December 5, 2022 | Matthew David December 5, 2022

Redshift to Snowflake Migration

Snowflake has become one of the most popular tools in the modern data stack for data lakes, data warehouses, and general-purpose data workloads. With more and more organizations looking to take advantage of Snowflake, that means a lot of data migrations from popular tools like AWS Redshift.

In this post, we'll walk through the steps for a typical data migration from Redshift to Snowflake. Then, we'll look at how a tool like Datafold's Data Diff can help with data validation and verification, protecting data teams from common pitfalls in data migrations.

How to approach a Redshift to Snowflake migration

The typical data migration from Redshift to Snowflake can be broken down into the following key steps:

Connect to Redshift with Python.
Migrate the Data Definition Language (DDL) for tables.
Export the data out of Redshift to AWS S3.
Import the data into Snowflake.
Validate and verify the data migration **(with Data Diff)**.

Let's walk through these steps one by one.

Connect to Redshift with Python

The first step in any migration away from Redshift is setting up a connection so that you can automate the pulling of the DDL for tables in Redshift. One of the simplest ways to do this is with Python and a well-known package called psycopg2, which is used extensively for Postgres.

```
$ pip3 install psycopg2
```

With connection information configured for the Redshift cluster, obtaining a connection using Python and psycopg2 is straightforward.

```
import psycopg2

def connect_to_redshift():
    try:
        conn = psycopg2.connect(
            dbname='migration_test',
            host='my-redshift-instance.us-west-2.redshift.amazo
            port=5439,
            user=f'{user}',
            password=f'{password}')
        return conn
    except Exception as err:
        print(err)
        exit(1)
```

With a connection to Redshift established, the next step is to pull the DDL for the table(s) that need to be migrated and created inside Snowflake. There are, of course, some slight differences in syntax for items like timestamps between the two systems, and these will need to be modified for use in Snowflake.

DDL Migration

We will be migrating a sample table called sales_reports from Snowflake to Redshift. Shown below is the DDL for that table in each stack. As you can see, the DDL statements are similar, with a slight variation in how the DECIMAL value in Redshift is called NUMBER in Snowflake.

Snowflake DDL:

```
CREATE TABLE IF NOT EXISTS sales_reports
(
   id INT,
   report_date DATE,
   sales_rep_id BIGINT,
   sales_amount NUMBER(6,2)
);
```

Redshift DDL:

```
CREATE TABLE IF NOT EXISTS sales_reports
(
   id INT,
   report_date DATE,
   sales_rep_id BIGINT,
   sales_amount DECIMAL(6,2)
);
```

With a connection created to the Redshift cluster, a cursor can be created to query the information schema for the table(s) needed for the migration.

```
cursor = conn.cursor()

cursor.execute("""
              SELECT table_schema, table_name
              FROM information_schema.tables
              WHERE table_type='BASE TABLE'
              AND table_name = 'sales_reports';
""")

cursor.execute(sql)
DDL = cursor.fetchall()
cursor.close()
```

With the DDL for the migration tables in hand, making any modifications needed and running the DDL create statements in Snowflake are straightforward tasks.

import snowflake.connector

```
import snowflake.connector

con = snowflake.connector.connect(user='XXXX', password='XXXX',
con.cursor().execute(”””
CREATE TABLE IF NOT EXISTS sales_reports
(
   id INT,
   report_date DATE,
   sales_rep_id BIGINT,
   sales_amount NUMBER(6,2)
);
""""")
```

Export Data out of Redshift to S3

With our schemas created in Snowflake, we're ready for the ingestion of data. The next step in the migration process is to export the data from each of the Redshift tables. This can be done easily with the UNLOAD command.

```
UNLOAD ('SELECT * FROM sales_report')
TO 's3://migration-project/sales/unload/sales_report_'
```

The default format from the UNLOAD command is pipe-delimited text files. The FORMAT AS option can be used to specify other file types like Parquet, JSON, or CSV.

Two of the main file format options commonly used are CSV or Parquet. If the data size is small, around 6GB, the PARALLEL OFF option can be used to unload the data to a single file; otherwise, the UNLOAD command will create the same number of files as the number of slices on the Redshift cluster.

Import Data into Snowflake

The importing of the S3 data from Redshift into Snowflake is a multistep process. First, we need to create a **File Format Object** in Snowflake, based on the previous files that were exported. In most instances, this will already exist in Snowflake. If CSV files were created, the new File Format Object would indicate CSV delimiter options.

```
CREATE or REPLACE file format migration_csv_format
        type = 'CSV'
  field_delimiter = '|'
  skip_header = 1;
```

The next step is to create a **Named Stage Object** that references the location of the Redshift migration files in their S3 bucket.

```
CREATE or REPLACE stage migration_csv_stage
        file_format = migration_csv_format
  url = 's3://migration-project/sales/unload/';
```

Lastly, we can run a COPY INTO statement to push our files from the S3 bucket into each Snowflake table.

```
COPY INTO sales_reports
        from @migration_csv_stage/sales/unload/
  pattern='.*sales_report_*.csv'
  on_error = 'skip_file';
```

Validate and Verify with Data Diff

With our data successfully migrated from Redshift and into Snowflake, we come to the most important part of any migration process: validating and verifying that all the data has been pulled and that it has been moved correctly.

Data validation is the most time consuming and challenging part of a migration project, typically done by a large team manually writing many different SQL commands to verify data and counts.

Fortunately, Data Diff from Datafold provides this functionality. Depending on the size of your migration, employing Data Diff could potentially save hundreds of engineering hours.

Data Diff can verify tables of all sizes—from hundreds to billions of records—providing row-level verification, monitoring and alerting, and even self-healing. Data Diff can be easily installed with Python's pip.

```
$ pip3 install data-diff
```

Running data-diff at the command line requires four arguments:

DB1_URI
TABLE1_NAME
DB2_URI
TABLE2_NAME

The database URI formats are familiar. In the case of our example migration, we want to compare Redshift to Snowflake, and so we specify each database URI along with the corresponding table.

```
$ data-diff \
snowflake://user:password@account/warehouse?database=database&s
sales_report \
redshift://username:password@hostname:5439/database \
sales_report
```

Using data-diff at the command line is a powerful and easy way to automate and run various verifications and checks for data migrations. As a CLI, data-diff is perfect for ad-hoc data verification as well as automated tasks built into CI/CD pipelines.

A common migration verification

When carrying out a large migration, missing even a single row can lead to inaccurate downstream results and mistrust in the migration process and new platform. Data Diff enables row-by-row comparisons for the initial migration as well as for ongoing data being loaded.

Two of the most common questions and problems related to data migrations are:

Is the data between the two systems the same?
If there are differences, exactly which rows are different?

Data Diff can easily answer these questions for migrations of both large and small datasets, even across many different tables.

Before running validation, it’s important to understand some of the command-line options available when using data-diff:

> The bisection-factor is the “segments per iteration.” For example, if a table has 1 million rows and we set the bisection factor to 10, then we will get 10 segments of 100,000 records each.
> The key-column option indicates the primary key for a table.
> The threads option lets you increase the number of worker threads used per database to improve performance.
> The min-age and max-age options allow the diff to compare rows older or younger than a configurable parameter. The valid units for these options are:

```
d, days, h, hours, min, minutes, mon, months, s, seconds, w, we
```

> The update-column option is an indicator for which column in the table holds the last updated timestamp for rows.

Let’s return to our example migration from Redshift to Snowflake. If our sales_report table had 1 million records, we might run data-diff with the following options:

```
data-diff \
snowflake://user:password@account/warehouse?database=database&s
sales_report \
redshift://username:password@hostname:5439/database sales_repor
```

```
--bisection-factor=10
--key-column=sales_id
--threads=3
--update-column=inserted_at
--min-age=5min
--columns=[sales_id, amount]
```

This command would compare the sales_report table in Snowflake to that of Redshift, with a million records split into 10 segments of at most 100,000 records each. Data Diff will compare the sales_id and amount columns between the two tables, using sales_id as the primary key column. Data Diff will also ignore rows inserted in the last five minutes.

For each row that is different, Data Diff will output the values for those columns passed in as key-column and update-column arguments, allowing for detailed verification and inspection of any possible migration issues.

The Benefits of Using Data Diff

It's easy to see that adding a tool like Data Diff is essential for any large data migration. The ability to use a single tool to compare any number of tables during a complex migration **saves engineering time and effort** and **reduces the possibility of missing or incorrect data validations**.

Because Data Diff is easy to install and run as a CLI tool, **integration into CI/CD and other data pipelines for automated execution** is simple. Continuously running data validation commands over several days and weeks during a large migration can ensure any problems are found and addressed quickly.

Conclusion

More and more organizations are adopting Snowflake, and many of them are coming from data stacks dependent on Redshift. Although the steps for migrating data from Redshift to Snowflake are clear and simple, the main challenge is the data validation and verification that is necessary to confirm a successful migration. In the past, comprehensive data validation required an immense amount of engineering resources and time. Now, with Data Diff from Datafold, post-migration data validation can be automated, yielding high-quality results in a fraction of the time.

Find out more about Data Diff here.
