International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 2, 191-196, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I2P128

Original Article

Real Time Data Synchronization and Historical Tracking using AWS Data Migration Service and Databricks

Vamshi Krishna Malthummeda
Independent Researcher, USA.

Received On: 16/03/2026 Revised On: 17/04/2026 Accepted On: 24/04/2026 Published On: 02/05/2026

Abstract: *There is a growing demand for synchronization of data between databases for workload separation, disaster recovery, historical tracking and for various other use cases. This article proposes a solution which uses AWS DMS (Data Migration Services) to migrate data from source databases into s3 buckets and store it in the form of parquet files. The data in these files is subjected to schema conversion, SCD (Slowly Changing Dimensions) type 2 transformations in Databricks Spark ingestion pipeline and finally appended to the UC (Unity Catalog) table. Finally, UC table data is loaded into various types of target databases. The proposed solution provides better schema validation, higher availability and requires much less configuration and maintenance compared to a solution developed in AWS EMR, which provides computing infrastructure plus Sqoop, which is used for extraction of data and Apache Spark. This solution allows for effective utilization of AWS DMS features like Broad Database Support, Serverless compute, Elastic scaling and Continuous Replication along with Cost-Effective, Self-Healing, Historical Tracking and Parallelization features of Databricks Ingestion pipelines.*

Keywords: *Real-Time Data Synchronization, Historical Data Tracking, AWS Data Migration Service (AWS DMS), Databricks, Cloud Data Integration, Change Data Capture (CDC), Data Replication, Big Data Analytics, ETL Automation, Data Lakehouse Architecture, Stream Processing, Data Engineering, Incremental Data Loading, Enterprise Data Migration, Scalable Data Pipeline, Data Governance.*

1. Introduction

1.1. Problem Description

The legacy Data Migration solution includes the following components:

- AWS EMR is set up with predefined number of cluster nodes along with their RAM & CPU core configurations.
- The JDBC connectors need to be installed for each of the source databases on the AWS EMR (The installation of these connectors increases the time to initialize the cluster)
- Sqoop is pre-installed in AWS EMR, which exports data from relational databases into S3 bucket using EMRFS.

However, Sqoop has the following limitations:

- Sqoop is not good at datatype mapping hence needs to be done outside of the tool.
- Sqoop executes multiple map-reduce jobs where data is written & read from disks which are inherently very slow operations.
- Apache Sqoop project retired in June 2021 and moved to Apache Attic.
- The format of Apache Sqoop output is CSV which has limitations like lack of schema enforcement.
- Apache Spark performs the extra task of explicit data type conversions in addition to other transformations. This is because of the input file format CSV which is unoptimized.
- Hive Metastore which has functionality, governance and data management issues compared to Databricks Unity Catalog.

The above legacy solution is very slow, error prone and a maintenance nightmare. Hence the solution below is proposed.

1.2. Solution Description

The proposed solution includes the following components:

- AWS Data Migration Service (DMS)
- Ingestion Pipeline in Databricks with Unity Catalog

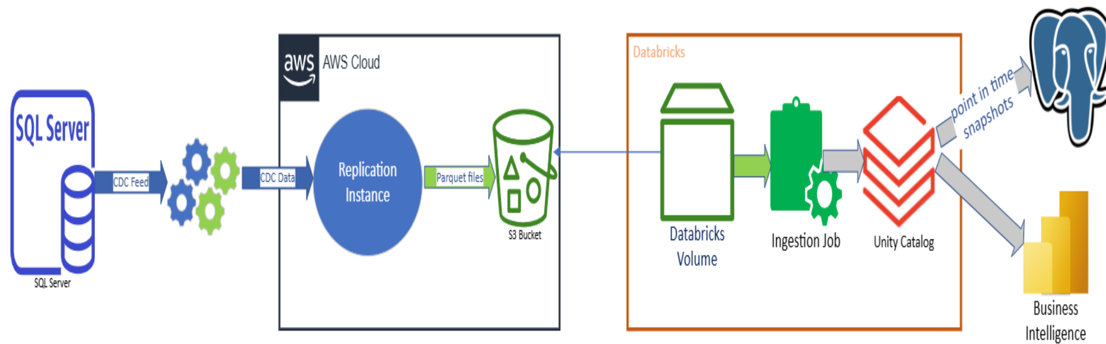

Fig 1: AWS DMS and Databricks-Based CDC Data Migration and Snapshot Architecture

2. AWS Data Migration Service (DMS)

As per AWS documentation, AWS Database Migration Service (AWS DMS) is a cloud service that makes it possible to migrate relational databases, data warehouses, NoSQL databases, and other types of data stores. You can use AWS DMS to migrate your data into the AWS Cloud or between combinations of cloud and on-premises setups. AWS DMS is a server in the AWS Cloud that runs replication software. AWS DMS is available in 2 flavors AWS DMS Serverless auto-scales based on the load and well suited for smaller, simpler data migrations. AWS DMS replication instance (also known as Standard) need manual resource management, well-suited for larger migrations.

2.1. Prerequisites for source SQL Server database

Below SQL server editions are supported when SQL server database is used as source for migration using AWS DMS

- The Enterprise, Standard, Workgroup, Developer, and Web editions support full-load replication.
- The Enterprise, Standard (version 2016 and higher), and Developer editions support CDC replication in addition to full load.

For CDC or ongoing replication there are 2 possible mechanisms available:

2.1.1. MS-Replication:

- In this mechanism, AWS DMS utilizes the distributor and publisher components of SQL Server's built-in replication.
- To automatically configure MS-Replication for change data capture (CDC), the AWS DMS endpoint user on the source SQL Server instance typically requires sysadmin privileges.

2.1.2. MS-CDC (Change Data Capture):

- It is enabled at both the database level and at the individual table level.
- A user with enough permissions can enable CDC for a database and individual tables by executing “sp_cdc_enable_db” and “sp_cdc_enable_table” stored procedures.
- Needs SQL server Agent running to create “cdc.<database_name>_capture” and “cdc.<database_name>_cleanup” jobs automatically when cdc on database is enabled.
- Need to adjust the Transaction Log Retention period based on how frequently the capture job scans the transaction log.

Once the source database is ready for replication, the following steps need to be performed to configure AWS DMS Replication:

- Create a Replication Instance (only for DMS Standard not for Serverless): This is the compute resource that performs data migration. Configure the instance class, storage, and VPC settings as required for the Replication Instance.
- Set Up Source and Target Endpoints: Define the connection details for your source and target locations. Specify whether it's a source or target endpoint and provide the necessary connection information (e.g., database type, server name, port, credentials, S3 bucket name, S3 folder name(<table_name>_incoming)). If target endpoint is S3, then provide the service access role ARN with the following permissions at the minimum: s3:PutObject, s3:DeleteObject, s3:ListBucket & s3:GetObject. Also need to provide the Endpoint Settings like dataformat(parquet) and compression_type(snappy)
- Create a Migration Task: This step defines how the data will migrate.

The following details will be provided:

- Task Name: Descriptive name of the task
- Replication Instance (only for Standard): Select the Replication Instance created earlier.
- Source Endpoint: Choose the source endpoint defined earlier (SQL Server database tables)
- Target Endpoint: Choose the target endpoint defined earlier (S3 bucket location)
- Migration Type: Full load/Full Load + CDC/CDC

Task Settings: Configure settings like table mappings, transformation rules, and error handling. You can define which tables to include or exclude and how data should be transformed during migration.

Need to create a Databricks Volume pointing to the table folder of the S3 bucket using the command below:

```
CREATE EXTERNAL VOLUME test_catalog.test_schema.test_table_incoming
LOCATION 's3://test_company-test_catalog-test_schema-us-east-1/test_table_incoming'
```

Start the migration task to migrate the data into S3 bucket table incoming folder.

3. Ingestion Pipeline Job

It gets triggered when the parquet file from DMS replication task gets dropped into the incoming folder (s3 file arrival trigger is set to watch for the files arrival into the incoming folder of test_catalog.test_schema.test_table volume). Ingestion Pipeline Jobs are triggered on a file arrival, on a schedule or manually.

The following steps are performed as part of the ingestion pipeline:
Once the files are processed the parquet files are moved into the processed volume.

The processed volume for a given table is created using the spark sql statement shown as below

```
CREATE EXTERNAL VOLUME test_catalog.test_schema. test_table_processed
LOCATION 's3://test_company-test_catalog-test_schema-us-east-1/test_table_processed'
```

The parquet file(s) are loaded into a spark dataframe as shown below.

```
df=spark.read.parquet("/Volumes/test_catalog/test_schema/test_table/incoming")
df.createOrReplaceTempView("DF")
```

In a CDC feed a particular record might have got inserted, updated and finally deleted. In the ingestion pipeline only the last operation on record will be considered. Below is the code to identify the last operation on a given record.

```
query = """SELECT A.test_key_id, A.test_field1, A.test_field2, A.test_field3, A.Op,
A.tx_commit_time FROM (SELECT test_key_id, test_field1, test_field2, test_field3,Op,
tx_commit_time, ROW_NUMBER() OVER (PARTITION BY test_key_id ORDER BY tx_commit_time DESC) AS row_num FROM DF)A where A.row_num=1"""
data_df = spark.sql(query)
```

In the CDC feed, the records will have 2 CDC related fields

1. “Op” which can hold either of 3 values “I” represents Insert, “U” represents Update and “D” represents Delete.
2. ”tx_commit_time” represents operation timestamp.

Next the above dataframe needs to be divided into upsert transactions and delete transactions dataframes using the code below:

```
upsert_df = data_df.filter("Op in ('I', 'U')")
delete_df = data_df.filter("Op = 'D'")
upsert_df.createOrReplaceTempView("updates")
delete_df. createOrReplaceTempView("deletes")
```

The following are the scenarios that need to be addressed as part of SCD Type2 implementation and for historical tracking.

Scenario 1: The last operation on record is “Insert” but the record already exists in UC target table. Then the CDC source record is compared against the target record and if there are any changes, then perform the following actions:

- Update the old record current field to false and also update the end_date field value to txt_commit_time field value of the source record.
- Insert the source record with current field value set to true, set txt_commit_time as effective_date and set end_date field value as null

If there are no changes then perform no operation.

Scenario 2: The last operation on record is “Insert” and the record does not exist in the UC target table

- Insert the source record with current field value set to true, txt_commit_time as effective_date and set end_date field value as null

Scenario 3: The last operation on record is “update” and the record does not exist in the UC target table

- Insert the source record with current field value set to true, txt_commit_time as effective_date and end_date field value as null

Scenario 4: The last operation on record is “update” and the record already exists in UC target table

If there are any changes in the CDC source record when compared to the target record, then perform the following actions:

- Update the old record current field value to "false" and update the end_date field to txt_commit_time field value of the source record.
- Insert the source record with current field value set to true, txt_commit_time as effective_date and end_date field value as null

If there are no changes then perform no operation

Scenario 5: The last operation on record is "delete" and the record already exists in UC target table then perform the following actions:

- Update the old record current field value set to false and update the end_date field to txt_commit_time filed value of the source record.

The following spark DML statement is executed to implement SCD type2.

```
upsert_query = """MERGE INTO test_catalog.test_schema.test_table tbl
USING
(
    SELECT updates.test_key_id as merge_key, updates.*
    FROM updates
    UNION ALL
    SELECT NULL as merge_key, updates.*
    FROM updates
    JOIN test_catalog.test_schema.test_table as test_tbl ON
    updates. test_key_id = test_tbl. test_key_id
    WHERE test_tbl.current = true AND (
        (test_tbl.test_field1 <> updates.test_field1)
        OR (test_tbl.test_field2 <> updates.test_field2)
        OR (test_tbl.test_field3 <> updates.test_field3))
) staged_updates ON tbl. test_key_id = merge_key
WHEN MATCHED AND tbl.current = true AND
(
    (tbl.test_field1 <> staged_updates.test_field1)
    OR (tbl.test_field2 <> staged_updates.test_field2)
    OR (tbl.test_field3 <> staged_updates.test_field3)
) THEN UPDATE SET tbl.current = false,
    tbl.end_date = staged_updates.tx_commit_time
WHEN NOT MATCHED THEN
    INSERT (test_key_id, test_field1, test_field2,
    test_field3, current, effective_date, end_date)
    VALUES (staged_updates.test_key_id, staged_updates.test_field1,
    staged_updates.test_field2, staged_updates.test_field3, true,
    staged_updates.tx_commit_time, NULL)"""
spark.sql(upsert_query).display()

delete_query= """MERGE INTO test_catalog.test_schema.test_table tbl USING deletes del
ON tbl.test_key_id = del.test_key_id
WHEN MATCHED THEN
UPDATE SET tbl.current = false,
tbl.end_date = del.tx_commit_time
WHEN NOT MATCHED THEN INSERT
(test_key_id, test_field1, test_field2,
test_field3, current, effective_date, end_date)
VALUES
(del.test_key_id, del.test_field1, del.test_field2,
del.test_field3, false, del.tx_commit_time, del.tx_commit_time)"""
spark.sql(delete_query).display()
```

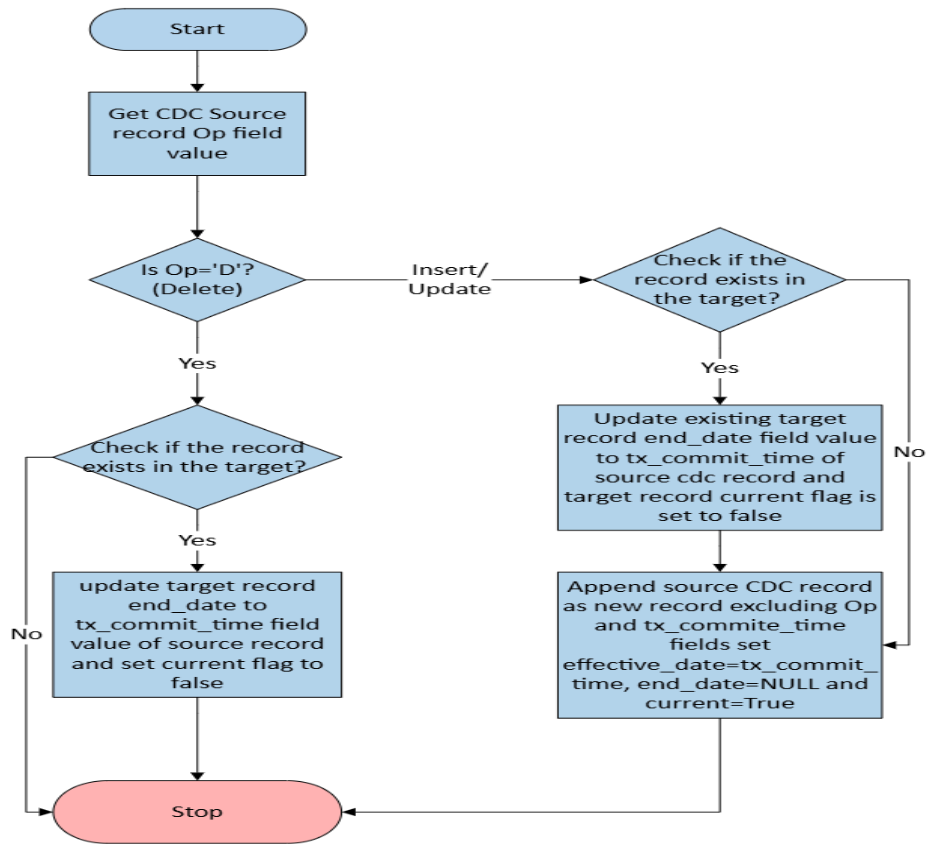

Fig 2: Export Current snapshot or snapshot in the past to another Relational Database

Here is the flowchart representing the scenarios discussed above:

Below is the query to extract current snapshot from the UC table:

```
current_snapshot_query = """select test_key_id, test_field1, test_field2, test_field3
from test_catalog.test_schema.test_table where current=true"""
snapshot_df = spark.sql(current_snapshot_query)
```

Below is the query to extract snapshot at some point in time from the UC table:

```
past_snapshot_query=f"""select test_key_id, test_field1, test_field2, test_field3
from test_catalog.test_schema.test_table where current=true and effective_date >= to_date('{snapshot_date}')
union
select test_key_id, test_field1, test_field2, test_field3
from test_catalog.test_schema.test_table where current=false and (to_date('{snapshot_date}') between effective_date and end_date)"""
snapshot_df = spark.sql(past_snapshot_query)
```

Below is the code snippet to push the snapshot into Postgres relational database

```
def build_db_query(table_name: str, keys: List[str], column_names: List[str]) -> str:
    cols_to_update = [col_name for col_name in column_names if col_name not in keys]
    query = f"""INSERT INTO {table_name} ({", ".join(keys) + ", " + ", ".join(cols_to_update)})
        VALUES ({",".join("%s" for _ in range(len(column_names)))})
        ON CONFLICT ({', '.join(keys)}) DO UPDATE SET
        ({', '.join([f"{col}" for col in cols_to_update])}) = ROW({', '.join([f"EXCLUDED.{col}" for col in cols_to_update])});"""
    return query
```

```
def upsert_udf(iterator):
    config_props = br_config.value
    database = config_props.get("database")
    host = config_props.get("host")
    user = config_props.get("user")
    password = config_props.get("password")
    db_provider_module = importlib.import_module(config_props.get("db_provider_module"))
    db_conn = db_provider_module.connect(
        host=host, user=user, password=password, database=database
    )
    for pdf in iterator:
        # can perform additional vectorized operations here on the pandas dataframe
        pdf = pdf.replace(float('nan'), None).replace({pd.NaT: None})
        rows = tuple(map(tuple, pdf.values))
        query = config_props.get("query")
        cursor = db_conn.cursor()
        try:
            # Use executemany to efficiently insert/update multiple rows
            cursor.executemany(query, rows)
            db_conn.commit()
        except Exception as e:
            db_conn.rollback()
            raise e
        finally:
            cursor.close()
        yield pdf
    db_conn.close()
```

```
cols = [col_name.lower() for col_name in snapshot_df.columns]
query = build_db_query('test_catalog.test_schema.test_table', ['test_key_id'], cols)
num_connections = int(config_dict.get("num_connections"))
snapshot_df = snapshot_df.coalesce(num_connections)
config_props = {"host": config_dict.get("host"), "user": config_dict.get("user"),
                "password": config_dict.get("password"), "database": config_dict.get("database")
                , "query": query,  "db_provider_module": config_dict.get("db_provider_module")
                }
br_config = spark.sparkContext.broadcast(config_props)
```

4. Conclusion

AWS DMS with Ingestion Framework presents a powerful solution for the data migration problem, provides high performance, high reliability, cost-efficient, low maintenance and faster execution compared to the legacy solution.

References

1. AWS Documentation: https://docs.aws.amazon.com/dms/
2. Databricks Documentation: https://docs.databricks.com/aws/en/sql/language-manual/sql-ref-volumes
3. SCD Type 2 Documentation: https://community.databricks.com/t5/technical-blog/how-to-implement-slowly-changing-dimensions-when-you-have/ba-p/40568
4. Bulk Upserts Pyspark: https://medium.com/@mvamsikhyd/implement-bulk-upserts-merge-into-rds-rdbms-from-apache-spark-5cbd60047a38