Recursive CTE on Databricks

Ryan Chynoweth

3 min read · Apr 20, 2022

Introduction

SQL at Databricks is one of the most popular languages for data modeling, data acquisition, and reporting. A somewhat common question we are asked is if we support Recursive Common Table Expressions (CTE). A recursive CTE is the process in which a query repeatedly executes, returns a subset, unions the data until the recursive process completes.

Here is an example of a TSQL Recursive CTE using the Adventure Works database:

```
CREATE PROCEDURE [dbo].[uspGetBillOfMaterials]
    @StartProductID [int],
    @CheckDate [datetime]
AS
BEGIN
    SET NOCOUNT ON;

    -- Use recursive query to generate a multi-level Bill of Material
(i.e. all level 1
    -- components of a level 0 assembly, all level 2 components of a
```

```
level 1 assembly)
    -- The CheckDate eliminates any components that are no longer
used in the product on this date.
    WITH [BOM_cte]([ProductAssemblyID], [ComponentID],
[ComponentDesc], [PerAssemblyQty], [StandardCost], [ListPrice],
[BOMLevel], [RecursionLevel]) -- CTE name and columns
    AS (
        SELECT b.[ProductAssemblyID], b.[ComponentID], p.[Name], b.
[PerAssemblyQty], p.[StandardCost], p.[ListPrice], b.[BOMLevel], 0 --
Get the initial list of components for the bike assembly
        FROM [Production].[BillOfMaterials] b
            INNER JOIN [Production].[Product] p
            ON b.[ComponentID] = p.[ProductID]
        WHERE b.[ProductAssemblyID] = @StartProductID
            AND @CheckDate >= b.[StartDate]
            AND @CheckDate <= ISNULL(b.[EndDate], @CheckDate)
        UNION ALL
        SELECT b.[ProductAssemblyID], b.[ComponentID], p.[Name], b.
[PerAssemblyQty], p.[StandardCost], p.[ListPrice], b.[BOMLevel],
[RecursionLevel] + 1 -- Join recursive member to anchor
        FROM [BOM_cte] cte
            INNER JOIN [Production].[BillOfMaterials] b
            ON b.[ProductAssemblyID] = cte.[ComponentID]
            INNER JOIN [Production].[Product] p
            ON b.[ComponentID] = p.[ProductID]
        WHERE @CheckDate >= b.[StartDate]
            AND @CheckDate <= ISNULL(b.[EndDate], @CheckDate)
        )
    -- Outer select from the CTE
    SELECT b.[ProductAssemblyID], b.[ComponentID], b.[ComponentDesc],
SUM(b.[PerAssemblyQty]) AS [TotalQuantity] , b.[StandardCost], b.
[ListPrice], b.[BOMLevel], b.[RecursionLevel]
    FROM [BOM_cte] b
    GROUP BY b.[ComponentID], b.[ComponentDesc], b.
[ProductAssemblyID], b.[BOMLevel], b.[RecursionLevel], b.
[StandardCost], b.[ListPrice]
    ORDER BY b.[BOMLevel], b.[ProductAssemblyID], b.[ComponentID]
    OPTION (MAXRECURSION 25)
END;
GO
```

Recursive CTEs are most commonly used to model hierarchical data. In the case above, we are looking to get all the parts associated with a specific assembly item. Another common use case is organizational structures.

Recursive Common Table Expression on Databricks

Unfortunately, Spark SQL does not natively support recursion as shown above. But luckily Databricks users are not restricted to using only SQL! Using PySpark we can reconstruct the above query using a simply Python loop to union dataframes.

In the TSQL example, you will notice that we are working with two different tables from the Adventure Works demo database: BillOfMaterials and Product.

Using PySpark the SQL code translates to the following:

```
i = 1
check_date = '2010-12-23'
start_product_id = 972 # provide a specific id

# bill_df corresponds to the "BOM_CTE" clause in the above query
df = spark.sql("""

SELECT b.ProductAssemblyID, b.ComponentID, p.Name, b.PerAssemblyQty,
p.StandardCost, p.ListPrice, b.BOMLevel, 0 as RecursionLevel

FROM BillOfMaterials b
    INNER JOIN Product p ON b.ComponentID = p.ProductID

WHERE b.ProductAssemblyID = {} AND '{}' >= b.StartDate AND '{}' <=
IFNULL(b.EndDate, '{}')
""".format(start_product_id, check_date, check_date, check_date))


# this view is our 'CTE' that we reference with each pass
df.createOrReplaceTempView('recursion_df')

while True:
# select data for this recursion level
bill_df = spark.sql("""

SELECT b.ProductAssemblyID, b.ComponentID, p.Name, b.PerAssemblyQty,
p.StandardCost, p.ListPrice, b.BOMLevel,  {} as RecursionLevel

FROM recursion_df cte
    INNER JOIN BillOfMaterials b ON b.ProductAssemblyID =
cte.ComponentID
    INNER JOIN Product p ON b.ComponentID = p.ProductID
```

```
    WHERE '{}' >= b.StartDate AND '{}' <= IFNULL(b.EndDate, '{}')
    """.format(i, check_date,check_date,check_date))

    # this view is our 'CTE' that we reference with each pass
    bill_df.createOrReplaceTempView('recursion_df')

    # add the results to the main output dataframe
    df = df.union(bill_df)

    # if there are no results at this recursion level then break
    if bill_df.count() == 0:
        df.createOrReplaceTempView("final_df")
        break
    else:
        i += 1
```

This may seem overly complex for many users, and maybe it is. However, if you notice we are able to utilize much of the same SQL query used in the original TSQL example using the **spark.sql** function. Additionally, the logic has mostly remained the same with small conversions to use Python syntax.

Conclusion

While the syntax and language conversion for Recursive CTEs are not ideal for SQL only users, it is important to point that it is possible on Databricks. So you do not lose functionality when moving to a Lakehouse, it just may change and in the end provide even more possibilities than a Cloud Data Warehouse.

Edit 10.03.22—check out this blog with a similar idea but with list comprehensions instead!

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

Databricks Databricks Sql Sql Spark Sql Pyspark

Written by Ryan Chynoweth

Edit profile

312 Followers

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

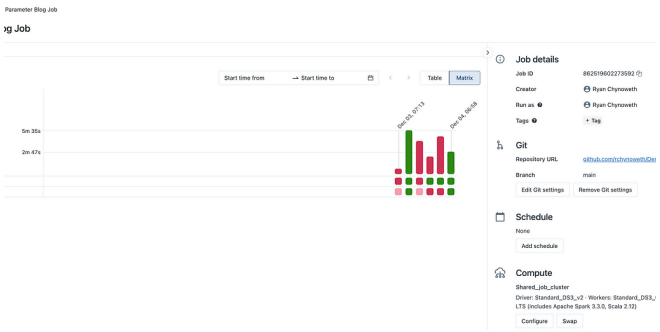
More from Ryan Chynoweth

Ryan Chynoweth

Delta Sharing: An Implementation Guide for Multi-Cloud Architecture

Introduction

8 min read · 3 days ago



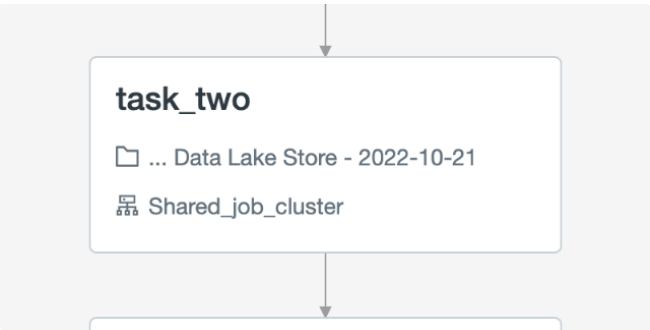
Ryan Chynoweth

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

3 2 50 3



Ryan Chynoweth

Converting Stored Procedures to Databricks

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

14 min read · Dec 29, 2022

116 5

Ryan Chynoweth

SQL Variables in Databricks

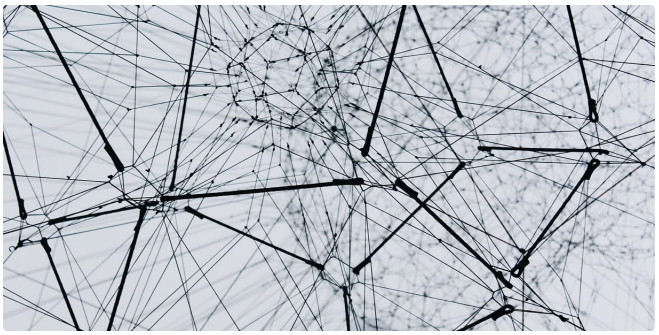
Last December we published a blog providing an overview of Converting Stored Procedure...

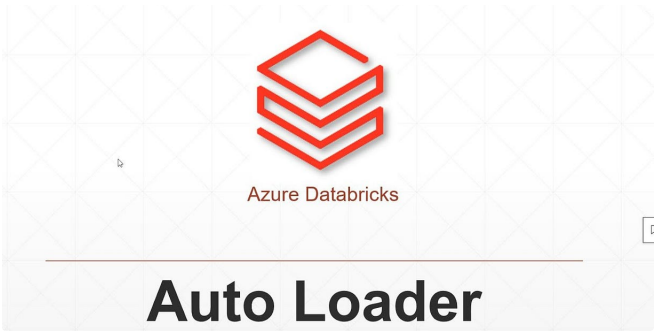
2 min read · Oct 6, 2023

21

See all from Ryan Chynoweth

Recommended from Medium





Subham Khandelwal in Dev Genius

PySpark—Optimize Joins in Spark

Shuffle Hash Join, Sort Merge Join, Broadcast joins and Bucketing for better Join...

8 min read · 4 days ago

65

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

dezimaldata

Databricks Spark Temporary Views with SQL and Python

Pranavk

Python Loading Dataframe in SQL Server using BCP

In this blog post, I will explain what are temporary views, how to create them using...

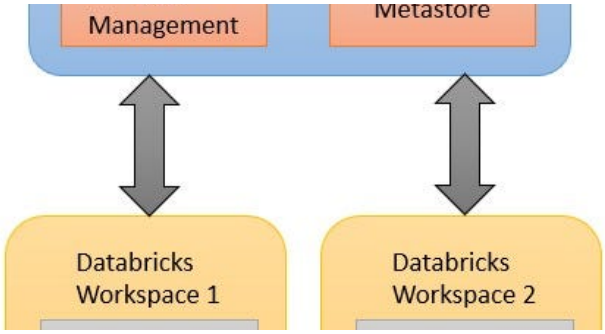
3 min read · Aug 19, 2023

8

Introduction

3 min read · Oct 11, 2023

55



Oindrila Chakraborty

Introduction to “Unity Catalog” in Databricks

What is “Unity Catalog”?

10 min read · Aug 14, 2023

18 2

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

See more recommendations