources, Spark configurations, Delta optimizations and Photon engine uplifts the performance significantly.

3. **Functionality testing:** Conduct comprehensive functional testing of your migrated workflows in Databricks. Verify that all job dependencies, transformations and outputs are functioning correctly. Test various scenarios, edge cases and data validation procedures.

4. **Integration and dependencies:** Validate the integration and dependencies of your Databricks Workflows with other systems and services. Ensure that any external data sources, APIs, repos or third-party tools used in your EMR workflows are properly integrated and functional in Databricks.

5. **Job monitoring and alerting:** Set up monitoring and alerting mechanisms within Databricks to track the performance and behavior of your workflows. Configure alerts for job failures, resource utilization thresholds, or any other relevant metrics to proactively identify and resolve issues. Databricks supports cloud native, open source tooling like Prometheus (contact your Databricks representative for the same) and third-party tooling like DataDog for setting up data pipeline observability.

6. **Cost analysis:** Analyze the cost implications of migrating to Databricks compared to your previous EMR setup. Assess the Databricks pricing model, resource allocation and job execution costs. Identify any cost optimization opportunities and fine-tune your Databricks cluster configurations as needed. For TCO calculations and DBU sizing, please reach out to your Databricks representative.

7. **User training and adoption:** Ensure that your team members are trained and familiarized with the Databricks environment. Provide documentation, workshops or training sessions to help them transition smoothly and utilize the Databricks features effectively. Databricks provides both in-person and self-paced training, and it is highly recommended that you leverage these resources at the earliest.

8. **Rollback plan:** Prepare a rollback plan in case you encounter unforeseen issues during migration. Have a backup strategy to revert to your EMR setup if necessary.

9 **User acceptance testing (UAT):** Involve key stakeholders and end users in UAT activities to validate that the migrated workflows meet their requirements. Collect feedback, address any concerns and make necessary adjustments before fully transitioning to Databricks.

By performing thorough validation, you can ensure the functionality, performance and data consistency of your migrated workflows. This will help mitigate risks and ensure a smooth and successful migration from EMR to Databricks.

KEY CONSIDERATIONS

1 **Instance types:** Often the instance types that perform best with EMR are not always the right options for Databricks Runtime. Consider using the 'i' series machines for more general-purpose workloads and compute-optimized instances for streaming workloads. Do refer to Databricks documentation for guidance on how to choose the right instance type and sizes for your workloads.

2 Cluster sizes with Databricks Runtime are often smaller compared to EMR for the same workload because of the optimizations in Databricks Runtime that are not available in EMR/OSS. As part of migration, look at cluster usage metrics (Ganglia, CloudWatch, etc.) and decide on the right size for the cluster.

3 A number of EMR workloads assume the availability of HDFS as a temporary storage as part of query execution. That is, queries either write temporary tables to HDFS to use as input to the next part of the query, or checkpoint a query to disk to break an overly large DAG. This assumption will not work on Databricks since there is no temporary HDFS storage available. Data read from S3 is completely held in memory. If data is read repeatedly from Parquet sources, Databricks will efficiently use the SSD to cache the data to avoid repeated reads.

4 Similarly, if you need to checkpoint a query, consider using S3 as temp storage. That said, reevaluate such queries as part of the migration process to see if you can restructure the queries without needing a checkpoint.

Phase 6: Downstream Tools Integration

Reducing data infrastructure dependencies and maintaining a single source of truth of data has led organizations to use Databricks SQL, which is a data warehousing product in Databricks Lakehouse. This ensures tighter integration between data and downstream applications and dashboards. Databricks SQL is a data warehousing product that has world-class performance for analytics workloads and support for high concurrency. Photon is a query engine that is built from scratch in C++ and vectorizes data to exploit both data-level and instruction-level parallelism.

Once data and transformation pipelines are migrated to Databricks Lakehouse, it's important to ensure business continuity for downstream applications and data consumers. Databricks Lakehouse validated large-scale BI integration with many of the market's most popular BI tools, including Tableau, Power BI, Qlik, ThoughtSpot, Sigma, Looker and more. For a particular set of dashboards or reports to work, it is recommended that you ensure that all upstream tables and views are migrated along with their associated pipelines and dependencies.

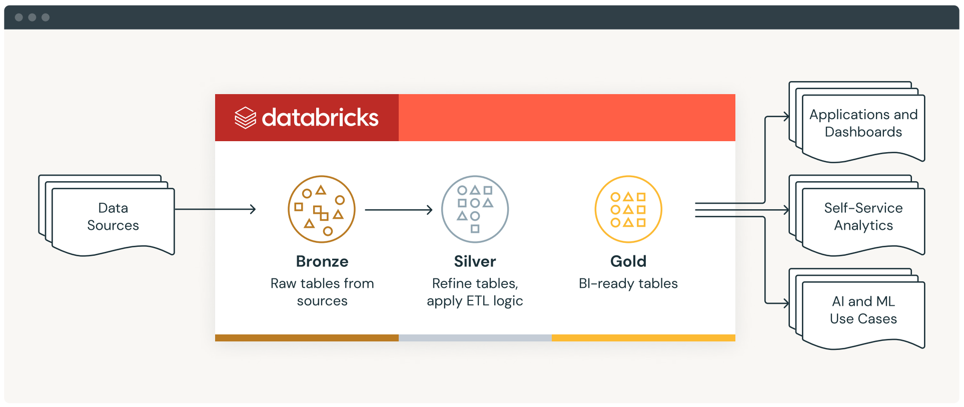


Figure #: Future-state architecture

If the schema of your tables and views has not changed after migration, re-referencing is usually a simple task of switching databases in your BI dashboard tools. If the table's schema changes, you will need to modify the table/view in Lakehouse to match the expected schema of the report/dashboard and publish it as a new data source for the report.

We recommend testing your approach with a small set of dashboards or reports and iterating through the remaining reporting layers during migration. During the report migration, a situation may arise where the BI tool's access to the cloud storage bucket needs to be elevated. This is because Databricks uses "Cloud Fetch" to support high-bandwidth data exchange. In this architecture, BI tools get a signed URL for a specific BI query, so BI tools download data in parallel directly from cloud storage. If the new permissions are not already set, you may need to enable them.

Best Practices

DATABRICKS PLATFORM

It's important to understand some basic concepts used in Databricks before you get started.

- Databricks Interface →
- Cluster Configuration →
- Cluster Policies →
- Data Governance →
- GDPR & CCPA Compliance →
- Delta Lake →
- Structured Streaming →
- CI/CD →

DELTA LAKE AND PERFORMANCE OPTIMIZATION

Optimize the performance of the migrated workload by tweaking the configuration of the Databricks environment and the workload itself. This includes identifying and eliminating any bottlenecks and improving the overall performance. Below are a few best practices to consider during performance tuning.

File Sizing

- Databricks Runtime automatically tunes file sizes based on table size and also based on workload — for example, to accelerate write-intensive operations
- File sizes can be manually adjusted by setting delta.targetFileSize as a table property or Spark configuration

Partitioning

- Avoid partitioning tables < 1TB
- Ideal size of partitions is > 1GB
- Use generated columns to avoid over-partitioning
- Partition on lower cardinality columns

Data Skipping

- Statistics will be automatically computed for you to facilitate data skipping
- Tracks file-level statistics like min, max, etc.
- Helps avoid scanning irrelevant files/data
- By default, Databricks Delta collects statistics on the first 32 columns defined in the table schema. This default value can be updated using the table property, `delta.dataSkippingNumIndexedCols`
- A best practice to keep in mind is to move numerical columns and high cardinality query predicates to the left of the 32nd ordinal position, and move strings and complex data types after the 32nd ordinal position of the table

Z-Ordering (Clustering)

- Effective on up to 3-5 columns
- Z-order on higher cardinality columns, columns for Z-ordering must be in the first 32 columns

Merge/Upsert

- Ensure you are using DBR 10.4+ to take advantage of Low Shuffle Merge
  - Avoids write amplification due to merge's use of fullOuterJoin
- With Low Shuffle Merge, fullOuterJoin is broken into an inner and leftOuterJoin followed by read > filter > write using file + rowId map
  - This helps optimize merge performance significantly

Generated Columns

- Special column type that gets defined based on a user-specified function over other columns in a Delta table
- Values for generated columns are computed at runtime
- Generated columns allow users to avoid over-/under-partitioning

Join Strategies

- Broadcast Hash Join / Broadcast Nested Loop Join
  - Requires one side of the join to be small
  - No shuffle, no sort, very fast
- Shuffle Hash Join
  - Needs to shuffle data, but avoids sort
  - Handles large tables, but will result in an out-of-memory error if data is skewed
- Sort Merge Join
  - Handles any data size
  - Requires shuffle and sort
  - Slower in most cases when table size is small due to excessive shuffle
- Shuffle Nested Loop Join/Cartesian Product
  - Does not require join keys
  - Extremely heavy operation; avoid at all costs if possible

Query Profile

- In the case of data warehouse usage, the SQL warehouse query profile is a powerful tool located inside the Databricks SQL workspace. Its objective is to troubleshoot slow-running queries, optimize query execution plans and analyze granular metrics to see where compute resources are being spent.
- The query profile provides value in these three capability areas:
  - Detailed information about the three main components of query execution, which are time spent in tasks, number of rows processed and memory consumption
  - Two types of graphical representations. A tree view to easily spot slow operations at a glance, and a graph view that breaks down how data is transformed across tasks.
  - Ability to understand mistakes and performance bottlenecks in queries
- Three common performance bottleneck problems surfaced by query profile are listed below:
  - Inefficient file pruning
  - Full table scans
  - Exploding joins (Cartesian product)

Analyze Table

- The ANALYZE TABLE command collects statistics on tables in Databricks and ensures that the query optimizer finds the most optimal query execution plan. SQL syntax is as follows:

```
ANALYZE TABLE my_table COMPUTE STATISTICS for COLUMNS col1, col2, col3
```

- One important point to remember is that you will want to prioritize statistics for columns that are frequently used in joins and other query predicates
- Best practice is to run ANALYZE TABLE as a separately scheduled job on a regular cadence (e.g., weekly or monthly)

GOVERNANCE AND SECURITY

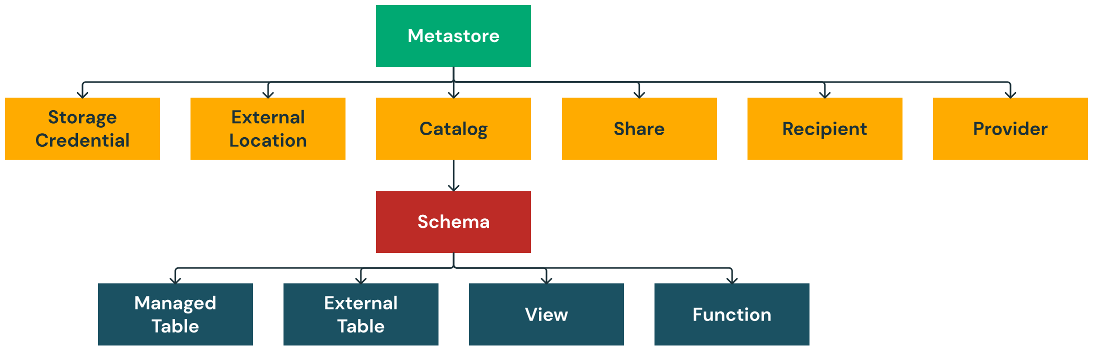
Reference Materials:

- Data Governance Guide →
- Unity Catalog →

Identity Management

- Identities exist at the Databricks account level. Identity federation allows for these account-level identities to be federated downward to workspaces.
- Single sign-on (SSO) can be set up to manage account-level identities
- Identity Types
  - Users
  - Groups
  - Service Principals

Privileges and Securable Objects



- Securable Objects →
- Inheritance Model →
- Privileges Types →

Compute

- Clusters & SQL Warehouses with Unity Catalog →

Need Help Migrating?

Regardless of size and complexity, the Databricks Professional Services team, along with an ecosystem of services partners and ISV partners, offers different levels of support (advisory, staff augmentation, scoped implementation) to accelerate your migration and ensure successful implementation. Aside from steps outlined in this migration guide, the services offered can include architecture design workshops, Databricks foundation setup, change management, cutover operations, and more.

To migrate Hive workloads running on EMR to DBSQL/Spark SQL, you have the option of utilizing tools like BladeBridge. Databricks, in collaboration with BladeBridge, has created automated tools for evaluating code complexity and migrating code (DDLs, DMLs) to adhere to best practices on Databricks Lakehouse. Moreover, Databricks partners have developed various automation tools to expedite the migration process.

Contact your Databricks representative or reach out to us using this form for more information. Rest assured that we can work with you and make your migration successful.

Appendix

APPENDIX 1: SPARK CODE DEVELOPMENT ON DATABRICKS

Users submitting Spark jobs to a EMR cluster via JAR files and scripts each get their own SparkContext, whereas Databricks shares a single SparkContext among all users on a Databricks cluster. When running a job on Databricks — either via Databricks notebooks or by uploading your own Java/Scala JAR files or Python scripts to DBFS (individual Python scripts or wheel or egg files) — the SparkContext is created for you. Since Databricks initializes the SparkContext, if you invoke a new SparkContext, your code will fail. For example, the following code will return an error:

```
from pyspark import SparkConf, SparkContext

conf = (SparkConf()
          .set("spark.executor.memory","2g"))
sc = SparkContext(conf = conf)
```

```
ValueError: Cannot run multiple SparkContexts at once; existing
SparkContext(app=Databricks Shell, master=spark://10.0.241.108:7077) created by __
init__ at /local_disk0/tmp/1585629397371-0/PythonShell.py:1335
```

Use the shared SparkContext created by Databricks:

```
%python
mySparkContext = SparkContext.getOrCreate()
mySparkSession = SparkSession.builder.getOrCreate()
```

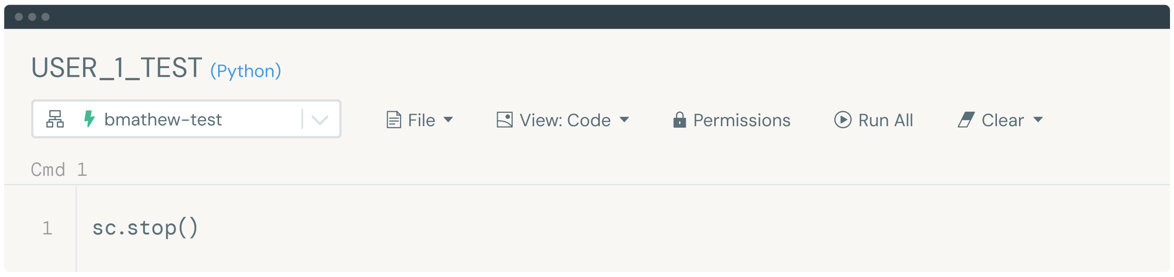
Returning to our example, we would modify the code:

```
from pyspark import SparkConf, SparkContext

conf = (SparkConf()
          .set("spark.executor.memory","2g"))
sc = SparkContext.getOrCreate(conf = conf)
```

Since Databricks creates a shared SparkContext for the cluster, you should not terminate the SparkContext as this could impact other users who are running jobs on the same cluster. For example, let's look at two practitioners using the same cluster.

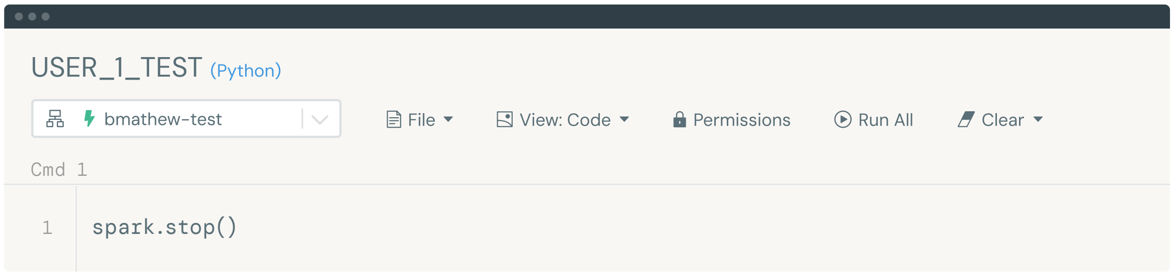
User 1 terminates the SparkContext by issuing these commands:



```
sc.stop()
```

```
The spark context has stopped and the driver is restarting. Your notebook will
be automatically reattached.
```

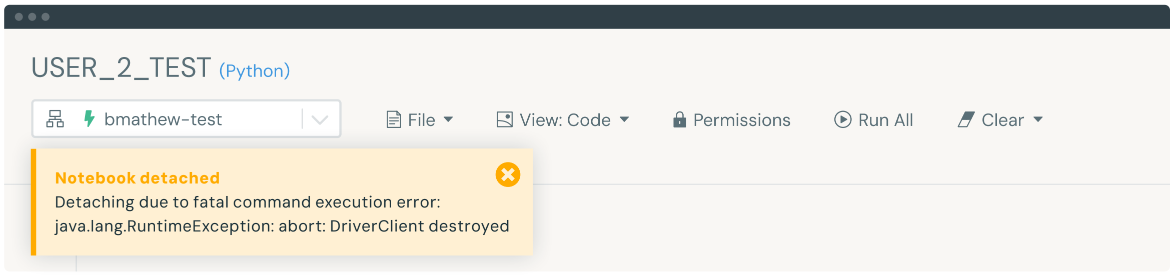
Or



```
spark.stop()
```

```
The spark context has stopped and the driver is restarting. Your notebook will
be automatically reattached.
```

User 2 is trying to run a job but now receives an error message because the SparkContext has stopped:



Notebook detached
Detaching due to fatal command execution error: java.lang.RuntimeException: abort: DriverClient destroyed

```
Internal error, sorry. Attach your notebook to a different cluster or restart the
current cluster.
```

Your job will run normally; however, it will end with the failure above.

For more information, please refer to the following documentation.

databricks

Databricks is the data and AI company. More than 9,000 organizations worldwide — including Comcast, Condé Nast, and over 50% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.