CTI DATA

White Paper

Migrating SQL Data to Databricks

A Foundation for Digital Transformation

Abstract

This whitepaper details a rapid method for migrating your SQL data to the Databricks SQL platform, illustrating the process through a clinical dataset while framing the benefits offuture-proofing the data architecture

Kurt Rosenfeld

krosenfeld@ctidata.com

CTI DATA

Migrating SQL Data to Databricks

A Foundation for Digital Transformation

Table of Contents

CTI DATA

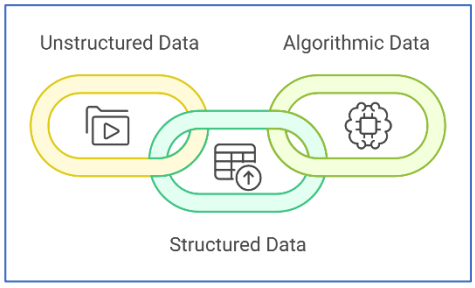
Migrating SQL Data to Databricks

A Foundation for Digital Transformation

1.0 Introduction

Shifting SQL workloads to a unified Data Infrastructure platform is increasingly important. Here's why: Historically, IT automation depended solely on structured data. Today, generative AI (GenAI) can surface information in the hordes of unstructured data, powering new automations.

Consequently, our systems must marry the unstructured "analog" data of the physical world with its structured "digital" twin. This requires a new perspective on IT data infrastructure. Existing infrastructure evolved fragmented: structured data residing in a row or columnar databases, object data residing as "blobs" in “containers” or “buckets,” and file data residing in file systems — each with distinct metadata and access controls.



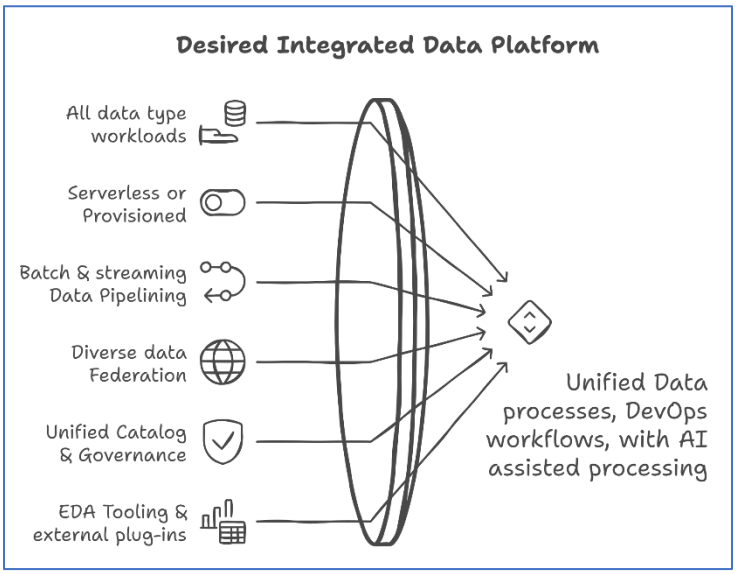
Such fragmentation is now a bottleneck; we need an infrastructure that treats structured and unstructured data as an integrated whole, along with a single metadata and relationship catalog, unified governance and control, within a uniform access platform.

The technology vendors have shoehorned their solutions towards this goal, adapting their SQL/NoSQL databases for object storage or extending their data lakes for SQL, with further adaptations for a new data class, the high-dimension vectors crucial to GenAI.

It’s a point-in-time mess, leaving the data engineer and end-user to work around the constraints of these platform legacies.

But it’s improving, and the vendors most relevant are the cloud data titans: Google, Microsoft, Amazon, Snowflake, and Databricks, with Salesforce and Oracle vying for their place.

The desired solution is as straightforward as depicted on the right.



The table on the next page calls out how well-positioned Databricks is for meeting this goal with the arrival of their serverless SQL capability in June 2022, deeply integrated with their open, multi-modal data platform heritage and processing tools.

Databricks' Position	
Any compute model	→ Serverless (SaaS) or managed (PaaS)
Any processing model	→ Python, R, Scala, Java, SQL
Rapid delivery	→ Built-in DevOps, DataOps, MLOps
Process all types of data	→ Data Lake with choices of structured abstractions
One data framework	→ From ingestion to micro-streaming
Single ecosystem, massive scale	→ Reduce technology silos, one platform for all needs
Cloud vendor abstraction	→ Mix and match any Cloud
Seamless data federation	→ External data is an equal citizen
Safe data sharing	→ Internal and external teams have governed access
Uniform governance & control	→ All data, all code, all security, all deployments
No vendor lock-in	→ 100% open technology

Consequently, running your SQL workloads on Databricks means you are futureproofing the data's value, regardless of how the data may need to evolve and integrate. Furthermore, even for one-off throw-away SQL workloads, Databricks is positioned as a perfectly competent platform.

To explore what is involved in migrating to Databricks, we need a legacy platform to make the details meaningful, so in this case, we assume Snowflake is the legacy.

These two platforms, with very different origins, now frequently compete as their technologies expanded and began to overlap. The chart below helps frame the current state but keep in mind we are singularly focused on the SQL aspects in this document.

Snowflake zero administration SQL service addressing analytic use cases	**Databricks** open-source computing and data service addressing data science use cases
Serverless SQL platform	Ecosystem tying together compute provisioning, structured and unstructured data management, and ML management Capable of serverless SQL operation (transparent dynamic provisioning of resources)
Supports non-SQL logic as a separate capability callable from SQL	Supports SQL processing as an equal citizen to its other computing capabilities
Data pipelining and orchestration require separate tooling	Integrates batch and real-time data pipelining and orchestration
Federates structured and semi-structured data	Federates any data
Governance and catalog addresses data only	Governed by a unified data and process manager with AI-assisted glossary generation
Plugins for popular BI platforms	AI-assisted exploratory data analysis (EDA) tooling with plugins for popular BI platforms

2.0 Database Migration Concerns

While SQL database platforms are conceptually similar – reflecting the power of SQL as a standard – all the platforms have feature sets with real differences, which, from a migration point-of-view, raise these migration concerns:

	Concern
Data	Adapting data types and converting formats
Logic	Adapting logic residing in the external programs (e.g. SQL statements) or in the data platform (e.g. Stored Procedures) to ensure functional equivalence
Process	Reproducing the program logic that controls data processing cycles
Security	Reproducing controls that prevent improper data access and creation
Operations	Reproducing infrastructure-as-code and other platform automation such as identity management, metering, and chargebacks
Meta-data	Transferring data descriptions and business rule descriptions

Applying these concerns specific to migrating Snowflake to Databricks, we note these strategies:

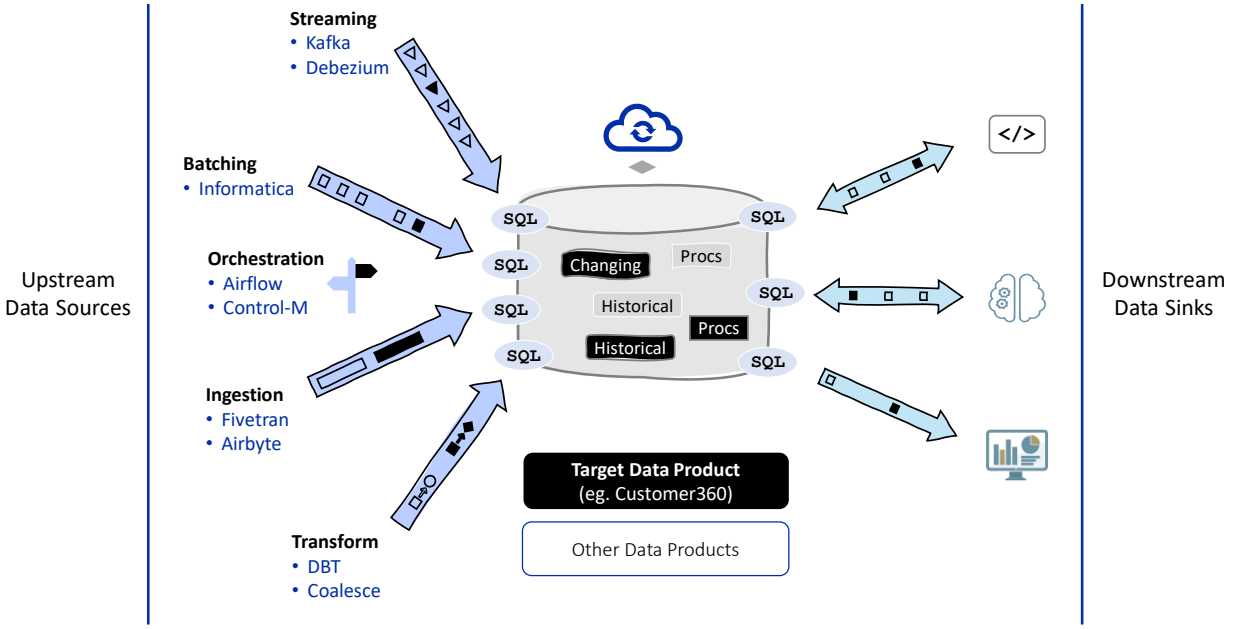
Technical Element	Migration Considerations	Migration Strategies
Data and Tables		
Data types	Similar, special handling for variants and geospatial	Review DDL, adapt dependent SQL
Views, materialized views	Same features and methods	Apply SQL dialect conversion automation
General Logic		
SQL dialect	Similar, ANSI SQL centric with minor differences	Apply SQL dialect conversion automation
Ingestion staging	Same features, similar methods	Repoint ingestion tool, or replace with Databricks Autoloader
Specialized Logic		
User Defined Functions	Same features with syntax & language differences	Convert syntaxes, maintain function signature
Stored Procedures and Triggers	Needs alternative solution	Replace with UDFs or Delta Live Tables

SQL refactoring	Identify high-cost queries (using query profiler)	Enhance with Databricks hashing/join features
Process Control		
CLI scripts	CLI's different, with little equivalence	Assess CLI script logic and recreate using Python, Databricks Jobs, etc.
Orchestration	Simple to convert Snowflake Tasks	Use richer features of Databricks Workflow (or repoint external tool)
Security		
Data privacy	Data masking, PII detection, clean rooms, audit differences	Map privacy setup, leverage Unity Catalog for data provenance
Identities, access control	Similar privilege/permission mgt, different row level security	Map and improve with Unity Catalog
Operational Control		
Data clustering	Clustering concepts similar, different syntaxes, less automated	Apply additional clustering based on query cost profiles
Resources, chargebacks	Different models, Databricks is more granular	Rethink to leverage Databricks tag-based granular capture
Elastic controls	Different methods/latency for automated scaling up/down	Assess scaling latency and use Databricks' more granular control

The top-of-mind topics are the first two, data and logic; we will dissect the first, the data topic, and touch on the logic topic, leaving a fuller examination for a later paper.

3.0 Data Product Trace Maps

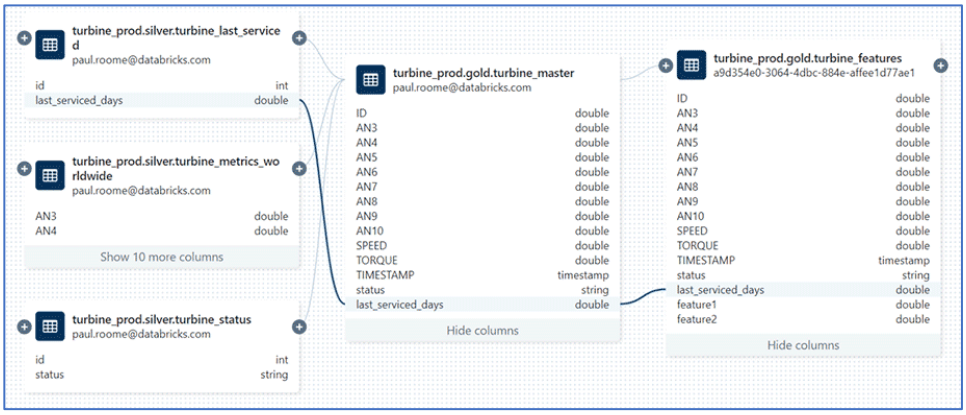
This diagram depicts the typical elements of an analytic data environment: various upstream services execute embedded SQL logic, stored procedures, or user-defined functions to produce or change data, with various downstream services consuming and sometimes producing or changing data.



We need a trace map of the dependency pathways for all the logic and related data in scope for the migration. A good practice is to scope the work by data product.

One of the beauties, from the point of view of mapping these dependency paths, is that SQL logic is almost always stateless: all state is stored in the data tables, the exception is when a stored procedure, trigger or custom function affects a variable.

This means modern data lineage tools can generate the trace map, like the diagram below, by probing the program objects of each deployed technology, extracting the logic, and identifying the discrete data elements that are touched, inferring the logic-to-data pathways.



But even so, a data engineer should inspect the results and fill in the gaps as necessary.

4.0 Migrating Logic

Logic migration is a distinctly different endeavor from data migration. The upstream and downstream technologies need two modifications to switch to the future platform: their data endpoint connections must be changed, which is simple enough, and their logic must be adapted to the differences in data types, data formats, logic dialects, and invocation methods.

Most SQL logic needs only minor changes. There is one exception: Stored Procedures and Triggers. Databricks SQL does not yet provide these features; if your data heavily depends on them, some "rethink" work will be necessary to recreate their functionality.

To get a feel for the SQL changes, consider the query below, which retrieves the top 5 pneumonia-related admissions for the past year with a mortality count. A typical question with a typical SQL expression.

```
SELECT
   concat(DR.DESCRIPTION,
      ' (', DR.DRG_CODE, ')')
               AS "DRG Description",
   count(DISTINCT A.HADM_ID)
               AS "Pneumonia Diagnosis",
   count(
      DISTINCT CASE
         WHEN A.HOSPITAL_EXPIRE_FLAG = TRUE
         THEN A.HADM_ID
         ELSE NULL
      END
   ) AS "Mortality Count"
FROM ADMISSIONS AS A
JOIN DIAGNOSES_ICD AS D
   ON A.HADM_ID = D.HADM_ID
JOIN D_ICD_DIAGNOSES AS DI
   ON D.ICD_VER_CODE = DI.ICD_VER_CODE
JOIN DRGCODES AS DR
   ON A.HADM_ID = DR.HADM_ID
WHERE
   date_part(YEAR, A.ADMITTIME) = 2024
   AND DI.LONG_TITLE ILIKE '%pneumonia%'
GROUP BY
   DR.DESCRIPTION,
   DR.DRG_CODE
ORDER BY
   "Pneumonia Diagnosis" DESC NULLS LAST
LIMIT 5
```

```
SELECT
   concat(DR.DESCRIPTION,
      ' (', DR.DRG_CODE, ')')
               AS `DRG Description,`
   count(DISTINCT A.HADM_ID)
               AS `Pneumonia Diagnosis`,
   count(
      DISTINCT CASE
         WHEN A.HOSPITAL_EXPIRE_FLAG = TRUE
         THEN A.HADM_ID
         ELSE NULL
      END
   ) AS `Mortality Count`
FROM ADMISSIONS AS A
JOIN DIAGNOSES_ICD AS D
   ON A.HADM_ID = D.HADM_ID
JOIN D_ICD_DIAGNOSES AS DI
   ON D.ICD_VER_CODE = DI.ICD_VER_CODE
JOIN DRGCODES AS DR
   ON A.HADM_ID = DR.HADM_ID
WHERE
   EXTRACT(YEAR FROM A.ADMITTIME) = 2024
   AND DI.LONG_TITLE ILIKE '%pneumonia%'
GROUP BY
   DR.DESCRIPTION,
   DR.DRG_CODE
ORDER BY
   `Pneumonia Diagnosis` DESC NULLS LAST
LIMIT 5
```

What changed? The Snowflake `date_part()` function is replaced with the ANSI SQL `extract()` function, and double quotes for enclosing identifiers are replaced with back-ticks (e.g. `Mortality Count`).

We used translation technology to make these changes, eliminating the labor of syntactic correction. However, the advancement of Large Language Models is fast displacing this method.

Nowadays, you can direct your LLM to review the reference documentation (see list below) along with your source code to provide conversion analysis and "first-pass" converted code. Although not yet perfect, the pace of improvement is remarkable. Databricks SQL has fast-evolving enhancements, making the LLM output generated from the reference documentation invaluable compared to almost immediately dated guides and other alternatives.

Snowflake and Databricks SQL Reference Documentation:

https://docs.snowflake.com/en/sql-reference/sql-all

https://docs.snowflake.com/en/sql-reference/functions-all

https://docs.databricks.com/en/sql/language-manual/sql-ref-datatypes.html

https://docs.databricks.com/en/sql/language-manual/sql-ref-functions-builtin-alpha.html

Snowflake Function	Databricks SQL Equivalent	Notes
`GET_DDL(object_name)`	`SHOW CREATE TABLE table_name` (for tables), query `information_schema.TABLES` or `information_schema.COLUMNS` for others	Databricks uses `SHOW CREATE` for tables. Use `information_schema` for programmatic metadata access.
`SYSTEM$GET_PREDECESSOR_RETURNED_COLUMNS(table_name)`	Not Directly Available. Consider Delta Lake Change Data Feed.	Snowflake-specific for CDC. Databricks uses Delta Lake CDC.
`SYSTEM$CLUSTERING_INFORMATION(table_name)`	Analyze `information_schema.COLUMNS` statistics; for Delta Lake, `DESCRIBE DETAIL table_name`.	Infer clustering from stats. Delta Lake shows partitioning/Z-Ordering.
`SYSTEM$TASK_HISTORY(...)`	Query `system.task_history` (if using Databricks Workflows).	Specific to Snowflake Tasks. Databricks Workflows has its own history table.
`SYSTEM$PIPE_STATUS('pipe_name')`	Monitor Auto Loader stream status via Spark Structured Streaming APIs or Delta Live Tables UI/APIs.	Specific to Snowflake Pipes. Databricks uses Auto Loader/DLT, monitored differently.
`CURRENT_REGION()`	Not Directly Available as a SQL function. Part of workspace config.	Databricks region is a configuration setting.
`GET_OBJECT_S3('s3://...')`	`spark.read.format('...').load('s3://...')` (replace '...' with format).	Databricks uses Spark's data source API.
`STAGE_FILES(...)` (external stages)	Use `dbutils.fs.ls('s3://...')` or cloud provider SDKs.	Databricks interacts with cloud storage directly.
`CONVERT_TIMEZONE(target_timezone,`	`from_utc_timestamp(to_utc_timestamp(timestamp, source_timezone), target_timezone)`	Convert to UTC first if source has timezone.

Snowflake Function	Databricks SQL Equivalent	Notes
source_timezone, timestamp)		
DATE_PART(date_or_time_part, expression)	extract(date_or_time_part FROM expression) or year(expression), month(expression), etc.	Databricks offers both general and specific date/time functions.
DATE_TRUNC(date_or_time_part, timestamp)	date_trunc(date_or_time_part, timestamp)	Function name is the same.
MASK(input_string, ...)	Requires custom UDF or use Unity Catalog's data masking.	Snowflake's MASK is built-in; Databricks needs UDF or Unity Catalog.
STRTOK_TO_ARRAY(string, delimiter)	split(string, delimiter)	Different function name.
SPLIT_TO_TABLE(input, delimiter)	explode(split(input, delimiter))	Use explode after splitting.
RLIKE(subject, pattern)	subject rlike pattern	Uses the rlike operator.
LIKEANY(string, pattern1, pattern2, ...)	string LIKE pattern1 OR string LIKE pattern2 OR ...	Requires explicit OR conditions.
SOUNDEX(string)	Requires custom UDF or external Spark library.	Snowflake built-in; Databricks needs UDF/external library.
LEVENSHTEIN(string1, string2)	Requires custom UDF or external Spark library.	Similar to SOUNDEX.
LISTAGG(expression [, delimiter]) [WITHIN GROUP (ORDER BY ...)] [ON OVERFLOW TRUNCATE 'string' [WITHOUT COUNT | WITH COUNT]]	array_join(collect_list(expression) WITHIN GROUP (ORDER BY ordering_expression), delimiter) OVER (PARTITION BY grouping_expression)	WITHIN GROUP moved inside collect_list. Manual overflow handling.
BITAND(expr1, expr2)	expr1 & expr2	
BITOR(expr1, expr2)	expr1 | expr2	
BITXOR(expr1, expr2)	expr1 ^ expr2	
BITNOT(expr1)	~ expr1	
IFF(condition, true_value, false_value)	CASE WHEN condition THEN true_value ELSE false_value END	Standard SQL CASE WHEN.
NVL(expr1, expr2)	coalesce(expr1, expr2)	Standard SQL coalesce.
ZEROIFNULL(numeric_expr)	CASE WHEN numeric_expr IS NULL THEN 0 ELSE numeric_expr END	Achieved using CASE WHEN.

Snowflake Function	Databricks SQL Equivalent	Notes
`TRY_CAST(source_value AS data_type)`	`TRY_CAST(source_value AS data_type)` (newer), or `CASE WHEN TRY_CAST(...) IS NOT NULL THEN TRY_CAST(...) END`	`TRY_CAST` available in recent runtimes.
`PARSE_JSON(string)`	`from_json(string, schema_of_json(string))`	Requires schema; `schema_of_json` can infer.
`GET_PATH(variant, path)`	`get_json_object(variant, path)` or `variant.path.to.element`	Both function and dot notation available.
`FLATTEN(input => array_or_object)`	`explode(input)`	Uses the `explode` function.
`XMLGET(xml, tag)`	Requires Spark XML libraries (e.g., `xml_xpath` after parsing).	Databricks uses Spark for XML processing.
`POLICY_CONTEXT(...)`	Not Directly Available. Use Unity Catalog features.	Databricks uses Unity Catalog for security policies.
`ARRAY_CONSTRUCT(...)`	`array(...)`	Databricks `array()` function.

5.0 Migrating Data

Historical data is unique in that it reflects business rules from the past, manifest in the data logic and the data inputs at the time, which may no longer be available. It is almost always preferable to migrate historical data rather than regenerate it by reconstituting the historical logic with its historical inputs.

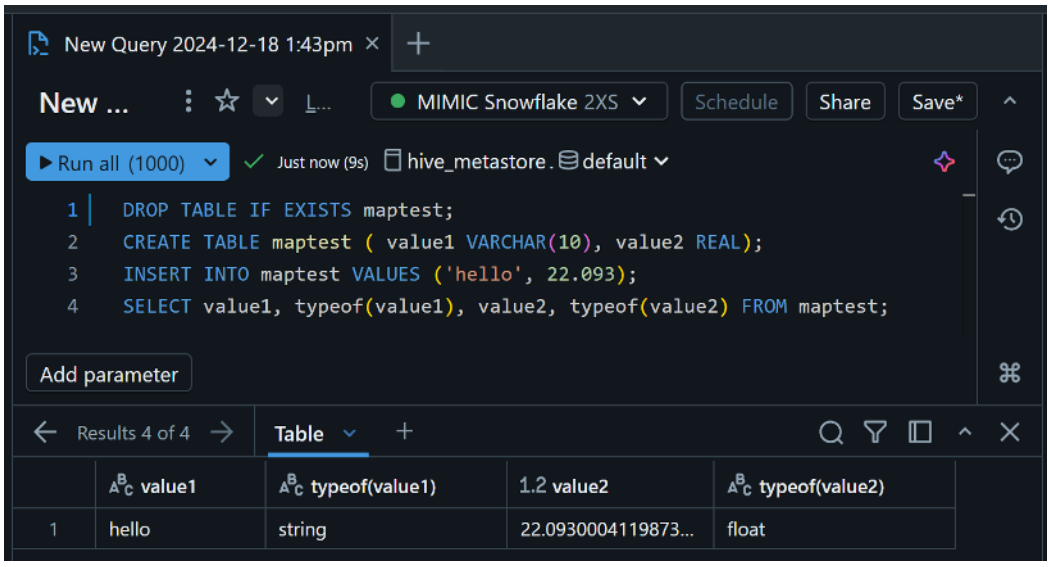
The migration process has one goal: reproduce the historical data adjusted for data type and format differences keeping the data rules intact. While there may be pressure to improve the data – be it to correct, enrich, or mask certain aspects – which is to be welcomed since these demands are a sponsorship opportunity – such improvements should be implemented as a separate step once the data is migrated, thereby isolating change and retaining proof of faithful data reproduction.

Once again, LLM advancements make understanding the differences a breeze; in this case, we engage the Databricks Assistant, which employs Databrick's own LLM service, to report ANSI SQL peculiars.

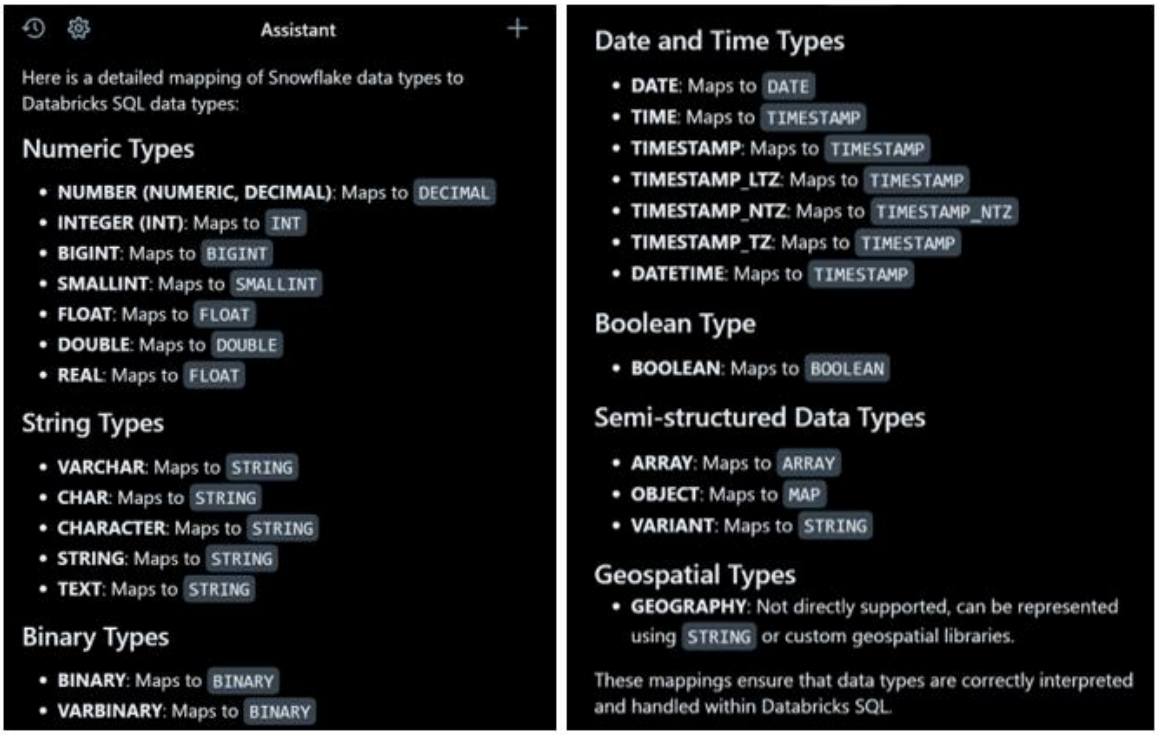
One main difference is that all character fields map to the `STRING` type; therefore, the character limit enforcement of `VARCHAR(n)` is lost and must be handled in logic if required.

[For the current Databricks data types, see https://docs.databricks.com/en/sql/language-manual/sql-ref-datatypes.html]

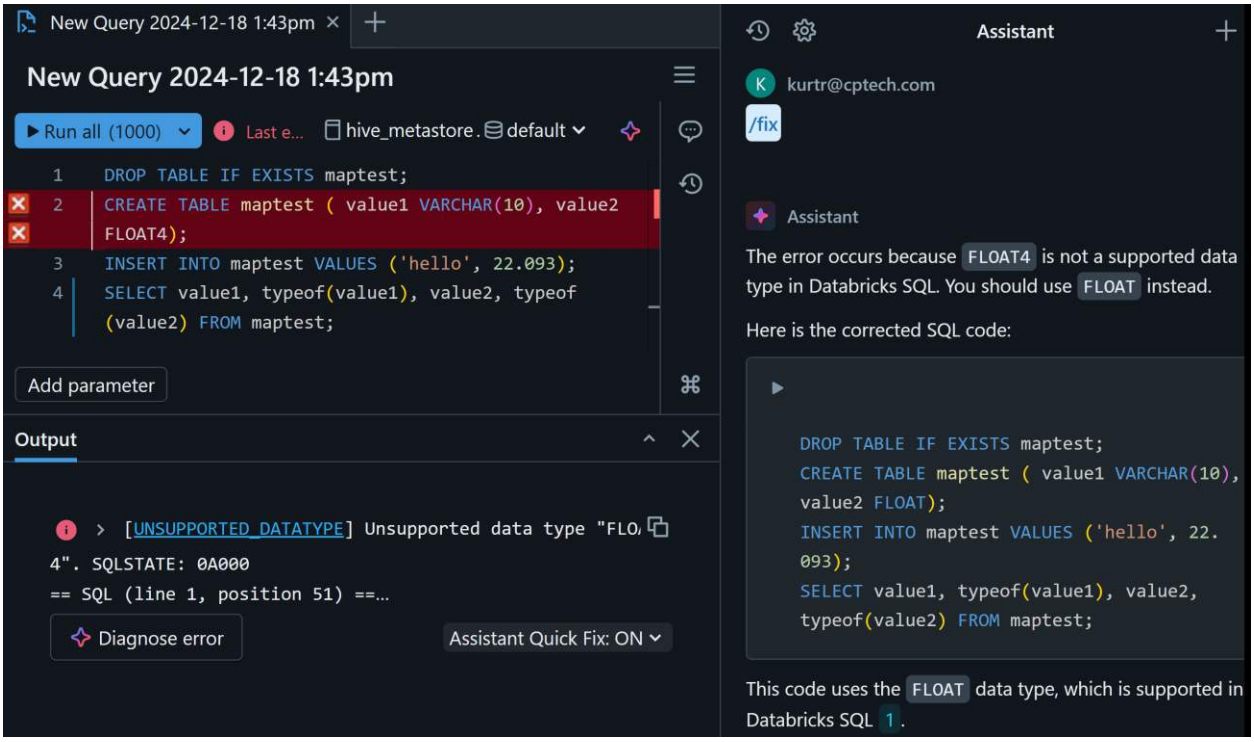
We observe below Databricks automatically handling this ANSI SQL behavior where VARCHAR(10) and REAL are converted to the Databricks SQL equivalent.



Since we will be migrating data from Snowflake, we must examine how Snowflake's non-ANSI data types will be handled.



The Databricks Assistant is a little too high level; Snowflake also supports FLOAT4 and FLOAT8. Databricks SQL errors out when presented with these, which is reasonable since they are not ANSI standard, and the Databricks Assistant recommends using FLOAT as the proper alternative.

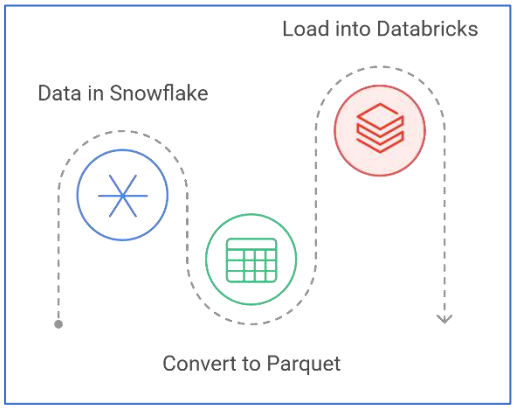


A SQL programmer might address data conversion by developing go-between migration logic: reading the source data via the Snowflake SQL client, casting the data to ANSI standard, and then writing the result to Databricks using the Databricks SQL client.

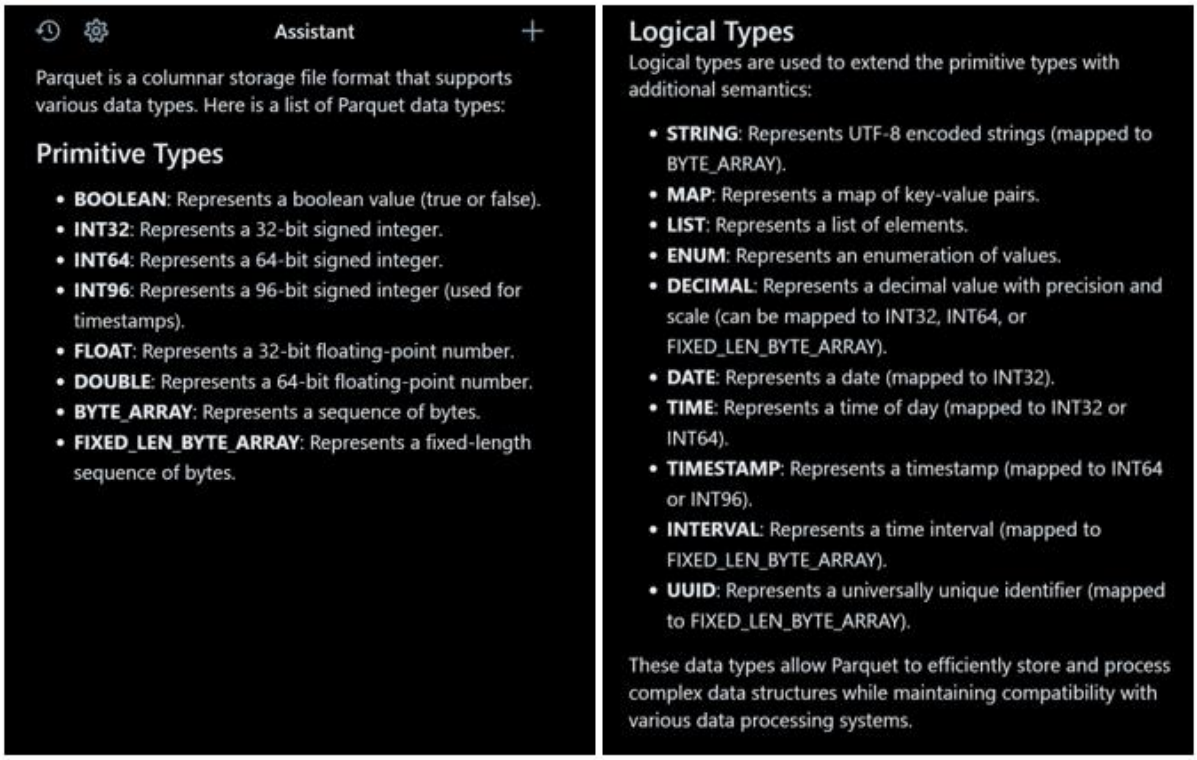
However, in the case of Snowflake, we can use Snowflake's internal conversion engine by way of Parquet, the highly efficient and flexible open data store format supported by the major data platforms.

When a Snowflake table is exported to Parquet, its native data types are converted to Parquet's. When imported to Databricks, the reverse happens, and since Databricks is heavily aligned with Parquet, little change is necessary. Essentially, Snowflake is exporting its tables into structures that align well with Databricks.

We can transfer tables with hundreds to billions of rows, bypassing the need for complex and computationally expensive conversion pipelines. Of course, some differences must be inspected afterward, such as `VARCHAR(n)`.



The Databricks Assistant provides the following elaboration regarding Parquet data types:



Assistant

Parquet is a columnar storage file format that supports various data types. Here is a list of Parquet data types:

Primitive Types

- **BOOLEAN**: Represents a boolean value (true or false).
- **INT32**: Represents a 32-bit signed integer.
- **INT64**: Represents a 64-bit signed integer.
- **INT96**: Represents a 96-bit signed integer (used for timestamps).
- **FLOAT**: Represents a 32-bit floating-point number.
- **DOUBLE**: Represents a 64-bit floating-point number.
- **BYTE_ARRAY**: Represents a sequence of bytes.
- **FIXED_LEN_BYTE_ARRAY**: Represents a fixed-length sequence of bytes.

Logical Types

Logical types are used to extend the primitive types with additional semantics:

- **STRING**: Represents UTF-8 encoded strings (mapped to BYTE_ARRAY).
- **MAP**: Represents a map of key-value pairs.
- **LIST**: Represents a list of elements.
- **ENUM**: Represents an enumeration of values.
- **DECIMAL**: Represents a decimal value with precision and scale (can be mapped to INT32, INT64, or FIXED_LEN_BYTE_ARRAY).
- **DATE**: Represents a date (mapped to INT32).
- **TIME**: Represents a time of day (mapped to INT32 or INT64).
- **TIMESTAMP**: Represents a timestamp (mapped to INT64 or INT96).
- **INTERVAL**: Represents a time interval (mapped to FIXED_LEN_BYTE_ARRAY).
- **UUID**: Represents a universally unique identifier (mapped to FIXED_LEN_BYTE_ARRAY).

These data types allow Parquet to efficiently store and process complex data structures while maintaining compatibility with various data processing systems.

Meta-data stored in Parquet elaborates the primitive types, ensuring proper interpretation as the appropriate ANSI SQL data type. Here is an example of this process after transferring a patient `ADMISSIONS` table from Snowflake to Databricks. The screenshots on the next page are from the native SQL workbooks of both products.

– Snowflake –

```
USE patient.silver;
DESCRIBE TABLE admissions
```

Results | Chart

	name	type
1	SUBJECT_ID	NUMBER(38,0)
2	HADM_ID	NUMBER(38,0)
3	ADMITTIME	TIMESTAMP_NTZ(9)
4	DISCHTIME	TIMESTAMP_NTZ(9)
5	DEATHTIME	TIMESTAMP_NTZ(9)
6	ADMISSION_TYPE	VARCHAR(255)
7	ADMIT_PROVIDER_ID	VARCHAR(10)
8	ADMISSION_LOCATION	VARCHAR(255)
9	DISCHARGE_LOCATION	VARCHAR(255)
10	INSURANCE	VARCHAR(255)
11	LANGUAGE	VARCHAR(255)
12	MARITAL_STATUS	VARCHAR(255)
13	RACE	VARCHAR(255)
14	EDREGTIME	TIMESTAMP_NTZ(9)
15	EDOUTTIME	TIMESTAMP_NTZ(9)
16	HOSPITAL_EXPIRE_FLAG	BOOLEAN
17	ADMDATE	TIMESTAMP_NTZ(9)
18	ADMITTIME_DAYOFMONTH	NUMBER(38,0)
19	DISCHTIME_DAYOFMONTH	NUMBER(38,0)
20	ADM_DISCH_INSAMEYEAR	NUMBER(38,0)

– Databricks –

```
USE patient_import.silver;
DESCRIBE TABLE admissions
```

Results 2 of 2 | Table

	col_name	data_type
1	SUBJECT_ID	decimal(38,0)
2	HADM_ID	decimal(38,0)
3	ADMITTIME	timestamp
4	DISCHTIME	timestamp
5	DEATHTIME	timestamp
6	ADMISSION_TYPE	string
7	ADMIT_PROVIDER_ID	string
8	ADMISSION_LOCATION	string
9	DISCHARGE_LOCATION	string
10	INSURANCE	string
11	LANGUAGE	string
12	MARITAL_STATUS	string
13	RACE	string
14	EDREGTIME	timestamp
15	EDOUTTIME	timestamp
16	HOSPITAL_EXPIRE_FLAG	boolean
17	ADMDATE	timestamp
18	ADMITTIME_DAYOFMONTH	decimal(38,0)
19	DISCHTIME_DAYOFMONTH	decimal(38,0)
20	ADM_DISCH_INSAMEYEAR	decimal(38,0)

A cardinality count provides some assurance the data is transferred correctly:

– Snowflake –

```
USE patient.silver;
select
    count(*) as "Records",
    count(distinct HADM_ID) as "Admissions",
    count(distinct SUBJECT_ID) as "Patients"
from ADMISSIONS
```

Results | Chart

	Records	Admissions	Patients
1	431231	431231	180733

– Databricks –

```
USE patient_import.silver;
select
    count(*) as `Records`,
    count(distinct HADM_ID) as `Admissions`,
    count(distinct SUBJECT_ID) as `Patients`
from ADMISSIONS
```

Results 2 of 2 | Table

	Records	Admissions	Patients
1	431231	431231	180733

6.0 Dissecting a Data Example

Our work in complex data solutions provides access to diverse data environments, one being an anonymized dataset of 4.75 million clinical diagnoses that we used to develop disease forecasting algorithms stored in Snowflake. In this walkthrough, we shall migrate the entire dataset to Databricks, reproducing the schema and stored values.

We export the Snowflake tables to Parquet and subsequently import them into Databricks using a Microsoft Azure storage container to hold the Parquet intermediary objects. We establish the container in the same region as our Snowflake and Databricks tenants for speed and reduced egress charges. As stated earlier, this is faster and cheaper than using a live database-to-database SQL connection to read from the source and write to the target, benefiting from Parquet as the data type go-between.

6.1 Environment Setup

For our purposes, Databricks provides two IDEs for working with SQL code. The first is their Jupyter-based notebook; although primarily for Python, it can also process SQL using the "% SQL" magic declaration of Jupyter. Its objects are labeled "Notebook" within the Databricks workspace explorer.

The second is a SQL IDE that processes SQL expressions; its objects are labeled "Query" within the workspace explorer. It is similar in concept to a Snowflake worksheet; for convenience, we'll call it the "SQL worksheet" environment.

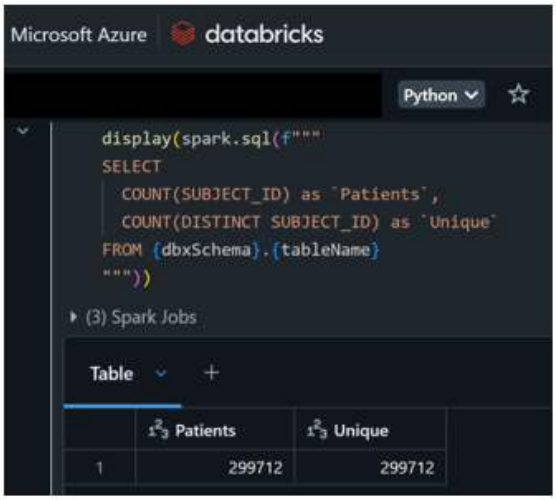
A SQL worksheet is attached to a serverless SQL configuration – called a "warehouse" – that defines the initial size of the compute that automatically responds to demand. In contrast, a Notebook is attached to a cluster configuration that identifies the compute plus runtime environment and must always be running to respond to demand.

There is an unusual exception: a Jupyter Notebook of SQL cells may execute on the serverless SQL warehouse; however, it will refuse if there are Python or other non-SQL cells.

Finally, an external IDE (such as Visual Studio) may also connect to a cluster or a SQL warehouse for performing development iterations.

Regardless of where execution occurs, whether on a cluster or serverless warehouse, all data is processed and stored within the Databricks platform and instantly accessible to both.

The following three screenshots demonstrate these options:



Databricks Notebook – Python Cell

SQL expression executed within a Python `spark()` call with the SQL statement passed as a string.

Variables are passed into the SQL statement using f-string replacements, with the variables previously set:

```
dbxSchema = 'patient_import.silver'
tableName = 'patients'
```

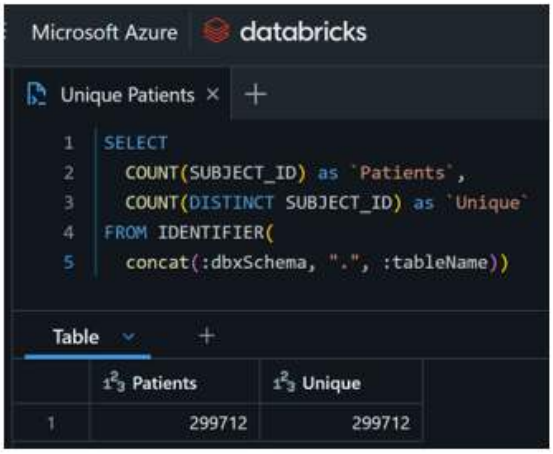
Databricks Notebook – SQL Cell

Alternatively, the notebook cell can be flipped to a SQL magic cell benefitting from IDE assistance in crafting the SQL and navigating errors.

Variables are passed as spark keys, with the keys previously set:

```
spark.conf.set(
  "stagefile.schema", dbxSchema)
```

Note: the same "key" approach could also replace the f-string variables used earlier.

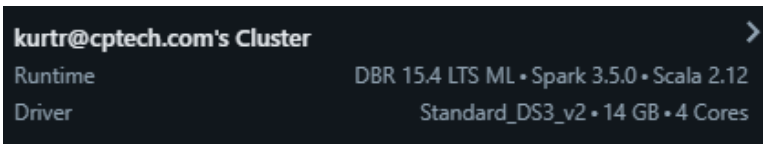


Databricks SQL editor worksheet

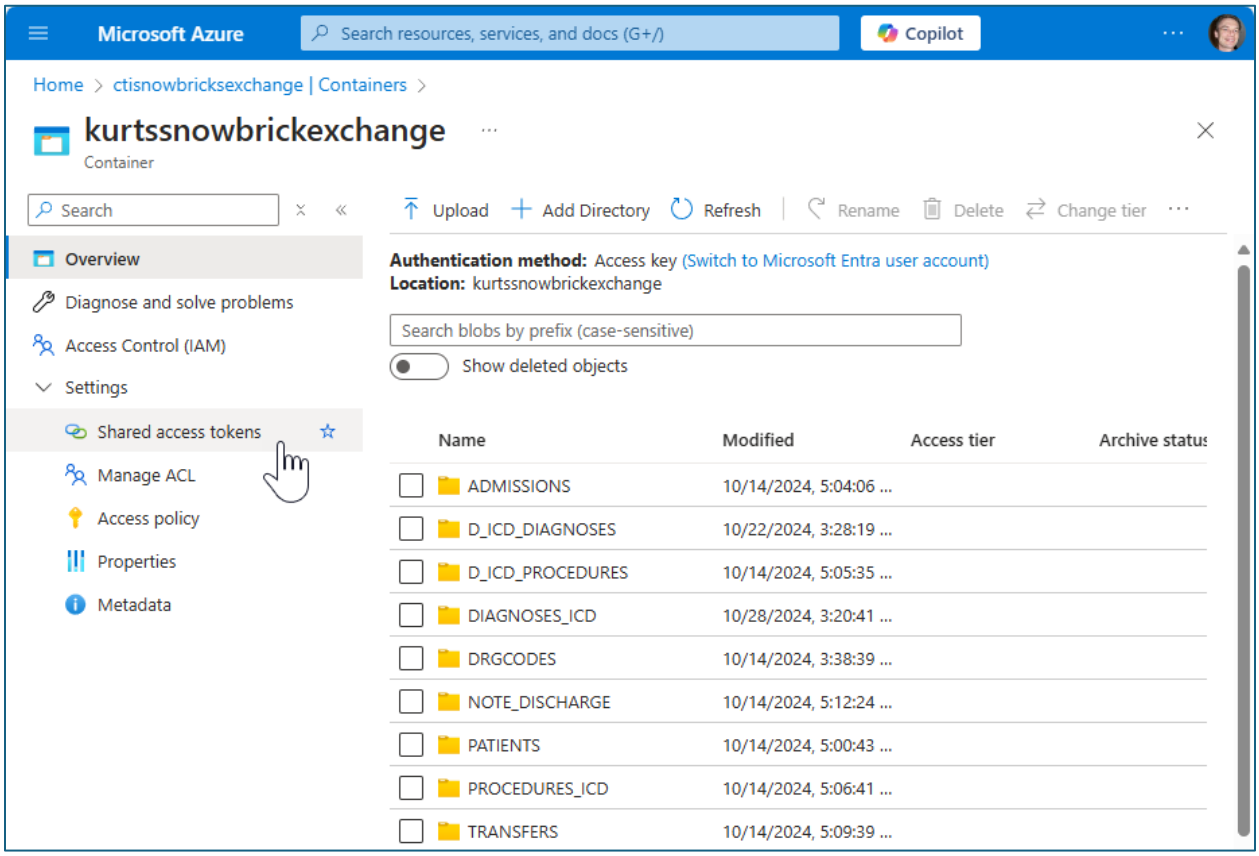
The same expression in the Databricks SQL editor benefits from a more robust SQL-centric interface.

Variables are passed as parameters (we've used the same names in the example, but they are independent of the Python variables)

Because our data migration code needs to coordinate data movement from Snowflake to Azure to Databricks, we shall use the flexibility of a Notebook to step through this sequence, sometimes switching between Python and SQL cells. We've provisioned a minimally sized cluster for this purpose.



We also need an ADLS storage container with a shared access signature (SAS) token that the Notebook will use. It's more interesting to see the container populated with the exported tables in the screenshot below, but it is empty until Step 1 is completed later.



6.2 Secrets and Run-Time Variables Setup

It is always preferable to use a "secrets vault" rather than environment variables to store and retrieve sensitive credentials; in this case, we've used Databricks Secrets. We set up the secrets by opening the cluster terminal and using the CLI.

```
#databricks secrets put-secret dbmigrate ADLS_ACCOUNT --string-value "<ADLS ACCOUNT NAME>"
#databricks secrets put-secret dbmigrate ADLS_SAS_TOKEN --string-value "<ADLS SAS TOKEN>"
#databricks secrets put-secret dbmigrate SNOW_USER --string-value "<SNOWFLAKE USER ACCOUNT>"
#databricks secrets put-secret dbmigrate SNOW_PWD --string-value "<SNOWFLAKE USER PASSWORD>"
#databricks secrets put-secret dbmigrate SNOW_ACCOUNT --string-value "<SNOWFLAKE ACCOUNT>"
```

The Databricks Notebook is “secrets aware”, displaying “redacted” for variables assigned from a Databricks Secret. We load the secrets into a Python dictionary

```
# Load all the project secrets into a dictionary for convenience
secretStore = "dbmigrate"
secrets = {s.key: dbutils.secrets.get(scope=secretStore, key=s.key)
           for s in dbutils.secrets.list(secretStore)}
secrets
```

```
{'ADLS_ACCOUNT': '[REDACTED]',
 'ADLS_SAS_TOKEN': '[REDACTED]',
 'SNOW_ACCOUNT': '[REDACTED]',
 'SNOW_PWD': '[REDACTED]',
 'SNOW_ROLE': '[REDACTED]',
 'SNOW_USER': '[REDACTED]'}
```

and use them to populate the connection parameters for Snowflake, Databricks, and ADLS.

```
### ADLS container identifiers
azStorageAccount, azSAStoken = secrets['ADLS_ACCOUNT'], secrets['ADLS_SAS_TOKEN']
azBlobContainer = "kurtssnowbrickexchange"

# snowflake identifiers
snowBlobURL = f'azure://{azStorageAccount}.blob.core.windows.net/{azBlobContainer}'
snowCreds = { 'warehouse': 'DEMO_WH',
              'user': secrets['SNOW_USER'],
              'account': secrets['SNOW_ACCOUNT'],
              'password': secrets['SNOW_PWD'] }
snowStage = "DATABRICKS_IMPORT"
snowSchema = "patient.silver"

### Databricks identifiers
dbxBlobURL = f"wasbs://{azBlobContainer}@{azStorageAccount}.blob.core.windows.net"
dbxSchema = "patient_import.silver"  # Unity Catalog name for the import location

# Populate spark keys that can serve as variables in the notebook SQL cells later
spark.conf.set("stagefile.accessToken", azSAStoken)
spark.conf.set("stagefile.catalog", dbxSchema.split('.')[0])
spark.conf.set("stagefile.schema", dbxSchema)
```

We create a convenience function, `snow_exec()`, that wraps the Snowflake Python library to return a dictionary of column names with row data.

```
# Setup the snowlfake connection used to execute the per-table exports in the main cycle

import snowflake.connector
# Even though we fully qualify the schema.table in the SQL statements later,
# we also force the schema default in the session connection for redundancy
snowCreds |= { 'database': snowSchema.split(".")[0], 'schema': snowSchema.split(".")[1] }
snow_con = snowflake.connector.connect(**snowCreds)

def snow_exec(sql, snow_con=snow_con):
    cur = snow_con.cursor().execute(sql)
    result = { 'columns': [desc[0] for desc in cur.description],
               'data': cur.fetchall() }
    cur.close()
    return result

import pandas as pd
# packaging SQL results in a dataframe gets nicely displayed in the Databricks notebook
prettify = lambda sql_result: pd.DataFrame(**sql_result)
```

Now, we're ready to work on our first target: transferring 4.75m rows of diagnosis data to Databricks.

Step 1: Export the Source Table from Snowflake to the Cloud Container

The ADLS container is identified to Snowflake as a stage area:

```
# Create the snowflake stage area connected to the external Azure container
# This only needs to be run once

snow_exec(f"""
CREATE STAGE IF NOT EXISTS {snowStage}
    URL = '{snowBlobURL}',
    CREDENTIALS = ( AZURE_SAS_TOKEN = '{azSAStoken}' )
    DIRECTORY = ( ENABLE = true );
""" )
```

```
{'columns': ['status'],
 'data': [('DATABRICKS_IMPORT already exists, statement succeeded.',)]}
```

This statement employs a Python f-string using the connection variables defined earlier:

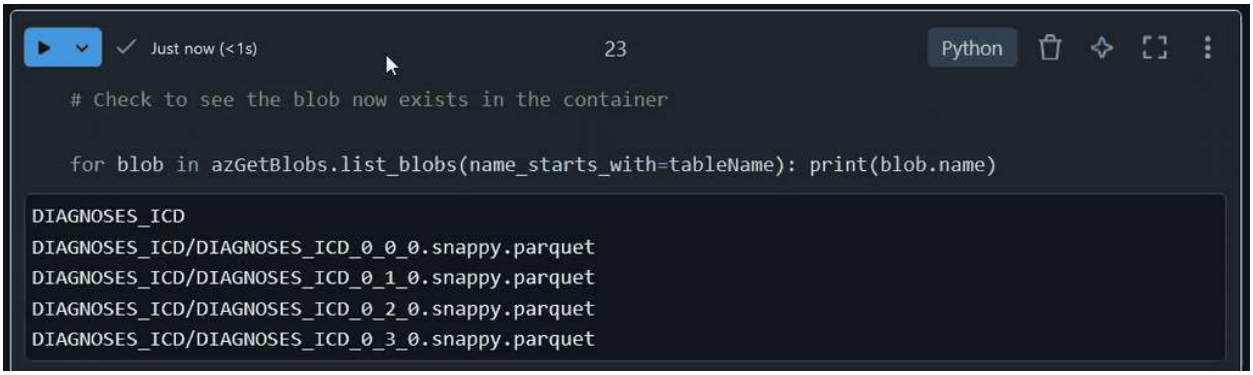
`snowStage`	The name within Snowflake for the stage area
`snowBlobURL`	Location of the stage's storage, in this case, the Azure container

azSAStoken	The Azure token that grants access to the container

We execute a `COPY INTO` to generate the collection of Parquet objects in the stage area:

tableName	DIAGNOSES_ICD

The Snowflake statement converts the table into snappy compressed Parquet objects, each about 25MB in size, under a folder with the same name as the table. We query ADLS to observe the objects:



These objects can now be loaded into Databricks.

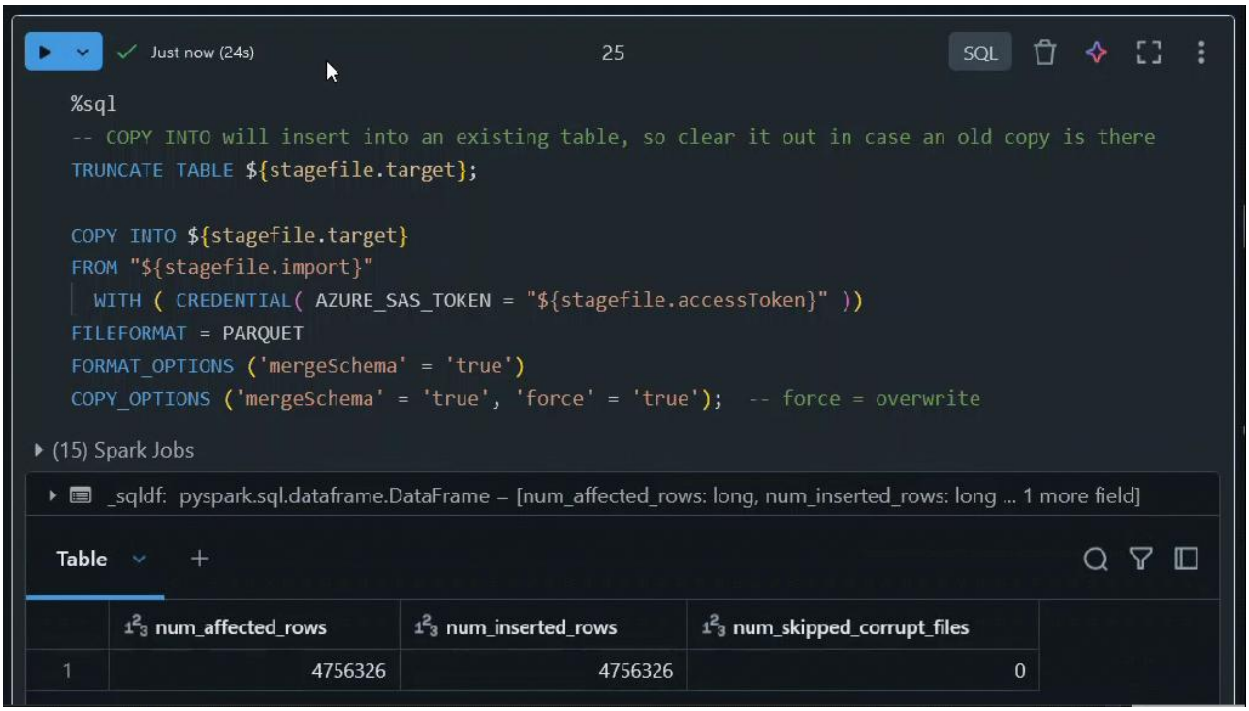
Step 2: Import the Parquet Objects into Databricks SQL under Unity Catalog

First, we set up identifiers to make the process repeatable for different tables:

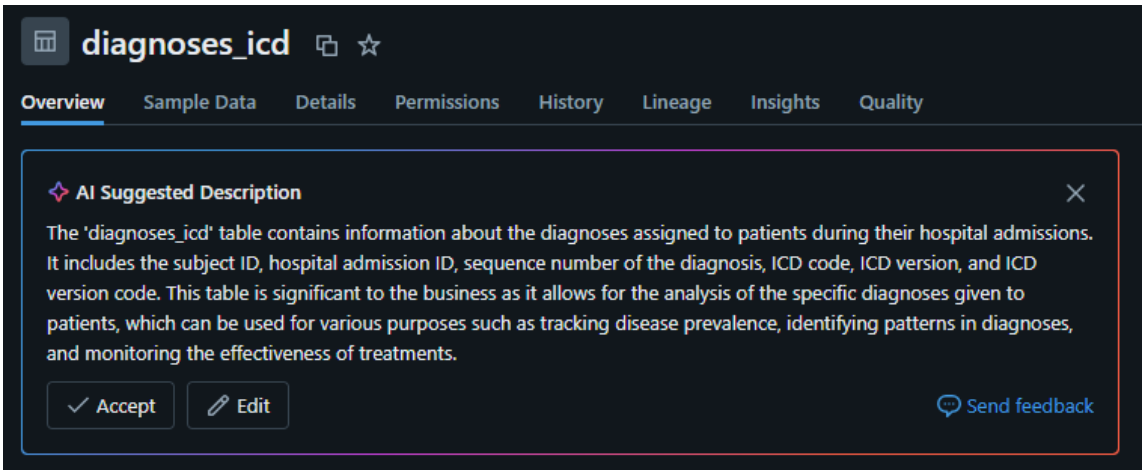
Notice the target table `patient_import.silver.DIAGNOSES_ICD` is described using the 3-level namespace of Databricks Unity Catalog: `<catalog>.<schema>.<data>` similarly to the way Snowflake manages datasets.

This requires the catalog and schema must already exist.

As the data is loaded, Unity Catalog automatically collects the meta-data from the Parquet objects. The load process is like the earlier export process but is reversed.



Notice the row count matches the earlier export count. The data is managed through Unity Catalog, which provides a remarkably helpful "starter" glossary definition, demonstrating how effectively GenAI seamlessly integrates into the user workflow.



Step 3: Rinse

The rinsing step is to validate the data. We've already noted the row counts match. We examine the meta-data equivalence:

– Snowflake –

```
# Get the Snowflake meta-data for the table
prettify(snow_exec(f"""
    DESCRIBE TABLE {snowSchema}.{tableName}
"""))
```

	name	type	kind	null?
0	SUBJECT_ID	NUMBER(38,0)	COLUMN	N
1	HADM_ID	NUMBER(38,0)	COLUMN	N
2	SEQ_NUM	NUMBER(38,0)	COLUMN	N
3	ICD_CODE	VARCHAR(255)	COLUMN	N
4	ICD_VERSION	NUMBER(38,0)	COLUMN	N
5	ICD_VER_CODE	VARCHAR(100)	COLUMN	Y

– Databricks –

```
%sql
-- Get the Databricks meta-data for the table
DESCRIBE TABLE ${stagefile.target}
```

Table +

	col_name	data_type	comment
1	SUBJECT_ID	decimal(38,0)	null
2	HADM_ID	decimal(38,0)	null
3	SEQ_NUM	decimal(38,0)	null
4	ICD_CODE	string	null
5	ICD_VERSION	decimal(38,0)	null
6	ICD_VER_CODE	string	null

As expected, `VARCHAR` has been converted to `STRING`. We test the cardinality of the 4.75 million rows by checking the counts of the distinct `SUBJECT_ID`:

– Snowflake –

```
# Get the Snowflake meta-data for the table
prettify(snow_exec(f"""
    SELECT COUNT(DISTINCT SUBJECT_ID)
    FROM {snowSchema}.{tableName}
"""))
```

	COUNT(DISTINCT SUBJECT_ID)
0	180640

– Databricks –

```
%sql
-- Get the Databricks meta-data for the table
SELECT COUNT(DISTINCT SUBJECT_ID)
FROM ${stagefile.target}
```

Table +

	count(DISTINCT SUBJECT_ID)
1	180640

Step 4: Repeat

The same process is repeated for the remaining schema tables.

Keep in mind the method we've presented is to demonstrate concepts. It lacks robust error handling and needs deeper data validation that should be automated.

Once all the necessary tables are transferred, a business query demonstrates accurate relationships, cardinality, consistency of field values, and their correct filtering and aggregation handling.

Here is the "pneumonia" business statement from earlier, first running in Snowflake, the original source:

```
# Snowflake Top 5 DRGs with the highest number of pneumonia diagnoses
snow_exec(f"USE {snowSchema}")
prettify(snow_exec(localize_sql(business_sql_test, "snowflake")))
```

	DRG Description	Pneumonia Diagnosis	Mortality Count
0	OTHER PNEUMONIA (139)	41	0
1	SEPTICEMIA & DISSEMINATED INFECTIONS (720)	36	8
2	HEART FAILURE (194)	13	0
3	MAJOR RESPIRATORY INFECTIONS & INFLAMMATIONS (...	9	1
4	TRACHEOSTOMY W MV 96+ HOURS W EXTENSIVE PROCED...	8	0

In the code, the variable `business_sql_test` holds the statement, which is syntax adapted for Snowflake using `localize_sql()`, a simple wrapper we created for the popular `SQLglot` library. Here is the same statement producing the same results, executing in Databricks:

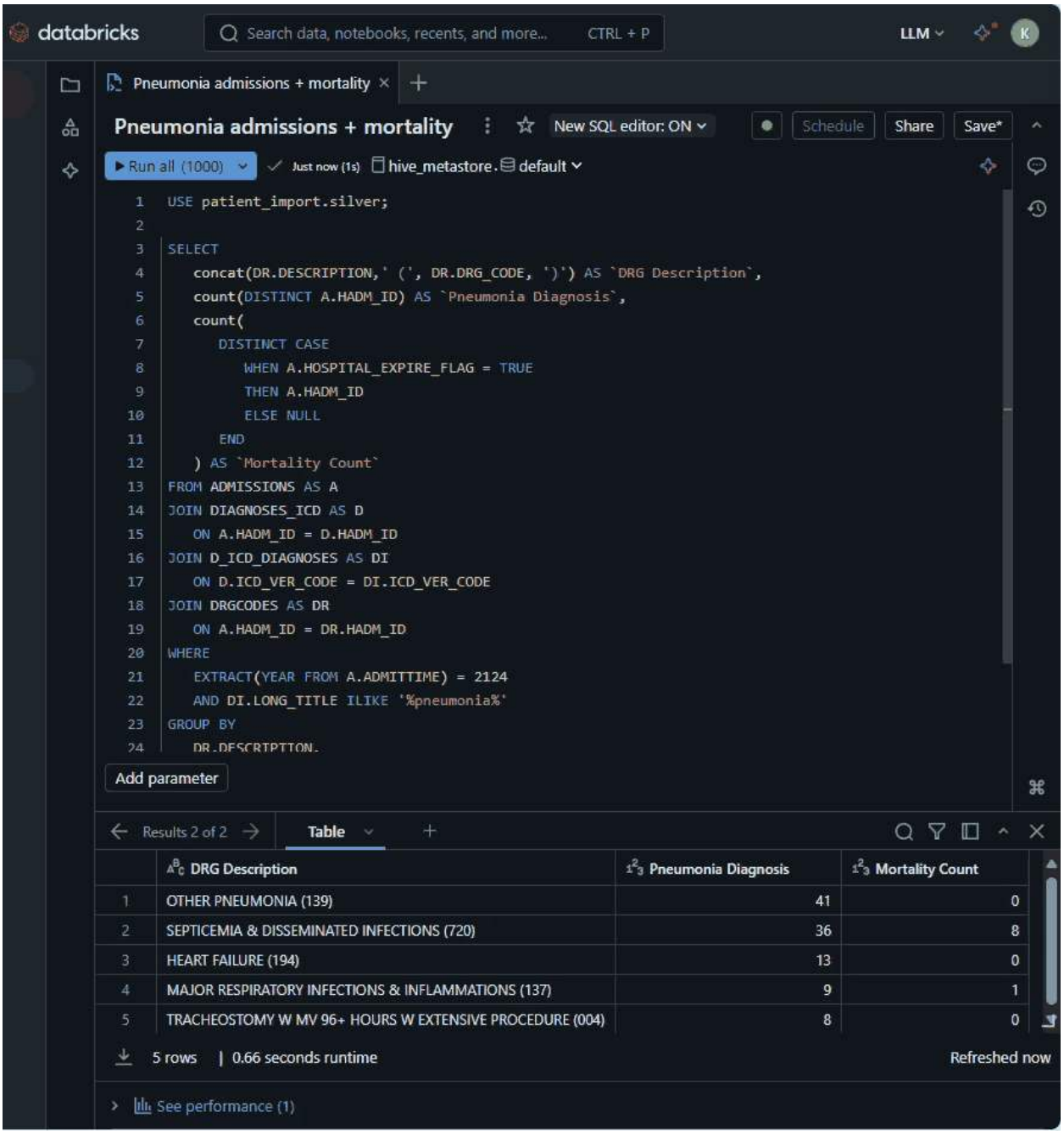
```
# Databricks Top 5 DRGs with the highest number of pneumonia diagnoses
spark.sql(f"USE {dbxSchema}")
display(spark.sql(localize_sql(business_sql_test, "databricks")))
```

▸ (6) Spark Jobs

Table +

	DRG Description	Pneumonia Diagnosis	Mortality Count
1	OTHER PNEUMONIA (139)	41	0
2	SEPTICEMIA & DISSEMINATED INFECTIONS (720)	36	8
3	HEART FAILURE (194)	13	0
4	MAJOR RESPIRATORY INFECTIONS & INFLAMMATIONS (137)	9	1
5	TRACHEOSTOMY W MV 96+ HOURS W EXTENSIVE PROCEDURE (004)	8	0

Although unnecessary, on the next page you can see the full statement running within Databricks SQL producing the same results, illustrating that Databricks SQL and the Databricks cluster are interacting with the same data store within the Databricks platform.

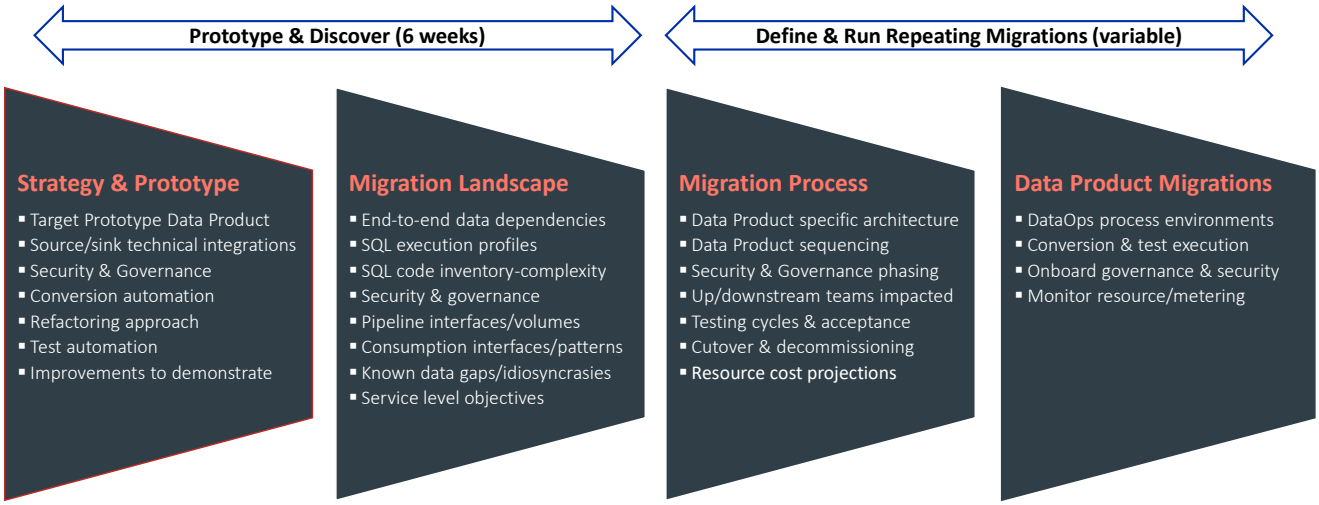

databricks
Search data, notebooks, recents, and more... CTRL + P
LLM
Pneumonia admissions + mortality
Pneumonia admissions + mortality
New SQL editor: ON
Schedule
Share
Save*
Run all (1000)
Just now (1s)
hive_metastore.default
USE patient_import.silver;
SELECT
concat(DR.DESCRIPTION,' (', DR.DRG_CODE, ')') AS `DRG Description`,
count(DISTINCT A.HADM_ID) AS `Pneumonia Diagnosis`,
count(
DISTINCT CASE
WHEN A.HOSPITAL_EXPIRE_FLAG = TRUE
THEN A.HADM_ID
ELSE NULL
END
) AS `Mortality Count`
FROM ADMISSIONS AS A
JOIN DIAGNOSES_ICD AS D
ON A.HADM_ID = D.HADM_ID
JOIN D_ICD_DIAGNOSES AS DI
ON D.ICD_VER_CODE = DI.ICD_VER_CODE
JOIN DRGCODES AS DR
ON A.HADM_ID = DR.HADM_ID
WHERE
EXTRACT(YEAR FROM A.ADMITTIME) = 2124
AND DI.LONG_TITLE ILIKE '%pneumonia%'
GROUP BY
DR.DESCRIPTION,
Add parameter
Results 2 of 2
Table
DRG Description
Pneumonia Diagnosis
Mortality Count
1 OTHER PNEUMONIA (139) 41 0
2 SEPTICEMIA & DISSEMINATED INFECTIONS (720) 36 8
3 HEART FAILURE (194) 13 0
4 MAJOR RESPIRATORY INFECTIONS & INFLAMMATIONS (137) 9 1
5 TRACHEOSTOMY W MV 96+ HOURS W EXTENSIVE PROCEDURE (004) 8 0
5 rows | 0.66 seconds runtime
Refreshed now
See performance (1)

7.0 Conclusion

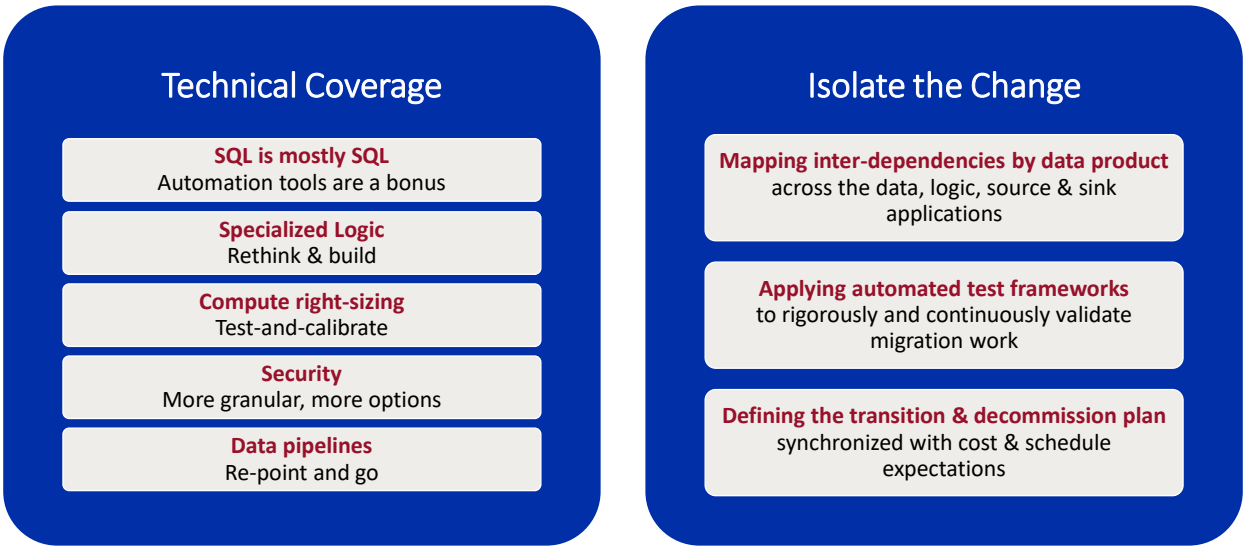
As we have shown, the mechanics of transferring data from Snowflake to Databricks are not complicated; large datasets can be relocated to the Databricks platform for initial experimentation within days.

On the other hand, moving data with all its application logic and other dependencies must be handled as you would any platform conversion process.

The key to success is early experimentation and scoping the work by data product, as in the outline below:



Start by targeting a simple data product to develop an appreciation for the work while exploring Databricks' capabilities during the process, and keep these principles in mind:



Beyond the Forklift

Examining if a forklift strategy is a lost opportunity is appropriate. Such changes are a chance to address technical debt; moving to Databricks opens new possibilities for rethinking the end-to-end environment with far-reaching benefits that can transform your data's time-to-value, accessibility, trust, and cost by way of these benefits:

- Consolidate data warehousing and data science, eliminating separate systems and their data transfer complexities.
- Consolidate data engineering under a unified pipeline, storage, and data processing stack.
- Complete complex analytical queries orders of magnitude faster by leveraging the Databricks massive scalability query engine.
- Unify business rules and data governance, eliminating inconsistencies and improving data trust.
- Replace analytic tooling with self-service GenAI-enabled Exploratory Data Analysis.

About CTI Data

Our data and analytics experts specialize in Digital Transformation, Advanced Analytics, AI/ML, and Data Marketplaces. This experience provides valuable insights and expertise. We are adept at understanding best practices, identifying potential pitfalls, and customizing solutions to meet your unique needs.

By partnering with us, you can drive value from digital transformation efforts as we examine your business strategy, analyze your current state, pinpoint opportunities, and develop a strategic roadmap that aligns technology investments with strategic goals. We commit to collaborating closely with you and sharing accountability for achieving mutual goals.

Contact us to explore our real-world case studies and learn more about how we have helped our clients grow and create business value.