Protrac to Smartabase data migration process flow

Step 1) Understand the data available in the Protrac database
Write a python script to connect to the Protrac database instance and using SQL write a query that lists all the tables and views within the database.
Save the list as a csv file or as a pdf so it can be used as a master reference document or data dictionary.

Services/ Software Used: Jupyter Notebook, Python.

Step 2) Determine what data you want to extract from the Protrac database
Using the list created, identify which tables and views have the data you wish to extract from the database.

Services/ Software Used:

Step 3) Add any new tables and views to the existing data extraction process
Using AWS DMS that is already configured and connected to the PgAdmin database, add any additional tables or views that you want extracted to the Client's data lake and set the Client Data Lake S3 location for where the data will be saved.
Source: PostgresDB
Target: S3

Services/ Software Used:

Step 4) Map the data tables and fields in Protrac to the Smartabase data structures that have been created
Create a data mapping table that lists the tables, views, and fields in Protrac to their matching tables and fields in Smartabase including any relationships or any data transformations required.

Services/ Software Used:

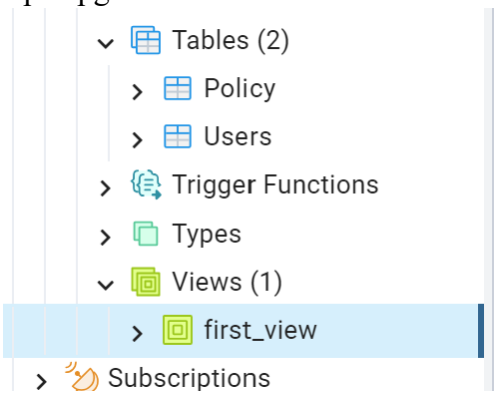
Step 5) Copy the Protrac data to Smartabase

i. Obtain a long living Smartabase API token
ii. Create a Python application that iterates through the mapping document created in step 4 and with each step, performs any data transformations and data joins using the data stored in the Swans Data Lake and maps it to the Smartabase structure. The application then connects to the Smartabase API using the token and calls the appropriate API's that will push the mapped data into Smartabase.

Services/ Software Used:

Step 1) Understand the data available in the Protrac database

Open pgAdmin and check all the tables and views within the PostgreSQL database.



Open Jupyter notebooks, create a .ipynb file to create a .csv file containing the list all the tables and views within the database.

Python Script:

```
pip install psycopg2-binary
pip install tablib
pip install reportlab
```

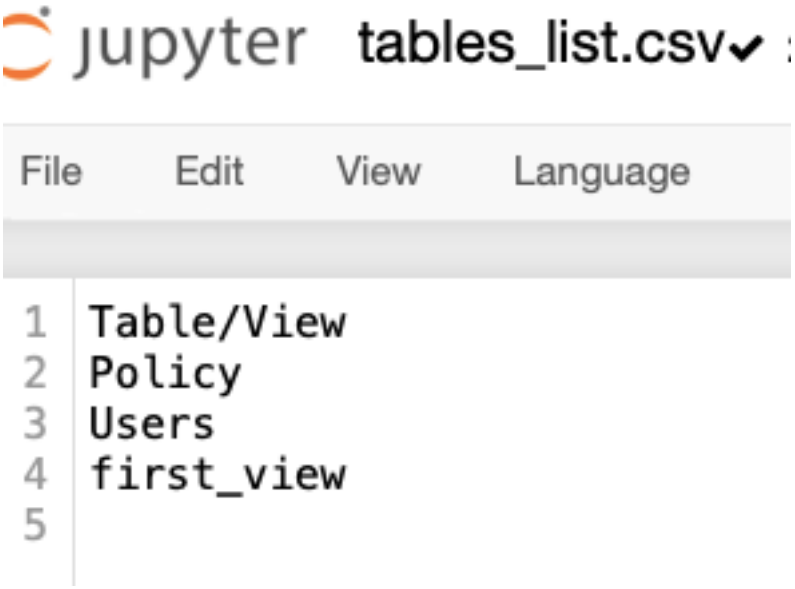
```
import psycopg2
import csv
from tablib import Dataset
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas

# Database connection parameters
db_params = {
    'host': 'localhost',
    'database': 'schneider_db',
    'user': 'postgres',
    'password': 're******',
}

def get_tables_and_views(connection):
    query = """
        SELECT table_name
```

```
            FROM information_schema.tables
            WHERE table_schema = 'public'
        """
        with connection.cursor() as cursor:
            cursor.execute(query)
            tables = cursor.fetchall()
        return tables

    def save_to_csv(tables, filename='tables_list.csv'):
        with open(filename, 'w', newline='') as csvfile:
            csv_writer = csv.writer(csvfile)
            csv_writer.writerow(['Table/View'])
            csv_writer.writerows(tables)



    def main():
        try:
            # Connect to the PostgreSQL database
            connection = psycopg2.connect(**db_params)

            # Retrieve tables and views
            tables = get_tables_and_views(connection)

            # Save to CSV
            save_to_csv(tables)

            print("Tables and views list saved successfully.")
            print(tables)

        except psycopg2.Error as e:
            print(f"Error: {e}")

        finally:
            if connection:
                connection.close()

    if __name__ == "__main__":
        main()
```

OUTPUT:

```
Tables and views list saved successfully.
[('Policy',), ('Users',), ('first_view',)]
```

.CSV file content



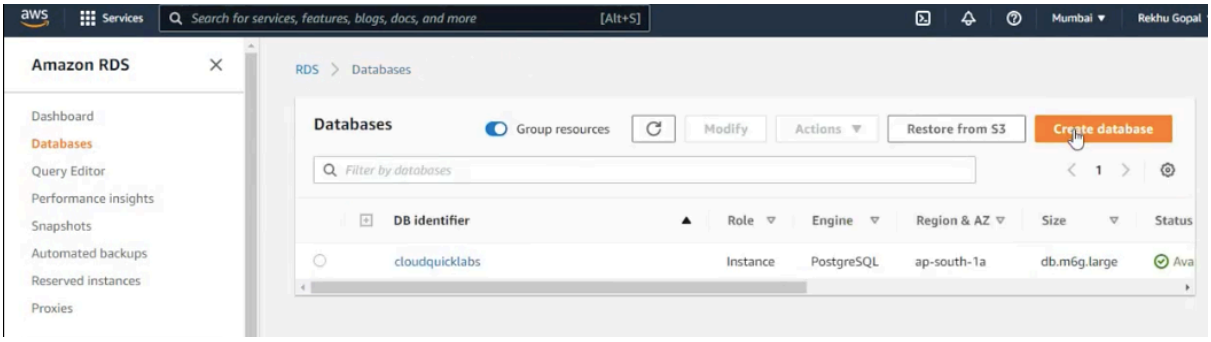
Note: The Python script should be running on the same local system where the pgAdmin (PostgreSQL) DBMS is.

Step 2) Determine what data you want to extract from the Protrac database

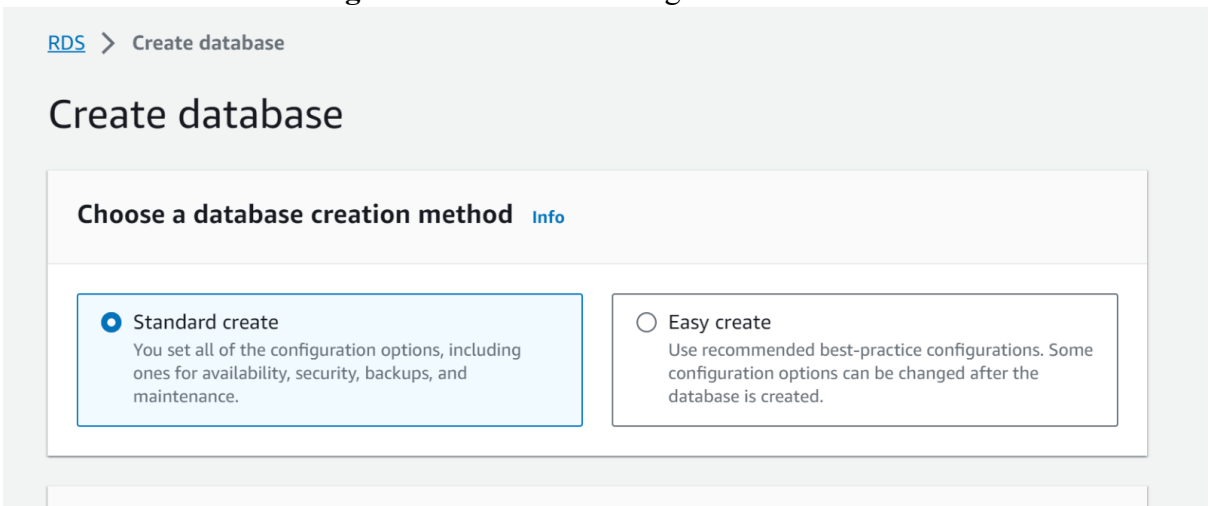
Using the list created in .CSV file above, identify which tables and views have the data you wish to extract from the database.

Step 3) Add any new tables and views to the existing data extraction process

Using AWS DMS that is already configured and connected to the Protrac database, add any additional tables or views that you want extracted to the Client's data lake and set the Client Data Lake S3 location for where the data will be saved.
Source: PostgresDB
Target: S3



Go to **RDS** & start **creating a database** as following:



Engine options

Engine type Info

- ○ Aurora (MySQL Compatible)
- ○ Aurora (PostgreSQL Compatible)
- ○ MySQL
- ○ MariaDB
- ◉ PostgreSQL
- ○ Oracle
- ○ Microsoft SQL Server
- ○ IBM Db2

Engine version Info

View the engine versions that support the following database features.

▼ Hide filters

Show versions that support the Multi-AZ DB cluster Info

Create a A Multi-AZ DB cluster with one primary DB instance and two readable standby DB instances. Multi-AZ DB clusters provide up to 2x faster transaction commit latency and automatic failover in typically under 35 seconds.

Engine Version

PostgreSQL 14.10-R1 ▼

Templates

Choose a sample template to meet your use case.

- ○ Production
  Use defaults for high availability and fast, consistent performance.
- ◉ Dev/Test
  This instance is intended for development use outside of a production environment.
- ○ Free tier
  Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS. Info

Availability and durability

Deployment options Info

The deployment options below are limited to those supported by the engine you selected above.

- ○ Multi-AZ DB Cluster
  Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.
- ○ Multi-AZ DB instance
  Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.
- ◉ Single DB instance
  Creates a single DB instance with no standby DB instances.

Settings

DB instance identifier Info

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

database-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

postgres

Public access Info

- (●) Yes
  RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.
- ( ) No
  RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

- (●) Choose existing
  Choose existing VPC security groups
- ( ) Create new
  Create new VPC security group

Existing VPC security groups

Choose one or more options

default ×

Availability Zone Info

No preference

RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

- [ ] Create an RDS Proxy Info
  RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see Amazon RDS Proxy pricing.

Estimated Monthly costs

DB instance	153.30 USD
Storage	23.00 USD
Total	**176.30 USD**

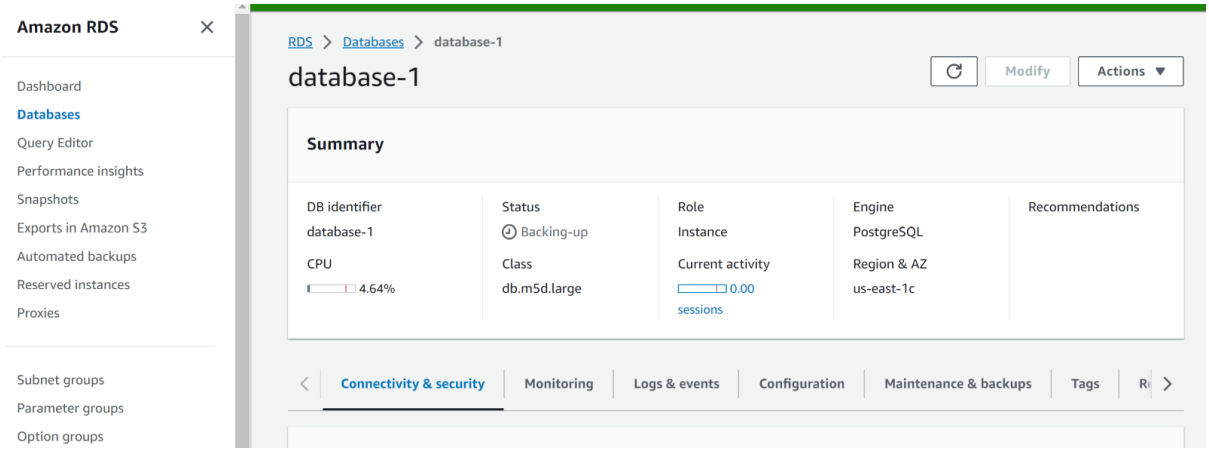
This billing estimate is based on on-demand usage as described in Amazon RDS Pricing. Estimate does not include costs for backup storage, IOs (if applicable), or data transfer.

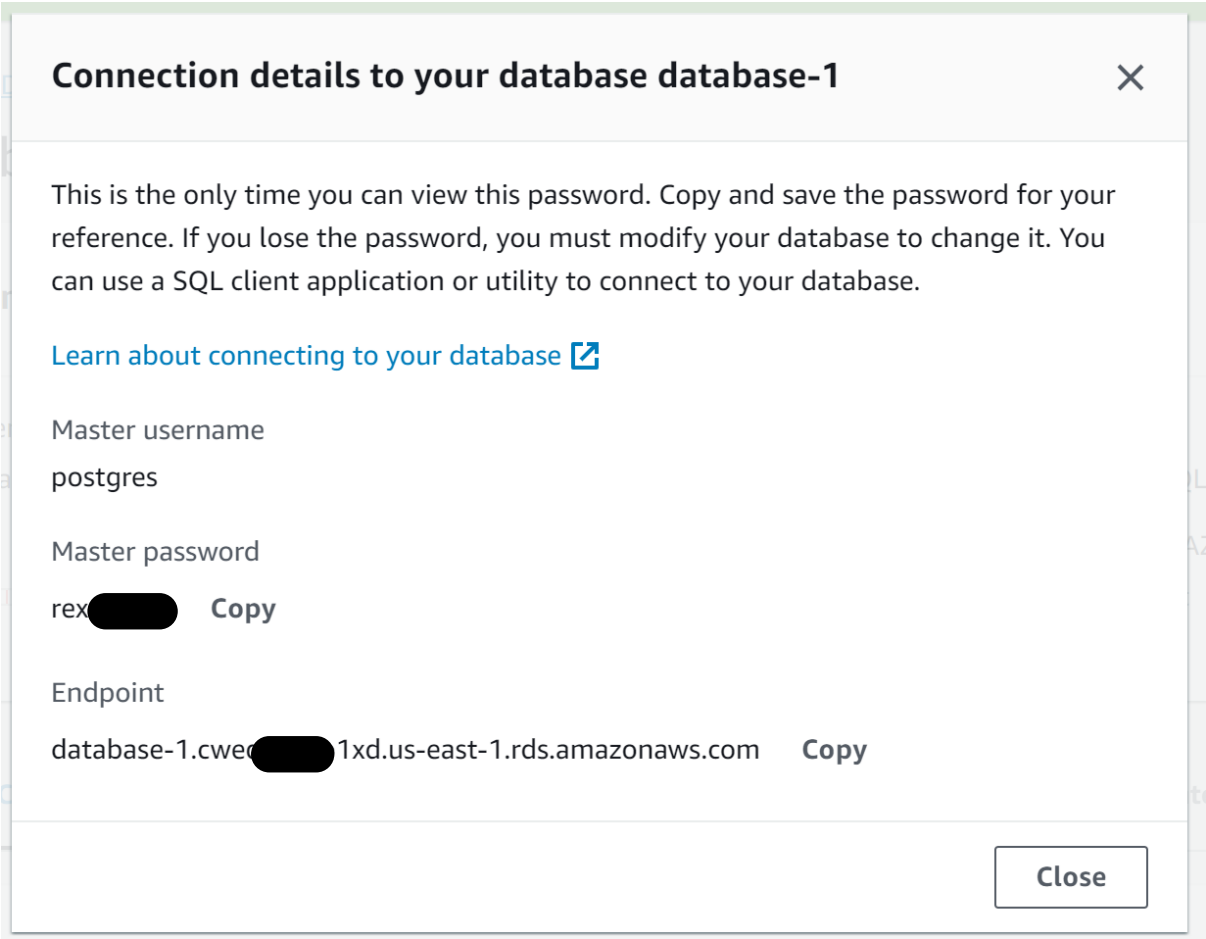
Estimate your monthly costs for the DB Instance using the AWS Simple Monthly Calculator.

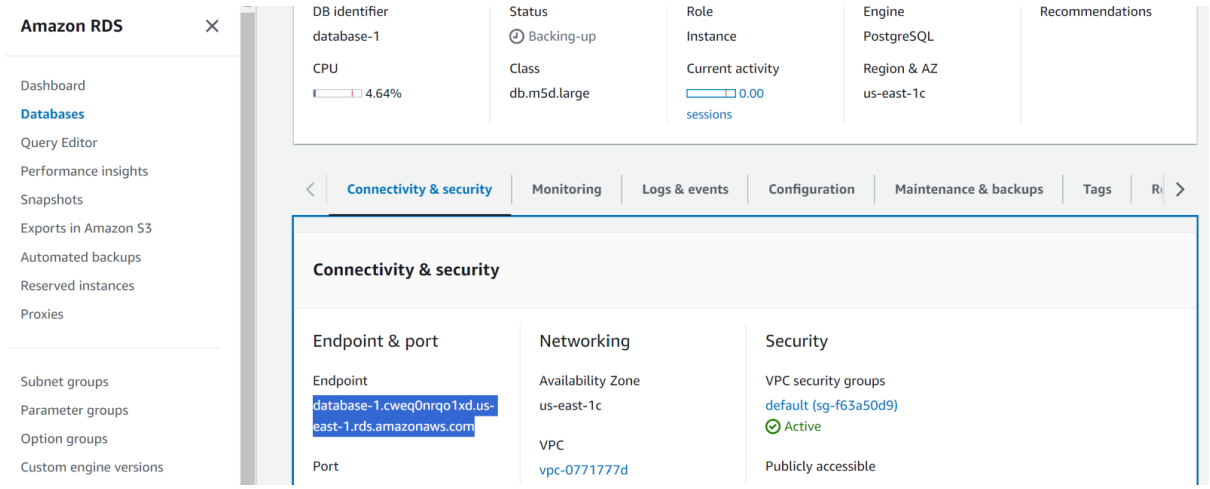
You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

Cancel **Create database**

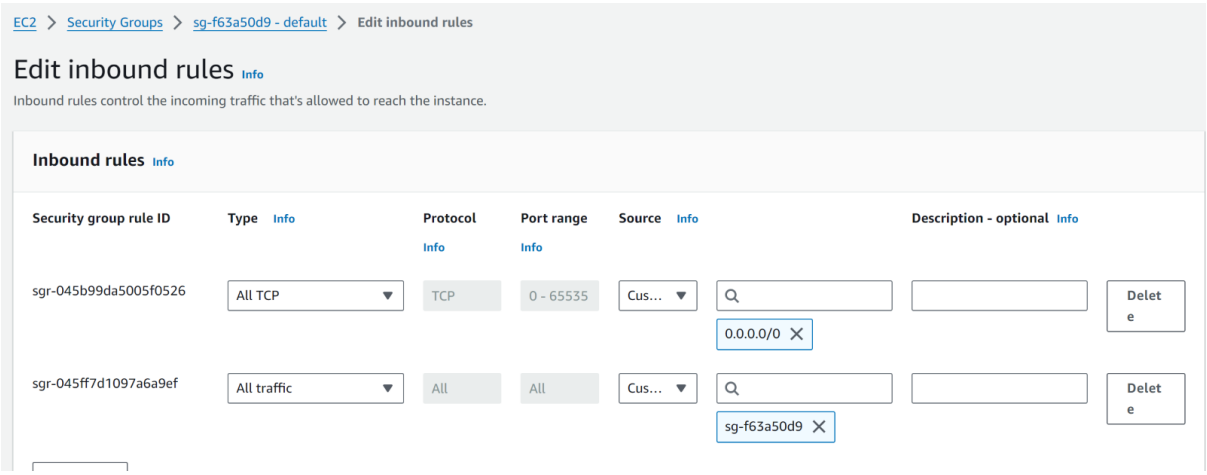
How to connect RDS instance with pgAdmin client:



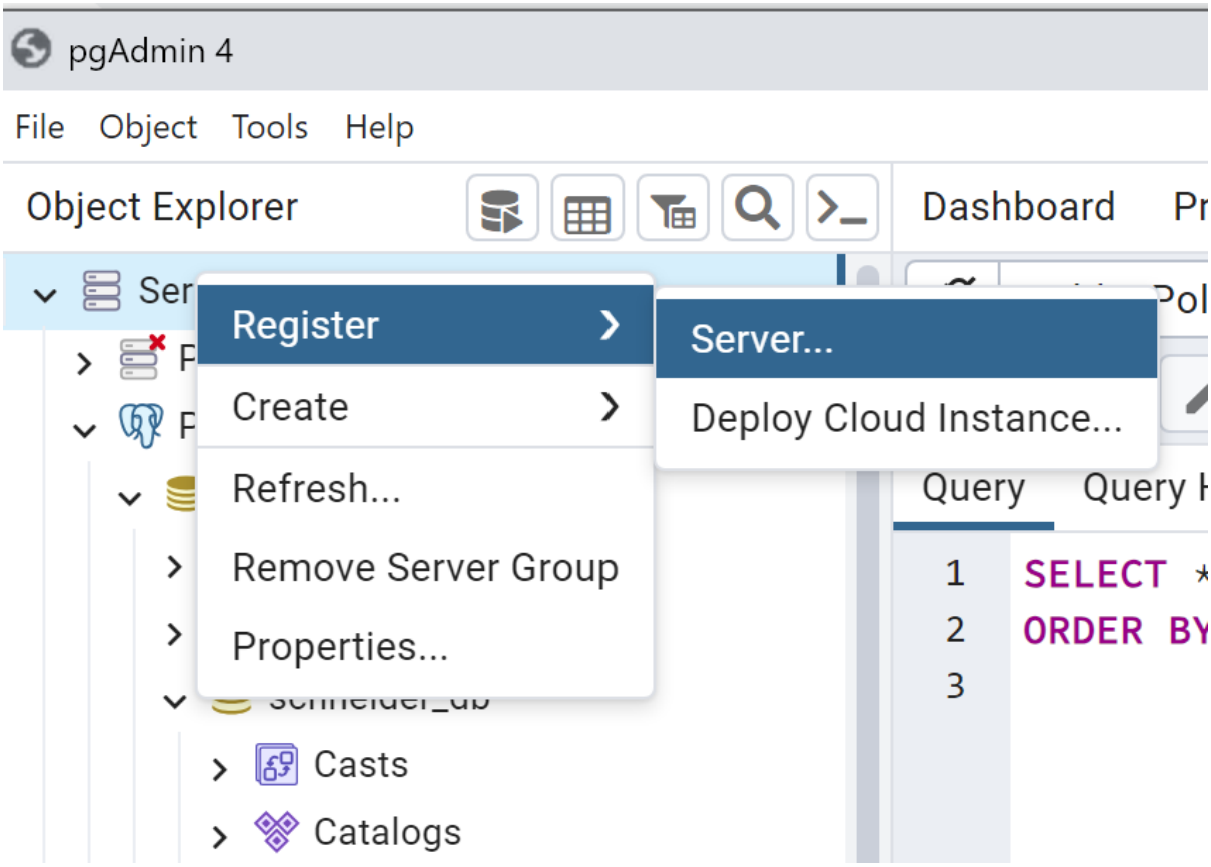


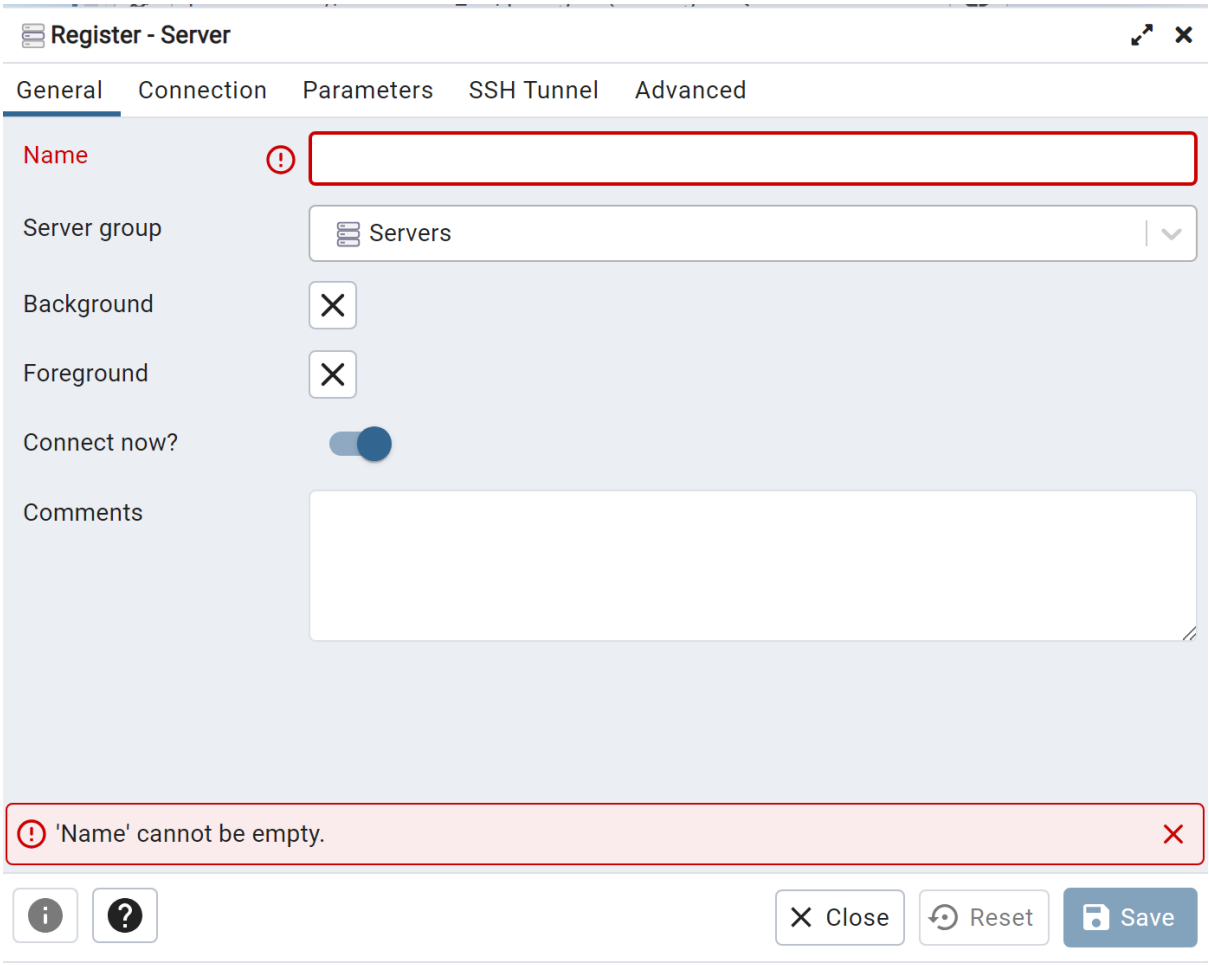


Creating required Inbound rules for RDS Postgres instance:

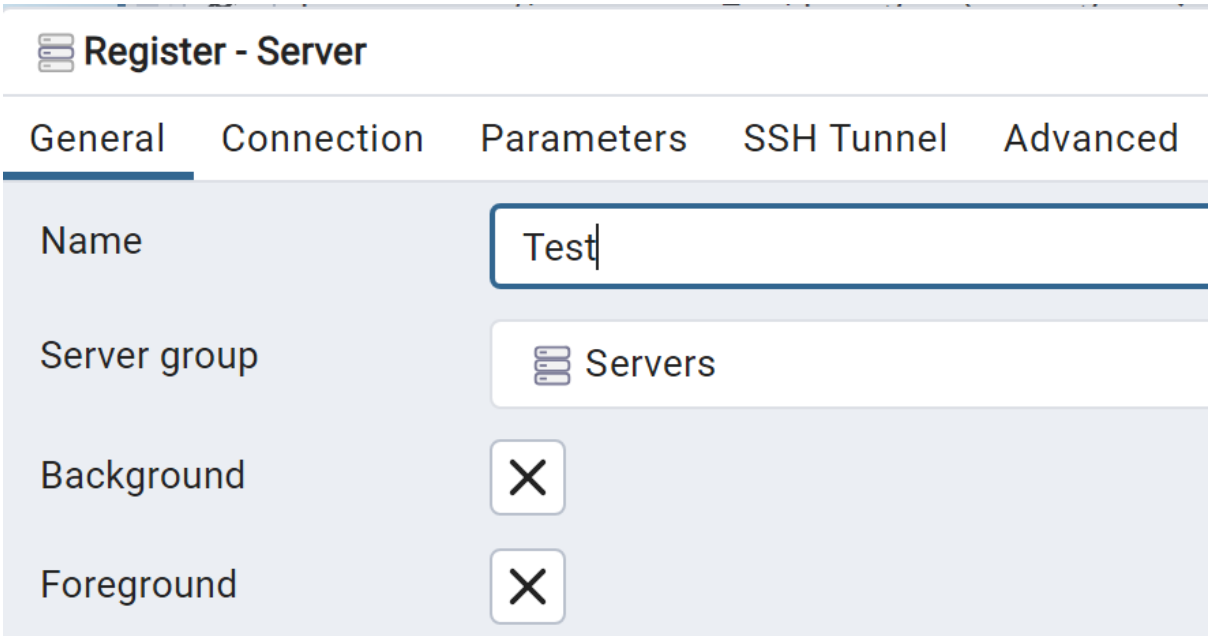


Go to pgAdmin, click Server

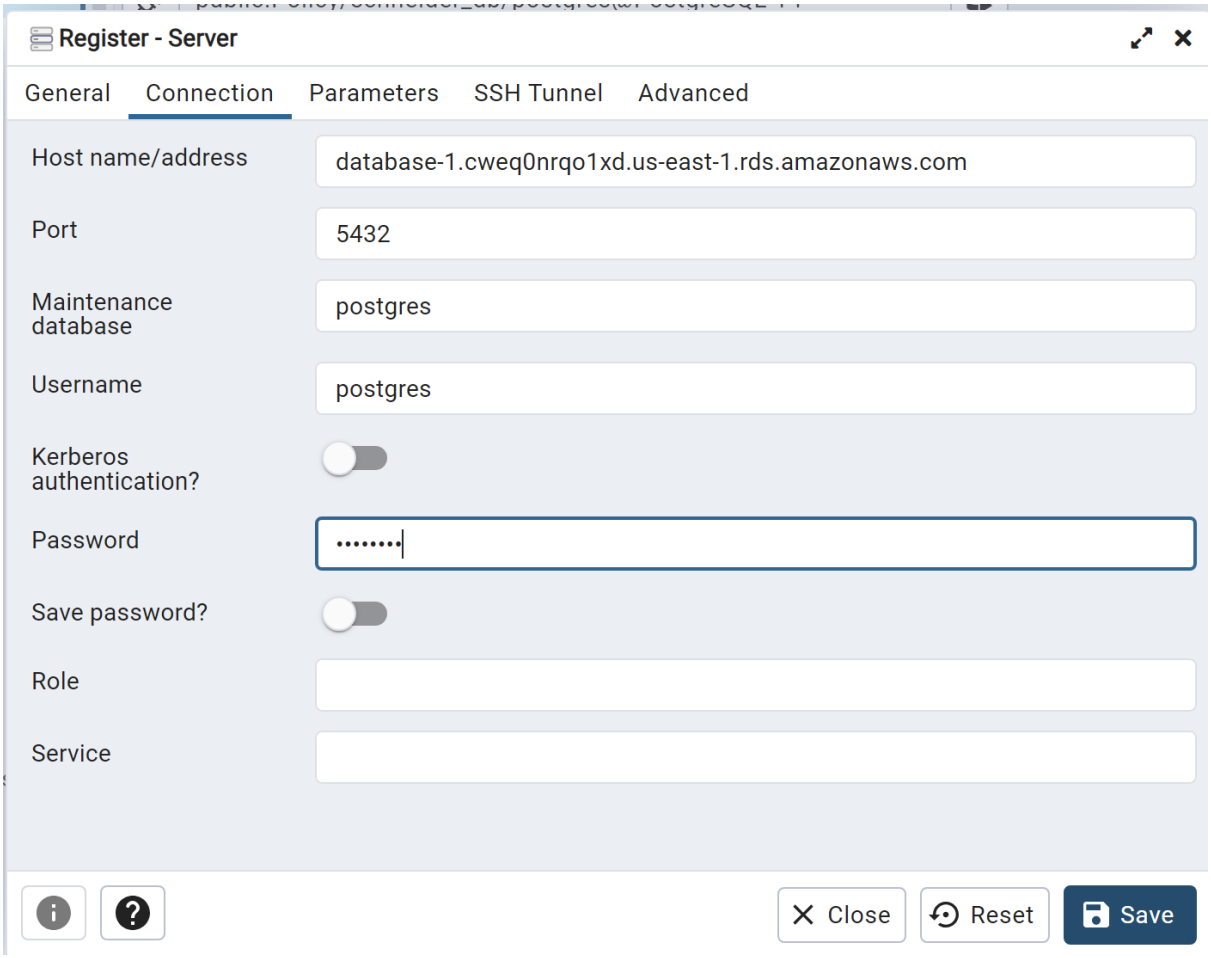

pgAdmin 4
File
Object
Tools
Help
Object Explorer
Dashboard
Register
Create
Refresh...
Remove Server Group
Properties...
Server...
Deploy Cloud Instance...
Query
1
2
3
SELECT *
ORDER BY
Casts
Catalogs



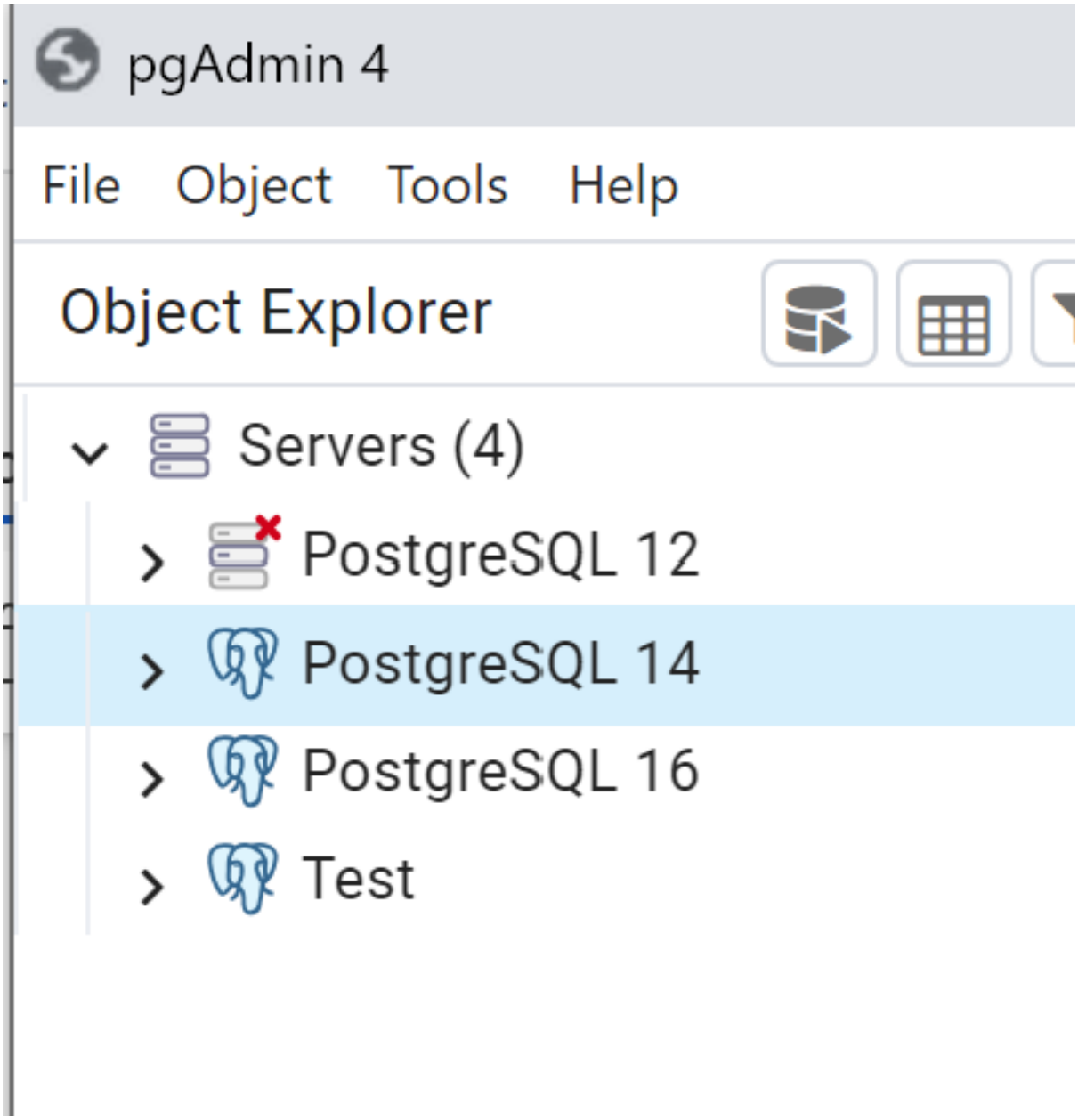
Give name **Test**



Go to Connection, paste Endpoint URL. Fill all details and click Save.

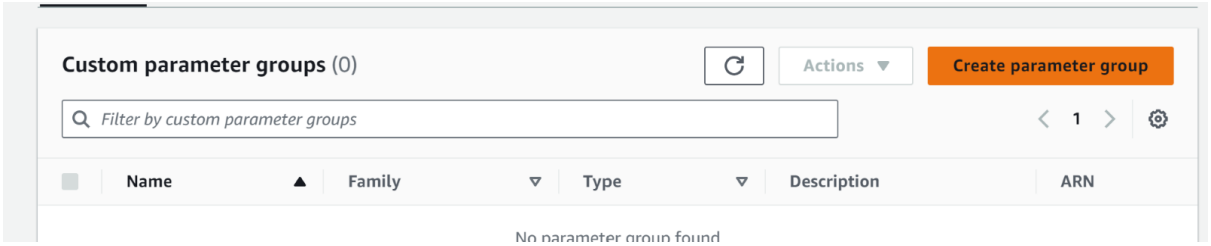


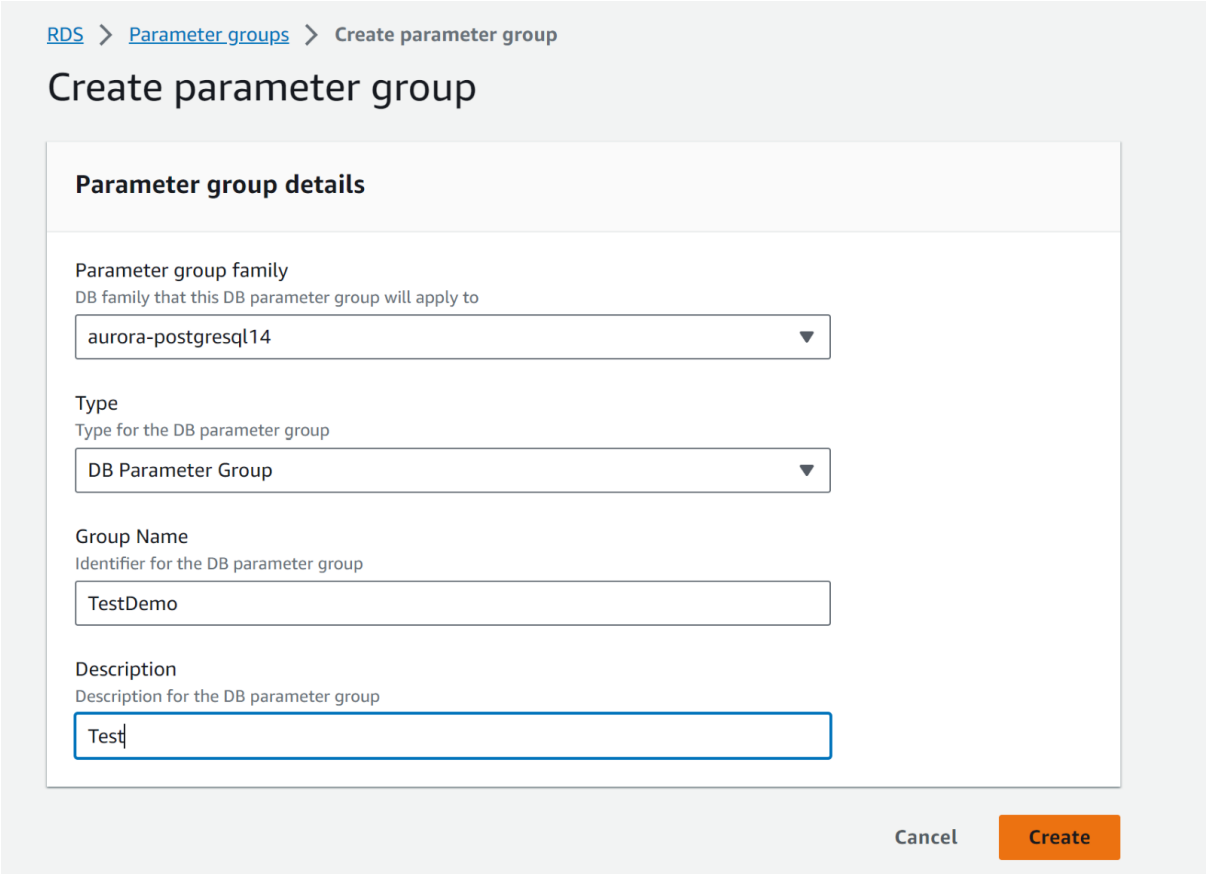
You can see the Test connection created below



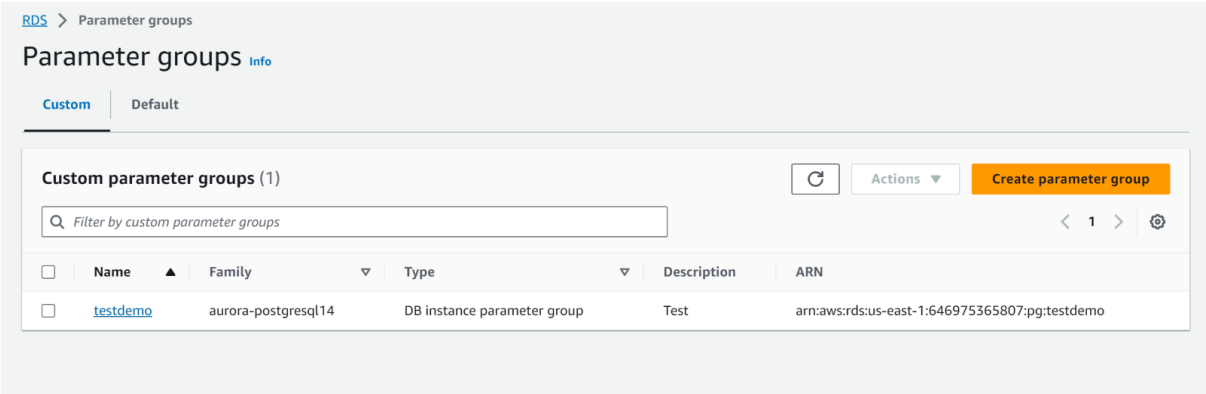
As we can see above, our **Test** server is created

Now, as we need to do the **CDC (change data capture)** to create a **Parameter Group**

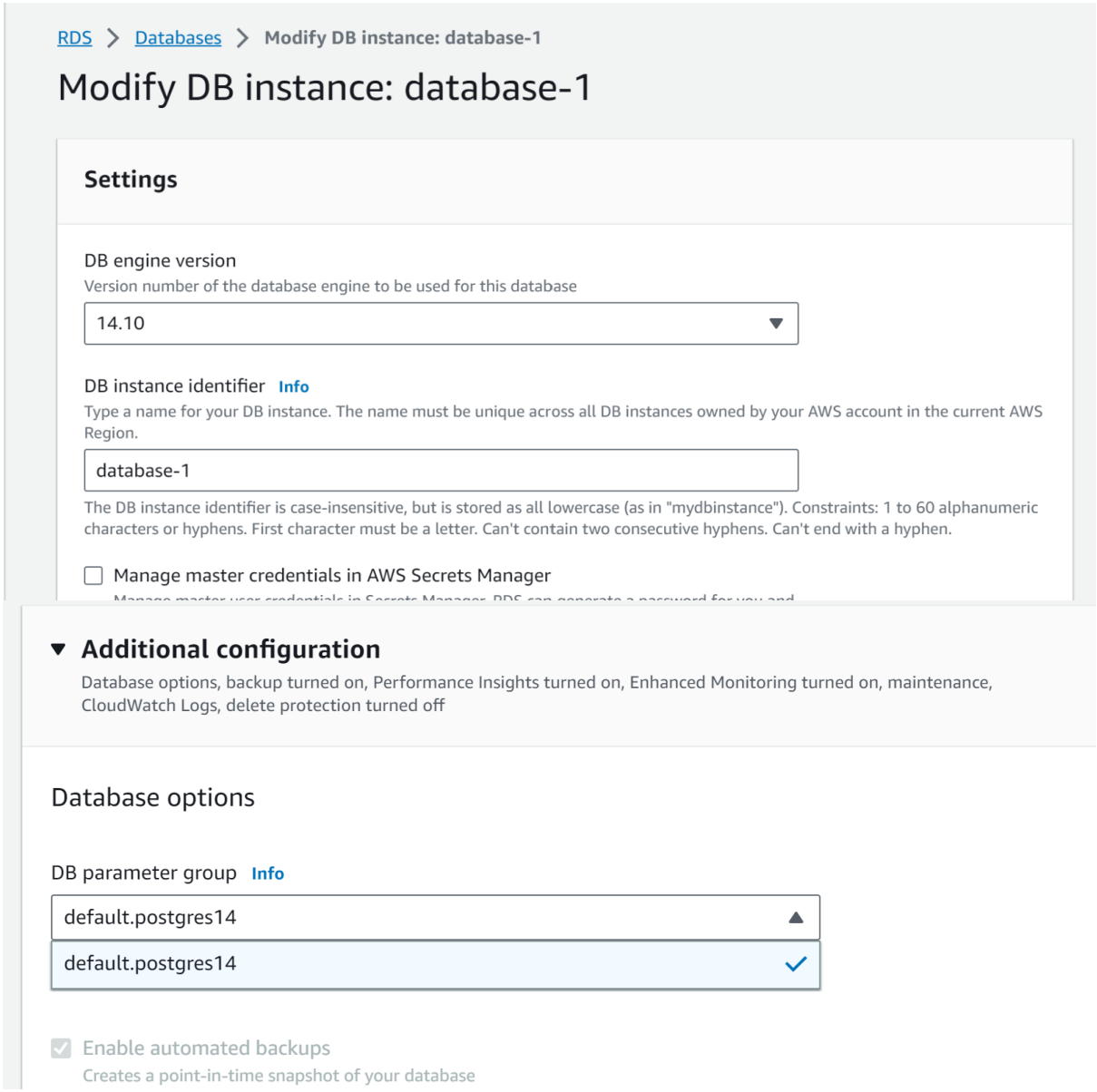




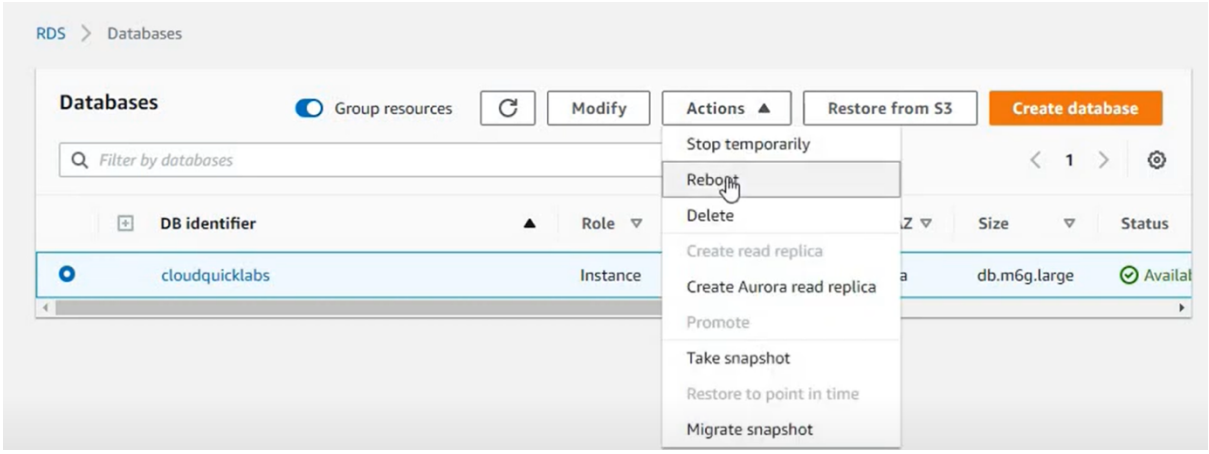
Go into recently created Parameter Group



Open the **RDS Database** & attach the recently created **Parameter Group** to it.



Reboot the RDS Database



Open pgAdmin & create the Table with SQL commands

Query | Query History

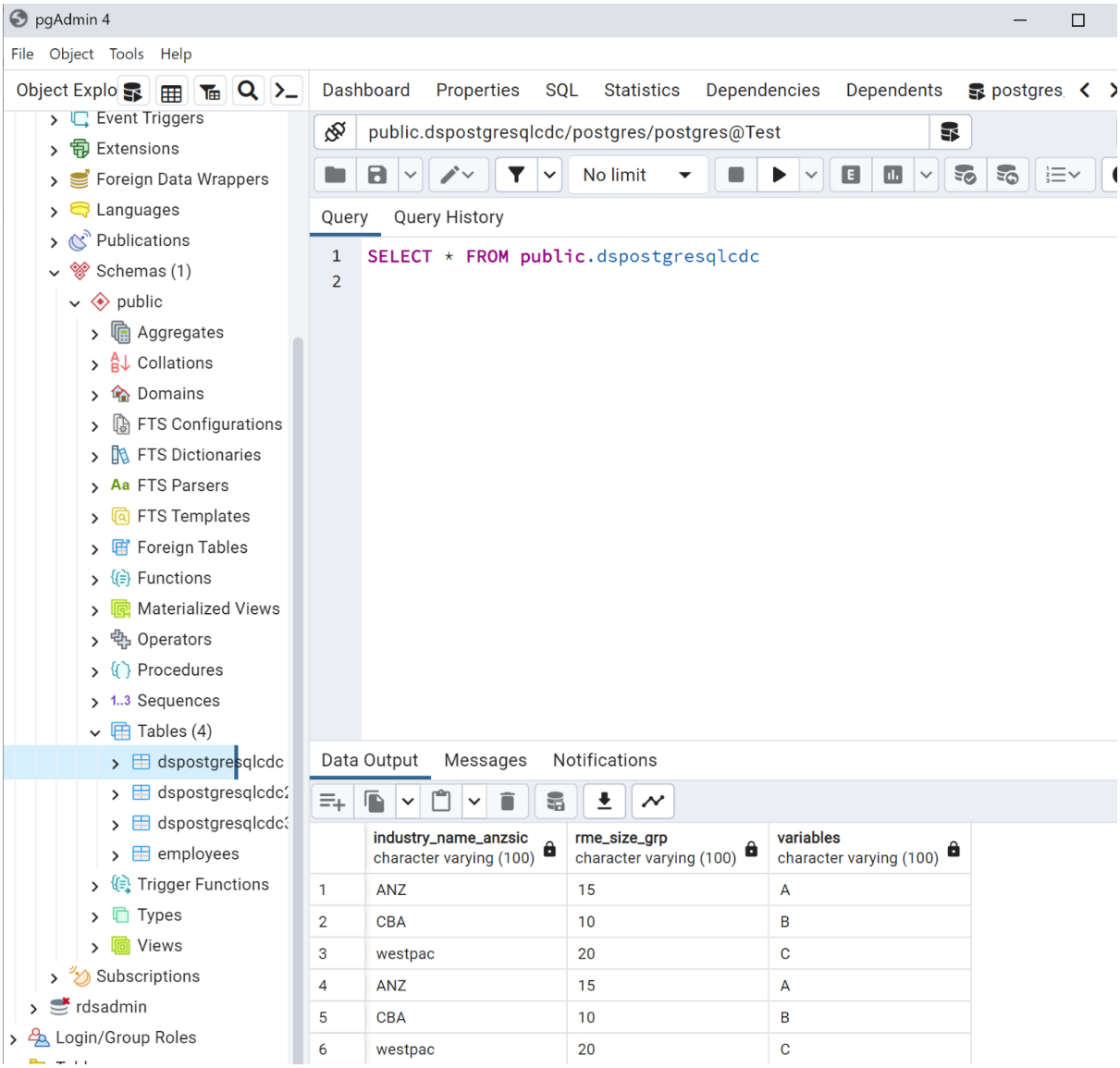
```
1  create table demo (
2    industry_name_anzsic varchar(100)    not null,
3    rme_size_grp         varchar(100)    not null,
4    variables            varchar(100)    not null
5  )
6
7
8  INSERT INTO demo (industry_name_anzsic, rme_size_grp, variables)
9  VALUES ('value1', 10, 'value3');
10
11 INSERT INTO demo (industry_name_anzsic, rme_size_grp, variables)
12 VALUES ('value1', 20, 'value3');
13
14 INSERT INTO demo (industry_name_anzsic, rme_size_grp, variables)
15 VALUES ('value1', 30, 'value3');
```

Data Output | Messages | Notifications

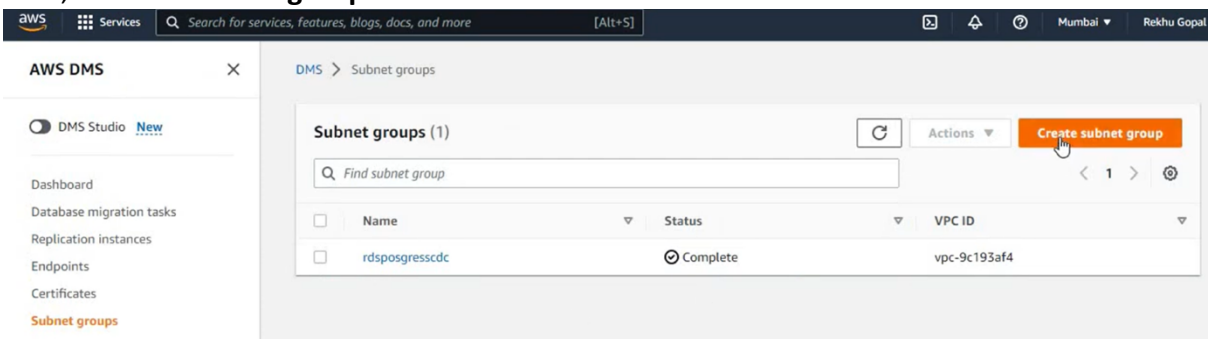
```
INSERT 0 1

Query returned successfully in 83 msec.
```

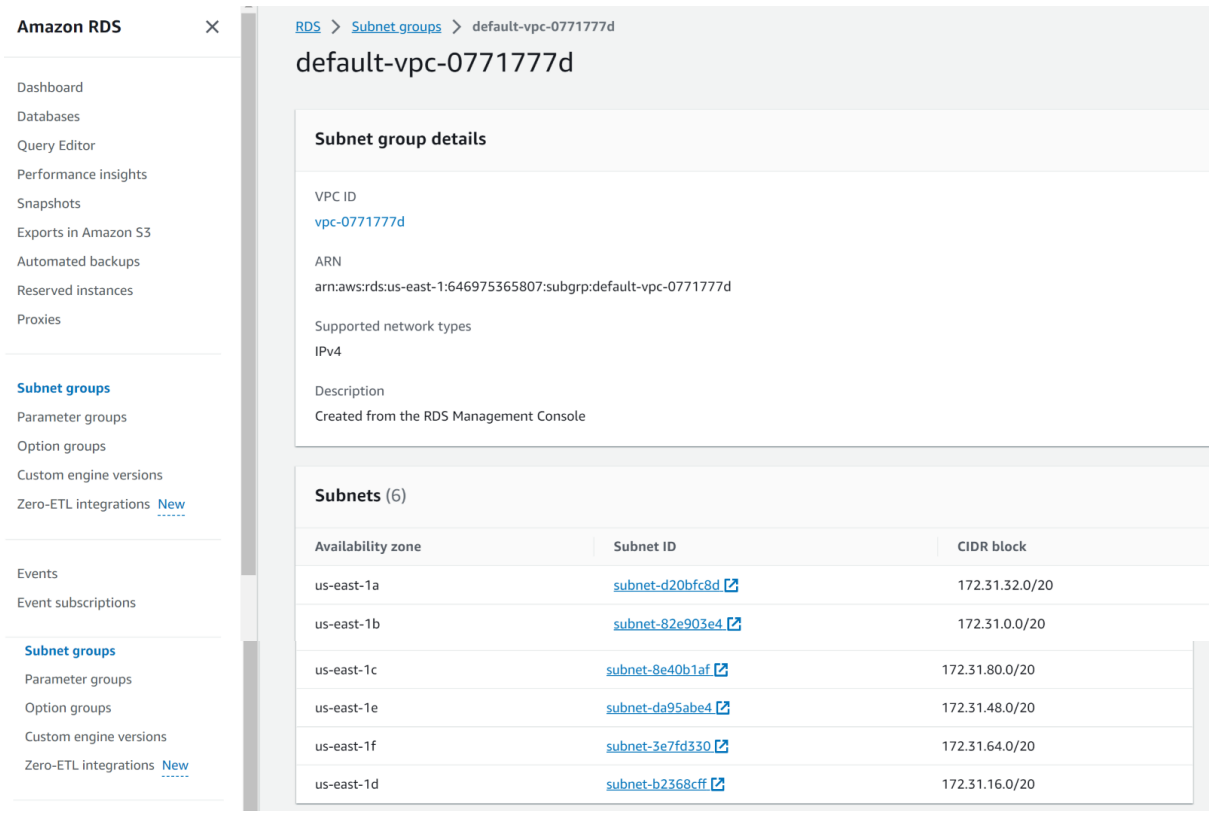
Github link:
https://github.com/RekhuGopal/PythonHacks/blob/main/AWS_RDS_Data_Migration_CDC_to_S3_Using_DMS/Create_Update.sql

Now, query the recently created tables

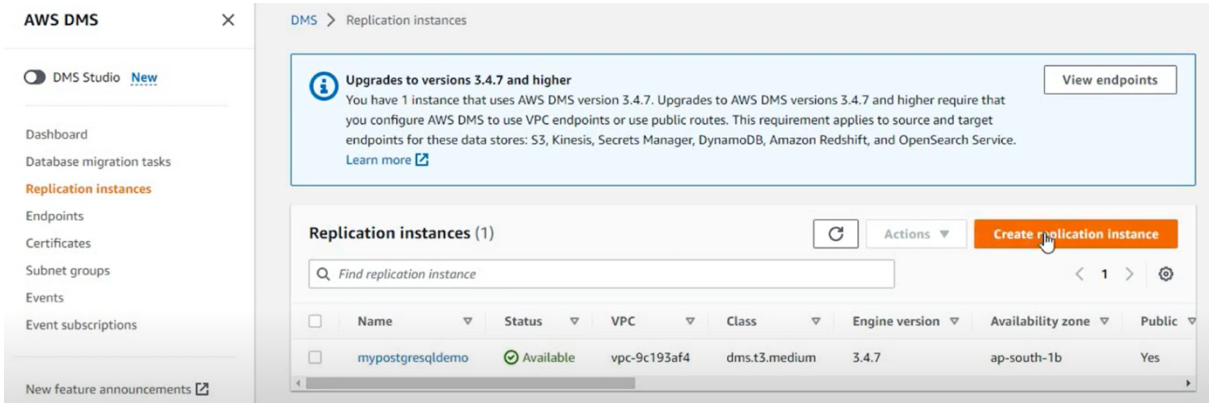


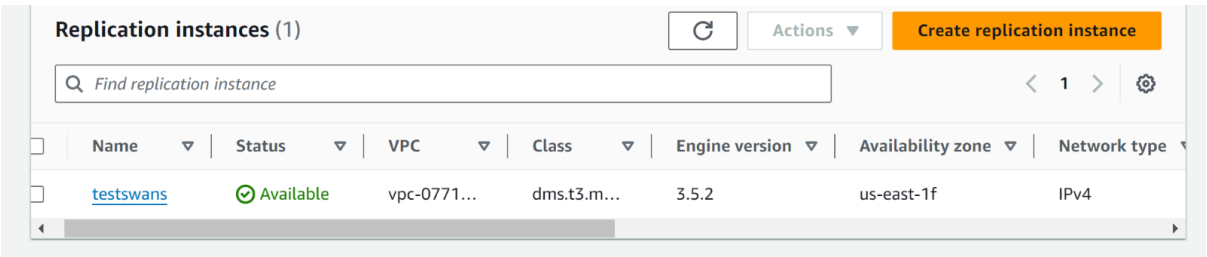
Now, create a **subnet group** with all the AZs available

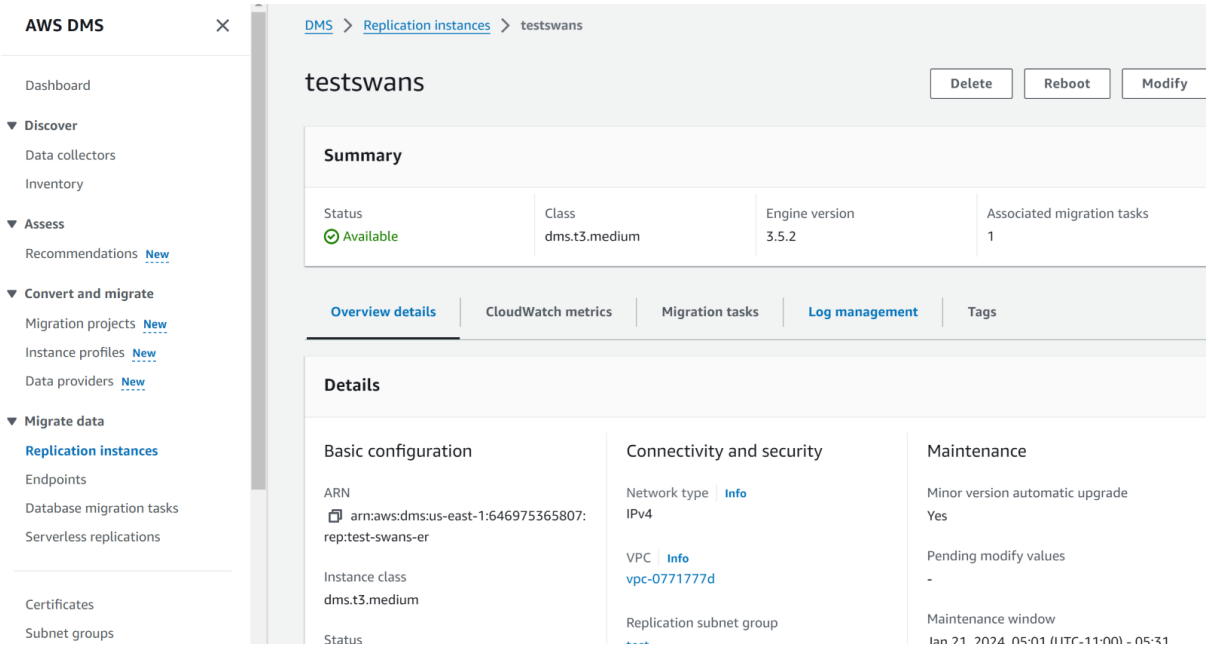




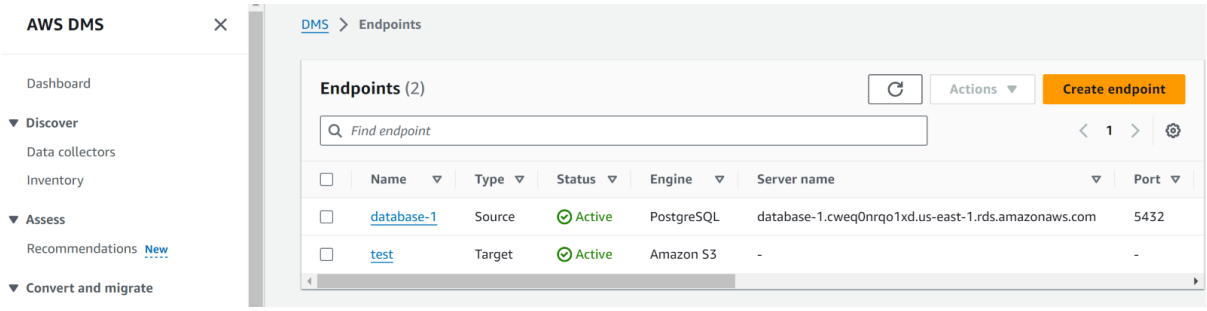
Now, create a Replication Instance

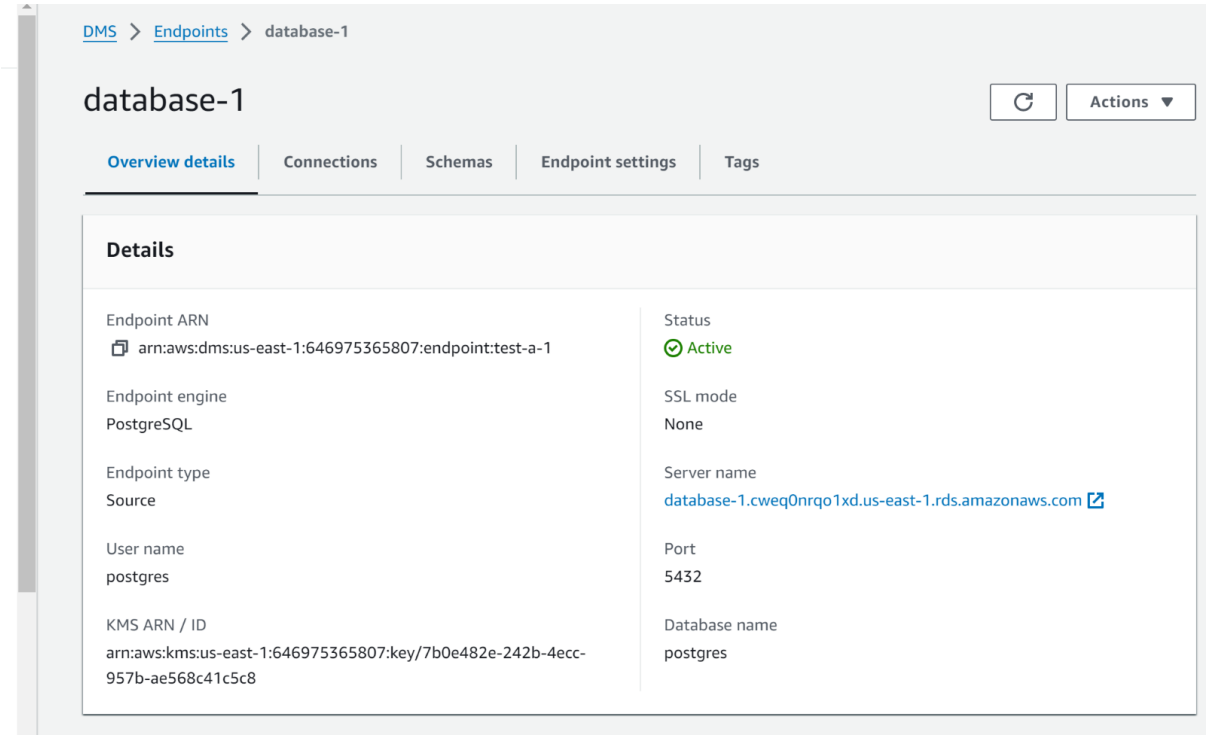


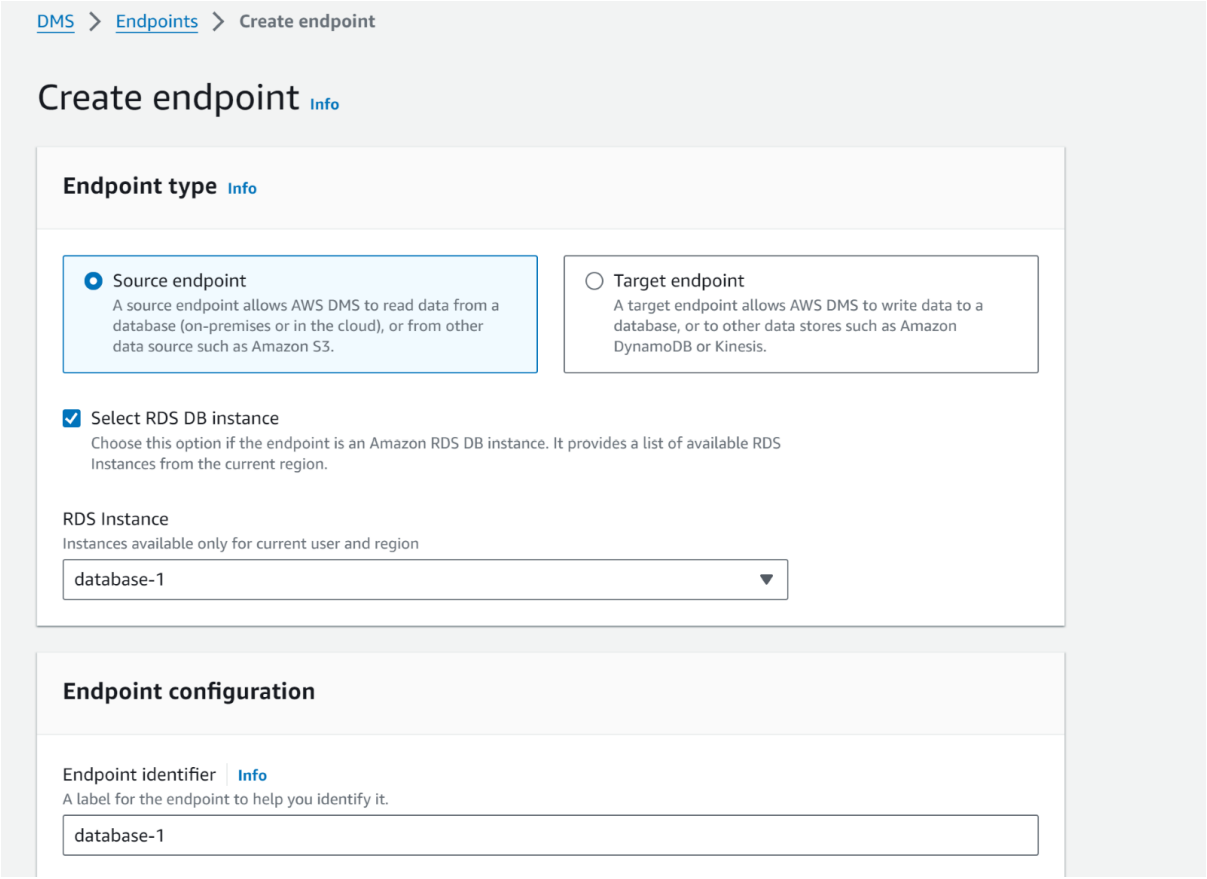




Now, create Source & Target Endpoints




DMS > Endpoints > database-1
database-1
Actions
Overview details
Connections
Schemas
Endpoint settings
Tags
Details
Endpoint ARN
arn:aws:dms:us-east-1:646975365807:endpoint:test-a-1
Status
Active
Endpoint engine
PostgreSQL
SSL mode
None
Endpoint type
Source
Server name
database-1.cweq0nrqo1xd.us-east-1.rds.amazonaws.com
User name
postgres
Port
5432
KMS ARN / ID
arn:aws:kms:us-east-1:646975365807:key/7b0e482e-242b-4ecc-957b-ae568c41c5c8
Database name
postgres


DMS > Endpoints > Create endpoint
Create endpoint Info
Endpoint type Info
Source endpoint
A source endpoint allows AWS DMS to read data from a database (on-premises or in the cloud), or from other data source such as Amazon S3.
Target endpoint
A target endpoint allows AWS DMS to write data to a database, or to other data stores such as Amazon DynamoDB or Kinesis.
Select RDS DB instance
Choose this option if the endpoint is an Amazon RDS DB instance. It provides a list of available RDS Instances from the current region.
RDS Instance
Instances available only for current user and region
database-1
Endpoint configuration
Endpoint identifier Info
A label for the endpoint to help you identify it.
database-1

Source engine

The type of database engine this endpoint is connected to. Learn more

PostgreSQL

Access to endpoint database | Info

- (•) AWS Secrets Manager
- ( ) Provide access information manually

Secret ID

Amazon Resource Name (ARN) of the secret used to manage access to your endpoint database. If you need a new secret ARN, create it from AWS Secrets Manager.

arn:aws:secret::123456789012

IAM role

IAM role that grants Amazon DMS permissions to access the specified secret (and any required KMS encryption key). For a new IAM role, create it from IAM.

arn:aws:iam::123456789012

Secure Socket Layer (SSL) mode | Info

The type of Secure Socket Layer enforcement

none

Database name

postgres

▼ Endpoint settings

DMS > Endpoints > test

test

Actions

Overview details | Connections | Endpoint settings | Tags

Details

Endpoint ARN
arn:aws:dms:us-east-1:646975365807:endpoint:NO77U2A3SKCSZ4L432PC6RGPU7USKUNG5KI2WRY

Endpoint engine
Amazon S3

Endpoint type
Target

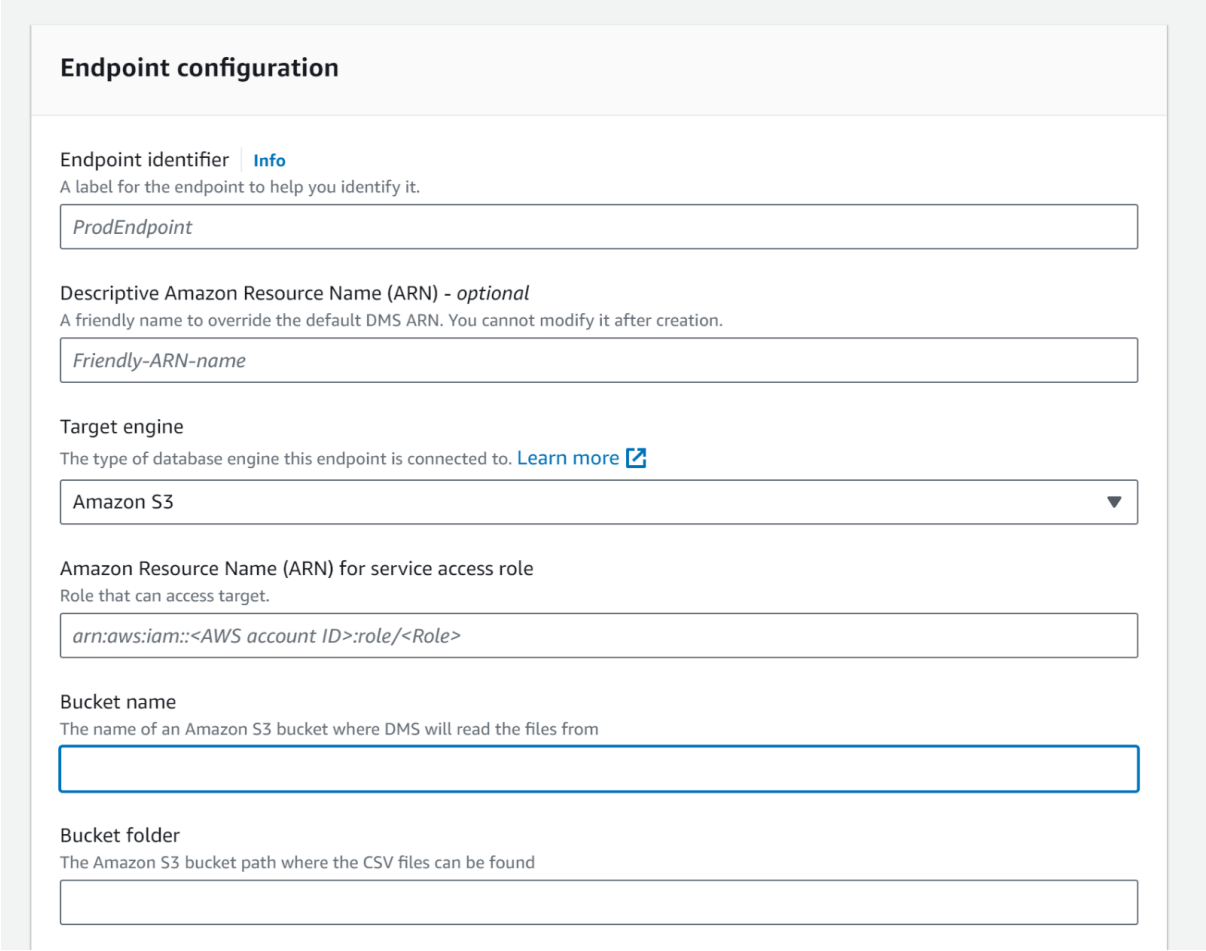
Amazon Resource Name (ARN) for service access role
arn:aws:iam::646975365807:role/DMSProtrac

Status
Active

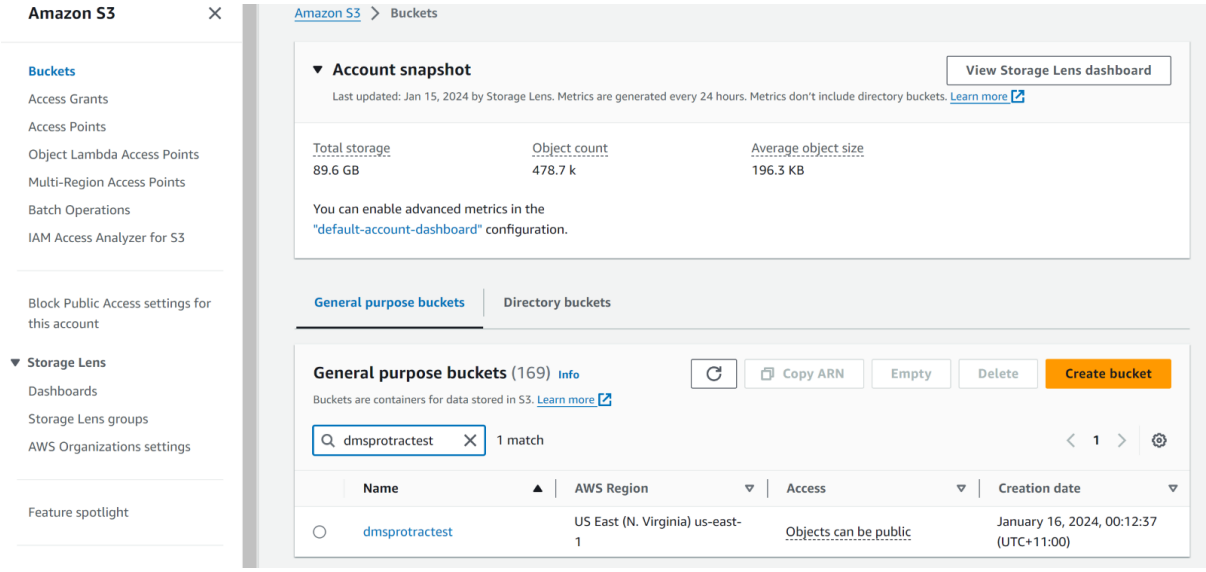
SSL mode
None

Extra connection attributes
bucketName=dmsprotractest;compressionType=NONE;csvDelimiter=,;csvRowDelimiter=\n;datePartitionEnabled=false;

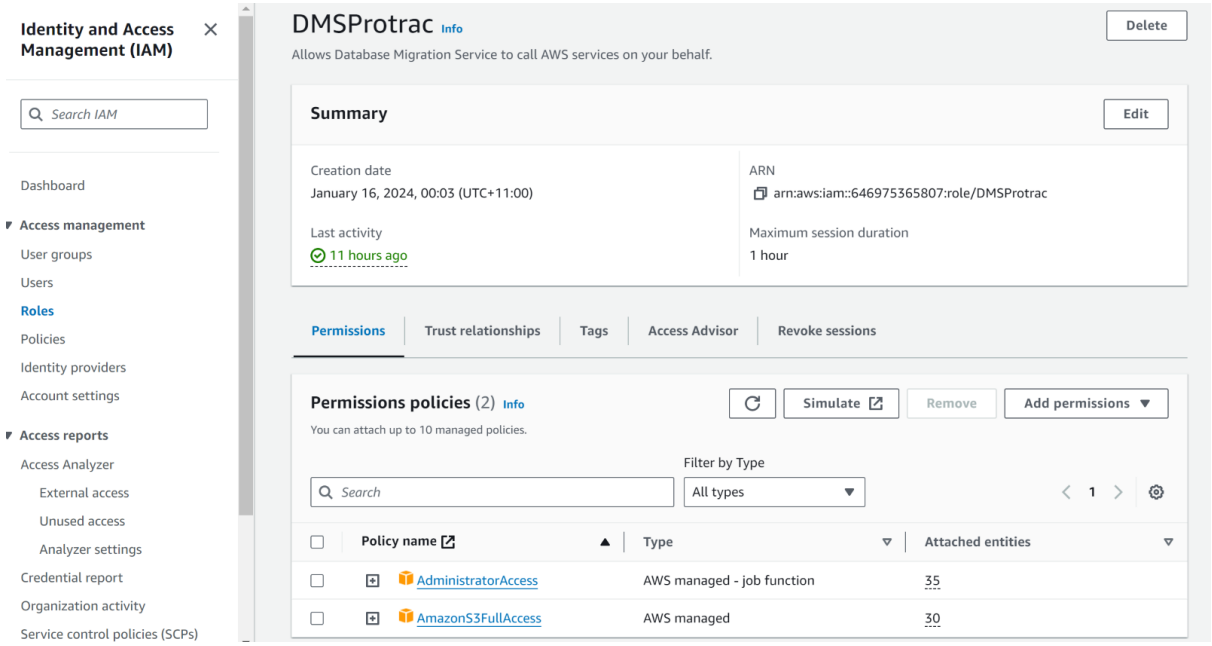
Bucket name
dmsprotractest



Copy the S3 Bucket name, that you created:

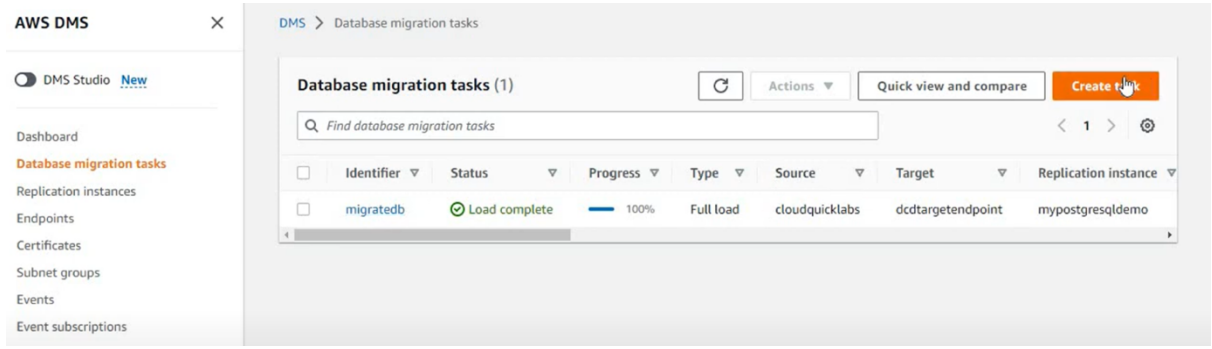


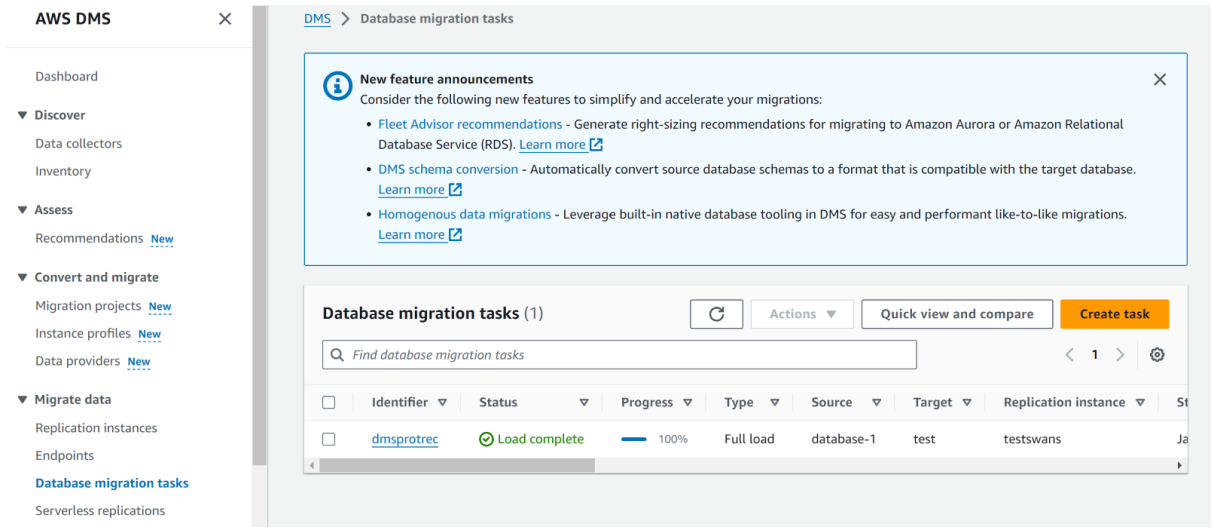
We need to create or attach an Amazon Resource Name(ARN), to create go to IAM Roles:
As we created one below

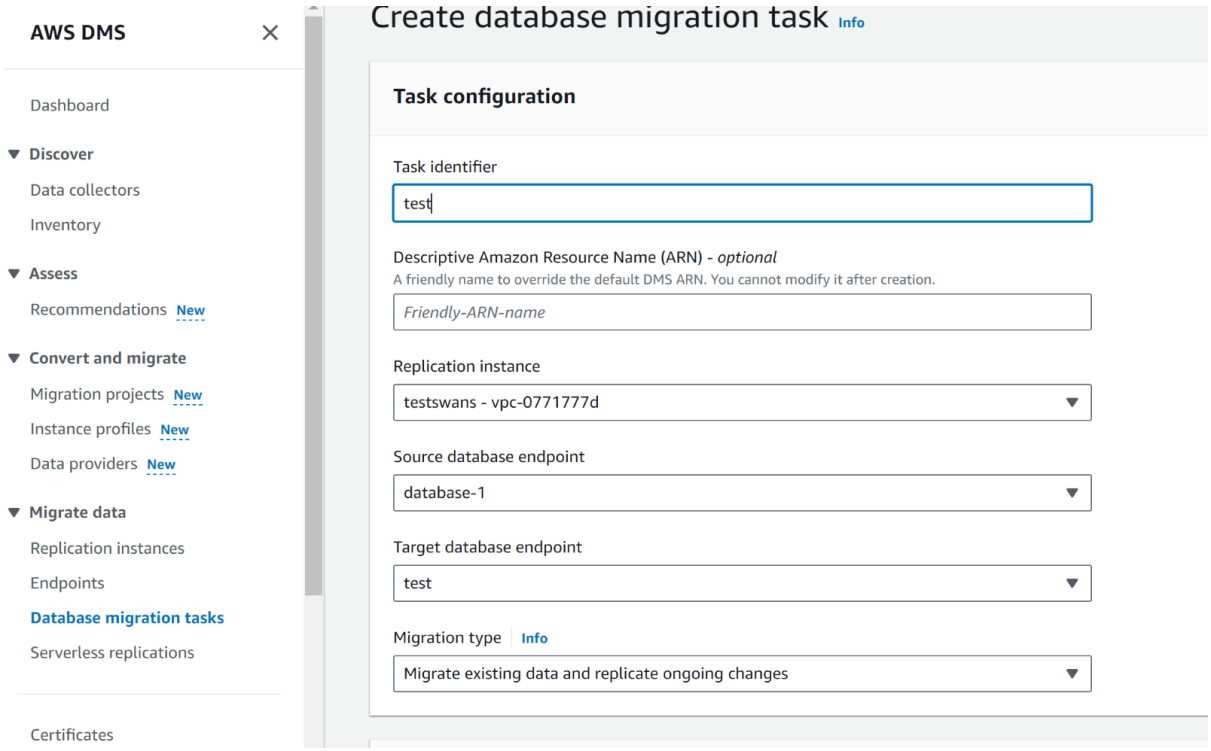


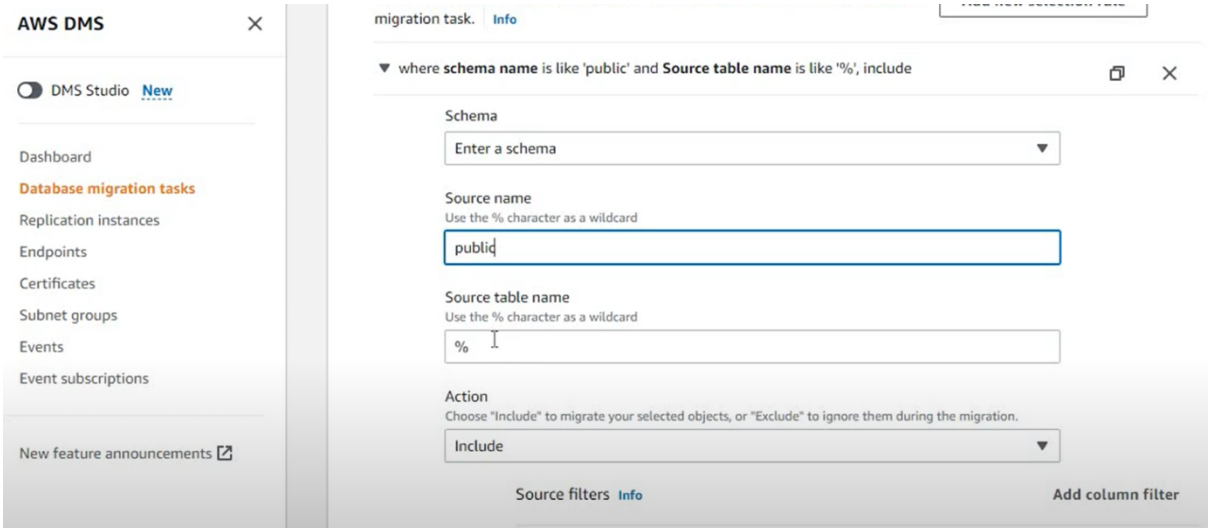
Get the ARN of this Role and paste above.

Finally, we can create our Data Migration Task:

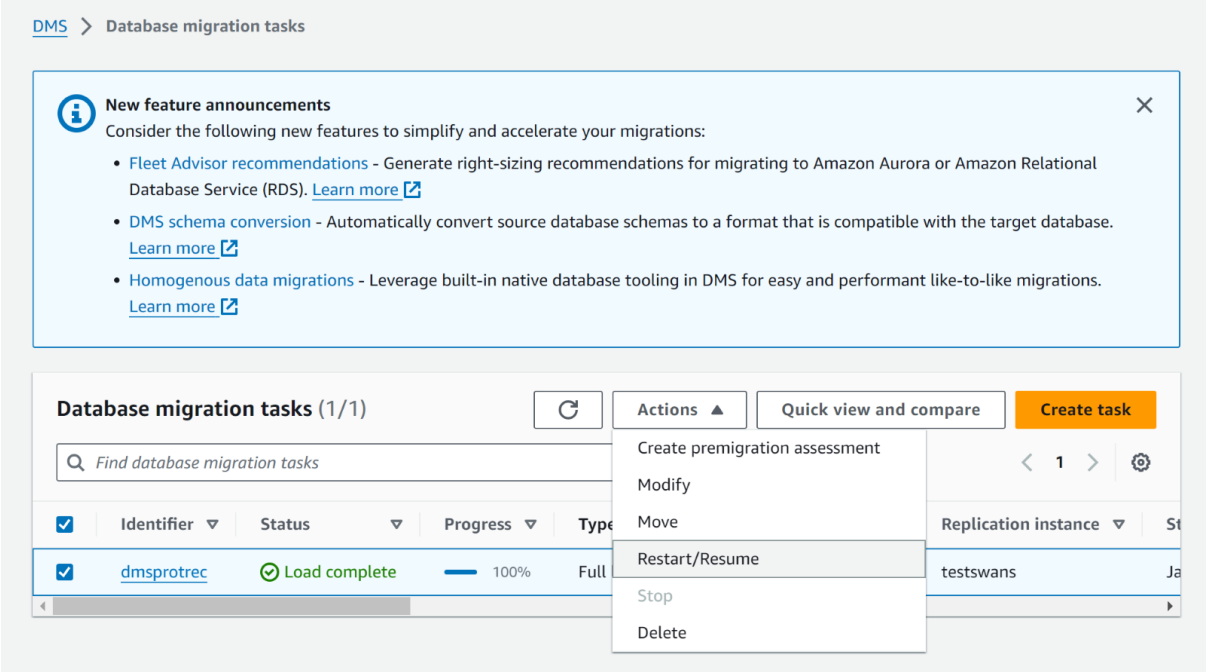


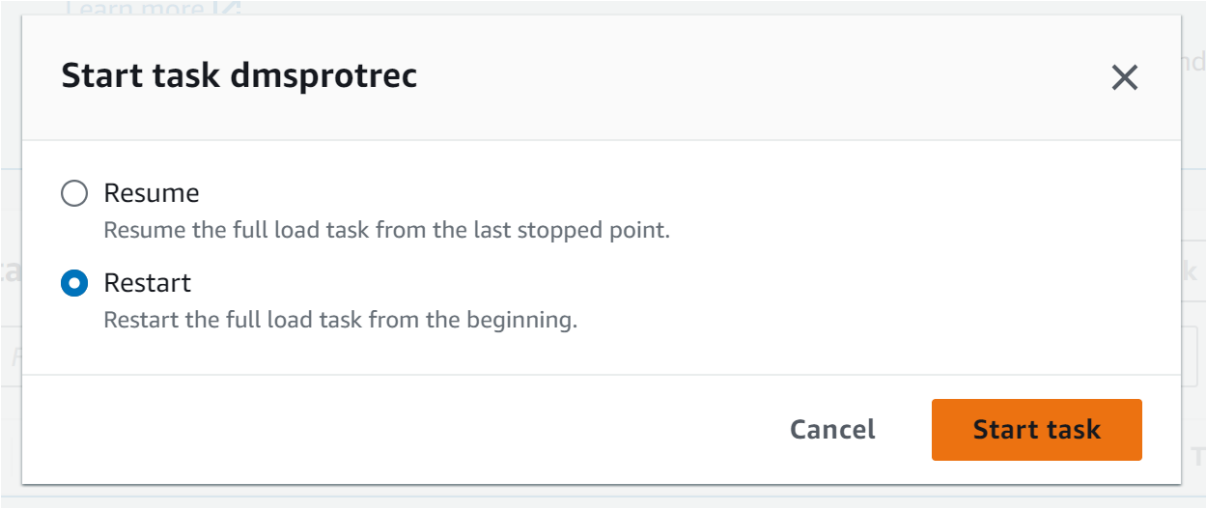


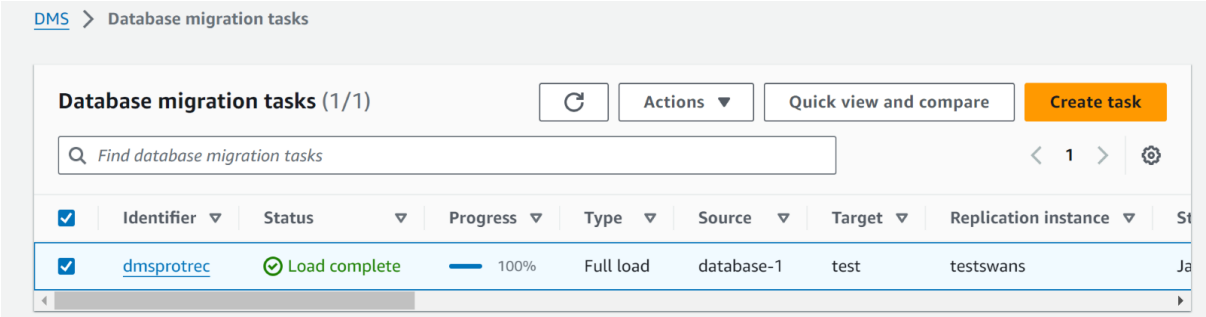




Now, Restart the Data Migration Task again

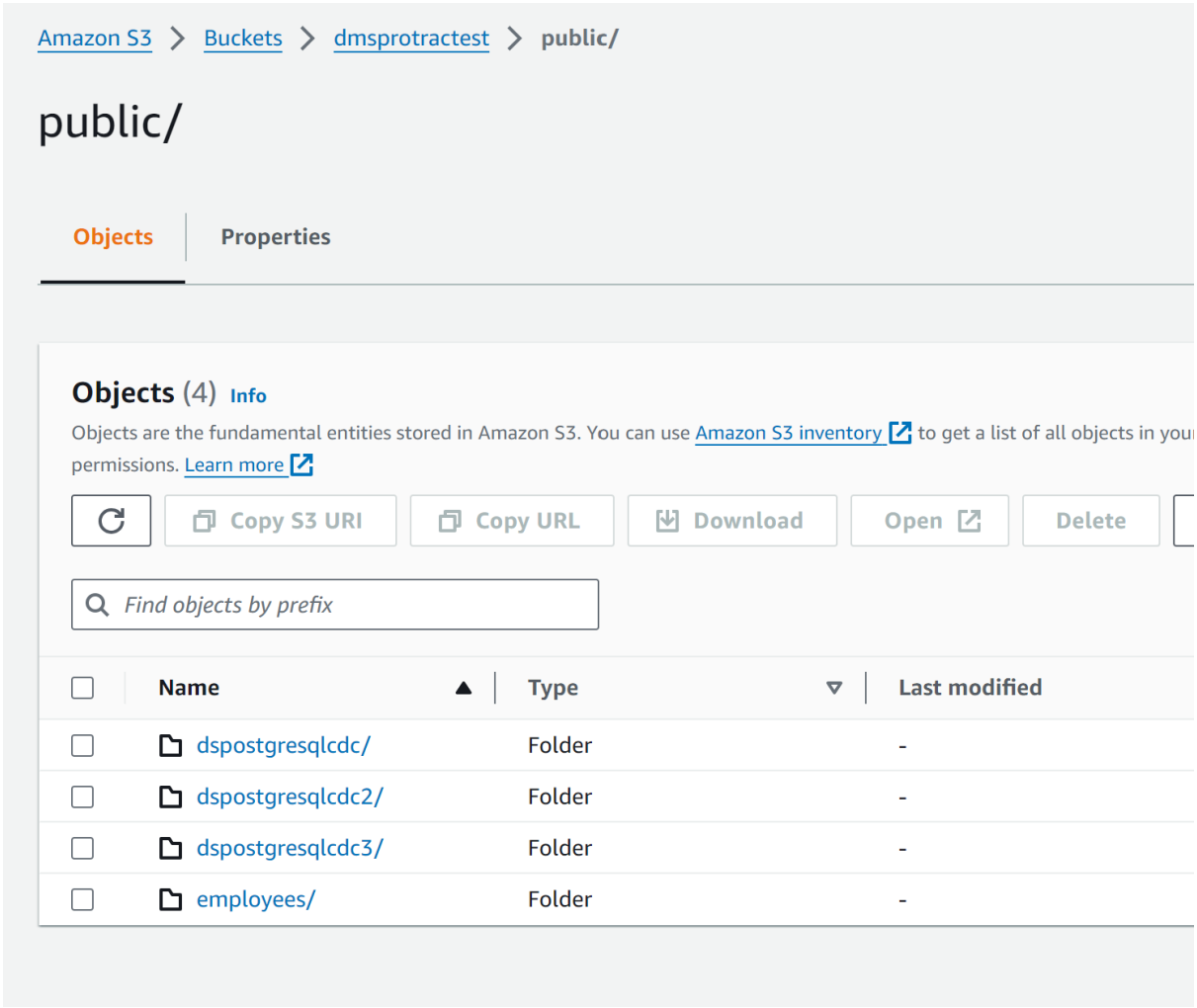




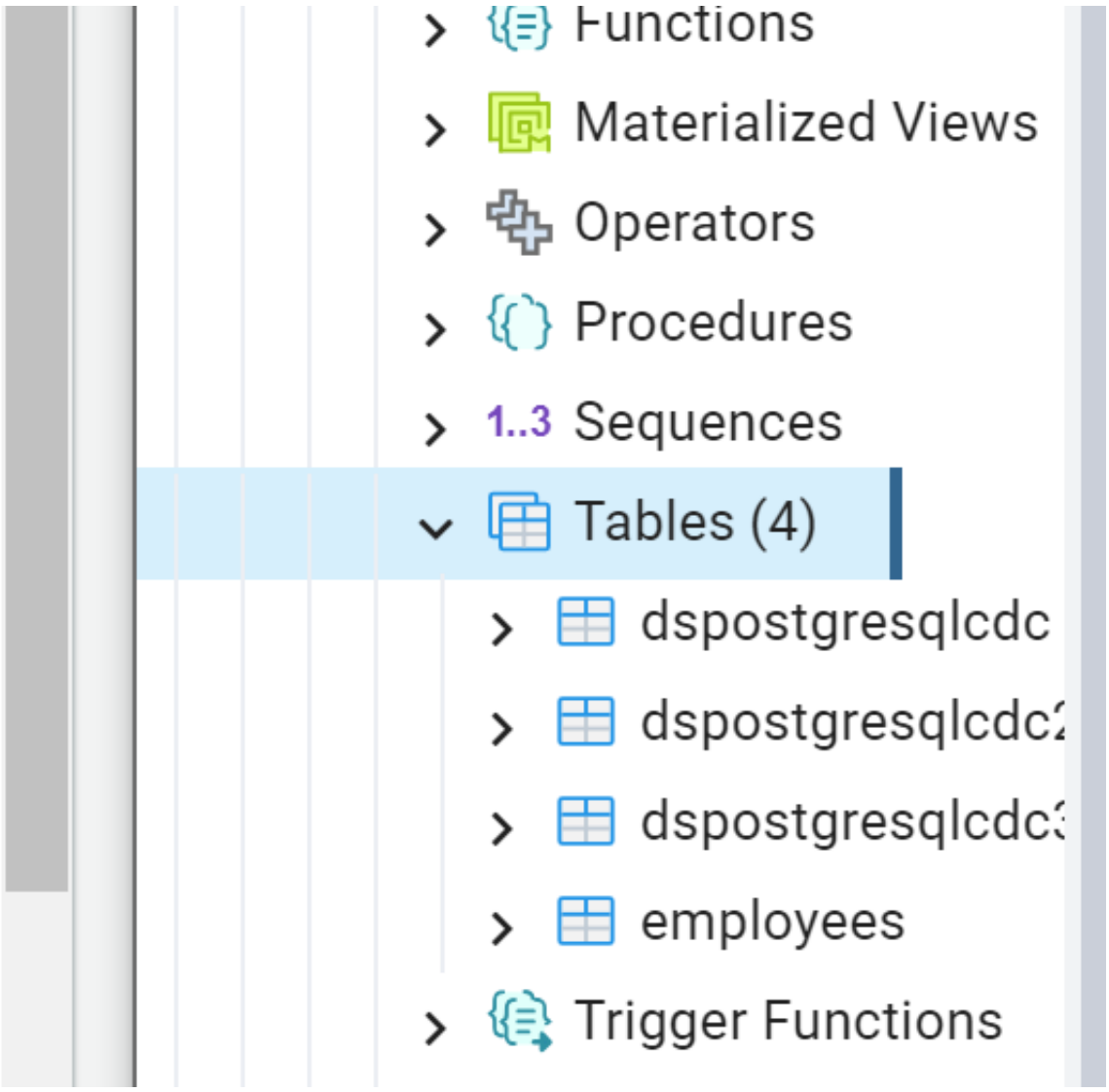


Finally, the load is completed

At the end, we can check the S3 Bucket to see if the Tables are migrated or not.

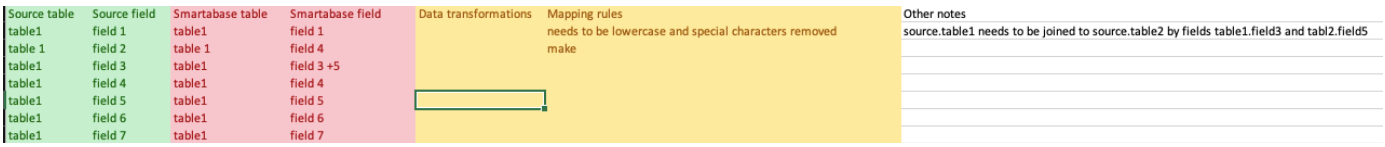


Also, cross-check with pgAdmin tables

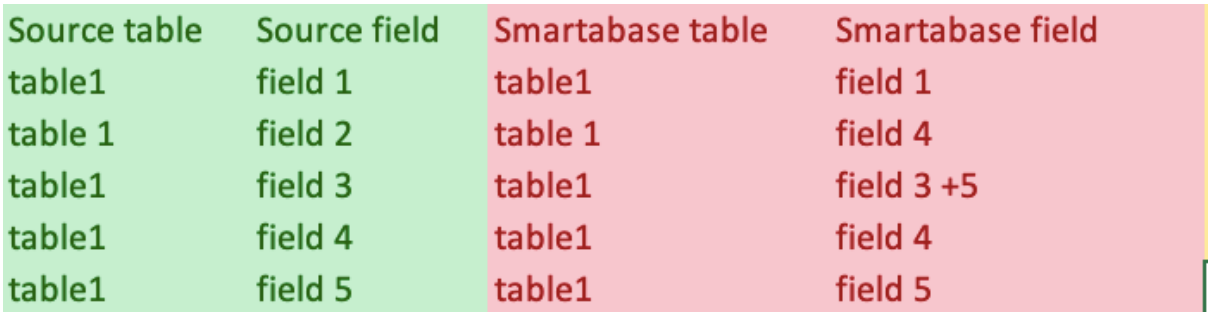


Hence, the tables have been migrated to S3 from pgAdmin.

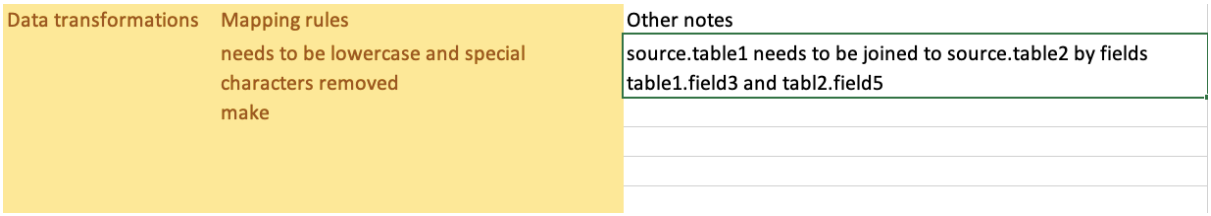
Step 4) Map the data tables and fields in Protrac to the Smartabase data structures that have been created



Source table	Source field	Smartabase table	Smartabase field	Data transformations	Mapping rules	Other notes
table1	field 1	table1	field 1		needs to be lowercase and special characters removed	source.table1 needs to be joined to source.table2 by fields table1.field3 and tabl2.field5
table 1	field 2	table 1	field 4		make	
table1	field 3	table1	field 3 +5			
table1	field 4	table1	field 4			
table1	field 5	table1	field 5			
table1	field 6	table1	field 6			
table1	field 7	table1	field 7			



Source table	Source field	Smartabase table	Smartabase field
table1	field 1	table1	field 1
table 1	field 2	table 1	field 4
table1	field 3	table1	field 3 +5
table1	field 4	table1	field 4
table1	field 5	table1	field 5



Data transformations	Mapping rules	Other notes
	needs to be lowercase and special characters removed make	source.table1 needs to be joined to source.table2 by fields table1.field3 and tabl2.field5

Step 5) Copy the Protrac data to Smartabase

To achieve the described process in a Python application, you would typically use libraries such as `pandas` for data manipulation, `boto3` for interacting with Amazon S3, and `requests` for making API calls to Smartabase. Below is a basic example of how you might structure this application:

1. Install required libraries:

```
pip install pandas openpyxl boto3 requests
```

2. Create a Python script (`mapping_application.py`):

```
import pandas as pd
import boto3
import requests
import json

# Load the mapping document (assuming it's an Excel file)
mapping_document_path = 'mapping_document.xlsx'
mapping_df = pd.read_excel(mapping_document_path)

# Function to perform data transformations
def perform_data_transformations(data):
    # Implement your data transformations logic here
    # Example: Convert date formats, clean data, etc.
    return data

# Function to perform data joins
def perform_data_joins(data1, data2):
    # Implement your data join logic here
    # Example: Use pandas merge or join functions
    merged_data = pd.merge(data1, data2, on='common_column', how='inner')
    return merged_data

# Function to fetch data from S3 Data Lake
def fetch_data_from_s3(bucket_name, object_key):
    s3 = boto3.client('s3')
    response = s3.get_object(Bucket=bucket_name, Key=object_key)
    data = pd.read_csv(response['Body'])
    return data

# Connect to the Smartabase API
def connect_to_smartabase(api_url, api_key):
    auth_url = f"{api_url}/auth/token"
    headers = {
        'Authorization': f'Bearer {api_key}',
        'Content-Type': 'application/x-www-form-urlencoded',
    }

    response = requests.post(auth_url, headers=headers)

    if response.status_code == 200:
        access_token = response.json().get('access_token')
        return access_token
    else:
```

```
        print(f"Authentication failed. Status code: {response.status_code}")
        return None

# Function to push data to Smartabase
def push_data_to_smartabase(api_url, endpoint, access_token, data):
    push_url = f"{api_url}/{endpoint}"
    headers = {
        'Authorization': f'Bearer {access_token}',
        'Content-Type': 'application/json',
    }

    response = requests.post(push_url, headers=headers, data=json.dumps(data))

    if response.status_code == 200:
        print("Data pushed to Smartabase successfully.")
    else:
        print(f"Failed to push data to Smartabase. Status code: {response.status_code}")
        print(response.text)

# Iterate through the mapping document
for index, row in mapping_df.iterrows():
    # Fetch data from S3 Data Lake
    s3_bucket = 'your_s3_bucket_name'
    s3_key = 'your_data_file.csv'
    s3_data = fetch_data_from_s3(s3_bucket, s3_key)

    # Perform data transformations and joins
    transformed_data = perform_data_transformations(s3_data)
    joined_data = perform_data_joins(transformed_data, other_data)

    # Map data to Smartabase structure (assuming there is a mapping logic)
    mapped_data = {
        'smartabase_field1': joined_data['data1_column'],
        'smartabase_field2': joined_data['data2_column'],
        # Add more fields as needed
    }

    # Connect to Smartabase API
    smartabase_api_url = "https://api.smartabase.com"
    smartabase_api_key = "your_api_key"  # Replace with your Smartabase API key
    access_token = connect_to_smartabase(smartabase_api_url, smartabase_api_key)

    if access_token:
```

```
    # Push mapped data to Smartabase
    push_data_to_smartabase(smartabase_api_url, 'your_api_endpoint',
access_token, mapped_data)
```

Replace placeholders like `'your_s3_bucket_name'`, `'your_data_file.csv'`, `'your_api_key'`, and `'your_api_endpoint'` with your actual S3 bucket name, data file in S3, Smartabase API key, and API endpoint.

This script assumes that your mapping document is stored in an Excel file (`mapping_document.xlsx`) and your data is in a CSV file in the S3 Data Lake. Customize the data transformation, data join, and mapping logic based on your specific requirements. Always ensure proper error handling and validation in a production environment.