Converting Stored Procedures to Databricks

Ryan Chynoweth

14 min read · Dec 29, 2022

Special thanks to co-author Kyle Hale, Sr. Specialist Solutions Architect at Databricks.

Introduction

A stored procedure is an executable set of commands that is recorded in a relational database management system as an object. More generally speaking, it is simply code that can be triggered or executed on a cadence. Once defined the executable routine can be referenced as an object within the host system.

Databricks does not have an explicit “stored procedure” object; however, the concept is fully supported and in a manner that gives engineers more functionality when compared to cloud data warehouses. Between notebooks, JARs, wheels, scripts, and all the power of SQL, Python, R, Scala, and Java, Databricks is well suited for making your stored procedures lakehouse-

friendly. These executables should be used as a task within a Databricks job. Jobs can be triggered via external orchestrators, executed using the built in CRON scheduler, or run continuously with Spark Structured Streaming or Delta Live Tables.

Databricks Workflows are a natural evolution of stored procedures. Sets of stored procedures were used to create complex dependencies between database objects in order to orchestrate data flows between tables. The directed acyclic graph that is generated by chaining stored procedures takes the form of Databricks Workflows and the tasks within the workflow encapsulate the spirit of a stored procedure.

A job within Databricks is the top level namespace used to run automated workloads. A job can have many tasks which are individual executables Databricks helps manage, monitor, and execute as requested. We will cover common SQL stored procedure patterns and how to convert them to Databricks as PySpark and Spark SQL.

Note: Some systems may support non-SQL stored procedures, this blog will just focus on SQL only conversions.

How to Approach Stored Procedure Conversions

The Databricks Lakehouse enables organizations to build end-to-end solutions for data engineering, data science, and data warehousing workloads. As data is ingested into Databricks from source systems, engineers begin converting legacy stored procedures to run on Databricks compute. But what options do they have?

Within Databricks there are two main orchestration engines, workflows and delta live tables. Delta Live Tables are an excellent way to create real-time data pipelines in a declarative and managed framework. For more information on Delta Live Tables, check out our documentation. However, since delta live tables can be used within a Workflow, we recommend converting procedures to tasks.

Databricks makes these conversions easier — Databricks SQL uses the ANSI SQL standard, so most data warehousing workloads can migrate to the lakehouse with minimal code changes. However, we will discuss scenarios where stored procedures may need to be refactored to a non-SQL language.

For example, converting existing stored procedures to a non-SQL language is ideal when conditional flow logic is required. In these cases, legacy SQL can be migrated to PySpark, and using the `spark.sql` function engineers can embed existing code without core logical changes. The `spark.sql` function allows users to write plain text SQL statements in the context of another interpreter. Due to the popularity of Python and SQL, we recommend programming in these languages for new workloads. The performance implications that once steered developers to Scala and Java do not exist with the DataFrame APIs and the advantages (e.g. hiring talent) that an organization will gain from using more popular languages will benefit you in the long run. Granted, if you have existing Scala code or your organization heavily uses R, then it is best to migrate workloads as the current language.

When migrating code to Databricks, developers can parameterize their code using widgets, SQL variables, and environment variables. There may be minor code adjustments required when drifting outside the ANSI SQL standard but can typically be solved with user defined functions.

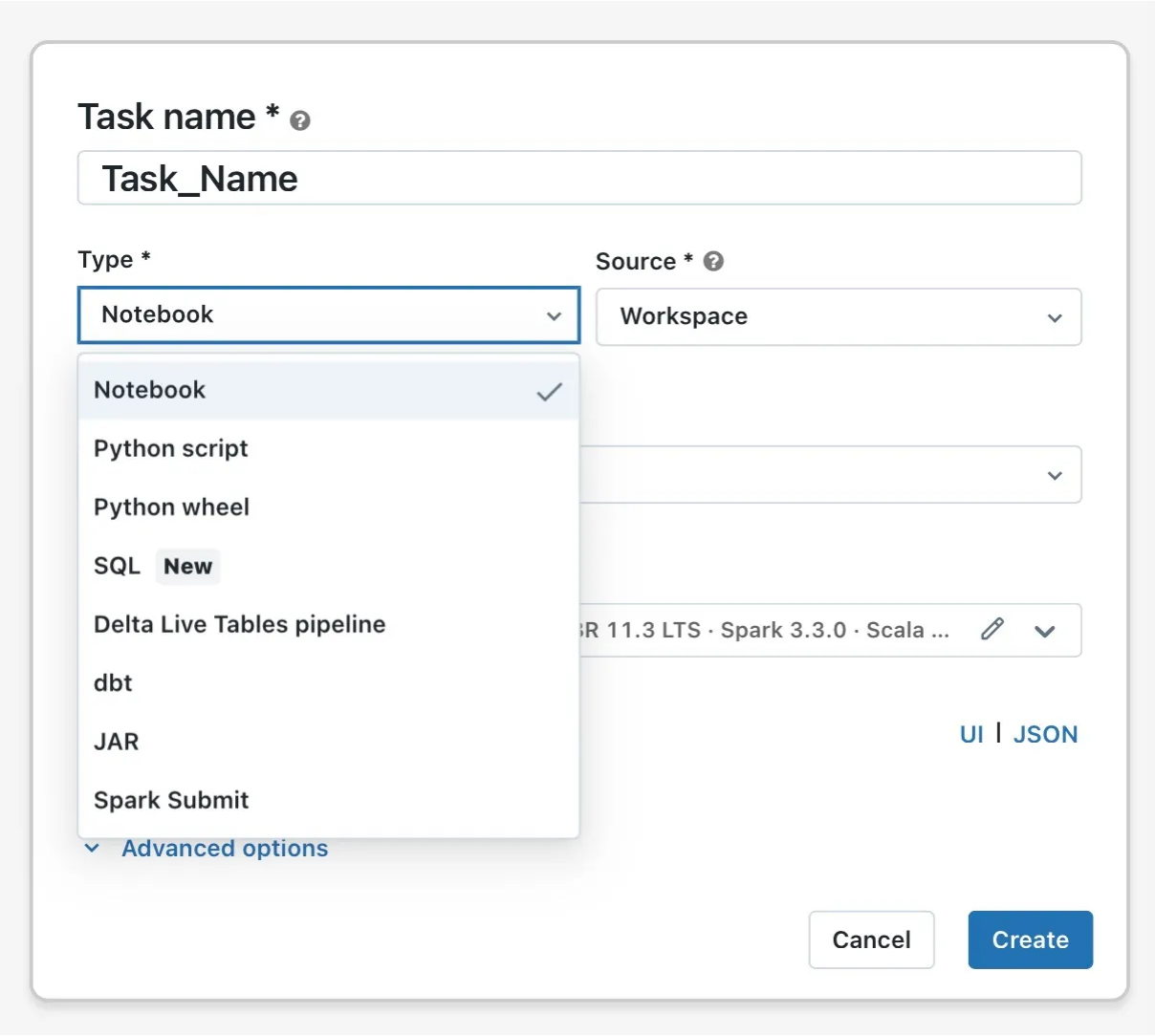
Databricks Workflows

After migrating legacy SQL code to Python and SQL with Databricks notebooks, what is the best way to execute and orchestrate these tasks in the lakehouse? For that, we have Databricks Workflows. But first, let's revisit how legacy data warehouse systems handled this.

One common pattern for developing data pipelines was to write multiple stored procedures that triggered each other. For example, a parent stored procedure would be called on a fixed schedule, which would then call child stored procedures, and in turn may reference even more procedures. Teradata heavily relied on external cron schedulers but users were responsible for coding the pipeline. Oracle databases would allow you to call stored procedures and schedule jobs on a recurring cadence. SQL Server's reliance on SQL Server Integration Services highlights the need for a visual data pipeline editor and by combining it with the SQL Agent users can easily schedule these jobs.

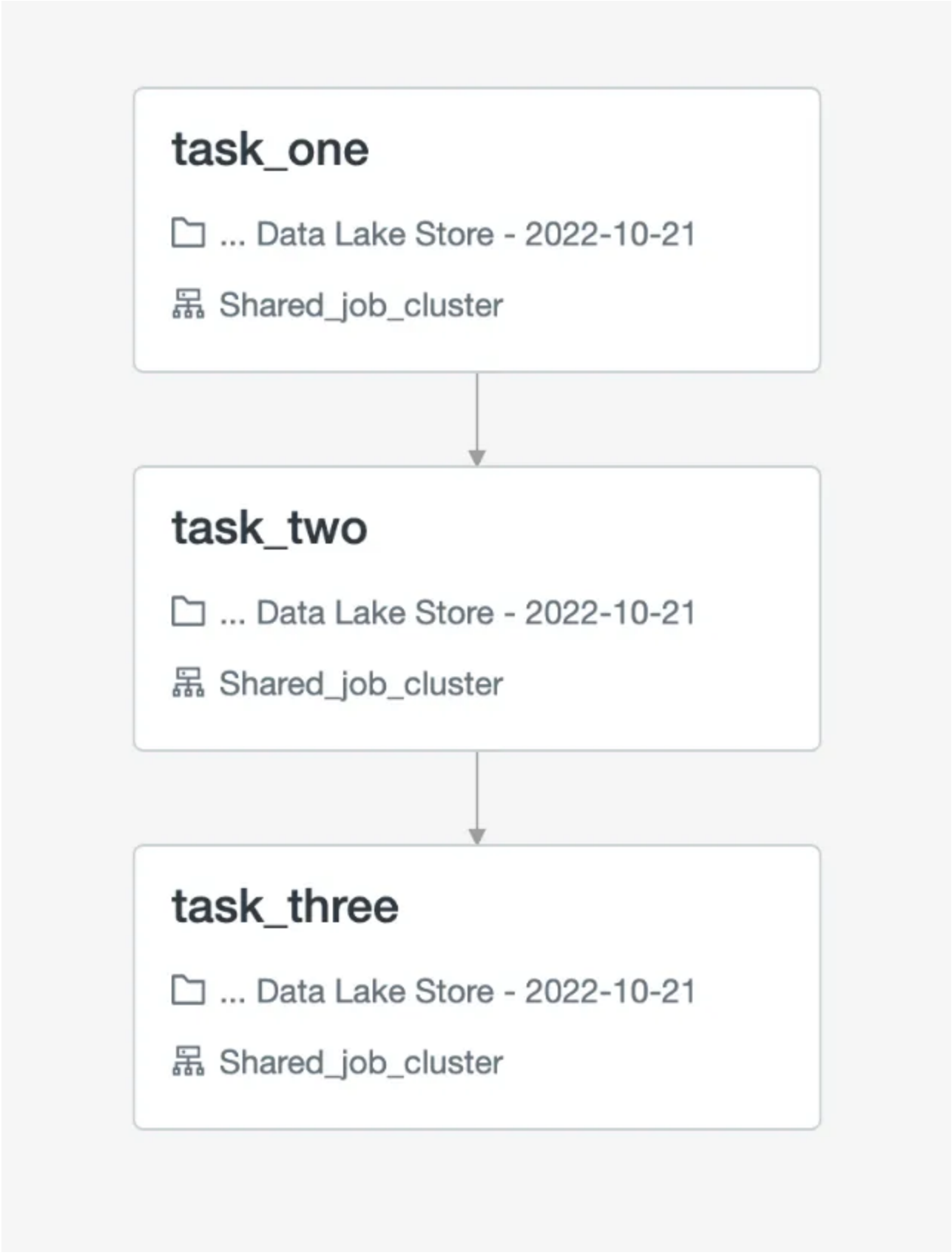
In all legacy cases the amount of effort to orchestrate, monitor, and manage data transformation activities was huge. So how does Databricks Workflows improve on this experience?

Databricks Workflows give users the capabilities to visually create pipelines and schedule as needed. Workflows can execute the following task types: notebooks, python scripts, python wheels, SQL scripts, Delta Live Table pipelines, dbt tasks, JARs, and spark-submit jobs. Individual tasks can pass variables between each other, use shared or isolated job clusters, seamlessly handle retries at the task level, and are integrated with git repositories which can incorporate your DataOps strategy.



Task Definition UI

Remember the parent-child stored procedure orchestration pattern we discussed earlier? If we were to translate the pipeline above to Databricks Workflows it would be represented as as shown below. Notice that each task is running on the “Shared_job_cluster” compute, using the same cluster for each task in the pipeline.

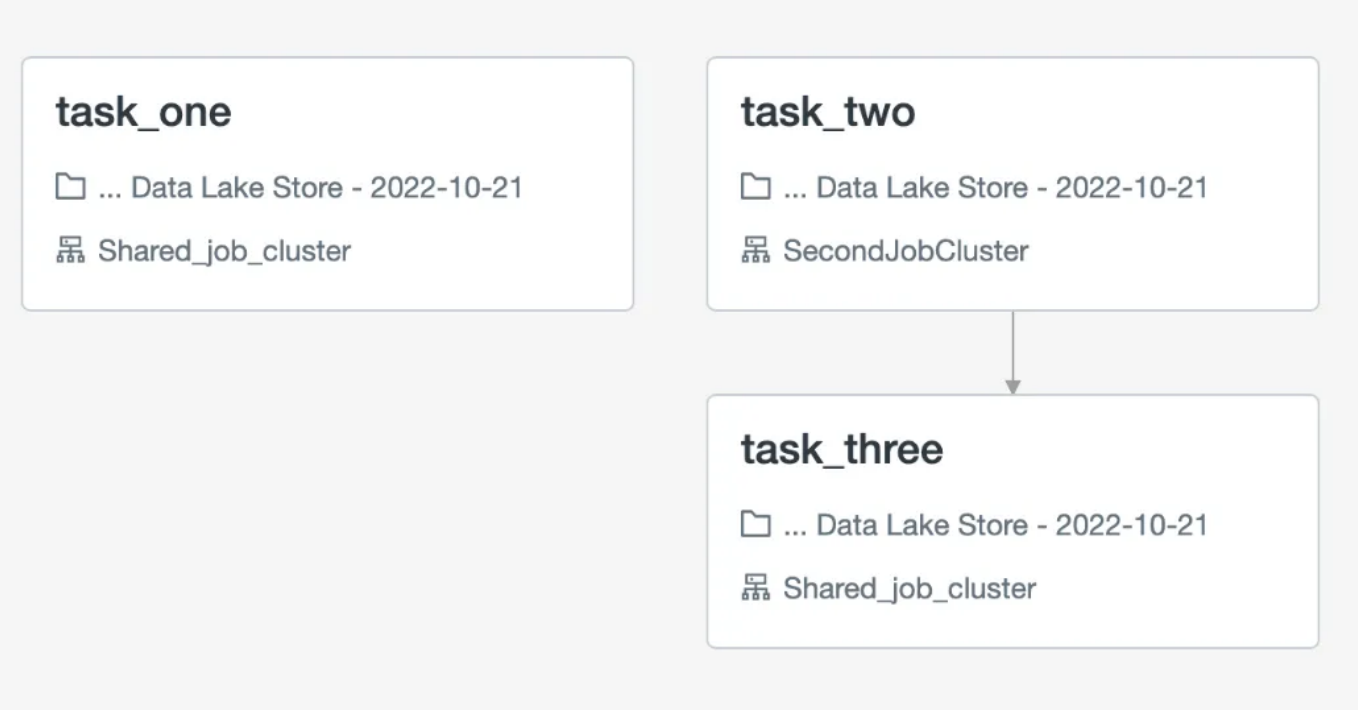


Example Parent-Child DAG

But what if the procedure from the legacy system did not have any dependencies between the child stored procedure? What if we just need to execute them all once a day in isolation of one another or what if two of the tasks were dependent on one another but the third one was not?

Not a problem for Workflows.

We can create a pipeline where `task_one` can run parallel to both `task_two` and `task_three`, while `task_three` is dependent on `task_two`. Below you will notice that we are using multiple job clusters, where `task_one` and `task_two` are running in isolated environments concurrently, while `task_three` runs after `task_two` and re-uses the cluster from `task_one`. In this case, `task_three` could share compute with `task_one` if `task_one` is not completed prior to `task_three` starting.



Alternate Workflow DAG

These simple examples only scratch the surface of what is possible with workflow orchestration. Developers can build pipelines where tasks fan in or out or establish conditional clauses within your tasks so that individual tasks are only executed if specific criteria is met.

An individual task within Databricks Workflows represents the same concept as executing a stored procedure in data warehouses. Not only that, but Workflows offer even more capabilities than stored procedures in a traditional data warehouse. The answer to “Does Databricks support stored procedures?” is a resounding “Yes, and more!”

Example Conversions

In this section, we will demonstrate a number of sample conversions from T-SQL to Spark SQL and PySpark. While T-SQL is the example, these concepts can be applied to various dialects of the SQL language.

Temporary Tables

Stored procedures will often use temporary tables to keep data for further processing, which can be especially useful when a source dataset is referenced multiple times within the same session. By design Apache Spark will keep data in memory unless the data size is too large then the dataset will spill to disk. With data caching and the fact that Spark holds data in memory, the need to materialize data as a temporary table is typically not required. It is recommended to use temporary views instead. Temporary views are conceptually the same as temporary tables in other systems. One of the major advantages of using a temporary view is the ability to reference a Spark DataFrame within a SQL statement.

Assume we have a process that selects data from a source table and writes to multiple output tables. To do so, we read our source table into a temporary table, execute different transformations, and write to multiple target tables.

Example

```
CREATE PROCEDURE WriteMultipleTables
AS
-- select into temp table
SELECT Id, Name, qty, ModifiedDate
INTO #tempTable
FROM my_schema.my_staging_source_table;

-- write to table 1 from staging
INSERT INTO my_schema.my_target_table1
SELECT *
FROM #tempTable
WHERE Name IS NOT NULL;

-- write to table 2 from staging with aggregates
INSERT INTO my_schema.my_target_table2
SELECT Id, Name, ModifiedDate, sum(qty) as qty_sum
FROM #tempTable
WHERE Name IS NOT NULL
GROUP BY Id, Name, ModifiedDate;
```

Spark SQL Example

```
-- create temporary view
CREATE OR REPLACE TEMPORARY VIEW tempTable
AS
SELECT Id, Name, qty, ModifiedDate
FROM my_schema.my_staging_source_table;

-- write to table 1 from staging
INSERT INTO my_schema.my_target_table1
SELECT *
```

```
FROM tempTable
WHERE Name IS NOT NULL;

-- write to table 2 from staging with aggregates
INSERT INTO my_schema.my_target_table2
SELECT Id, Name, ModifiedDate, sum(qty) as qty_sum
FROM tempTable
WHERE Name IS NOT NULL
GROUP BY Id, Name, ModifiedDate;
```

PySpark Example

```
df = spark.read.table('my_schema.my_staging_source_table')

df.createOrReplaceTempView('tempTable')

spark.sql("""
  INSERT INTO my_schema.my_target_table1
  SELECT *
  FROM tempTable
  WHERE Name IS NOT NULL
""")

spark.sql("""
  INSERT INTO my_schema.my_target_table1
  SELECT Id, Name, ModifiedDate, sum(qty) as qty_sum
  FROM tempTable
  WHERE Name IS NOT NULL
  GROUP BY Id, Name, ModifiedDate
""")
```

As you can see in both examples above there is little variation between the core SQL logic that was expressed in the source systems. The Python syntax drifts slightly from the original code but engineers will have more functionality available to them using Python than purely SQL alone.

Common Table Expressions

One of the top questions we receive from customers that are migrating to a lakehouse is whether or not Databricks supports *common table expressions* (CTEs). Common table expressions are extremely common in data warehousing and allow engineers to write complex queries while maintaining high readability of the code. Engineers use CTEs to create datasets that can be referenced within a session in a subsequent SQL query.

Example

```
WITH cte as (
  SELECT *
  FROM my_schema.my_staging_source_table
)

SELECT * FROM cte
```

Spark SQL CTE Example

```
WITH cte as (
  SELECT *
  FROM my_schema.my_staging_source_table
)

SELECT * FROM cte
```

Your eyes are not fooling you — the two examples above are exactly the same! That is because common table expressions are supported in Spark SQL. One area that becomes a bit more complicated is recursive CTEs. At the

time of writing Spark SQL does not support recursive CTEs, however, using PySpark developers can refactor their code to achieve this.

Parameters and Variables

NOTICE! — As of Databricks Runtime 14.1, SQL variables are now supported universally on Databricks. Please review my newly published blog. The code below will work for older runtimes.

A best practice in programming is the DRY Principle — Don't repeat yourself! — which reduces the amount of repetitive code in software. Parameters and variables allow data engineers to reduce the number of code objects required by allowing for more dynamic execution depending on the values provided at runtime. Enterprise data warehouse systems can be very large so reducing the amount of code in the system by making it reusable is key for governance.

Let's take the following example that uses both parameters and variables. Notice the syntax changes for using variables and parameters in Databricks notebooks compared to legacy data warehouses.

Example

```
CREATE PROCEDURE my_stored_procedure @CostCenter INT
AS

DECLARE @year_variable INT;
DECLARE @total_qty BIGINT;
SET @year_variable = YEAR(GETDATE());

SET @total_qty = (
  SELECT sum(qty)
  FROM my_schema.my_staging_source_table
```

```
  WHERE YEAR(ModifiedDate) = @year_variable and CostCenter = @CostCenter;
);

CREATE TABLE my_schema.cost_center_qty_agg
AS

WITH cte as (
  SELECT year, sum(qty) as summed_qty
  FROM my_schema.my_staging_source_table
  WHERE CostCenter = @CostCenter
  GROUP BY YEAR
)

SELECT *
FROM cte
WHERE summed_qty > @total_qty ;
```

Spark SQL Example

```
CREATE WIDGET TEXT CostCenter DEFAULT '';

SET var.year_variable = YEAR(CURRENT_DATE());

SET var.total_qty = (
  SELECT sum(qty)
  FROM my_schema.my_staging_source_table
  WHERE YEAR(ModifiedDate) = ${var.year_variable} and CostCenter = $CostCenter;
);

CREATE TABLE my_schema.cost_center_qty_agg
AS
WITH cte as (
  SELECT year, sum(qty) as summed_qty
  FROM my_schema.my_staging_source_table
  WHERE CostCenter = $CostCenter
  GROUP BY YEAR
)

SELECT *
FROM cte
WHERE summed_qty > ${var.total_qty} ;
```

PySpark Example

```
import datetime

dbutils.widgets.text('CostCenter', '')
cost_center = dbutils.widgets.get('CostCenter')

year_variable = datetime.date.today().year

total_qty = spark.sql(f"""
  SELECT sum(qty)
  FROM my_schema.my_staging_source_table
  WHERE YEAR(ModifiedDate) = {year_variable} and CostCenter = {cost_center}
""").collect()[0][0]

spark.sql(f"""
  CREATE TABLE my_schema.cost_center_qty_agg
  AS
  WITH cte as (
  SELECT year, sum(qty) as summed_qty
  FROM my_schema.my_staging_source_table
  WHERE CostCenter = {cost_center}
  GROUP BY YEAR
  )
  SELECT *
  FROM cte
  WHERE summed_qty > {total_qty}
""")
```

Converting SQL is fairly straightforward as it is mostly small syntax changes between systems. One recommendation to make conversion slightly easier would be related to variable naming. Naming objects in programming is difficult for engineers, but when you start translating code from system to system with different style conventions it can be near impossible.

Thinking strictly about migration scenarios, one way to simplify the effort would be keeping the names of your existing variables as is. If you have a variable `CostCenter` and you want to convert the script to Python, then leave

it as camel case instead of trying to follow the Python style guide and change every single reference to that object. While reading the code may be a little strange due to inconsistent style, functionally there is no difference. By ignoring stylistic conversions the migration will go faster and save engineers the headache of trying to find every reference to the variable in the code to simply change the way it looks. Every engineer who has spent time searching for a missing comma or incorrect indentation in Python would agree!

With regards to SQL variables and Notebook widgets in Databricks, it is important to note the following:

- Variables use the following syntax: `<prefix>.<variable_name>` where `<prefix>` can be any string value.
  - It is a best practice to be consistent with your prefix for readability purposes. In the examples in this blog, we have used `var` as a standard to indicate it is a variable.
  - Assigning a value to a variable is done as follows: `SET <prefix>.<variable_name> = …`
  - Using a variable in a SQL statement is done as follows: `${<prefix>.<variable_name>}`
  - If you assign a variable the result set of a query, it is done so lazily and is not evaluated until there is an action.
- Widgets are parameters that are passed in as string values so you may need to convert the data type.
  - Widgets can be referenced with a preceding dollar sign (`$`) e.g. `$my_widget_name`.

Conditional Statements

Conditional statements (if, else, etc.) are used to define when a set of code should be executed based on the specified condition. When developing ETL pipelines there are many scenarios where conditional flow is used, and these pipelines will need to be converted to PySpark. Spark SQL does not support conditional statements. Please note that there is a built-in if function that is used within the context of a DML statement and not for conditional flows related to pipeline orchestration.

Using the following example of IF ELSE from Microsoft's AdventureWorks database, let's see how we would translate it into Spark SQL:

Example

```
CREATE PROCEDURE CalculateWeight
AS

DECLARE @maxWeight FLOAT, @productKey INTEGER ;
SET @maxWeight = 100.00 ;
SET @productKey = 424 ;

IF @maxWeight <= (SELECT Weight from DimProduct WHERE ProductKey = @productKey)
  SELECT @productKey AS ProductKey
    , EnglishDescription
    , Weight
    , 'This product is too heavy to ship and is only available for pickup.' AS S

  FROM my_schema.DimProduct WHERE ProductKey = @productKey

ELSE
  SELECT @productKey AS ProductKey
    , EnglishDescription
    , Weight
    , 'This product is available for shipping or pickup.' AS ShippingStatus
  FROM my_schema.DimProduct WHERE ProductKey = @productKey
```

PySpark Example

```
maxWeight = 100
productKey = 424

w = spark.sql(f"SELECT Weight from DimProduct WHERE ProductKey = {productKey}").

if maxWeight <= w:
  spark.sql(f"""
    SELECT {productKey} AS ProductKey, EnglishDescription, Weight, 'This product
    FROM DimProduct WHERE ProductKey = {productKey}
  """)
else :
  spark.sql(f"""
    SELECT {productKey} AS ProductKey, EnglishDescription, Weight, 'This product
    FROM DimProduct WHERE ProductKey = {productKey}
  """)
```

Loops

Loops are used to iteratively execute a set of code until a breaking condition is met. While loops are primarily used in data warehousing for two reasons:

- Breaking down the batch processing of a large amount of data into smaller amounts due to limited resources
- Repeating the same pipeline steps with different parameters

Let's use an example of breaking down a larger data set to complete smaller batch inserts into a target table.

Loop Example

```
CREATE PROCEDURE BatchInserts
AS

DECLARE @counter int;
```

```
DECLARE @rowcount int ;
DECLARE @batchsize int;

SET @batchsize = 10000;
SET @counter = 0;
SET @rowcount = SELECT count(1) FROM my_source_table;

WHILE @counter <= @rowcount
  BEGIN
    INSERT INTO my_schema.my_target_table
    SELECT TOP (@batchsize) * FROM my_schema.my_source_table WHERE (id > @counte

    SET @counter = @counter + @batchsize
  END
```

Spark SQL Example

```
INSERT INTO my_target_table
SELECT * FROM my_source_table
```

PySpark Example

```
# this is the same as spark.sql("SELECT * FROM my_source_table")
df = spark.read.table('my_source_table')

df.write.mode("append").saveAsTable("my_target_table")
```

Look, no loops! Databricks' powerful massively parallel processing engine and workload isolation makes batch inserting data to preserve compute resources irrelevant. And if you *do* need to operate on individual rows or

partitions simply use our `foreach` functionality to distribute the step across your cluster in parallel, again avoiding loops.

Similarly, a loop is no longer required to repeatedly execute the same workload under different conditions because of notebook widgets and environment variables. These can be leveraged to schedule the same notebook with different values and can be executed in parallel on isolated job clusters.

Functions

Since Databricks SQL is ANSI SQL by default, most of the built-in functions available in legacy systems are also available in Databricks. However, your code may have functions in it that are only available in your specific data warehouse, or custom functions that your organization wrote to solve specific domain problems.

In the "Parameters and Variables" section of this document, you will notice that we used a couple of functions to determine the value of the current year i.e. SET @year_variable = YEAR(GETDATE());. The GETDATE() function is non-ANSI T-SQL and not available in Spark SQL. In this scenario there are two options:

1. Change `GETDATE()` in your code to the ANSI supported function, `CURRENT_TIMESTAMP()`
2. Create a UDF called `GETDATE()` that returns the value of `CURRENT_TIMESTAMP()` so you do not have to change your code

Option one is ideal when a function is not widely used; updating a few pieces of code to use the new function is no big deal. Option two is better when

there are many references to the same function across the code base. In this case, we would use SQL UDFs and look something like this:

```
CREATE FUNCTION GETDATE()
RETURNS TIMESTAMP
COMMENT 'Function used to map the T-SQL getdate() function in Databricks'
LANGUAGE SQL
RETURN CURRENT_TIMESTAMP();
```

Now there are two functions that are logically equivalent to each other: `GETDATE()` and `CURRENT_TIMESTAMP()`.

Exception Handling

Ideally, exception handling would not be required. We all would be able to envision every possible scenario for processing data and gracefully determine how best to operate in that situation. For us mere mortals, logging errors and exceptions is required to allow individuals to investigate issues and remediate them.

Legacy data warehouses allowed the ability to capture state and exception information that could be saved to relational databases. Spark SQL does not have a `try/catch` block, but all the other languages supported in Databricks do! There are two different ways to convert a SQL block to PySpark. The first method shows how to easily convert the same process to Databricks without refactoring and the second method likely requires refactoring to align with the `log4j` library using the `logging` python module.

If you are interested in learning more about logging in Databricks with Python, check out this blog by Ivan Trusov, a Solutions Architect at

Databricks. In this example, we will exclude much of the required configuration and focus on the exception handling itself.

Example

```
CREATE PROCEDURE MyExceptionProcedure
AS

BEGIN TRY
  INSERT INTO my_schema.my_target_table
  SELECT *
  FROM my_schema.my_source_table
END TRY

BEGIN CATCH
  INSERT INTO dbo.DB_Errors
  VALUES
  (SUSER_SNAME(),
  ERROR_NUMBER(),
  ERROR_STATE(),
  ERROR_SEVERITY(),
  ERROR_LINE(),
  ERROR_PROCEDURE(),
  ERROR_MESSAGE(),
  GETDATE());
END CATCH
```

PySpark Example — Table Logging

```
try:

spark.sql("""
  INSERT INTO my_schema.my_target_table
  SELECT * FROM my_schema.my_source_table
""")

except Exception as e:
```

```
spark.sql(f"""
  INSERT INTO dbo.DB_ERRORS
  VALUES({str(e)}, {current_timestamp()}
""")
```

PySpark Example — Using a Logging Library

```
logger = LoggerProvider().get_logger(spark)

try:
  spark.sql("""
    INSERT INTO my_schema.my_target_table
    SELECT * FROM my_schema.my_source_table
""")

except Exception as e:
  logger.fatal(str(e))
```

Conclusion

Databricks Workflows are an extremely powerful tool that can be used to orchestrate robust data pipelines at scale. Stored procedures easily map to individual tasks within the pipeline to create dependencies between each other to serve business solutions.

Converting legacy SQL code to Databricks is simple, and we have a number of different resources and tools to assist with code conversions and accelerate your migration to the lakehouse.

Disclaimer: these are my own thoughts and opinions and not a reflection of my employer.

Databricks | Stored Procedure | Spark | Sql | Data Warehouse

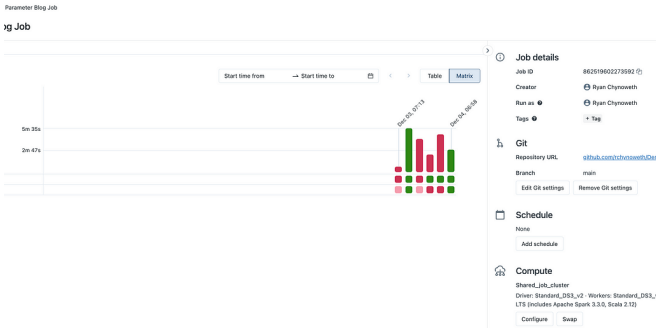
Written by Ryan Chynoweth

Edit profile

312 Followers

Senior Solutions Architect Databricks — anything shared is my own thoughts and opinions

More from Ryan Chynoweth



Ryan Chynoweth

Delta Sharing: An Implementation Guide for Multi-Cloud Architecture

Introduction

8 min read · 3 days ago

3 2

Ryan Chynoweth

Task Parameters and Values in Databricks Workflows

Databricks provides a set of powerful and dynamic orchestration capabilities that are...

11 min read · Dec 7, 2022

50 3

Ryan Chynoweth

Recursive CTE on Databricks

Introduction

3 min read · Apr 20, 2022

33

Ryan Chynoweth

SQL Variables in Databricks

Last December we published a blog providing an overview of Converting Stored Procedure...

2 min read · Oct 6, 2023

21

See all from Ryan Chynoweth

Recommended from Medium

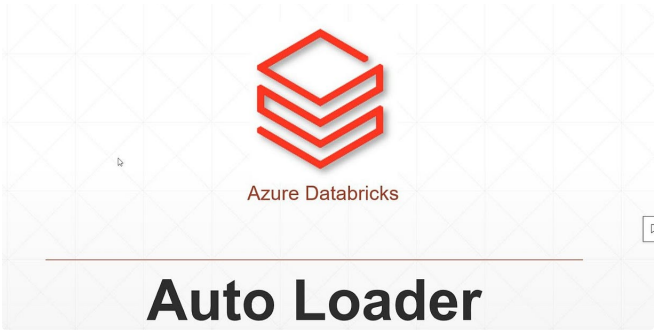
Daan Rademaker

Do-it-yourself, building your own Databricks Docker Container

In my previous LinkedIn article, I aimed to persuade you of the numerous advantages o...

7 min read · Oct 16, 2023

26



SIRIGIRI HARI KRISHNA in Towards Dev

Auto Loader

Autoloader simplifies reading various data file types from popular cloud locations like...

5 min read · Dec 9, 2023

4 1

Lists

ChatGPT

23 stories · 371 saves

Natural Language Processing

1053 stories · 530 saves

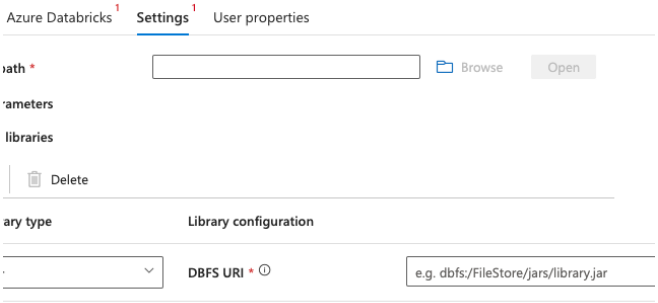
Manasreddy

How to pass: Databricks Data Engineer Professional Certification

Conquering the Databricks Data Engineer Professional Exam: A Definitive Guide

3 min read · Aug 24, 2023

8



Matt Bradley

Azure Data Factory tips for running Databricks jobs

Following on from an earlier blog around using spot instances with Azure Data...

4 min read · Nov 2, 2023

14

Eduardo Senior

REST API Data Ingestion with PySpark

Putting executors to work.

6 min read · Oct 5, 2023

231

Karim Faiz

Mastering DBT: From Zero to Hero

Introduction

✦ · 12 min read · Dec 26, 2023

2

See more recommendations