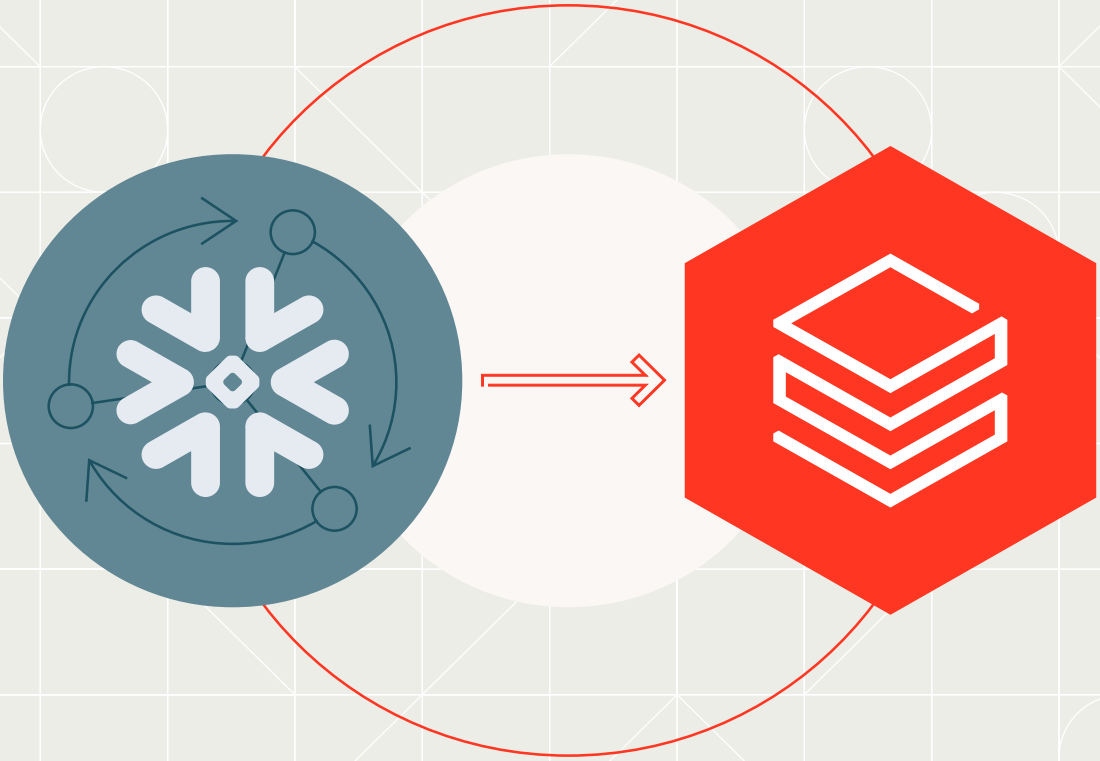
databricks

Guide

Snowflake to Databricks Migration Guide



databricks

Table of Contents

databricks

Table of Contents

databricks

Preface

The purpose of this document is to provide an overview of the process of migrating workloads from Snowflake 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.

Migration Strategy

When migrating from Snowflake 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, workload types, and migration goals. The strategy employed in the migration process is influenced by several key factors, including the urgency and timelines, workload dependency (integrated vs. isolated pipelines), shared vs. isolated warehouses, current architectural limitations, road map backlog, new business 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, such as use case, schema, data mart, or data pipeline. It is recommended to adopt a phased migration approach to mitigate risks, learn and identify patterns, and show progress early in the process. From a high-level strategy perspective, there are two popular approaches to migration.

1. Lead with migrating data ingestion and transformation workloads by landing all data in cloud storage (Amazon S3, Azure Data Lake Storage Gen 2, or Google Cloud Storage), taking advantage of the commoditized cloud storage, in an open table format like Delta Lake or Iceberg. Perform ETL — on both batch and streaming data — using Databricks and move cleaned, aggregated, ready-to-serve data to consumers.

   On top of this there are plenty of features in Databricks Lakehouse to support end-to-end ETL orchestration using Lakeflow Spark Declarative Pipelines (SDP) and Lakeflow Jobs, unified governance and data lineage using Unity Catalog, and cross-organization data collaboration using Delta Sharing.

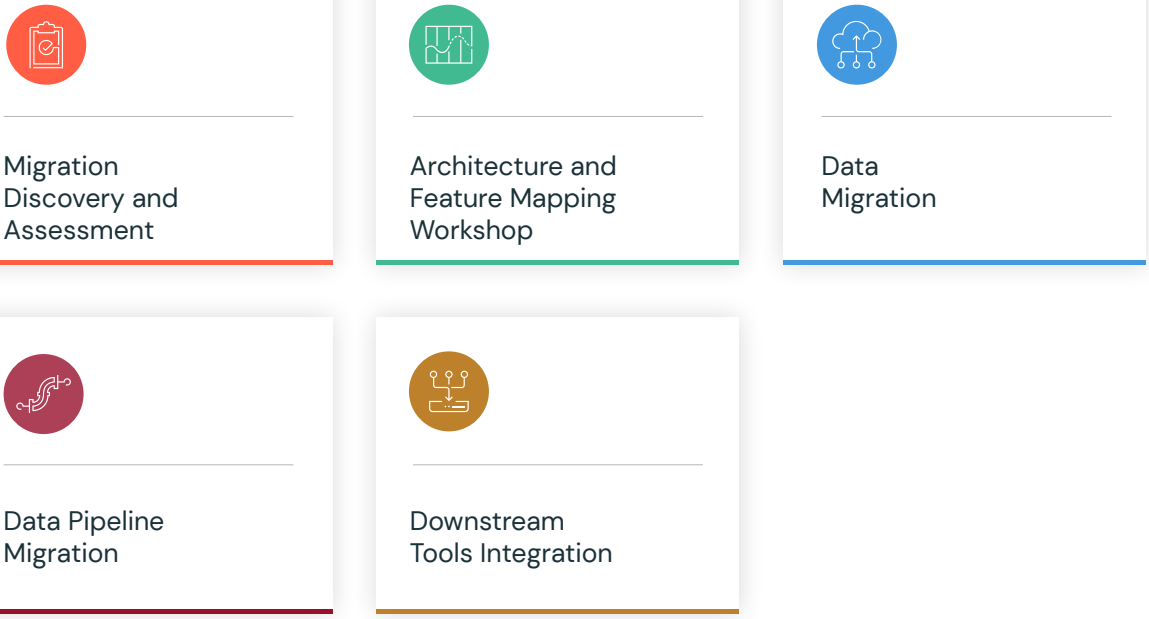
   This approach allows you to use Snowflake, in the interim, for serving downstream applications and BI dashboards. This way you could minimize the disruption to end users and slowly migrate downstream applications to the data consumption (Gold) layer on Databricks eventually.

2 | Lead with modernizing the reporting layer by replicating the data warehouse "Gold/presentation" layer tables from Snowflake in Databricks. This quickly unlocks the value by breaking data silos and enabling cross-functional reporting and cross-functional data science projects. Then work on reconfiguring the ETL in Databricks and cut off the ingestion and transformation in Snowflake. Snowflake's read and write support for externally managed Iceberg tables greatly simplifies the migration journey by supporting both platforms simultaneously.

In the next few sections, we will dive into the migration process, focusing on the approach and leading with the first approach described above: migrating ETL workloads first and then migrating BI workloads.

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 Snowflake are relatively easy because of ANSI SQL support in Apache Spark™ and both platforms being cloud-based managed applications, a lot of concepts are similar. The migration process typically consists of the following steps, but can vary depending on the situation and needs:



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 scale out to more developers and consumers. Databricks Academy is a good place to get started with some self-learning.

In general, migrating from one platform to another 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 Snowflake to Databricks. The Databricks Brickbuilder Solutions and Accelerators have preferred partners who have demonstrated a unique ability in migrating Snowflake workloads to lakehouse successfully.

Phase 1: Migration Discovery and Assessment

Before migrating any data or workloads, one or more migration assessments should be conducted in order to:

- Understand the types of workloads (ETL, BI, ingress/egress, etc.) and their size by warehouses and databases
- Understand the scope of data and queries/workloads to be migrated
- Understand the upstream and downstream technologies and applications involved in the architecture
- Understand the current security setup and protocols
- Understand the level of effort required to complete the migration
- Collect information relevant to developing estimates for infrastructure costs and person-hour

Databricks strongly recommends using automation tools (**Snowflake Profiler** and/or the **Lakebridge Code Analyzer**) to expedite gathering migration-related information.

> Snowflake Profiler is a tool to estimate a Total Cost of Ownership (TCO) of the Snowflake to Databricks migration and helps to highlight the financial advantages of migrating to Databricks. It reads system tables in Snowflake (e.g., snowflake.account_usage.query_history) and other warehouse tables, and returns insights such as workload types, long-running ETL queries, and surface background optimization costs. This analysis provides guidance on identifying warehouses/databases/DAGs contributing to high costs and supports prioritization. The profiler classifies queries by complexity, evaluates function compatibility, and extracts information for data migration (clustering keys, table size).

Lakebridge Analyzer tool helps with migration complexity estimates and it may be also used to estimate cost associated with the migration itself. Analyzer is also used to get deeper insights on data types, DDL (data definition language), DML (data manipulation language) code complexity, and data dependencies. This tool parses Snowflake SQL code and generates:

- An inventory of the code base: Table DDLs, Views, Stored Procedures, Functions, Tasks, Sequences, etc.
- Complexity of each script categorized from low to very complex based on the number of statements
- List of data types and functions
- List of code and table cross-references that can provide support in understanding a table's popularity

The outcome of this phase is a migration scope covering information across queries/workloads, data/tables, and usage of any Snowflake proprietary features/tools.

This is also a great time to evaluate and prioritize the effort and determine which parts of your Data Warehouse will benefit the most and potentially discard unused or legacy reporting and analytics that may have become obsolete but still exist in the environment. Databricks Specialists (SSA) can help at this stage.

Phase 2: Architecture and Feature Mapping Workshop

When planning your Snowflake migration, it is important to correctly map Snowflake capabilities to Databricks Lakehouse capabilities. The second phase of the migration journey is understanding the current-state architecture and Snowflake features in order to map them to the future-state architecture and Databricks features. In this phase, we design the ideal lakehouse architecture that serves all the data and AI use cases, and design how each component of the current architecture needs to be modernized to map to the target architecture.

We will then compare/contrast current- and future-state architecture diagrams and align on the intermediate phases of the architecture as the migration progresses through each phase. As an example, the intermediate architecture will require running ETL pipelines in both Snowflake and Databricks in parallel for some period of time, so we will want to architect an optimal solution to ensure data stays in sync and external systems/tooling can continue to function without impact to the rest of the organization.

Following architectural alignment, we will conduct a deep dive into all features currently used in Snowflake. Phase 1 will help surface this information, but Phase 2 will dive deeper into how they are used to ensure each existing Snowflake feature/functionality is mapped to the appropriate Databricks feature/functionality.

Snowflake vs. Databricks Feature Mapping Example

FEATURE	SNOWFLAKE	DATABRICKS
Compute	One type of compute for all workloads called Snowflake Virtual Warehouse	Serverless SQL Warehouses Serverless Compute for Notebooks/Jobs Classic Compute • All-purpose clusters • Job clusters • SDP clusters • Classic SQL Warehouses
Storage	Snowflake's own storage layer	Cloud storage (Amazon S3, Azure Data Lake Storage Gen2, Google Cloud Storage)
Format	Snowflake FDN format (proprietary compressed format) Open source Iceberg format (Parquet)	Open source Delta and Iceberg format (Parquet)

FEATURE	SNOWFLAKE	DATABRICKS
Architecture Layers	Cloud services layer Query processing Database storage	Control plane Data plane
Stages/Tables	External stage Internal stage External tables Internal tables	External tables Managed tables Lakehouse Federation
Interface	Snowpark Snowsight SnowSQL CLI	Databricks collaborative notebooks Databricks SQL workspace Databricks CLI
Database Objects	Tables, temporary tables, transient tables, views, materialized views, dynamic tables Hybrid tables Snowpark notebooks Stored procedures UDFs	Tables, temporary views, views, materialized views, streaming tables Lakebase Databricks notebooks Stored procedures UDFs
Metadata Catalog	Horizon Catalog Snowflake Open Catalog (based on Apache Polaris)	Unity Catalog
Data Ingestion	COPY INTO Bulk Loading Snowpipe Snowpipe Streaming Snowpark Load data UI Integrations via Partner Connect	COPY INTO CONVERT TO DELTA Lakeflow Spark Declarative Pipelines Auto Loader Spark Streaming DataFrame Reader Add data UI Integrations via Partner Connect
Data Types	Data Types in Snowflake	Data Types in Databricks
Workload Management	Load monitoring chart, custom query tags	Cluster configuration (policies), Prometheus and Grafana supported metrics
Security	System defined roles and custom roles. Hierarchy of roles Table-/column-/row-level security	System roles: account admins, workspace admins, metastore admin, account users and workspace users Users, groups and service principals Object based controls using access control lists (ACLs) Table-/column-/row-level security

FEATURE	SNOWFLAKE	DATABRICKS
Data Clustering	Clustering	Liquid Clustering
Programming Language	SQL, Python, Scala, Java, JavaScript,	External tables Managed tables Lakehouse Federation
Data Integration	Open Flow External tools (dbt, Fivetran, Airbyte, Matillion, Talend, Pentaho, Informatica, etc.)	Lakeflow Connect External tools (dbt, Fivetran, Airbyte, Matillion, Prophecy, Informatica, Talend, etc.)
Orchestration	Snowflake Tasks External third-party tools (e.g., Airflow)	Databricks Workflows External third-party tools (e.g., Airflow)
Machine Learning	Python tools on Snowpark, external third-party tools (e.g., Alteryx, DataRobot)	Databricks ML (Runtime with OSS ML packages, MLflow, Feature Store, AutoML)
Change Data Capture	Snowflake Streams	Delta Change Data Feed Lakeflow Spark Declarative Pipelines
Time Travel	Snowflake Time Travel	Delta Time Travel
Data Sharing	Snowflake Secure Data Sharing Snowflake-to-Snowflake Snowflake Marketplace	Delta Sharing • Databricks-to-Databricks • Databricks open sharing Databricks Marketplace
Pricing Unit	Snowflake credits Snowflake storage	Databricks units (DBUs)

The table above is an example of mapping key features between Snowflake and Databricks. A similar exercise comparing all relevant features for your environment should be performed in this step.

Typically, by the end of this phase we have a good handle on the scope and complexity of the migration, and can come up with a more accurate migration plan and cost estimate.

Phase 3: Data Migration

Before you start the migration process, one or more Databricks workspaces will need to be provisioned if not available. The decision to create one or more workspaces typically revolves around the following considerations:

- **Separation of environments:** Requiring different workspaces for development, staging, production, and other environments
- **Separation of business units:** Requiring different workspaces for marketing, finance, risk management, and other departments
- **Implementation of modern data architectures:** Requiring different workspaces to support modern data architectures, such as Data Mesh architecture, to decentralize data ownership for different domains.

Once the workspace is set up, the first step of the migration is to migrate the data. Databricks is optimized for cloud object storage: Amazon S3, ADLS2 and Google Cloud Storage. In addition to cloud storage, Databricks can read/write from other storage endpoints, including relational databases (Oracle, SQL Server, Teradata, etc.), HDFS, Apache Hive, NoSQL (HBase, Cassandra, Neo4j, MongoDB), in-memory cache (Redis, RocksDB), message bus (Kafka, Kinesis), files (delimited text files, JSON, Parquet, ORC, Avro), JDBC/ODBC sources/sinks, and many more.

Lakeflow Connect (LFC) offers a number of connectors from databases like SQL Server, PostgreSQL, MySQL/MariaDB, Oracle, Big Query, etc. and tools/platforms like Salesforce, Workday, ServiceNow, Dynamics360 etc. It includes the Query Based (QBC) connector to Snowflake. All LFC connectors are based on Lakehouse Federation (LF) and Lakeflow Spark Declarative Pipelines (SDP) technologies and provide CDC functionality.

Lakebase is the serverless Postgres database integrated with the Lakehouse. Lakebase complements Delta Lake helping with migration of operational/serving workloads that need PostgreSQL compatibility and low latency. Some Snowflake workloads are actually transactional (OLTP) rather than analytical (OLAP). Such workflows can be migrated to Lakebase, for example Hybrid tables (OLTP) and Low-latency lookups. ML Feature Serving can be migrated to Feature Store and/or Lakebase.

When migrating data out of Snowflake, there are key decisions that need to be made. Some of these could include:

- What is the target design for the tables being migrated?
- Should the destination retain the same hierarchy of catalogs, databases, schemas and tables?
- Should there be any cleanup or organization of the existing data footprint in Snowflake?
- Using Snowflake's interoperability and support for externally managed Iceberg tables.

Once these are decided, the data migration can proceed. Generally speaking, we do not recommend introducing many changes in the schema structure during migration. Given the Schema Evolution capability in Delta, it is a common practice to evolve the schema after it is put in Delta. This approach also allows us to easily compare the data in Snowflake and Databricks during parallel runs. As investment continues in open formats such as Delta and Iceberg, and Snowflake's capabilities for externally managed Iceberg tables evolve, opportunities to simplify the migration and interoperability will only become more common.

RECOMMENDED APPROACH

It is important to note that not every data migration will follow the same pattern, but Databricks recommends the following flow:

1. Migrate EDW (enterprise data warehouse) and staging tables (optional) into Delta Lake medallion data architecture
2. Migrate/build data pipelines that incrementally populate Bronze/Silver/Gold in Delta Lake
3. Backfill Bronze/Silver/Gold tables as needed

Typically, data in a Snowflake ELT pipeline moves through a staging or landing zone (Bronze), then an optional integration layer (Silver), and ends up in a data model in an EDW/reporting layer (Gold). Whatever data model (dimensional model, data vault, etc.) is implemented in Snowflake can be implemented in the Databricks Lakehouse in a more performant manner using Delta Lake. Data architecture in Databricks Lakehouse follows Delta Lake — Bronze, Silver, Gold paradigm — and the data model belongs in the Gold layer.

SCHEMA MIGRATION

- Before the tables are offloaded to Databricks, the schema of the tables must be created in Databricks. The recommended way of doing that is by using Lakehouse Federation (LF) or Lakeflow Connect (LFC), in this case tables schemas will be migrated automatically.
- Other option: if you have DDL scripts, you could leverage that with some tweaks to data types used in the DDLs. DDLs can also be extracted from Snowflake using the get_ddl function. Once the scripts are extracted, automation tools (e.g., Lakebridge Converter) can handle the conversion of Snowflake DDLs to Databricks DDLs. Refer to the guidance around matching data types in both Databricks and Snowflake in Appendix 2.

DATA MIGRATION APPROACHES

In terms of implementation, several approaches have been validated by Databricks for migrating the data and are already in production use by various customers. For the initial Gold table migration, options include:

1. Snowflake's support for Iceberg tables and Databricks Apache Iceberg reads provide a powerful way to leverage existing routines and interoperate with both platforms while migrating workflows from one environment to the other.
    - Leverages both of the company's investments in supporting Iceberg and external and managed catalogs.
    - Provides a single copy of data until workloads are cutover.
2. Lakeflow Connect (LFC) Snowflake connector (Query-Based) for direct incremental ingestion. This connector requires Foreign Catalog to be created based on External Connection to Snowflake — refer to Appendix 4.

3 | Leveraging Lakehouse Federation (LF) to read data from Snowflake and create external catalogs in UC.

- Lakehouse Federation allows Databricks to query external data sources, including Snowflake, without moving data. This is valuable during migration for maintaining business continuity.
- LF bypasses the write to Parquet or Iceberg, but requires running Snowflake virtual warehouses and Databricks clusters in parallel, which can be costly
- The preferred method to map metadata from Snowflake to Databricks
- LF allows gradual cutover:
  - Migrate tables incrementally while maintaining full data access
  - Create views that union migrated and non-migrated data
- LF is useful for dependency management:
  - Keep downstream reports working during migration
  - Query Snowflake for tables with complex dependencies not yet migrated
- This method is commonly used for small tables up to 15GB in size and a manageable number of tables, but for larger tables, the first approach above is recommended. LF is for validation and transition, not production workloads

4 | Traditional but still widely used methods:

- Leveraging Snowflake's COPY INTO command to push data out of Snowflake and into cloud storage in Parquet format (file stage) then write to Delta Lake format tables using one of the following options:
  - Auto Loader
  - Databricks COPY INTO command
  - Spark batch/streaming APIs
- Leveraging Databricks Spark Connector from Spark code, often in combination with Snowflake Connector for Python

5 | Leveraging data ingestion partners from Databricks Partner Connect

After the initial Gold table offload, recurring jobs should be set up to continuously sync data from Snowflake to Databricks for those Gold tables until data pipelines are migrated to Databricks and are feeding data to those Gold tables. For continuous replication and real-time sync, options include:

- Leveraging Snowflake's support for externally managed Iceberg tables and Databricks Unity Catalog's ability to allow external engines, like Snowflake, read and write to Unity Catalog-managed Iceberg tables.
- Leveraging the Lakeflow Connect to ingest data from Snowflake in a CDC (change data capture) fashion
- Leveraging partner CDC tools like Fivetran, Airbyte, etc. Repointing dbt models to write to Databricks.
- Leveraging Lakehouse Federation for metadata and smaller tables

As you go through the migration, the current architecture slowly changes as you start offloading data and workloads in a phased manner. Figure 3 shows the architecture during the data migration phase.

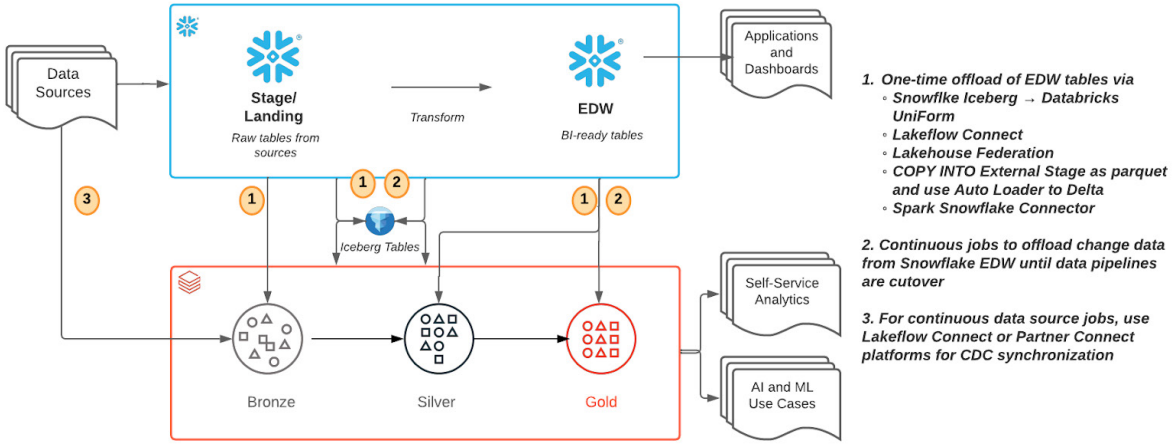


Figure 1: Transient state architecture: post-data migration

After validation and testing, the Bronze and Gold layer data are available for immediate use in Databricks by end users for ad hoc analysis and machine learning, while data pipelines are being offloaded to Databricks in parallel. This is the advantage of the lakehouse architecture: to make the data available for different use cases instantly without having to move data around.

KEY CONSIDERATIONS: SCHEMA MIGRATION AND DATA MIGRATION

1. **Data Modeling:** As part of the migration, there might be a need to refactor the data model or reproduce a similar data model in an automated and scalable manner. It can be done with the help of AI tools or visual data modeling tools from Databricks Partner Connect. All those approaches can help to reverse engineer a Snowflake data model (table structures) and implement them in Databricks easily or just to migrate the visual data models.

2. When migrating DDLs of a table, and for all auto-migrated by LF, LFC, AI or partner tools tables it is important to check the schema of the source data. Migrated schema quality assurance is important, the incorrectly converted column types in DDL generated from the Snowflake table may cause unnecessary casting on our ingestion pipeline once after the data is migrated from Snowflake.

3. In Snowflake and Databricks, keeping schemas in sync during the transient stage can become critical if there are changes introduced to table schemas during migration. Approaches such as making a change in a central Master Data Model (MDM) first and then implementing it in both Snowflake and Databricks are gaining traction.

Phase 4: 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 Snowflake 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 Snowflake, the orchestration is typically done using external tools such as dbt, Airflow, Matillion or by using Snowflake Tasks. There are generally two options when migrating these workflows.

1. Use Lakeflow Jobs to orchestrate the migrated pipelines. In addition, Lakeflow Spark Declarative Pipelines (SDP) can be used for building reliable and efficient data processing pipelines. Using SDP 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. Tasks in Snowflake get created as Lakeflow Jobs. It also has out-of-the-box functionality for SCD Type 1 and Type 2 tables.

2. It is possible to use the external tools like Airflow for orchestration and repoint these tools from Snowflake compute to Databricks compute. You would be just translating Snowflake SQL queries to Spark SQL queries in the orchestration job while retaining most of the orchestration elements.

SOURCE/SINK MIGRATION

Similar to orchestration, in most cases an external ETL tool is seen in Snowflake architecture to extract data from source systems, transform (optional) and load it into the Snowflake staging layer.

1 | Source data connections

- Ingestion pipelines using Snowpipe are replaced with Databricks Auto Loader, Spark Structured Streaming or Spark DataFrame APIs. Lakeflow Spark Declarative Pipelines (SDP) supports Auto Loader and Spark DataFrame APIs, the output of the SDP pipelines is Streaming Tables and/or Materialized Views.
- Zerobus Ingest ± set of APIs and SDKs that you can use to push data to managed tables. This is different from other LFC connectors, which are usually pull-based. Zerobus Ingest aims to simplify writes where applications do not have the ability to buffer and write directly to object storage. Ex. IoT devices may not have the ability to buffer data, or you want to offload data coming from a web application, which is an ephemeral application.
- Ingestion pipelines using tools such as Fivetran and Qlik Replicate can be duplicated and configured to point to Databricks Delta Lake instead of Snowflake staging. Delta is an open source format and widely supported as the target data format for popular data ingestion tools. Though, with Lakeflow connect (LFC) Snowflake connector and Lakeflow Federation it became possible to move from Partner Connect solutions to Databricks native CDC solution completely.
- Native Spark integrations (e.g., Kafka) are leveraged to refactor the streams

2 | Sink data connections

- Ingestion tools and framework will now generate data in Delta Lake format instead of Snowflake format

3 | Snowflake Streams and CDC Migration

- Option 1: Lakeflow Spark Declarative Pipelines (SDP) with CDC
  - Auto-CDC, SCD Type 1 and 2, Apply Changes, Apply Changes from Snapshot
- Option 2: Structured Streaming with Delta. This approach:
  - Automatically tracks progress via checkpoints
  - Processes changes incrementally
  - Integrates with SDP Pipelines
- Option 3: Delta Change Data Feed (CDF)
  - Delta CDF tracks row-level changes (inserts, updates, deletes) on Delta tables. Change Types Returned:
    - _change_type = 'insert': New rows
    - _change_type = 'update_preimage': Row before update
    - _change_type = 'update_postimage': Row after update
    - _change_type = 'delete': Deleted rows

QUERY MIGRATION AND REFACTORING

Queries here refer to any DML query that transforms the data or to ad hoc data analysis queries run by the user for data analysis. The interface from where queries are initiated could be directly in Snowflake or coming from an external ETL tool such as dbt or Matillion.

In situations where SQL queries are triggered from an external ETL platform such as dbt, the refactoring is straightforward, especially for user-written SQL queries. Any Snowflake-specific integration features should be refactored, which in most cases is equivalent to tweaking underlying Snowflake SQL query.

Migrate from Snowflake SQL to Databricks SQL, identifying and replacing any incompatible/proprietary Snowflake SQL functions or syntax. A few options for tackling this are:

- (Recommended) Use Lakebridge Converter to automate lift-and-shift portion of query migration
- (Not recommended) Develop custom script in-house to convert Snowflake SQL to Databricks SQL
- (Not recommended) Manually convert Snowflake SQL to Databricks SQL

Given that both Snowflake and Databricks support ANSI SQL standards, a large portion of the Snowflake SQL query can be automatically converted to Databricks syntax to accelerate the migration. The Lakebridge conversion tool supports schema conversion (tables and views), SQL queries (select statements, expressions, functions, user-defined functions, etc.), stored procedures and data loading utilities such as COPY INTO. The conversion configuration is externalized, meaning conversion rules can be extended by users during migration projects to handle new code pattern sets to achieve a greater percentage of automation.

Refer to Appendix 3 for some examples on SQL translation differences between Snowflake and Databricks. Check out this handy reference guide to help you get started on Databricks using SQL programming in no time!

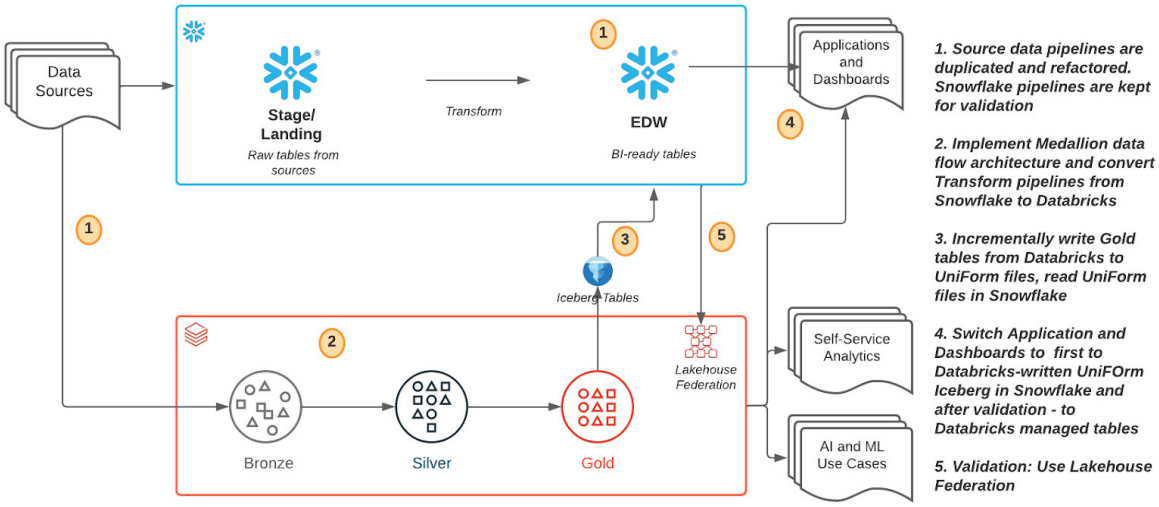


Figure 2: Transient state architecture during pipeline migration

STORED PROCEDURE MIGRATION

Databricks supports SQL stored procedures (DBR 14.0+) with procedural logic, enabling closer 1:1 migration from Snowflake. For complex procedures, Python UDFs or notebooks may be more appropriate.

PROCEDURE CHARACTERISTICS	RECOMMENDED DATABRICKS TARGET	RATIONALE
Simple SQL DML (INSERT/UPDATE/DELETE)	SQL Stored Procedure	Direct 1:1 migration
SQL with basic control flow (IF/LOOP)	SQL Stored Procedure	DBR 14.0+ supports procedural SQL
JavaScript with data processing	Python UDF or Notebook	Python is a better match to convert Java Script than SQL
JavaScript with API calls	Python Notebook + Workflows	External integrations need orchestration
Complex multi-step ETL	Lakeflow Spark Declarative Pipelines	Better monitoring, recovery, lineage
Procedures calling other procedures	Databricks Jobs	Orchestrate notebook/task dependencies
Scheduled batch procedures	Lakeflow Jobs	Native scheduling with alerting
Real-time/event-driven procedures (Snowpipes/Streams/Tasks)	Structured Streaming	Continuous processing model

Optimization Recommendations

- Use DataFrame operations instead of SQL loops
- Cache intermediate results
- Use broadcast for small lookup tables
- Partition operations appropriately

Migration Checklist for Stored Procedures

- Pre-Migration:
  - Inventory all stored procedures in Snowflake
  - Classify by complexity (simple SQL, JavaScript, multi-step)
  - Document dependencies between procedures
  - Identify procedures with external API calls
  - Document transaction requirements
  - Identify scheduling/trigger patterns
  - Gather test cases and expected outputs
- Migration:
  - Set up target catalog/schema structure in Unity Catalog
  - Migrate simple SQL procedures first
  - Convert JavaScript procedures to Python UDFs/notebooks
  - Implement complex procedures as Jobs
  - Set up error handling and logging
  - Configure monitoring and alerting

DYNAMIC TABLES MIGRATION

Snowflake Dynamic Tables provide automated, incremental data transformation with declarative SQL definitions. Databricks Lakeflow Spark Declarative Pipelines (SDP) offers equivalent and enhanced capabilities for building reliable, maintainable data pipelines.

Mapping

SNOWFLAKE DYNAMIC TABLES	DATABRICKS SDP	NOTES
Dynamic Table	Materialized View or Streaming Table	Depends on refresh pattern
TARGET_LAG	Pipeline trigger interval	Controls data freshness
DOWNSTREAM refresh mode	Automatic dependency management	SDP handles automatically
INCREMENTAL refresh	Streaming Tables/Materialized Views	Automatic incremental processing
FULL refresh	Streaming Tables/Materialized Views	Complete recomputation
Warehouse assignment	Serverless or Pipeline cluster	Compute isolation

Key Advantages of SDP over Dynamic Tables

- Built-in Data Quality: Declarative expectations with configurable violation handling
- Native SCD Support: APPLY CHANGES for Type 1 and Type 2 slowly changing dimensions
- Unified Batch/Streaming: Same syntax for batch and streaming workloads
- Enhanced Observability: Built-in lineage, monitoring, and event logging
- Schema Evolution: Automatic handling of schema changes
- Development Mode: Test pipelines without affecting production data
- Recovery and Replay: Automatic checkpoint management and failure recovery

Migration Patterns by Use Case

- Simple Aggregation Dynamic Table → Materialized View
- Multi-Stage Pipeline (Bronze → Silver → Gold)
- SCD Type 2 with Dynamic Tables → SDP APPLY CHANGES
- Dynamic Table with Data Quality → SDP Expectations

MIGRATION VALIDATION

Data/Metadata

- Validation is mostly done around the data in both the platforms.
- Establish a unit testing for sink-to-sink comparisons. As there might be thousands of tables migrated, it is manually impossible to compare the data values in Snowflake and Databricks. Generally a testing framework with a script to compare values automatically in both the platforms is used. Some example data points to compare include:
    - Check if the table exists
    - Check the counts of rows and columns across the tables
    - Calculate the sum of numeric columns and compare
    - Calculate the distinct count of values in string columns and compare
- Run the pipelines in parallel for a week or two and review the comparison results to ensure the data is flowing correctly.
- Use Lakebridge validation Reconcile tool to validate the migration and to get a Daily Data Validation Issues Report
- For more advanced table data and schema comparison, tools like Datacompy can be used.
- Lakehouse Federation Parallel Validation:
    - LF helps to compare data between migrated Delta tables and source Snowflake tables
    - Allows to run validation queries that join both sources
- Delta Sharing also sharing Iceberg tables with Snowflake and provides direct access to foreign Iceberg tables from Snowflake among other sources

Stored Procedures/Query

- Run unit tests for each migrated procedure
- Compare outputs with Snowflake (parallel testing)
- Validate transaction behavior
- Performance benchmark against Snowflake and Optimization
- Validate scheduling and triggers

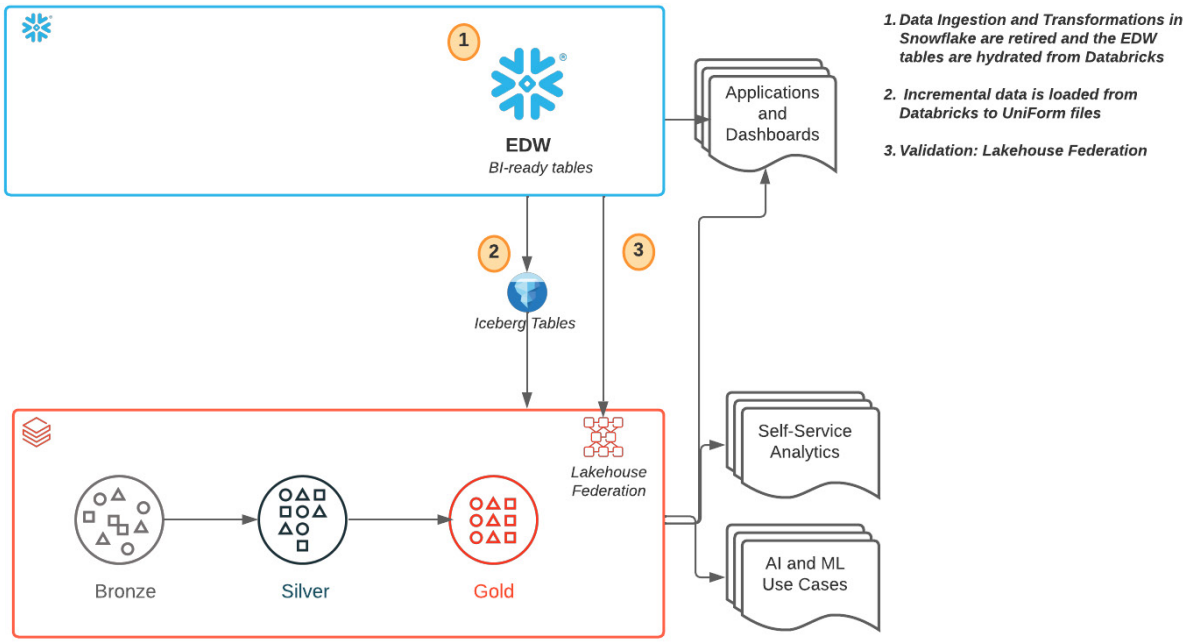


Figure 3: Transient state architecture — post-data and ETL pipeline migration

KEY CONSIDERATIONS: DATA PIPELINE MIGRATION

Data Pipeline Refactoring and Optimization

1. During the migration, there is a high probability for some portion of data pipeline queries to be refactored and/or optimized. This includes but is not limited to:
   - Rewriting queries to avoid nonperformant join strategies
   - Changing query filters due to changes in clustering keys
   - Adjusting queries to account for function syntax changes

   Join strategies, clustering keys best practices recommendations to overcome inefficiencies and improve query performance can be found in Delta Lake & Performance Optimization section

2. Consider reengineering some ETL pipelines to leverage new capabilities in Lakeflow Spark Declarative Pipelines such as SCD Type 2 which are not straightforward to implement in Snowflake using Snowflake Streams and Tasks. One popular use case where reengineering is almost always considered is modernizing CDC ingestion and streaming workloads using the power of Spark Structured Streaming and Lakeflow Spark Declarative Pipelines. Although this results in additional migration effort, this is critical for long-term cost reduction and any value-add your team would like to realize.

3. Consider creating a Git repository of the queries being migrated, and refresh this repository frequently if the queries/pipelines are allowed to change during the migration so that there are fewer conflicts to resolve during the final migration. It is recommended to impose code freeze during migration if possible.

Data Pipeline Cutover

During data pipeline migration, there will be a period during which data pipelines will be running in Databricks and Snowflake concurrently. This is expected, but in order to minimize the costs associated with this, we recommend defining the following:

- Cutover schedule
- Criteria for approving/disapproving production readiness in Databricks
- Criteria for approving/disapproving data pipeline deprecation in Snowflake
- Consider using Snowflakes support for externally managed tables as a cross over technique, to support both platforms simultaneously until it is time to cutover.
- Upstream/downstream integration validation
- Communication strategy for all applicable stakeholders

The approach described until now ensures the ETL pipelines are fully migrated and running in Databricks and the Gold layer data in Snowflake is kept in sync with the Gold layer of Databricks. While it is possible to incur data movement costs between the platforms, the significant savings gained from ETL costs would easily offset these costs. Additionally, if the team chose to use the externally managed Iceberg tables, they could potentially reduce the redundant costs in the pipelines. In the next phase of migration, the architecture is evolved to support business intelligence and other serving use cases.

Phase 5: Downstream Tools Integration

To further consolidate data platform infrastructure and maintain a single source of truth of data, organizations have adopted Databricks SQL, a data warehousing product in Databricks Lakehouse, to meet their data warehousing needs and support downstream applications and business intelligence dashboards.

Databricks SQL offers world-class price/performance for analytics workloads as well as support for high-concurrency use cases with auto-scaling SQL warehouses. Databricks SQL includes Photon, which is a query engine built from scratch in C++ and is vectorized to exploit both data-level and instruction-level parallelism.

Once data and transformation pipelines are migrated to the Databricks Lakehouse, it is critical to ensure business continuity of downstream applications and data consumers. Databricks Lakehouse has validated large-scale BI integrations with many popular BI tools in the market such as Tableau, Power BI, Qlik, ThoughtSpot, Sigma, Looker, and more. The expectation for a given set of dashboards or reports to work is to ensure all the upstream tables and views are migrated along with the associated pipelines and dependencies.

As described in the blog (see section 3.5 Repointing BI workloads), one of the common ways to repoint BI workloads after a data migration is by testing sample reports and working by renaming the data source/tables names of existing tables and pointing to the new ones.

Typically, if the schema of the tables and views post-migration hasn't changed, the repointing is a straightforward exercise of how you handle switching databases on the BI dashboard tool. If the schema of the tables has changed, you will need to modify the tables/views in the lakehouse to match the expected schema of the report/dashboard and publish it as a new data source for the reports.

Figure 4: Future-state architecture

We recommend testing the approach with a small set of dashboards or reports and iterating through the remainder of the reporting layer throughout the migration.

During the reports migration, a potential situation you may run into is the need to expand the permission of BI tool access to cloud storage buckets. This is because Databricks uses "Cloud Fetch" to support high-bandwidth data exchange. With this architecture, for a given BI query, the BI tool gets back pre-signed URLs, so that the BI tool downloads data in parallel directly from cloud storage. This might require enabling new access permissions if not already configured.

Best Practices

The Databricks Lakehouse Platform provides a number of ways to evolve and optimize the platform to achieve the best performance at the lowest cost when using and administering Databricks. Here are some best practices to implement in Databricks pertaining to different areas of ETL workloads.

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. Leveraging Liquid Clustering and Predictive Optimization should greatly simplify or reduce the need for further tuning. The following discussion provides insight into what Liquid Clustering can do automatically.

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

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

Liquid Clustering (recommended)

Liquid Clustering is the recommended data layout optimization strategy for all new Delta tables and should be the default choice over legacy partitioning and Z-ORDER.

1 | Benefits of Liquid Clustering:

- Eliminates the need for manual OPTIMIZE ZORDER commands
- Works incrementally—only reorganizes data that benefits from clustering
- Supports up to 4 clustering columns. Specifying more than four columns will result in an error
- Can be modified without rewriting all data (unlike partitioning)
- Automatically handles data layout as new data arrives
- Unpartitioned tables benefit automatically from ingestion time clustering. Ingestion time provides similar query benefits to partitioning strategies based on datetime fields without any need to optimize or tune your data (DBR 11.3+)
- Automatic Liquid Clustering is available when you enable Predictive Optimization. In this case Databricks automatically monitors the table for new data, determines when clustering would benefit query performance, runs OPTIMIZE in the background during low-usage periods and reorganizes data incrementally (only files that need it)

2 | Implementation:

- New tables: CREATE TABLE t (...) CLUSTER BY (col1, col2)
- Existing tables: ALTER TABLE t CLUSTER BY (col1, col2)
- Trigger clustering: OPTIMIZE t (runs automatically with Predictive Optimization)

3 | Column Selection for Liquid Clustering:

- Choose columns frequently used in WHERE clauses and JOIN conditions
- Prioritize high-cardinality columns that provide good data skipping
- Order columns by filter frequency (most filtered first)

2 | Migration from Snowflake Clustering Keys:

- Snowflake clustering keys generally map well to Liquid Clustering columns
- If Snowflake table uses CLUSTER BY (a, b, c), use the same columns for Liquid Clustering
- Test query performance and adjust columns based on actual Databricks query patterns

Z-Ordering (if needed)

Liquid Clustering should be the default, but consider legacy approaches in some cases

1 | When to Use Legacy Partitioning + Z-ORDER:

- Tables have extremely high write throughput where clustering overhead impacts latency
- Existing tables are already well-optimized and performing adequately
- Compliance: specific compliance requirements mandate partition-based data isolation
- Data Sharing Requirements at Partition Boundary: if your use case demands sharing slices of data strictly at partition boundaries (for example, sharing all records for a given year/month/day), legacy partitioning makes such operations straightforward
- Backward Compatibility: if you have existing systems or readers (older DBR versions, non-Delta platforms) that cannot support the newer Delta table protocols required for liquid clustering, deletion vectors preferred for Liquid Clustering legacy partitioning ensures compatibility. Liquid Clustering upgrades table protocols, which older systems may not read.

2 | Legacy Z-ORDER Guidelines (if needed):

- Use on higher cardinality columns
- Columns for Z-ordering must be in the first 32 columns of the table schema
- Effective for up to 3-4 columns; beyond that, benefits diminish
- Requires manual OPTIMIZE commands or scheduled jobs
- Partitioning for tables < 1TB is not recommended. For tables > 1TB recommended to partition by low-cardinality column, Z-order by high-cardinality columns

3 | Main features comparison between Liquid Clustering and Z-Order

FEATURE	LIQUID CLUSTERING	LEGACY Z-ORDER
Manual OPTIMIZE needed	No (with Predictive Opt)	Yes
Column limit	4	3-4 effective
Schema changes	No rewrite needed	Requires rewrite

Merge/Upsert

1 | Ensure you are using DBR 16.4+ to take advantage of Low Shuffle Merge
- Avoids write amplification due to merge's use of fullOuterJoin

2 | 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

1 | Special column type that gets defined based on a user-specified function over other columns in a Delta table

2 | Values for generated columns are computed at runtime

3 | Generated columns allow users to avoid over-/under-partitioning

Join Strategies

Preferred join strategies (in descending order):

1 | Broadcast Hash Join / Broadcast Nested Loop Join
- Requires one side of the join to be small
- No shuffle, no sort, very fast

2 | 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

3 | Sort Merge Join
- Handles any data size
- Requires shuffle and sort
- Slower in most cases when table size is small due to excessive shuffle

4 | Shuffle Nested Loop Join / Cartesian Product
- Does not require join keys
- Extremely heavy operation; avoid at all costs if possible

Query Profile

1. 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.
2. 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
3. Three common performance bottleneck problems surfaced by query profile are listed below:
   - Inefficient file pruning
   - Full table scans
   - Exploding joins (Cartesian product)

Analyze Table

Preferred join strategies (in descending order):

1. 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  ALL COLUMNS
```

2. If All Columns is not supported (MAP or VARIANT Data Types, etc.) it is important to prioritize statistics for columns that are frequently used in joins and other query predicates
3. Best practice is to use Predictive Optimization or run ANALYZE TABLE as a separately scheduled job on a regular cadence (e.g., after every load, daily or weekly)

Predictive Optimization

Predictive Optimization is an automated maintenance feature that eliminates the need for manual table optimization jobs. It should be enabled for all Unity Catalog managed tables

1. What Predictive Optimization Does:
   - Automatically runs OPTIMIZE on tables that would benefit from compaction
   - Automatically runs VACUUM to remove old files and reduce storage costs
   - Automatically runs ANALYZE to collect table statistics
   - Monitors query patterns to prioritize optimization for frequently accessed tables
   - Schedules maintenance during low-usage periods to minimize impact
2. Migration Recommendation:
   - When migrating from Snowflake, enable Predictive Optimization on all managed tables to replicate Snowflake's automatic micro-partition maintenance without manual intervention. This eliminates the need to create and maintain scheduled OPTIMIZE and VACUUM jobs.

GOVERNANCE AND SECURITY

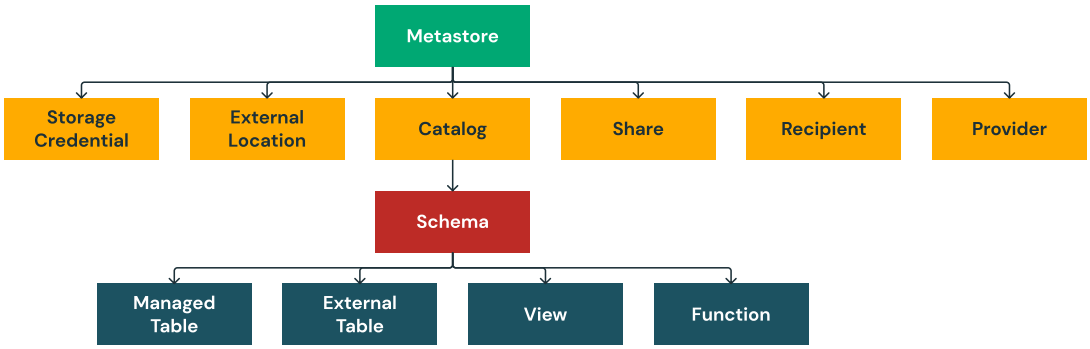
Reference Materials:

- Data Governance Guide
- Unity Catalog

Identity Management

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

Privileges and Securable Objects



- Securable Objects
- Inheritance Model
- Privileges Types

Compute

- Clusters & SQL Warehouses with Unity Catalog

COST OPTIMIZATION

Migrating from Snowflake to Databricks provides opportunities to optimize costs through proper configuration and best practices.

Compute Cost Management

1. Serverless SQL Warehouses (Recommended for BI):
   - Pay only for query execution time, not idle time
   - Auto-scales based on demand
   - No cluster management overhead
   - Best for unpredictable or spiky BI workloads

2. Job Clusters for ETL:
   - Use job clusters instead of all-purpose clusters for production pipelines
   - Job clusters terminate automatically after job completion
   - Significantly cheaper than keeping all-purpose clusters running

3 | Cluster Policies:

- Implement cluster policies to prevent over-provisioning
- Set maximum cluster sizes based on workload requirements
- Enforce auto-termination (recommended: 10-30 minutes for development)
- Restrict expensive instance types to approved use cases

4 | Job Clusters for ETL:

- Enable Photon for SQL-heavy workloads (significant speedup for analytical queries)
- Photon is included with SQL Warehouses
- For clusters, use Photon-enabled runtimes when running SQL/DataFrame operations

Storage Cost Management

1 | VACUUM Regularly:

- Remove old file versions to reduce storage costs
- Default retention: 7 days (configurable)
- Run: VACUUM table_name RETAIN 168 HOURS
- Or enable Predictive Optimization for automatic VACUUM

2 | Optimize File Sizes:

- Small files increase storage overhead and query latency
- Run OPTIMIZE to compact small files
- Enable Predictive Optimization for automatic optimization

3 | Retention Settings: Reduce time travel retention for non-critical tables

4 | Shallow Clones for Development:

- Use shallow clones instead of full copies for dev/test environments
- Changes to clone don't affect source

Query Cost Optimization

1. Result Caching:
   - RSQL Warehouses cache query results automatically
   - Identical queries return cached results instantly
   - No additional cost for cached queries
2. Materialized Views:
   - Pre-compute expensive aggregations
   - Automatically refreshed
   - Reduce repeated computation costs
3. Query Tagging for Cost Attribution
   - Tag queries to track costs by team/project
   - Monitor in Query History and create cost reports
3. Partition Pruning:
   - Ensure queries filter on partition columns
   - Avoid full table scans with appropriate WHERE clauses
   - Use EXPLAIN to verify partition pruning

Key Optimization Strategies Post-Migration

1. Start with Serverless SQL Warehouses for BI workloads
2. Use Job clusters (not all-purpose) for scheduled ETL
3. Enable Predictive Optimization on all managed tables
4. Implement cluster policies to prevent over-provisioning
5. Monitor with Cost Analysis dashboard in account console
6. Set up alerts for unusual spending patterns
7. Review Query History for expensive queries to optimize

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 the steps outlined in this migration guide, the services offered can include architecture design workshops, Databricks foundation setup, change management, cutover operations, and more.

With Lakebridge, Databricks has developed automated tooling for code complexity assessment and code migration (DDLs, DMLs) that produces outcomes tuned to best practices on Databricks Lakehouse. The conversion tool is available for use either with your preferred services vendor or a services vendor recommended by Databricks. Additionally, Databricks partners have developed several other automation tools to accelerate your migration.

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: DELTA VS. SNOWFLAKE - STORAGE FORMAT COMPARISON

	DELTA	SNOWFLAKE
Default Format Type	Columnar (OSS Parquet)	Columnar (proprietary FDN format)
IO Unit	File	Micro-partition
Size of IO Unit	16MB - 1GB (depending on table size, also configurable)	16 MB
Sort Order Within IO Unit	None	None
Sort Order Between IO Units	Liquid Clustering or Z-order	Clustering (default on ingest, optionally based on keys)
Column Statistics collected on...	Default on first 32 columns, configurable to more (unlimited)	All columns
Stats updated by...	Write operations	Write operations
Caching	FIFO data and result sets on local memory/SSD	FIFO data and result sets on local memory/SSD
Tricks to Reduce IO	Pruning via stats, liquid clustering, z-order, partitioning, bloom filters, compression	Pruning via stats, compression

APPENDIX 2: DATA TYPES

SQL data type conversions are relevant primarily within the DDL conversion. But DML statements can also have explicit data type conversions (using CAST or short format like colName::datatype). Data types supported in Snowflake can be easily mapped to the data types supported in Databricks. Data types not directly supported can be easily implemented using STRING types and processed using JSON processing function.

In certain instances, the process of type promotion may happen, which pertains to the process of casting a type into another type within the same type family which contains all possible values of the original type. As a concrete illustration, TINYINT has a range from -128 to 127, and all its possible values can be safely promoted to the INTEGER type. Refer to the link here and Databricks' official release notes for the full list of supported SQL data types.

Below is the summary of data types conversion rules.

DATA TYPE CATEGORY	SNOWFLAKE DATA TYPE	CONVERTED DATA TYPE	NOTES
Character	1. VARCHAR 2. CHAR 3. CHARACTER 4. STRING 5. TEXT 6. BINARY 7. VARBINARY	1. VARCHAR → STRING 2. CHAR → STRING 3. CHARACTER → STRING 4. STRING → STRING 5. TEXT → STRING 6. BINARY → BINARY 7. VARBINARY → BINARY	
Numeric	8. NUMBER 9. DECIMAL 10. NUMERIC 11. INT 12. INTEGER 13. BIGINT 14. SMALLINT 15. TINYINT 16. BYTEINT 17. FLOAT 18. FLOAT4 19. FLOAT8 20. DOUBLE 21. DOUBLE PRECISION 22. REAL	8. NUMBER → NUMERIC 9. DECIMAL → DECIMAL * 10. NUMERIC → NUMERIC 11. INT → INT 12. INTEGER → INTEGER 13. BIGINT → BIGINT 14. SMALLINT → SMALLINT 15. TINYINT → TINYINT 16. BYTEINT → BYTE 17. FLOAT → DOUBLE ** 18. FLOAT4 → DOUBLE ** 19. FLOAT8 → DOUBLE ** 20. DOUBLE → DOUBLE 21. DOUBLE PRECISION → DOUBLE 22. REAL → DOUBLE	* Important Note about differences in supported values ranges for Integer type columns. In Snowflake all Integer types are Synonymous with NUMBER, and defaults to NUMBER(38, 0). This means different types like TINYINT and BIGINT columns can have the same value range. Where as on Databricks, supported column value range depends on the actual integer types * Note that there is a difference in default value of precision for DECIMAL and NUMERIC. Snowflake SQL defaults to 38 where as Databricks SQL default to 10 ** Note on float data types from Snowflake docs "The names FLOAT, FLOAT4, and FLOAT8 are for compatibility with other systems; Snowflake treats all three as 64-bit floating-point numbers"

DATA TYPE CATEGORY	SNOWFLAKE DATA TYPE	CONVERTED DATA TYPE	NOTES
Boolean	23. BOOLEAN	23. BOOLEAN → BOOLEAN	
DateTime	24. DATE 25. DATETIME 26. TIME 27. TIMESTAMP 28. TIMESTAMP_LTZ 29. TIMESTAMP_NTZ 30. TIMESTAMP_TZ	24. DATE → DATE 25. DATETIME * 26. TIME → NOT SUPPORTED (use STRING or TIMESTAMP) 27. TIMESTAMP → TIMESTAMP 28. TIMESTAMP_LTZ ** 29. TIMESTAMP_NTZ ** 30. TIMESTAMP_TZ **	* DATETIME is an alias for TIMESTAMP_NTZ in Snowflake ** The usual practice is to create a new column to store an indicator for Timezone while the Timestamp is used for storing the actual value.
Semi-structured Data Types	31. VARIANT 32. VARIANT 33. VARIANT 34. OBJECT 35. ARRAY 36. ARRAY (heterogenious)	31. VARIANT → VARIANT * 32. VARIANT → STRING 33. VARIANT → STRUCT ** 34. OBJECT → STRUCT 35. ARRAY → ARRAY <TYPE> * 36. ARRAY → ARRAY<VARIANT> ***	* Note that Array type in Snowflake can support values with heterogeneous data types by using variant type whereas Array type values in Databricks need to be homogenous ** Best performance when schema is known and stable (Schema-on-Write) *** VARIANT elements for mixed types
Geospatial	37. GEOGRAPHY 38. GEOMETRY	37. GEOGRAPHY 38. GEOMETRY	

APPENDIX 3: EXAMPLE SQL DIFFERENCES

Databricks SQL supports ANSI standard SQL dialect by default. This supports seamless migration on hundreds or thousands of queries from data warehouses. Most of the SQL you have in Snowflake can be dropped in and will just work on Databricks SQL. To make this process simpler for customers, we continue to add SQL features that remove the need to rewrite queries. There are, however, certain query patterns that need to be adjusted. Here are some SQL differences:

FEATURE	SNOWFLAKE	DATABRICKS
Convention	Naming convention used to refer to tables: **<schema>.<database>.<table>**	Naming convention used to refer to tables: **<catalog>.<database>.<table>**
Unquoted Identifiers	Snowflake SQL unquoted identifiers start with a letter or an underscore and can contain letters, digits and dollar sign.	Mostly compatible with identifiers on Databricks except they cannot contain a dollar sign.
Quoted Identifiers	Snowflake SQL quoted identifiers are enclosed with double quotes(")	Databrick SQL quoted identifiers are enclosed using backtick characters (`)
SQL Variables	Session variables are defined and accessed using SET, UNSET, and SHOW VARIABLES statements	For Batch workloads Spark session parameters can be set and variable substitution in SQL is enabled by default using syntax :varName. Additionally, Widgets in Notebooks and Query Parameters in Databricks SQL can be used for passing arguments
Time Travel	AT or BEFORE clauses are used for Time Travel in Snowflake Example: `SELECT * FROM table_x AT(TIMESTAMP => '2019-01-29 00:37:58'::timestamp)`	Databricks supports time travel using temporal specification along with Delta Lake table name. Databricks Syntax (with timestamp) maps to the AT clause in Snowflake Syntax as it is inclusive. Example: `SELECT * FROM table_x TIMESTAMP AS OF '2019-01-29 00:37:58'`
Change Tracking	`CHANGE_TRACKING = TRUE | FALSE` table attribute is used in Snowflake to enable change tracking.	In Databricks the change tracking can be enabled using the table property: `TBLPROPERTIES (delta.enableChangeDataFeed = true)`
Delete Records	1) `DELETE FROM TABLE_A;` If using a USING clause in DELETE 2) `DELETE FROM TABLE_A USING (SELECT X FROM TABLE_B) as TABLE_B WHERE TABLE_A.X = TABLE_B.X;`	1) `DELETE FROM TABLE_A;` Use the MERGE operation: 2) `MERGE INTO TABLE_A USING (SELECT X FROM TABLE_B) as TABLE_B ON TABLE_A.X = TABLE_B.X WHEN MATCHED THEN DELETE;`

FEATURE	SNOWFLAKE	DATABRICKS
Update Records	1) `UPDATE TABLE_A SET COL_A='A' WHERE COL_B='B';` If using a FROM clause in UPDATE 2) `UPDATE TABLE_A SET COL_A= TABLE_B.COL_A FROM TABLE_B WHERE TABLE_A.X = TABLE_B.X;`	1) `UPDATE TABLE_A SET COL_A='A' WHERE COL_B='B';` Use the MERGE operation: 2) `MERGE INTO TABLE_A USING (SELECT COL_A FROM TABLE_B) as TABLE_B ON TABLE_A.X = TABLE_B.X WHEN MATCHED THEN UPDATE;`
Loading Data to Tables	`COPY INTO` command is used to load files in Stage to an existing table	The `COPY INTO` command is comparable in functionality.
Unloading Data from Tables	`COPY INTO` command is used to unload from a table to Stage location.	Databricks doesn't use COPY INTO to unload. Instead Use a. INSERT OVERWRITE DIRECTORY (only on Databricks Runtime) b. Use EXTERNAL TABLE definition pointing to required relocation c. Use one of the Spark DataFrame API

Flatten/Lateral vs. Explode/Lateral view	`SELECT d.id, f.value::STRING as item FROM data d, LATERAL FLATTEN(input => d.items) f;`	Databricks uses EXPLODE() and LATERAL VIEW EXPLODE instead ofSnowflake's FLATTEN() table function. Databricks uses LATERAL VIEW syntax instead of Snowflake's LATERAL
		
		`SELECT d.id, item FROM data d EXPLODE(items) as item;`
		
		or
		
		`SELECT d.id, item FROM data d LATERAL VIEW EXPLODE(d.items) t AS item;`

APPENDIX 4: LAKEFLOW CONNECT SNOWFLAKE QUERY-BASED CONNECTOR

Snowflake QBC connector can be created via API and require a Connection and Foreign Catalog. Here is the code to create a pipeline:

```
pipeline_spec = """
{
 "name": "REPLACE_WITH_PIPELINE_NAME",
 "ingestion_definition": {
     "ingest_from_uc_foreign_catalog": "true",
     "connection_name": "REPLACE_WITH_CONNECTION_NAME",
     "objects": [
       {
         "table": {
           "source_catalog": "REPLACE_WITH_SOURCE_CATALOG",
           "source_schema": "REPLACE_WITH_SOURCE_SCHEMA",
           "source_table": "REPLACE_WITH_SOURCE_TABLE",
           "destination_catalog": "REPLACE_WITH_DESTINATION_CATALOG",
           "destination_schema": "REPLACE_WITH_DESTINATION_SCHEMA",
           "table_configuration": {
             "primary_keys": ["REPLACE_WITH_PRIMARY_KEY_COLUMN"],
             "query_based_connector_config": {
               "cursor_columns": ["REPLACE_WITH_CURSOR_COLUMN"]
             },
             "scd_type": "SCD_TYPE_2"
           }
         }
       }
     ]
 },
 "catalog": "REPLACE_WITH_DESTINATION_CATALOG",
 "schema": "REPLACE_WITH_DESTINATION_SCHEMA",
 "channel": "PREVIEW"
}
"""

notebook_context = dbutils.notebook.entry_point.getDbutils().notebook().getContext()
api_token = notebook_context.apiToken().get()
workspace_url = notebook_context.apiUrl().get()
api_url = f"{workspace_url}/api/2.0/pipelines"
headers = {
   'Authorization': 'Bearer {}'.format(api_token),
   'Content-Type': 'application/json'
}

response = requests.post(url=api_url, headers=headers, data=pipeline
spec)
print (response.text)
```