**I have tables in postgres:**

TIDDOHOF has columns doc\_title,doc\_status

TIDDRHOF has columns dr\_status

TIDENADL has columns ec\_number, ec\_revision, document\_nbr,doc\_nbr,doc\_sheet\_nbr,document\_sub\_type, document\_type,facility, adl\_occur\_flag

**I want use aws glue etl using pyspark to have table ENGINEERING\_CHANGE\_AFFECTED\_DOC\_LST in s3 having columns from postgres tables**

-Affctd\_doc\_id generate ec\_number,ec\_revision,facility,document\_type,document\_sub\_type,document\_nbr,doc\_sheet\_nbr,adl\_occur\_flag and is a primary key

-Src\_sys\_cd set to “NAMS”

-Eng\_chg\_nbr trim blank straight copy renamed from ec\_number

-eng\_chg\_revsn\_nbr trim blank straight copy renamed from ec\_revision

-fclty\_loc\_id

-doc\_nbr trim blank straight copy renamed from document\_nbr

-doc\_revsn\_sts\_cd trim blank straight copy renamed dr\_status

-doc\_sht\_nbr trim blank straight copy renamed after doc\_sheet\_nbr

-doc\_sub\_typ\_cd trim blank straight copy renamed after document\_sub\_type

-doc\_typ\_cd trim blank straight copy renamed after document\_type

-fclty\_cd trim blank straight copy renamed after facility

Then I want to implement SCD2 and soft delete on the fact that any records changed from tables in postgres affect tables in s3 and load the final results in redshift. I want to use delta format and the final to include also columns:

Eff\_strt\_dttm for start date

Eff\_end\_dttm for end date

Etl\_insrt\_usr\_id etl user id

Etl\_insrt\_dttm etl insert time

Etl\_upd\_usr\_id etl update user id

Etl\_upd\_dttm etl update time

The Delta table is stored with an pri\_key, which is unique to each and every change and is used to track the changes. The pri\_key is created for every insert, update, and delete, and can be used to find all the changes

The pri\_key is created using the SHA256 hashing algorithm

The scd use merge function

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

import pyspark.sql.functions as F

# Initialize GlueContext

glueContext = GlueContext(SparkContext.getOrCreate())

# Read from PostgreSQL tables

tiddohof = glueContext.create\_dynamic\_frame.from\_options(

connection\_type="postgresql",

connection\_options={"url": "jdbc:postgresql://<your-db-url>", "dbtable": "TIDDOHOF", "user": "<user>", "password": "<password>"}

)

tiddrhof = glueContext.create\_dynamic\_frame.from\_options(

connection\_type="postgresql",

connection\_options={"url": "jdbc:postgresql://<your-db-url>", "dbtable": "TIDDRHOF", "user": "<user>", "password": "<password>"}

)

tidenadl = glueContext.create\_dynamic\_frame.from\_options(

connection\_type="postgresql",

connection\_options={"url": "jdbc:postgresql://<your-db-url>", "dbtable": "TIDENADL", "user": "<user>", "password": "<password>"}

)

# Convert DynamicFrames to DataFrames for more complex transformations

df\_tiddohof = tiddohof.toDF()

df\_tiddrhof = tiddrhof.toDF()

df\_tidenadl = tidenadl.toDF()

# Perform necessary transformations and renames

# Assuming a join or other transformation logic to integrate the dataframes

# This is a simplified transformation assuming direct mappings and renames

df\_result = df\_tidenadl \

.withColumn("Affctd\_doc\_id", F.concat\_ws('\_', 'ec\_number', 'ec\_revision', 'facility', 'document\_type', 'document\_sub\_type', 'document\_nbr', 'doc\_sheet\_nbr', 'adl\_occur\_flag')) \

.withColumn("Src\_sys\_cd", F.lit("NAMS")) \

.withColumn("Eng\_chg\_nbr", F.trim(df\_tidenadl.ec\_number)) \

.withColumn("eng\_chg\_revsn\_nbr", F.trim(df\_tidenadl.ec\_revision)) \

.withColumn("fclty\_loc\_id", df\_tidenadl.facility) \

.withColumn("doc\_nbr", F.trim(df\_tidenadl.document\_nbr)) \

.withColumnRenamed("dr\_status", "doc\_revsn\_sts\_cd") \

.withColumn("doc\_sht\_nbr", F.trim(df\_tidenadl.doc\_sheet\_nbr)) \

.withColumnRenamed("document\_sub\_type", "doc\_sub\_typ\_cd") \

.withColumnRenamed("document\_type", "doc\_typ\_cd") \

.withColumnRenamed("facility", "fclty\_cd")

# Convert DataFrame back to DynamicFrame

dynamic\_frame\_result = DynamicFrame.fromDF(df\_result, glueContext, "dynamic\_frame\_result")

# Write the result to S3

glueContext.write\_dynamic\_frame.from\_options(

frame=dynamic\_frame\_result,

connection\_type="s3",

connection\_options={"path": "s3://<your-bucket-name>/<path>/"},

format="parquet" # or your desired format

)

**Step 1: Capture Changes in PostgreSQL**

You would need to capture changes in your PostgreSQL databases. This can be done using:

* PostgreSQL's logical replication to stream changes.
* A Change Data Capture (CDC) tool like Debezium to capture changes and forward them to a Kafka topic, S3, or another destination.

**Step 2: Process Changes with AWS Glue and Spark**

Assuming you have your changes ready to be processed, the next step is to use AWS Glue to process these changes and apply the SCD Type 2 logic using Spark.

pythonCopy code

from pyspark.sql import SparkSession from pyspark.sql.functions import lit, col, sha2, concat\_ws, current\_timestamp from delta.tables import DeltaTable # Initialize Spark session for Delta support spark = SparkSession.builder.appName("SCD2 and Soft Deletes").getOrCreate() # Assuming delta\_table\_path points to your S3 Delta Lake location delta\_table\_path = "s3://<your-bucket-name>/<delta-table-path>" # Load the change data into a DataFrame # This could be from a staging area on S3 where changes are stored change\_data\_df = spark.read.json("s3://<your-staging-area-path>/changes/") # Add necessary columns for SCD2, including a hash column as pri\_key for uniqueness change\_data\_df = change\_data\_df.withColumn("pri\_key", sha2(concat\_ws("|", \*change\_data\_df.columns), 256)) \ .withColumn("Eff\_strt\_dttm", current\_timestamp()) \ .withColumn("Eff\_end\_dttm", lit(None)) \ .withColumn("Etl\_insrt\_dttm", current\_timestamp()) \ .withColumn("Etl\_upd\_dttm", current\_timestamp()) # Read the existing Delta Lake or create if not exists try: delta\_table = DeltaTable.forPath(spark, delta\_table\_path) except: change\_data\_df.write.format("delta").save(delta\_table\_path) delta\_table = DeltaTable.forPath(spark, delta\_table\_path) # Apply SCD Type 2 logic using merge delta\_table.alias("target").merge( change\_data\_df.alias("source"), "target.pri\_key = source.pri\_key" ).whenMatchedUpdate( condition="source.is\_deleted = true", # Assuming 'is\_deleted' indicates a soft delete set={ "Eff\_end\_dttm": current\_timestamp() } ).whenMatchedUpdate( condition="", set={ # Update attributes except 'Eff\_strt\_dttm', 'pri\_key', etc. } ).whenNotMatchedInsert( values={ "pri\_key": "source.pri\_key", # Insert all other necessary attributes } ).execute()

**Step 3: Load Data into Redshift**

After processing your data in S3 with Delta Lake format and applying SCD2 and soft deletes, you can load this data into Redshift. This process can be done using AWS Glue's built-in functionalities to connect to Redshift or using the COPY command for larger datasets.

pythonCopy code

# Example AWS Glue job script to load data into Redshift (conceptual) glueContext.write\_dynamic\_frame.from\_options( frame=dynamic\_frame\_result, connection\_type="redshift", connection\_options={ "url": "<redshift-url>", "dbtable": "<target-table-name>", "user": "<username>", "password": "<password>" }, transformation\_ctx="datasink2" )

**Important Notes:**

* **Change Data Capture**: The initial step of capturing changes in PostgreSQL is crucial. Depending on your setup, you may use built-in PostgreSQL capabilities, AWS Database Migration Service (DMS) for ongoing replication, or third-party tools.
* **Delta Lake on S3**: Make sure your Spark session is configured to handle Delta Lake operations on S3.
* **Redshift Load**: The method for loading data into Redshift can vary based on your data size and frequency of updates. For large datasets or frequent updates, consider using Redshift's **COPY** command for efficiency.
* **Permissions**: Ensure AWS IAM roles and policies are correctly set up for access between services (Glue, S3, Redshift).

This example provides a high-level overview. You'll need to adapt it to fit your specific data structures, capture mechanisms, and AWS environment.



**You**

in step 1 for capture change i want to use glue

ChatGPT

**ChatGPT**

To capture changes in your PostgreSQL database using AWS Glue for a scenario like implementing SCD Type 2 and soft deletes, you would typically follow a process that involves setting up an AWS Glue job to periodically scan your source tables and identify changes. This approach might involve comparing snapshots of the data at different times to detect inserts, updates, and deletes. However, it's worth noting that AWS Glue, by itself, does not provide a built-in, real-time change data capture (CDC) mechanism like some other tools (e.g., Debezium for Kafka). Instead, you can simulate a CDC mechanism by using AWS Glue's capabilities alongside other AWS services.

Here is an approach to simulate capturing changes using AWS Glue:

**Step 1: Initial Setup**

1. **Create a Baseline**: Initially, load your PostgreSQL tables into S3 as a baseline for comparison. This dataset serves as the point of reference for detecting changes.
2. **Timestamps and Versioning**: Ensure each table in PostgreSQL has timestamp columns (e.g., **created\_at**, **updated\_at**) and, if possible, a version number. This setup helps in identifying new and updated records.

**Step 2: Periodic AWS Glue Jobs for Change Detection**

1. **Schedule AWS Glue Jobs**: Create and schedule AWS Glue ETL jobs to run at regular intervals (e.g., hourly, daily, depending on your requirements). Each job performs the following tasks:
   * **Extract**: Read the latest state of your PostgreSQL tables.
   * **Transform**: Compare this latest state against the previously stored snapshot in S3 to identify inserts, updates, and "soft" deletes. This comparison can be done based on **created\_at**, **updated\_at** timestamps, and version numbers.
   * **Load**: Write the detected changes into a separate area in S3 in a format suitable for further processing, such as Delta format for SCD2 handling.

**Step 3: Implementing Change Logic in the AWS Glue Job**

Here's a simplified example of how the AWS Glue job script might look to detect and process changes:

pythonCopy code

import sys from awsglue.context import GlueContext from awsglue.transforms import \* from awsglue.utils import getResolvedOptions from pyspark.context import SparkContext from pyspark.sql.functions import lit, expr, current\_timestamp sc = SparkContext() glueContext = GlueContext(sc) spark = glueContext.spark\_session # Define connection options to read from PostgreSQL connection\_options = { "url": "jdbc:postgresql://<your-db-url>/<db-name>", "dbtable": "<your-table-name>", "user": "<your-username>", "password": "<your-password>", "redshiftTmpDir": args["TempDir"], "aws\_iam\_role": "<your-glue-role>" } # Read the current snapshot of the table current\_df = glueContext.create\_dynamic\_frame.from\_options( connection\_type="postgresql", connection\_options=connection\_options ).toDF() # Assuming you have an existing S3 path where the previous snapshot is stored previous\_snapshot\_path = "s3://<your-bucket-name>/previous\_snapshot/" previous\_df = spark.read.format("parquet").load(previous\_snapshot\_path) # Example logic to identify new and updated records # Adjust the logic based on your table structure and requirements changes\_df = current\_df.join(previous\_df, "<primary-key>", "outer") \ .where(current\_df["updated\_at"] > previous\_df["updated\_at"]) \ .select(current\_df["\*"], lit("update").alias("change\_type")) # Write changes to S3 in Delta format for further processing changes\_df.write.format("delta").save("s3://<your-bucket-name>/changes/") # Update the snapshot current\_df.write.format("parquet").mode("overwrite").save(previous\_snapshot\_path)

**Notes:**

* This script is a high-level template and requires customization to fit your specific schema, tables, and change detection logic.
* You need to handle deletes separately, possibly by tracking records that exist in the previous snapshot but not in the current state, marking them as "soft deleted."
* For handling SCD Type 2 and soft deletes in Delta format as described in previous steps, you would process the changes captured and stored by this Glue job.

Remember, the effectiveness of this approach depends on your data volume, change frequency, and the AWS Glue job's schedule. For large datasets or high-frequency changes, consider more direct CDC solutions or AWS DMS for ongoing replication needs.

Top of Form