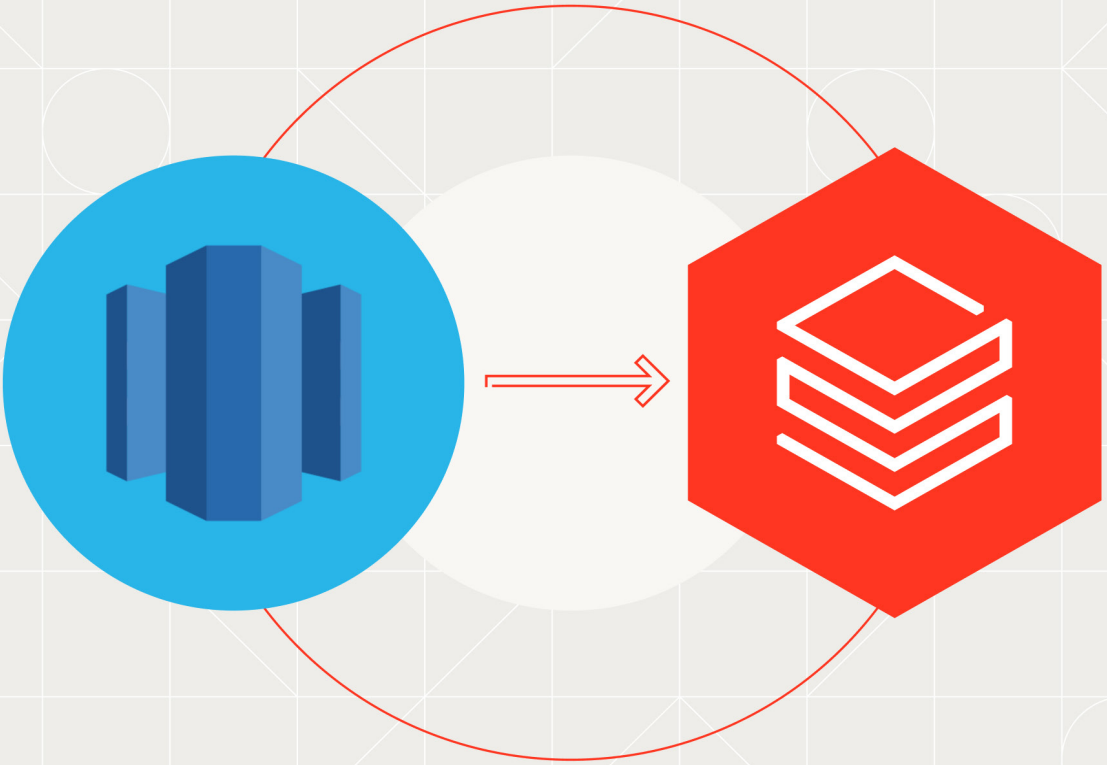
databricks

Guide

Amazon Redshift to Databricks Migration Guide



databricks

Table of Contents

Preface

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

Migration Strategy

When migrating from Amazon Redshift to Databricks, it is crucial to plan and execute the process carefully to ensure a successful outcome. Adopting a structured approach minimizes risks and increases 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 (ETL, BI, ML, etc.)
- Business criticality of use cases
- Current projects in flight and their delivery timelines
- Migration goals (cost reduction, cutover deadlines, user change management, etc.)

Broadly there are two areas to consider when selecting the migration strategy:

- Migration approach
- Technical execution strategy

MIGRATION APPROACH

The choice of the migration approach typically revolves around two main options: a big-bang migration or a phased migration.

- A big-bang migration involves a faster implementation and cutover process, enabling organizations to swiftly transition off their entire Amazon Redshift infrastructure
- A phased migration involves executing the migration in stages such as a specific use case, schema, data mart or data pipeline

It is recommended to adopt a phased migration approach to mitigate risks, show progress and demonstrate value early in the process. This is the better-suited approach for a large Amazon Redshift environment with several databases and business teams.

While a phased migration is ideal for large Amazon Redshift environments, a big-bang approach can be more suitable for smaller Amazon Redshift footprints or when the scope of the migration is relatively limited. This approach is more beneficial because it allows for a quicker and less complex migration process. However, it is crucial to carefully assess the impact and potential risks associated with a big-bang migration, particularly in larger and more complex environments.

TECHNICAL EXECUTION STRATEGY

Once you settle on the migration approach, 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
- Access to migration tools and migration effort

From a high-level strategy perspective, there are two popular execution strategies for migrations.

ETL-first approach: Lead with migrating data ingestion and transformation workloads by landing all data in the cloud storage (Amazon S3, Azure Data Lake Storage Gen2, or Google Cloud Storage), in an open, analytics-optimized format like Delta Lake, and performing ETL on both batch and streaming based data.

Delta Live Tables and Databricks Workflows can help simplify the movement, cleansing and aggregation of data assets for formal business consumption.

This approach allows Amazon Redshift use, 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.

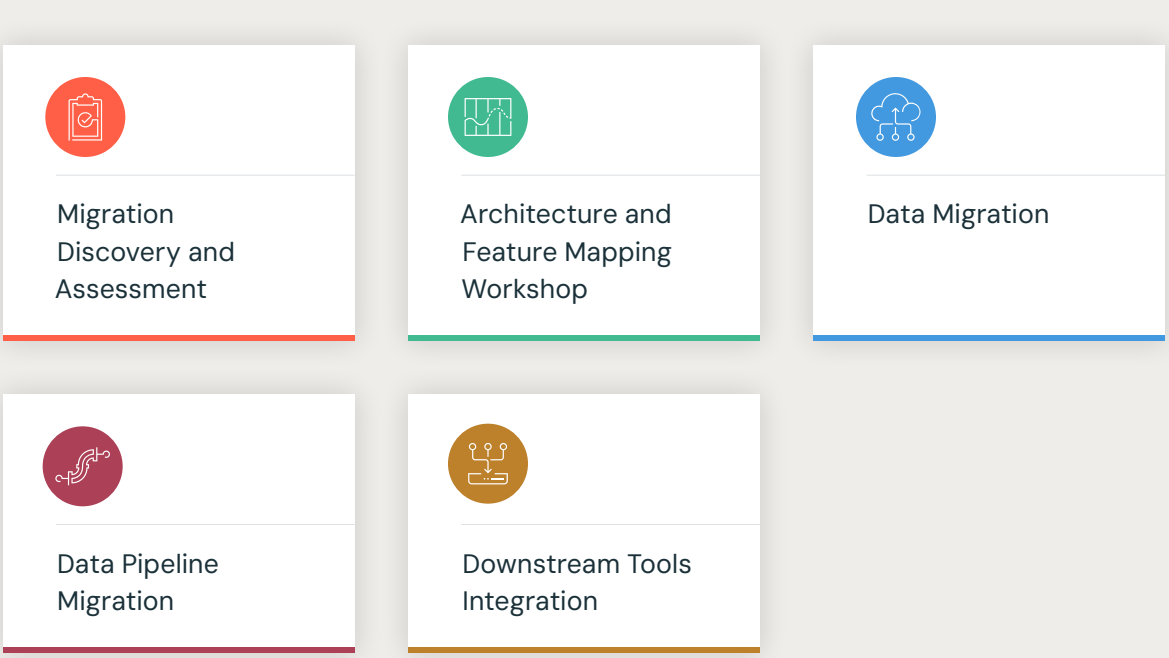
BI-first approach: Lead with modernizing the reporting layer by replicating the Amazon Redshift data warehouse Gold, or "presentation," layer tables to Databricks. This quickly unlocks value by breaking data silos and enabling cross-functional reporting capabilities and data science projects. Subsequently, build and "replumb" the ETL data pipelines in Databricks and cut over from those running in Amazon Redshift.

In the next few sections, we will dive into more detail on the migration process of the ETL first execution strategy.

Overview of the Migration Process

Enterprise data warehouse (EDW)/ETL migrations from legacy on-premises platforms to the cloud are typically complex and lengthy engagements. However, migrations are relatively simpler from Amazon Redshift to Databricks because the data is already in the cloud.

The migration process typically consists of the following steps, but can vary depending on customer 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.

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

Phase 1: Migration Discovery and Assessment

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

- The types of workloads (ETL, BI, ingress/egress, etc.) and their size by warehouses and databases
- The scope of data and queries/workloads to be migrated
- The upstream and downstream technologies and applications involved in the architecture
- The current security setup and protocols
- Estimates for infrastructure costs

Using the insights from migration assessments positions you better for success as it provides an understanding of the level of complexity and the effort required to migrate to Databricks. Furthermore, it enables you to make informed, data-driven decisions in prioritizing use cases/workloads to maximize business value. Databricks strongly recommends using automation tools, such as the Databricks **Amazon Redshift Profiler** and/or the **BladeBridge Code Analyzer,** to expedite the process of gathering migration-related information.

AMAZON REDSHIFT PROFILER

The Amazon Redshift Profiler reads system views and catalog tables (e.g., PG_TABLE_DEF) and returns insights such as workload types, long-running ETL queries and user access patterns. The profiler classifies queries into T-shirt sizes for complexity, evaluates function calls for compatibility, and extracts other metadata information to aid with the data migration. This analysis provides guidance on identifying databases and pipelines that contribute to high costs and complexity. The results assist in workload prioritization and migration execution planning. Work with your Databricks representative to get access to the Profiler.

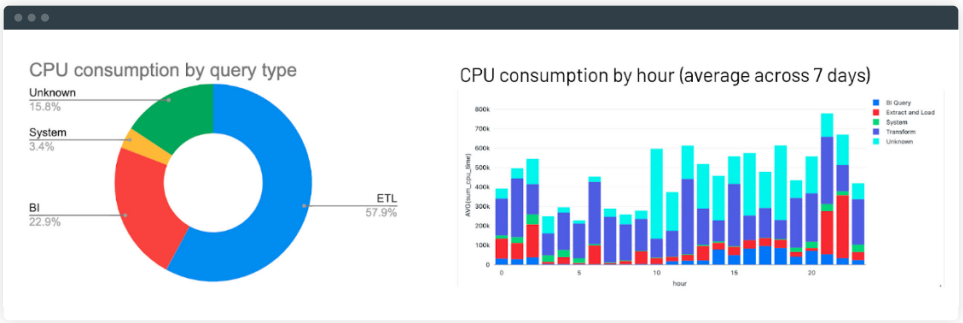


Figure 1: Sample output from example profiler result

BLADEBRIDGE CODE ANALYZER

The BladeBridge Code Analyzer collects deeper insights on data types, DDL (data definition language), DML (data manipulation language) code complexity and data dependencies. The tool parses Amazon Redshift SQL code artifacts to generate:

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

The complexity counts are used to size the software costs for using the BladeBridge Code Converter and help you estimate the number of hours you should forecast for the migration project. This is available free of cost — a Databricks solutions architect can assist you in running the tool.

Phase 2: Architecture and Feature Mapping Workshop

When planning your Amazon Redshift migration, it is important to correctly map Amazon Redshift capabilities to Databricks Lakehouse capabilities. In a typical Amazon Redshift–based architecture, it's common to see Amazon Redshift intertwined with various AWS offerings and third-party utilities. Thus, it becomes crucial to assess the integrations that will persist in the final architecture versus the ones that will be substituted with lakehouse capabilities.

In this phase, we design and map the path for each component of the current architecture that needs to be modernized to the target lakehouse architecture that aims to serve the data and AI use cases.

Additionally, we will need to compare/contrast current- and future-state architecture diagrams to align on any intermediate phases of the architecture. As an example, the intermediate architecture may require running data transformation pipelines in both Amazon Redshift and Databricks in parallel. This will therefore involve a design solution to ensure data remains in sync so that 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 Amazon Redshift. The initial discovery (phase 1) will help surface this information, but

this phase will dive deeper into how they are used to ensure each existing Amazon Redshift feature/functionality is mapped accordingly in Databricks.

AMAZON REDSHIFT VS. DATABRICKS FEATURE MAPPING EXAMPLE

OBJECTS/WORKLOAD	AMAZON REDSHIFT	DATABRICKS
Compute	Amazon Redshift Clusters	Databricks Clusters optimized for workload types with a runtime: ▪ All-purpose for interactive/developer use ▪ Jobs for scheduled pipelines ▪ SQL warehouse for BI workloads and ad hoc SQL queries
Storage	Amazon Redshift Managed Storage and S3	Cloud storage (Amazon S3, Azure Blob Storage, Azure Data Lake Storage Gen2, Google Cloud Storage)
Tables	Amazon Redshift Tables	Delta Tables
Format	Amazon Redshift proprietary	Delta Lake (Parquet)
Interface	Amazon Redshift Query Editor v2.0 Redshift Data API	Databricks collaborative notebooks Databricks SQL workspace Databricks CLI
Database Objects	Tables, Temporary Tables, Views, Materialized Views, Stored Procedures, UDFs	Tables, Views, Temporary Views, Materialized Views, UDFs
Metadata Catalog	Built-in catalog, Glue Catalog	Unity Catalog
Data Sharing	Amazon Redshift Data Sharing AWS Data Exchange	Delta Sharing Delta Sharing Marketplace
Data Ingestion	AWS Glue, Amazon EMR, custom pipelines using connectors, COPY (S3), AWS Kinesis, third-party tools	COPY INTO CONVERT TO DELTA Auto Loader DataFrame Reader Structured Streaming APIs for Kafka and Kinesis Integrations via Partner Connect Add data UI
Data Types	Data Types in Amazon Redshift	Data Types in Databricks
Workload Management	Available through WLM	Intelligent Workload Management (IWM), cluster configuration (policies), cluster metrics
Access Control	IAM, Glue Catalog	IAM, role-based access, Unity Catalog
Sorting	Multipart Sort Keys	Z-Ordering

FEATURES	AMAZON REDSHIFT	DATABRICKS
Distribution Styles	Auto/Even/Key/All	Not applicable in Databricks
Programming Language	SQL, Python (for UDFs)	SQL, Python, R, Scala, Java
Data Integration	EMR, Glue, External tools (dbt, Matillion, Talend, Pentaho, Informatica, etc.)	Delta Live Tables, External tools (dbt, Matillion, Prophecy, Informatica, Talend, Fivetran, etc.)
Orchestration	Glue, data pipeline, Airflow	Databricks Workflows, Airflow, Azure Data Factory, Dagster, Python SDK, Terraform Provider
Machine Learning	AWS SageMaker, Amazon Redshift ML	Databricks ML (Runtime with OSS ML packages, MLflow, Feature Store, AutoML)

The table above is an example of mapping key features between Amazon Redshift 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 formulate a more accurate migration plan and migration 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. Refer to Databricks documentation for instructions on workspace setup.

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 is migrating the data. The data migration involves both metadata (table DDLs, views, etc.) and actual table data. Databricks is optimized for cloud object storage: Amazon S3, ADLS Gen2 and Google Cloud Storage. The actual table data that

is migrated is stored in open format (Delta) in the cloud object storage, unlike Amazon Redshift, where the data is stored in a proprietary format.

When migrating data out of Amazon Redshift, there are key questions that need to be answered. 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, tables?
- Should there be any cleanup of duplicated data sets or organization of the existing data footprint in Amazon Redshift?

Once these are determined, 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 into Delta. This approach also allows us to easily compare the data in Amazon Redshift and Databricks during parallel runs.

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) tables into Delta Lake medallion (Bronze/Silver/Gold) data architecture
2. Migrate/build data pipelines that incrementally populate Bronze/Silver/Gold in Delta Lake
3. Backfill Bronze/Silver/Gold tables as needed

SCHEMA MIGRATION

Before the tables are offloaded to Databricks, the schema of the tables must be created in Databricks. If you have DDL scripts, you could leverage them with some tweaks to data types used in the DDLs. Once the scripts are extracted, automation tools (e.g., BladeBridge code converter) can handle converting Amazon Redshift DDLs to Databricks DDLs. Refer to guidance around matching data types in both Databricks and Amazon Redshift in Appendix 1.

DATA MODELING IN THE LAKEHOUSE

Typically, data for Amazon Redshift can be coming from several sources such as Amazon S3, Amazon EMR, Amazon DynamoDB, or data sources on remote hosts, and ends up in a data

model that represents the EDW tables. Whatever data model (dimensional model, data vault, etc.) is implemented in Amazon Redshift can be implemented in the Databricks Lakehouse in a more performant manner using Delta Lake. Data architecture in Databricks Lakehouse follows the medallion architecture — Bronze, Silver, Gold paradigm — and the data model belongs in the Gold layer.

DATA MIGRATION

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. Leveraging Amazon Redshift's UNLOAD command to push data out of Amazon Redshift and into cloud storage in Parquet format, then using one of these options to load into Databricks to write to Delta Lake format tables using one of the following options:
   - Auto Loader
   - Databricks COPY INTO command
   - Spark batch/streaming APIs
2. Leveraging the Spark Amazon Redshift connector to read data from Amazon Redshift into a Spark DataFrame. The DataFrame is then saved as a Delta Lake format table.
3. Leveraging data ingestion partners such as Hevo from Databricks Partner Connect for quick data migration using no-code and automation with built-in schema management, high availability (HA) and autoscaling.

After the initial Gold table offload, recurring jobs should be set up to continuously sync data from Amazon Redshift to Databricks for those Gold tables until data pipelines have cut over and solely feed data to those Gold tables in Databricks. For continuous replication and real-time sync, leverage real-time change data capture tools from Databricks Partner Connect with partners like Hevo Data. Databricks and any partners involved will work with your team to align on the best approach(es) for your team's requirements.

As you go through the migration, the current architecture slowly changes as you start offloading data and workloads in a phased manner. Figure 2 shows the architecture during the data migration phase.

After validation and testing, the Bronze and Gold layer data is 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.

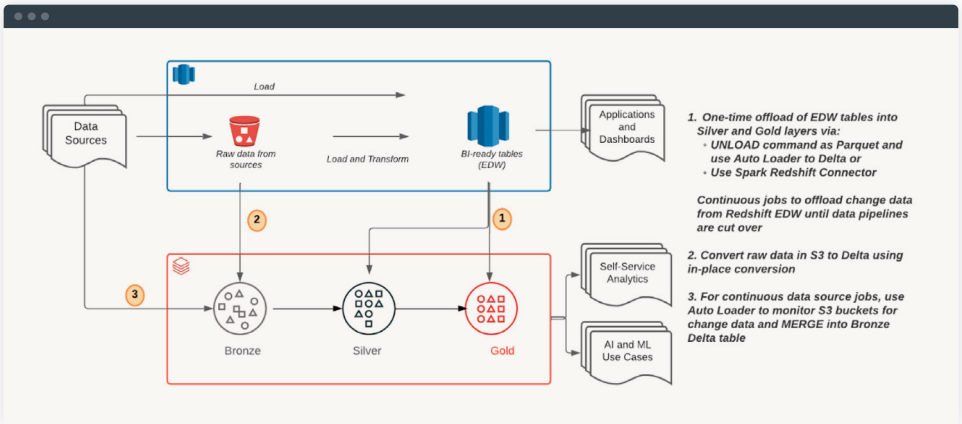


Figure 2: Transient state architecture: post-data migration

KEY CONSIDERATIONS: SCHEMA MIGRATION AND DATA MIGRATION

- Distribution styles and sort keys: Distribution keys and sort keys are commonly found in table DDLs in Amazon Redshift. This detail is used by the Amazon Redshift query planner and engine to efficiently run the query. Databricks uses different query execution planning and optimization strategies. While distribution styles are not applicable as is in Databricks, the sort keys are very similar to the Z-Ordering strategy. Refer to the best practices section in this document for more details on these strategies.
- 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. Visual data modeling tools such as Quest ERWIN or SqlDBM, from Databricks Partner Connect, can accelerate this development and deployment of the data model in a few clicks. Both of these tools can reverse engineer an Amazon Redshift data model (table structures) and implement them in Databricks easily.
- When migrating DDLs of a table, it is important to check the schema of the source data. For example, let's say one of the data sources is in Parquet format. Some numeric column types in DDL generated from the Amazon Redshift table will be different from the type in the source Parquet files. If we don't use the source column types during DDL creation, we will be forced to have unnecessary casting on our ingestion pipeline after the data is migrated from Amazon Redshift.
- In Amazon Redshift and Databricks, keeping schemas in sync during the transient stage can become critical if there are changes introduced to table schemas during migration. Making a change in a central MDM (master data model) first — leveraging tools like SqlDBM — and then implementing it in both Amazon Redshift and Databricks is a popular approach.

- Amazon Redshift users may be used to Automatic Analyze and Automatic Vacuum features for table maintenance in addition to manually performing these commands. While you can run different table optimizations such as Optimize, Analyze, ZORDER and Vacuum today in Databricks manually, the Auto Maintenance capabilities for Optimize, Vacuum, Analyze and Clustering are being introduced in Databricks with the latest DBR versions. Refer to the latest product version and supported functionality for details on availability of these features.

Phase 4: Data Pipeline Migration

Data pipeline migrations from Amazon Redshift to Databricks consist of several key components: compute model migration, orchestration, source/sink migration, query migration and refactoring. Understanding the end-to-end view of the pipelines from data sources to the consumption layer, including the governance aspects, is critical to effectively migrate the workloads.

COMPUTE MODEL MIGRATION

As compared to Amazon Redshift, Databricks offers much more flexibility of compute options for the entire end-to-end pipeline needs: from core ETL, BI, ML, streaming, etc. In addition to simple engineering performance, this degree of flexibility is a large part of what enables customers to save on their infrastructure costs by allowing users to "form fit" their computing needs to their exact problem.

Consider a scale of 1 to 10, where 1 is a completely monolithic model that runs all workloads for your entire data platform on a single cluster, and 10 is a completely serverless model where all workloads run on a totally serverless ephemeral compute model. In Databricks, you can choose ANY number on that scale in a way that solves your problems best. This is amazing for savings, resource efficiency and cluster management simplicity, but does require some thinking as you migrate from a more monolithic data-warehousing environment to the Databricks Lakehouse.

In Databricks you have the following compute types: All-purpose, Jobs, DBSQL, DLT, and Serverless. For each of these, you can elect to use Databricks Spark or the Databricks Photon runtime, which is often orders of magnitude faster, but more expensive on a per-unit basis. For data warehousing style workloads (SQL queries with lots of filters, joins, aggregations), Photon is well worth the extra price, as this is precisely what it was built for, but if there are parts of your architecture where performance is not as key or they are not very complex, then you can still elect to simply use the normal DBR Spark runtime at the cheaper price.

All-purpose: This is compute that users and applications can share and run code in different languages. It can be "always-on" or have any auto-termination rules for non-use to ensure clusters are on while unutilized for long periods of time (like the weekend). This compute type is generally best for longer-standing workloads that are less predictable and not automated in jobs (ML experiment, ad hoc Python/SQL analytics, development). This cluster can also autoscale and be any size required for your needs. In addition, you can create as many of these as you want with different rules.

Jobs: This compute type allows you to tell a job how many resources it can use while running a task and will create an ephemeral cluster (active only for the time that the job is running, then automatically turns off and is NOT reused) for that specific job run. Within a job, you have a great deal of flexibility. You can share clusters for many tasks, each task can have their own cluster, and everything in between. This is the primary consideration when thinking about designing warehousing jobs on a lakehouse. Usually in classical data warehousing workloads, ETL, analytics and BI are all required to share compute resources, resulting in lots of queuing and slowing down of queries. Now you can share resources where it makes sense and isolate resources where it makes sense. In general, here are some best practices for designing the computing model for your workloads:

1. **Share a cluster when tasks share the same data:** If different tasks in a job read from the same data, or if that data is used over and over and/or is large, then the shared caching of the cluster will greatly reduce the runtimes for the job as a whole. So when tasks share data sources, it usually makes sense to share clusters.

2. **Isolate clusters where bottleneck occurs:** If there are particular tasks that are especially greedy and bottleneck the whole cluster, then you can often get a great deal of performance improvement by provisioning a separate cluster for those tasks. Not only will the task finish faster, but other tasks will not need to fight as hard for scarce resources. At first, this seems more expensive (since you are creating more clusters), but since the clusters disappear as soon as they are done, it can often result in very significant performance and cost gains for the pipeline as a whole. This is indeed the idea of "form-fitting" your compute model to your specific data processing needs.

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. There are several ways to orchestrate Amazon Redshift–based ETL tasks. Some of them include using AWS services such as AWS Glue, AWS Data Pipeline, AWS Step Functions and AWS Lambda. The other common ways are using Apache Airflow or custom Python-based scripting or using third-party tools such as Matillion.

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. It also has out-of-the-box functionality to SCD Type 1 and Type 2 tables. In this method you are reengineering the orchestration layer.
2. It is also possible to use the external tools such as Airflow for orchestration and repoint these tools from Amazon Redshift compute to Databricks compute. You would be just translating Amazon Redshift SQL queries to Spark SQL queries in the orchestration job while retaining most of the orchestration elements.

   However, it is recommended to use Databricks Workflows for better integration, simplicity and lineage.

SOURCE/SINK MIGRATION

The most popular pattern data ingested to Amazon Redshift is using S3 as an intermediary state for staging and then loading to Amazon Redshift. Similar to orchestration, in most cases an external ETL tool is seen in Amazon Redshift architecture to extract data from source systems, transform it (optional) and load it into the Amazon Redshift warehouse.

1. Source data connections
   - Ingestion pipelines using tools such as Fivetran and Qlik Replicate can be duplicated and configured to point to Databricks Delta Lake instead of Amazon Redshift. Delta is an open source format and widely supported as the target data format for popular data ingestion tools.
   - Ingestion pipelines using S3 as an intermediary stage with a `COPY` command or a Amazon Redshift Job that reads from S3 are replaced with Databricks Auto Loader or Spark DataFrame APIs that read directly from S3. Delta Live Tables supports Auto Loader and Spark DataFrame APIs.
   - Native Spark integrations (e.g., Kafka) are leveraged to refactor the ingestion pipelines reading stream data from Amazon Kinesis Data Streams or Amazon MSK
2. Sink data connections
   - Ingestion tools and framework will now generate data in Delta Lake format instead of Amazon Redshift native format

QUERY MIGRATION AND REFACTORING

Queries here refer to any DML query that transforms the data or the ad hoc data analysis queries run by the user for data analysis. The interface from where queries are initiated could be directly in Amazon Redshift 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 Matillion, the refactoring is straightforward, especially for user-written SQL queries. Any Amazon Redshift-specific integration features should be refactored, which in most cases is equivalent to tweaking the underlying Amazon Redshift SQL query. Migrating from Amazon Redshift SQL to Spark SQL requires identifying and replacing any incompatible/proprietary Amazon Redshift SQL functions or syntax. A few options for tackling this are:

1. (Recommended) Use BladeBridge Converter to automate lift-and-shift portion of query migration
2. (Not recommended) Develop custom script in-house to convert Amazon Redshift SQL to Spark SQL
3. (Not recommended) Manually convert Amazon Redshift SQL to Spark SQL

A common question we run into is how Stored Procedures are migrated since Databricks does not have a database object called "Stored Procedures." This is addressed easily given that Databricks supports different programming frameworks and tools between notebooks, JARs, scripts and jobs and all the power of SQL, Python, R, Scala and Java. Stored Procedures in most cases are executable code that can be triggered or executed on a cadence. The recommended approach is using Databricks Workflows with the stored procedure (one or more) defined as tasks within a Databricks Job. The tasks could be the Amazon Redshift SQL code patterns converted into Databricks SQL, or PySpark code that covers code patterns such as conditional statements, loops, functions and exception handling statements. Check out this blog that describes in detail how different code patterns that go into a Stored Procedure are converted to Databricks. Here is guidance around commonly used database objects and SQL query patterns:

- Python-based UDFs can largely be a lift-and-shift while Java-based UDFs need to be converted to SQL based or Python based
- Recursive CTEs are not natively supported in Spark SQL, but using PySpark, it can be easily achieved. Example 1, Example 2.
- Amazon Redshift Data API supports multi-statement transactions. While this is not natively supported currently in Delta, the functionality can be implemented using custom development using PySpark and Delta.
- Materialized Views in Databricks SQL is in private preview and can be used in regular SQL or Delta Live Tables with governance handled by Unity Catalog

Given that both Amazon Redshift and Databricks support industry-standard SQL, a large portion of the Amazon Redshift SQL queries can be automatically converted to Databricks syntax to accelerate the migration. The BladeBridge 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 the COPY command. 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. Check out this short demonstration of the conversion tool.

Refer to Appendix 3 for some examples of SQL translation differences between Amazon Redshift SQL and Databricks. Check out this handy cheat sheet packed with essential tips and tricks to help you get started on Databricks using SQL programming in no time!

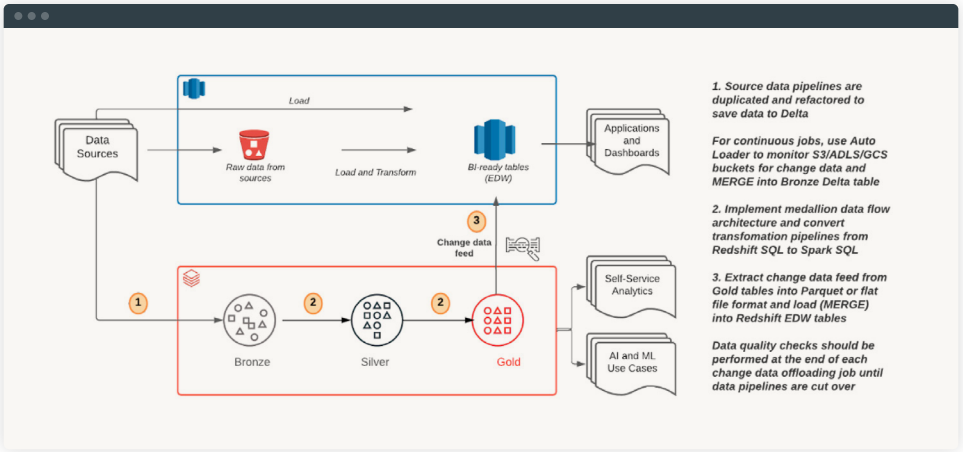


Figure 3: Transient state architecture during pipeline migration

MIGRATION VALIDATION

Validation is mostly done for the data in both the platforms. As there might be thousands of tables migrated, it is manually impossible to compare the data values in Amazon Redshift 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. For more advanced table data and schema comparison, tools like Datacompy can be used.

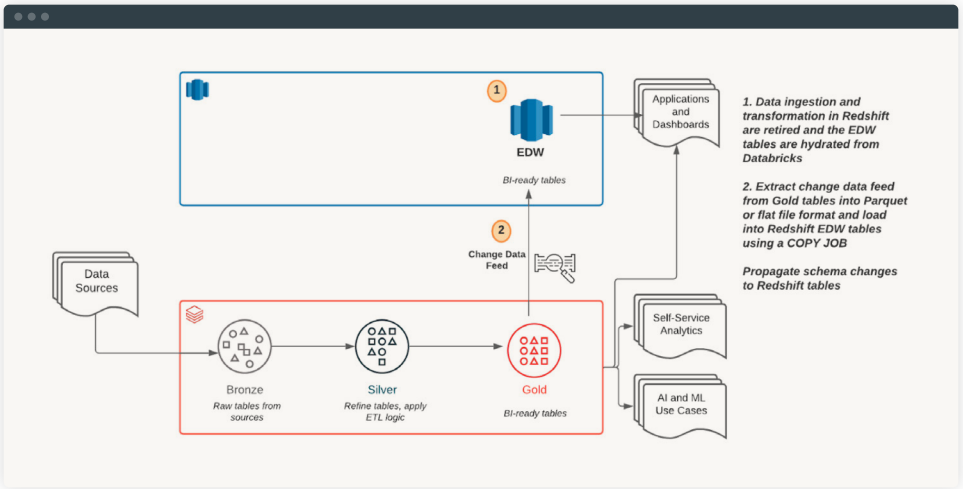


Figure 4: Transient state architecture — post-data and ETL pipeline migration

KEY CONSIDERATIONS: DATA PIPELINE MIGRATION

Data Pipeline Refactoring and Optimization

- 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:
  1. Rewriting queries to avoid nonperformant join strategies
  2. Changing JOIN and GROUP BY columns due to changes in dist keys
  3. Changing query filters due to changes in sort keys
  4. Adjusting queries to account for function syntax changes

  Here are some strategies to overcome inefficiencies and improve query performance:

  A. Join strategies
     a. Avoid Cartesian products if at all possible (these are heavy in any query engine)
     b. Avoid self, exploding joins that result in Cartesian products (self-join with sliding window calculations as keys); instead, shift logic into a CTE or view so that the sliding window calculations are materialized pre-join
     c. Preferred join strategies (in descending order):
        i. Broadcast Hash Join
        ii. Shuffle Hash Join
        iii. Sort Merge Join
        iv. Shuffle Nested Loop Join (Cartesian Product)

  B. Data Partitioning
     a. Select high cardinality columns for Z-ordering
     b. Z-order is effective for up to 3-5 columns
     c. For tables > 1TB — we recommend partitioning by low cardinality columns and Z-Ordering by high cardinality columns

  C. See Delta Lake & Performance Optimization section for other recommendations.

- Consider reengineering some ETL pipelines to leverage new capabilities in Delta Live Tables such as SCD Type 2, which are not straightforward to implement in Amazon Redshift, and are handled using views, temporary tables and row functions. 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 Delta Live Tables. 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.

- 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 over which data pipelines will be running in Databricks and Amazon Redshift concurrently. This is expected, but in order to minimize the costs associated with this, we recommend defining the following:

1. Cutover schedule
2. Criteria for approving/disapproving production readiness in Databricks
3. Criteria for approving/disapproving data pipeline deprecation in Amazon Redshift
4. Upstream/downstream integration validation
5. 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 Amazon Redshift 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. 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 autoscaling 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 this 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.

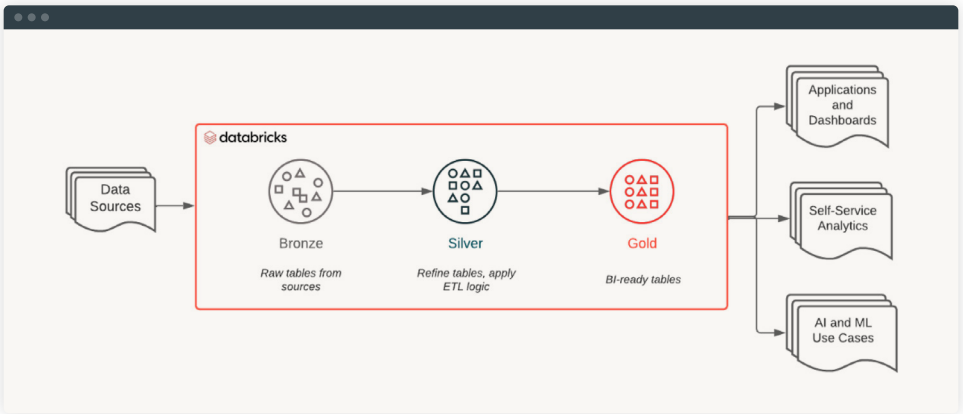


Figure 5: Future-state architecture

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

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

COMPUTE

One of the great advantages of migrating from Amazon Redshift to Databricks is the availability of a spectrum of compute options for different types of workloads. You would be using ephemeral shared nothing clusters vs. a monolithic cluster design in Amazon Redshift. For example, you would use Jobs Clusters for scheduled ETL jobs and use all-purpose clusters and SQL warehouses for interactive workloads when choosing compute types by workload types. The same capability comes in handy to isolate/combine jobs and pipelines based on their nature of work. As a general rule of thumb you could follow this guidance:

- Isolate jobs/pipelines to separate compute where they compete heavily for compute resources. For example, jobs with heavy writes and heavy reads on the same cluster can be separated.
- Share compute resources (use the same cluster) with jobs/queries when they read a lot of the same data or reuse results from the previous task

We recommend using DBR 12.2LTS+ for data warehouse migrations. The newer Databricks Runtime support features such as deletion vectors will greatly fit data warehousing workloads.

See the documentation for details on different compute options and how they work.

- Clusters & SQL Warehouses with Unity Catalog

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

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

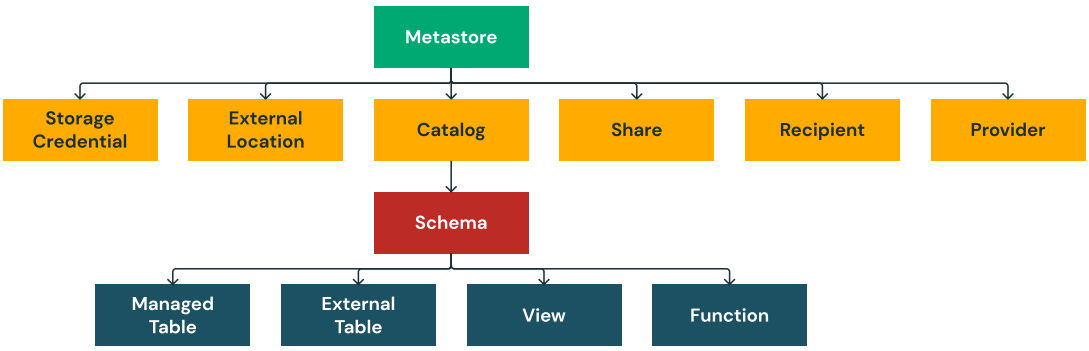
Reference Materials:

- 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

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.

Working with BladeBridge, 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. AMAZON REDSHIFT — STORAGE FORMAT COMPARISON

	AMAZON REDSHIFT	DELTA
Default Format Type	Columnar (proprietary)	Columnar (OSS Parquet)
IO Unit	Database block	File
Size of IO Unit	Amazon Redshift uses a block size of 1MB	16MB–1GB (depending on table size, also configurable)
Sort Order Between IO Units	Uses distribution styles and sort keys	Ingestion Time Clustering + Z-order
Column Statistics collected on...	All columns	Default on first 32 columns, configurable to more (unlimited)
Stats updated by...	Continuously monitoring and automatically performing analyze operations	Write operations
Caching	Caching based on the number of entries in the cache and the instance type of your Amazon Redshift cluster	FIFO data and result sets on local memory/SSD
Tricks to Reduce IO	Pruning via stats, compression, proper distributions	Pruning via stats, partitioning, bloom filters, 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 Amazon Redshift can be easily mapped to the data types supported in Databricks.

Below is the summary of data types conversion rules.

DATA TYPE CATEGORY	AMAZON REDSHIFT DATA TYPE	CONVERTED DATA TYPE	NOTES
Character	1 CHAR 2 CHARACTER 3 NCHAR 4 VARCHAR 5 NVARCHAR 6 VARBYTE	1 CHAR → STRING 2 CHARACTER → STRING 3 NCHAR → STRING 4 VARCHAR → STRING 5 NVARCHAR → STRING 6 VARBYTE → BINARY	
Numeric	7 SMALLINT 8 INT2 9 INTEGER 10 INT 11 INT4 12 BIGINT 13 INT8 14 DECIMAL 15 NUMERIC 16 REAL 17 FLOAT4 18 DOUBLE PRECISION	7 SMALLINT → SMALLINT 8 INT2 → SMALLINT 9 INTEGER → INTEGER 10 INT → INT 11 INT4 → INT 12 BIGINT → BIGINT 13 INT4 → BIGINT 14 DECIMAL → DECIMAL * 15 NUMERIC → NUMERIC 16 REAL → DOUBLE 17 FLOAT4 → DOUBLE 18 DOUBLE PRECISION → DOUBLE	* Note that there is a difference in the default value of precision for DECIMAL and NUMERIC. Amazon Redshift SQL defaults to 38, whereas Databricks SQL defaults to 10.
Boolean	19 BOOLEAN	20 BOOLEAN → BOOLEAN	
DateTime	21 DATE 22 TIME 23 TIMETZ 24 TIMESTAMP 25 TIMESTAMPTZ 26 INTERVAL	21 DATE → DATE 22 TIME → NOT SUPPORTED (use STRING or TIMESTAMP) 23 TIMETZ → NOT SUPPORTED (use STRING or TIMESTAMP) 24 TIMESTAMP → TIMESTAMP 25 TIMESTAMPTZ → TIMESTAMP * 26 INTERVAL → INTERVAL	* 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	27 ARRAY (Amazon Redshift Spectrum) 28 STRUCT (Amazon Redshift Spectrum) 29 MAP (Amazon Redshift Spectrum) 30 HLLSKETCH 31 SUPER	27 ARRAY → ARRAY 28 STRUCT → STRUCT 29 MAP → MAP 30 HLLSKETCH ** 31 SUPER ***	** While there is no built-in data type, the spark-alchemy package supports advanced HLL processing → Advanced Analytics with HyperLogLog Functions in Apache Spark *** Can be mapped to STRING and values processed using built-in unstructured data parsing functions
Geospatial	32 GEOGRAPHY 33 GEOMETRY	32 GEOGRAPHY * 33 GEOMETRY *	* While there are no built-in geospatial data types in Delta Lake tables, there are a plethora of options to process geospatial data at scale with the Databricks platform by leveraging the open source community-developed libraries. Refer to the following blogs: →Processing Geospatial Data at Scale With Databricks →Building a Geospatial Lakehouse, Part 1 →Building a Geospatial Lakehouse, Part 2

APPENDIX 3: EXAMPLE SQL DIFFERENCES

	AMAZON REDSHIFT	DATABRICKS
Object Naming	Naming convention used to refer to tables: ***<database>.<schema >.<table>***	Naming convention used to refer to tables: ***<catalog>.<schema>.<table>*** or ***<catalog>.<database>.<table>***
Unquoted Identifiers	Amazon Redshift unquoted identifiers start with an alphabetic character or an underscore and can contain ASCII alphanumeric characters, underscores and dollar sign	Mostly compatible with identifiers on Amazon Redshift except they cannot contain a dollar sign
Quoted Identifiers	Amazon Redshift quoted identifiers are enclosed with double quotes(")	Databricks SQL quoted identifiers are enclosed using backtick characters (`)
SQL Variables	Variables are defined at BLOCK level and accessed using DECLARE and := 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
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;
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;
Merging Records	MERGE INTO target_table USING source_table ON match_condition WHEN MATCHED THEN { UPDATE SET col_name = { expr } [,...] | DELETE } WHEN NOT MATCHED THEN INSERT VALUES ({ expr } [, ...])	Unlike Amazon Redshift, Databricks supports both NOT MATCHED BY SOURCE and NOT MATCHED BY TARGET. MERGE INTO target_table_name USING source_table_reference ON merge_condition WHEN MATCHED THEN matched_action WHEN NOT MATCHED [BY TARGET] THEN not_matched_action WHEN NOT MATCHED BY SOURCE THEN not_matched_by_source_action
Loading Data to Tables	COPY command is used to load data from stage files into an existing table	The COPY INTO command is comparable in functionality
Unloading Data from Tables	UNLOAD command is used to unload from a table to stage location (S3 bucket)	Databricks doesn't offer UNLOAD to unload. Instead use: • INSERT OVERWRITE DIRECTORY (only on Databricks Runtime) • Use EXTERNAL TABLE definition pointing to required relocation • Use one of the Spark DataFrame API

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.