**Configuration of Northwind sample Database from Git-Hup repository to Local machine steps Documentation**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

This is the link from GitHub from where I configured database to my local machine.

<https://github.com/microsoft/sql-server-samples.git>

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

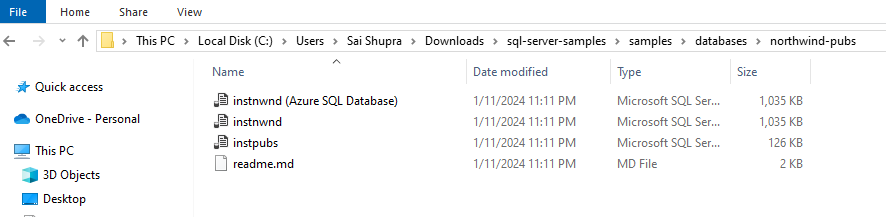
Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



Another way to fetch databases from GitHub repository is I generated one of the python script to fetch databases as below and configured in my local



SOLUTION First Approach: Using AWS Schema conversion tool.

Manually created a database named Northwind-pubs and created Tables using the scripts.

A screenshot of a computer

Description automatically generated

And created views using scripts.

A screenshot of a computer

Description automatically generated

Now my recommendation to use AWS SCT tool to transfer data from SQL server to PostgreSQL this is possible when all environment in one place I don’t have some access so unable to do it properly below are some issues I faced when doing connection, but it is executable when all environment is at one place

A screenshot of a computer

Description automatically generated

This can be eliminated by using required drivers when all compatible software used, I have some Limitation so facing issue.

**SECOND APPROACH AWS CLOUD SOLUTION:**

Architecture DIAGRAM is not created due to some restrictions.

Approach 1: Using S3 and Lambda for Incremental Data

Workflow:

SQL Server data is transferred to S3.

Lambda function triggers new data in S3.

Lambda processes the incremental data and updates the PostgreSQL database.

Pros:

Serverless architecture for cost efficiency.

Incremental processing can reduce the load on the database.

Cons:

Complexity in handling incremental data processing.

Potential delays in data propagation.

Considerations:

Ensure Lambda function scalability and error handling.

Monitor and manage S3 costs based on data storage and transfer.

Approach 2: Using AWS Glue and Athena

Workflow:

AWS Glue Crawler fetches data from SQL Server and catalogs it.

Use Athena for querying data in the Glue catalog.

Load data from Athena to Amazon RDS for PostgreSQL.

Pros:

Simplifies ETL (Extract, Transform, Load) processes using Glue.

Athena enables SQL queries directly on data in S3.

Cons:

AWS Glue may have associated costs.

Additional considerations for managing schema changes.

Considerations:

Regularly update Glue crawlers to reflect changes in SQL Server schema.

Optimize Athena queries for performance.

Common Considerations for Both Approaches:

High Availability (HA):

Replicating data to multiple availability zones (AZs) is a good practice for minimizing downtime. Ensure PostgreSQL in RDS is configured for multi-AZ deployment.

Security:

Implement appropriate security measures, such as encryption in transit and at rest, to protect sensitive data.

Monitoring and Logging:

Set up CloudWatch logs and metrics for Lambda, Glue, and RDS to monitor performance and detect issues.

Testing:

Conduct thorough testing, including performance testing and data consistency checks, before and after the cutover.

Cost Management:

Regularly review and optimize costs associated with Lambda, Glue, S3 storage, and RDS.

Documentation:

Document the entire process, including configurations, for reference and future troubleshooting.

Resources used to develop this document.

1. SSMS- from Microsoft documentation
2. PostgreSQL- EBD PostgreSQL documentation
3. AWS Schema conversion tool -AWS documentation
4. AWS cloud services -AWS documentation
5. ChatGPT – For Information structure

**Approach 1: Using S3 and Lambda for Incremental Data**

**Terraform configuration (main.tf):**

provider "aws" {

region = "your-region"

}

# S3 bucket for data transfer

resource "aws\_s3\_bucket" "data\_bucket" {

bucket = "your-bucket-name"

}

# Lambda function to process incremental data

resource "aws\_lambda\_function" "data\_processor" {

filename = "lambda\_function.zip" # Replace with your code file

source\_code\_hash = filebase64sha256("lambda\_function.zip")

handler = "your.handler"

runtime = "python3.9" # Adjust as needed

role = aws\_iam\_role.lambda\_role.arn

}

# IAM role for Lambda function

resource "aws\_iam\_role" "lambda\_role" {

assume\_role\_policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Action": "sts:AssumeRole",

"Principal": {

"Service": "lambda.amazonaws.com"

},

"Effect": "Allow",

"Sid": ""

}

]

}

EOF

}

# S3 event trigger for Lambda function

resource "aws\_s3\_bucket\_notification" "data trigger" {

bucket = aws\_s3\_bucket.data\_bucket.id

lambda function {

lambda\_function\_arn = aws\_lambda\_function.data\_processor.arn

events = ["s3:ObjectCreated:\*"]

}

}

# PostgreSQL database in RDS

resource "aws\_db\_instance" "postgres\_db" {

# ... database configuration details ...

}

**Approach 2: Using AWS Glue and Athena**

**Terraform configuration (main.tf):**

# ... S3 bucket resource from Approach 1 ...

# Glue crawler to catalog SQL Server data

resource "aws\_glue\_crawler" "data\_crawler" {

database\_name = "your-database-name"

name = "your-crawler-name"

role = aws\_iam\_role.glue\_role.arn

s3\_target {

path = "s3://your-bucket-name/data"

}

}

# IAM role for Glue crawler

resource "aws\_iam\_role" "glue\_role" {

# ... role policy for Glue access ...

}

# Athena database for querying data

resource "aws\_athena\_database" "data\_database" {

name = "your-database-name"

}

# PostgreSQL database in RDS (same as Approach 1)