Week 3 Exercises

List of all assignments for the last week together at one place. Please find all DDL and all other needed queries for the exercises in following file on GitHub: `Week3/week3_exercises.sql`

14 SQL API

This exercise is dedicated to using the SQL API. We will go through authorization and sending the API request to Snowflake. For sending the API requests we are going to use Postman.

Let's start with authorization. We will use key pair authentication and for that we need generate private & public key and then assign it to our user.

1. Let's check if our user does not have assigned some key by performing `desc user <my_username>`

We will get a list of user's parameters where one is called RSA_PUBLIC_KEY.

2. We can generate the encrypted private key with following command:

```
openssl genrsa 2048 | openssl pkcs8 -topk8 -v2 des3 -inform PEM -out rsa_key.p8
```

We need to provide our passphrase.

3. Once we have a private key, we can generate the public key:

```
openssl rsa -in rsa_key.p8 -pubout -out rsa_key.pub
```

4. Now we have to assign the public key to our user - remove the public key delimiters

```
 Alter user <username> set rsa_public_key = 'key'
```

5. We can verify that connection with the key pair authentication works in SnowSQL

```
snowsql -a <accountIdentifier> -u <username> --private-key-path <path to private key>
```

6. We have working key pair authentication so we can proceed and generate the JWT token which we also need for authorisation. Again we can use SnowSQL for that:

```
snowsql -a <account> -u <user> --private-key-path <path to private key> —generate-jwt
```

7. Now we have everything what we need for sending an API request. Our requests will have following headers:

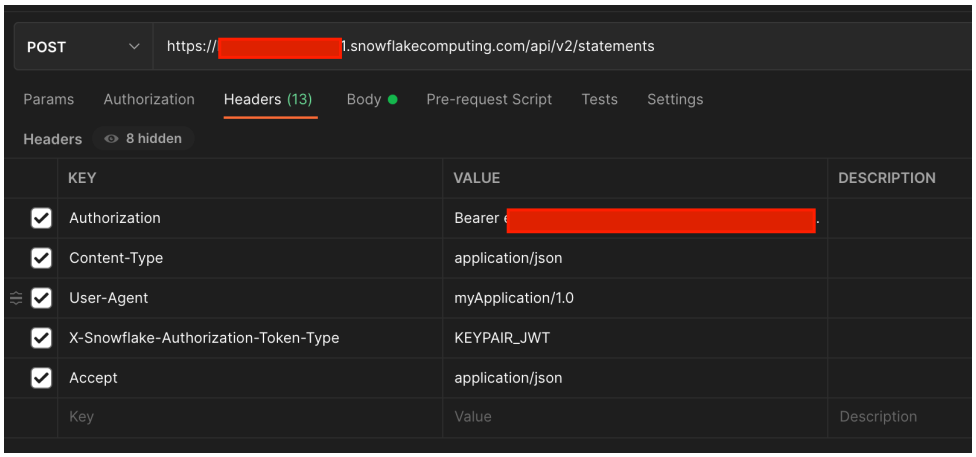
```
authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT
```

We also need to provide a request body. Let's try with simple SQL query selecting data from one of our tables:

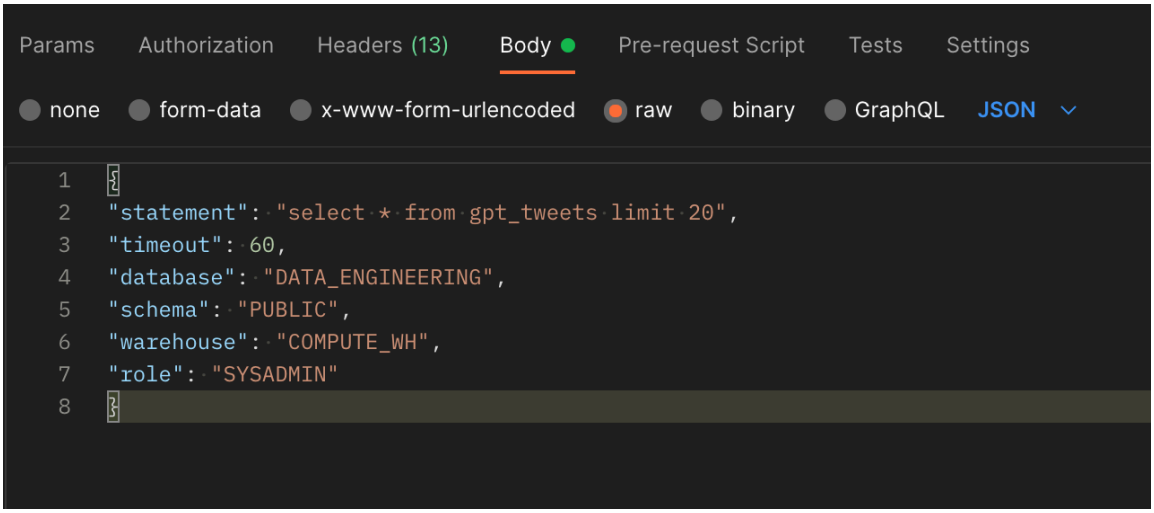
```
{
"statement": "select * from gpt_tweets limit 20",
"timeout": 60,
"database": "DATA_ENGINEERING",
"schema": "PUBLIC",
```

```
"warehouse": "COMPUTE_WH",
"role": "SYSADMIN"
}
```

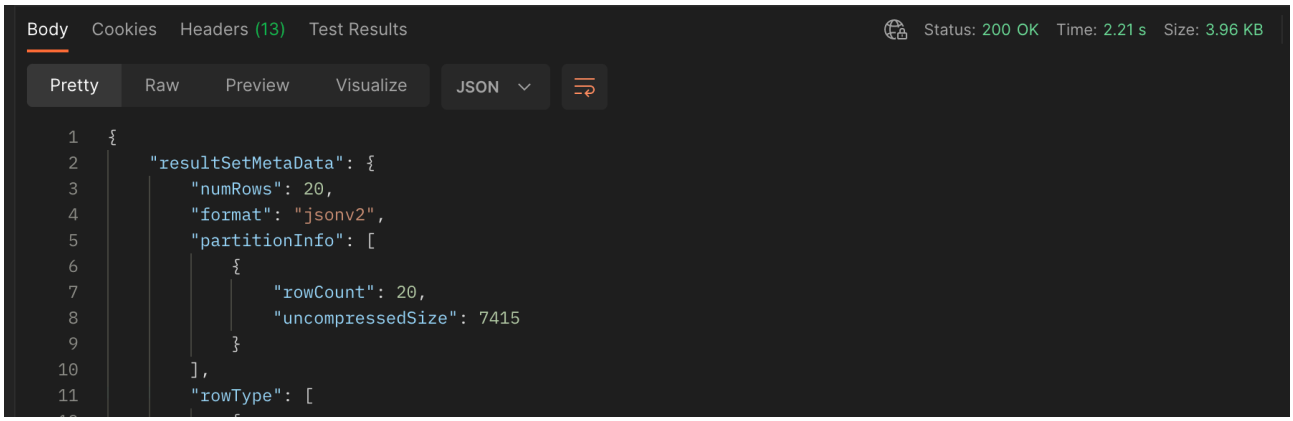
8. Let's use postman for sending the request:



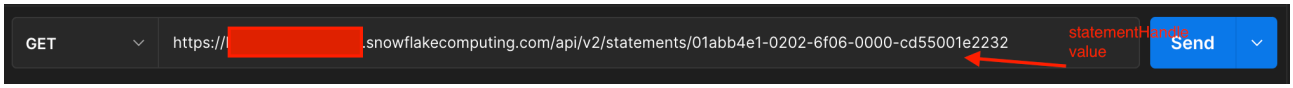
Request body specification:



And the response:



As a part of the response we also get the statementHandle value which we can use with another API endpoint to verify the status of the request - useful especially for long running queries.



16 External function

We are going to focus on creation of the external function in this exercise. We will try to leverage Amazon Translate API provided by Boto AWS SDK and translate and return German translation for input strings in English.

Let's start with creation of the lambda function in AWS which will be doing the invocation of the Translate API. But first we need to make a note of our AWS Account ID. We can find this value in My Account menu.

We can go and create a Lambda function:

- Default option:start from scratch
- Give it name `translate`
- Choose Python 3.8 as a language
- Expand "Choose or create an execution role" and select "Create a new role with basic Lambda permissions."

Open the function and deploy the code which you can find on Github in `week3/external_fnc_lambda.py` and deploy the code.

Now we need to add a IAM policy for AWS Translate service. Go to configuration of the lambda, and click on Role name in Execution role menu. This will redirect you to IAM console. Here under permission tab, click on Add permissions and attach Policies and search for **TranslateFullAccess** policy, select it and click attach.

Now we need to do an integration between Snowflake and AWS. We have to create an IAM role for that.

Let's create a new IAM role:

- Choose "AWS account" when asked to select the type of trusted entity.
- Choose another AWS account and paste your Account ID for now.
- Click next to skip the permission policies
- Name the role `SFLambdaRole and click` create role.
- Make a note of the Role ARN value. We will need it later.

arn:aws:iam::749382034063:role/SFLambdaRole

Let's move on. Now we need an API gateway for the integration. Go to API gateway console and select REST API by clicking the build button:

- Select NEW API
- Name it `TranslatePI`
- Make an Endpoint Type Regional

Click create API and we will be in the /Methods window now. From the "Actions" drop-down, select "Create Resource." Let's use `sf-proxy` as a name for this resource and click create.

Go again into the "Actions" menu and click "Create Method". Select POST from the drop down menu and click on the check button. Now you have the stup window available:

- Choose Lambda function as integration type.
- Check the Use Lambda Proxy integration
- In the Lambda function name paste the name of the function we have created before.

Click save and ok. Now we have to deploy the API. From “Actions” menu select “Deploy API”:

- Select [New Stage] in deployment stage
- Use some stage name e.g. test and click deploy

Now we are redirected to stage editor page. Under stages, expand your stage until you see “POST” under your resource name. Click on “POST” and make a note of the “INVOKE URL” field for this POST request.

Now we have to secure our API so that only your Snowflake account can access it.

In the API Gateway console, go to your API POST method and click on "Method Request.”

Inside the Method Request, click on the pencil symbol next to "Authorization" and choose "AWS_IAM." Click the checkmark just to the right to save. Then, click on "Method execution." Find the ARN under "Method Request." And make a note of that value.

Next, go to "Resource Policy" from the right panel. You'll be setting the resource policy for the API Gateway to specify who is authorized to invoke the gateway endpoint. Once there, paste in this resource policy:

Paste there this resource policy:

```
{
    "Version": "2012-10-17",
    "Statement":
    [
        {
        "Effect": "Allow",
        "Principal":
            {
            "AWS": "arn:aws:sts::<12-digit-number>:assumed-role/
<external_function_role>/snowflake"
            },
        "Action": "execute-api:Invoke",
        "Resource": “<your method request arn>/*/POST/MyResourceName”
        }
    ]
}
```

- Replace the <12-digit-number > with your Account ID
- Replace the <external_function_role> with the Role name which you have created
- Replace the “Resource” value with your Method Request ARN value.

Click save. And we need to redeploy the API as it was secured now. Go back to resources and click on Actions and Deploy API where you will select the stage you have created earlier.

Now we have lambda function, IAM role and policy + API gateway. Let’s continue in Snowflake where we have to create an API integration.

You can find the SQL statements in the `week3/exercises.sql` on GitHub. You have to replace two values in the DDL script. You have to use your IAM Role which we have created -

`SFLambdaRole` - please use the ARN of the role, not a name. Then you need API invocation URL. this will go into the `api_allowed_prefixes` field.

Once done, run the desc integration command. We have to make a note of following property values:

- API_AWS_IAM_USER_ARN
- API_AWS_EXTERNAL_ID

Like we did for storage integration. Now we have to setup the trust relationship between Snowflake and AWS for our api integration. Go to AWS console, search for IAM and our `SFLambdaRole` role.

Go to “Trust relationship” tab and click on edit trust policy button there. Find the value of “Statement”.”Principal”.”AWS”. There you have to replace the arm value with arn value from the previous step. So use there the value of API_AWS_IAM_USER_ARN from the describe integration command.

Then find the “Statement”.”Condition” field which is currently empty. Paste the following string between the curly braces:

```
"StringEquals": { "sts:ExternalId": "xxx" }
```

You have to replace the xxx with the second value from desc integration command. Put there value `from API_AWS_EXTERNAL_ID field and` click on update policy button.

Uff, we are almost ready. Now it is time to create the external function in Snowflake. Please run the DDL statement for the external function. You just need to replace two values there again:

- Replace the `< api_integration_name >` with the name of your API integration
- Replace the `< invocation_url >` value with your resource invocation URL.

And we are finally done! Let’s try our external function with passing some English string and get the translation as a response.

17 Snowpark transformations

We are going to practise basic data frame transformations related to Snowpark for Python. We will test out the new Python worksheets which are in public preview to find out its limitations and then we will continue with local Python environment and doing the development in Jupyter notebook.

Please find following files with source code on Github:

- `Week3/python_worksheet.txt` - Python code for the Python worksheet in Snowsight.
- `Week3/snowpark_transformations.ipynb` - Jupyter notebook with Snowpark code which you should run from your local

Please create a new Python worksheet in Snowsight and paste the content of `python_worksheet.txt` file into that. Then you can experiment with the code and run it.

Please download the Jupyter Notebook and run it inside your local Python environment. You can go through it statement by statement to find out how the transformations are built.

In the end, there is a task to create one more Snowpark transformation by using the Snowpark API. **You should try to find how long is the shortest and longest tweets.**

18 Deploying Python UDFs

This exercise is dedicated to deploying Python UDFs which are being developed locally to Snowflake. We are going to try deploy the function manually first and the we will introduce a new community based tool SnowCLI which can make the work easier.

1. Let's start with creation of UDF via Snowpark session

We need to create an internal stage which will be holding our UDF Python code. You can use following DDL command to create a stage:

```
create or replace stage udf_stage;
```

Please activate your local Python environment. You can download the from `week3/18_udf_deploy_man.py` from GitHub with source code for this exercise. If you prefer the Jupiter notebooks, you can download this one: `week3/18_udf_deploy_man.ipynb`

You can go through code and try to deploy the UDF into Snowflake via Snowpark and then call it.

2. In the second part of the exercise we will try to use SnowCLI for creation and deployment of the UDFs.

1. Let's install the snowcli into our python environment:
```
pip install snowflake-cli-labs
```

2. Then you should be able to get a help for the cli by using command: `snow —help`
You can see what everything is possible to do with snowcli.

3. Let's create a new folder where we will place our new udf:
```
mkdir my_udf
cd my_udf
```

4. Now we can call snowcli command `snow function init`

It will populate our directory with bunch of files where one of it is `app.py` which will be the handler for our function.

5. You can try to run the function locally by running python `app.py`

6. Let's try to install also some packages. Open the requirements.txt file and put there `pandas`.

7. Then install it by running `pip install -r requirements.txt`

8. Now we can import pandas into `app.py`

9. Let's update our `app.py` to create a simple data frame and return it back

```
import pandas as pd

def hello() -> dict:
    df = pd.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
    return df.to_dict()
```

10. We have to change the return type of the function in `app.toml` file from `string` to `variant`

```
[default]
input_parameters = "()"
return_type = "variant"
name = "helloFunction"
handler = "app.hello"
file = "app.zip"
```

11. We have tested it out locally and now we want to publish it in Snowflake. So first we have to authenticate ourselves. Snowcli takes the connection details from SnowSQL which we have already configured. The config file should be in `~/.snowsql/config`

12. Run the `snow login` command
    - Confirm the path for config file
    - Put the connection name you want to use

13. Now we are going to do a deployment. Run the command:
    `snow function create`
    `SnowCLI` will create an app.zip file, upload it to the stage and create the UDF in Snowflake.

14. Once the functions is deployed we can test it via SnowCLI with command:
    `snow function execute -f "helloFunction()"`

15. You can again check the stages in snowflake. You will find out that there is a new stage called Deployment. It holds the zip file with the function code. As a last command you can also try to run the UDF in Snowsight:
    - `Show stages;`
    - `Ls @DEPLOYMENTS;`
    - `Select helloFunction();`

19 Governance features

This exercise is dedicated to governance features. We will try to use classification functions same as create and assign a row access policy to the table.

Classification features

1. Let's use our snowpipe_landing table for testing the classification function. Use that table as a function parameter. You can see what kind of semantic and privacy categories have been discovered in the table

2. You can also check the system function to find out what are possible values:
   `system$get_tag_allowed_values('snowflake.core.semantic_category')`
   `system$get_tag_allowed_values('snowflake.core.privacy_category')`

3. Now we can automatically associate those proposed categories to tags by calling a stored procedure ASSOCIATE_SEMANTIC_CATEGORIES(). It accepts as input parameter the output of the previous command.

4. It will automatically assign the tag_name and tag_value to attributes in the table. You can check out those in Snowflake.account_usage.tag_references view. But be aware of 2 hours latency of that system view. Meaning that data won't be visible right after the stored procedure finishes.

Row access policies

Let's try to create a row access policy and protect the data in one table. We will use the mapping table with sales region. Based on user role and region the data from the table will be filtered out. You can find all the queries in GitHub in the file dedicated to this week.

We are going to use the schema owner role which is `sysadmin` in our case.

1. Let's create a mapping table called manager_region with two columns:
    - manager_role will be varchar
    - Region wil be varchar

2. We will populate the table with 3 regions

```
insert into manager_region values ('SALES_EU', 'EU');
insert into manager_region values ('SALES_APAC', 'APAC');
insert into manager_region values ('SALES_US', 'US');
```

3. Let's prepare the data - we are going to use our snowpipe_landing table as a source and we will create an empty table with same structure

4. We will add two columns in the table which is region and revenue to simulate we have various users per region with different revenue values.

5. Now we will populate the table with content of snowpipe_landing table and we will randomly select one of the regions + generate some random number as a revenue.

6. Now we need more roles to test the policy and simulate we are users from different regions. Let's create roles SALES_EXECUTIVE and SALES_EU and grant them needed privileges in order to be able to query the table and use our warehouse.

7. We will grant those roles to our user in order to be able to use those roles and test it

8. Now we will create a row access policy

9. Attach it to the table

10. Use those created roles and test the policy

20 CI/CD

This exercise will be dedicated to building up the CI/CD pipeline for snowflake objects by using GitHub Actions and schema change tool.

Let's start with preparations of all needed objects which includes
- New python virtual environment
- New GitHub repository
- Some Db objects to manage
- Github action to process the changes

1. Let's start with creating a new GitHub repository on GitHub.com use. My repository will be called `olt_sf_cicd_demo`
2. We are going to create a new python virtual environment. I am using miniconda so I will create a new one with command: `conda create —n cicd_demo`
3. Activate it `conda activate cicd_demo` and open the root directory `cd cicd_demo`
4. `Let's clone our remote repository`
    - `Git clone` https://github.com/USERNAME/REPOSITORY.git
5. Now we need to have some db objects for deploying. Let's create one table, one stage and one file format. We will but everything into separate directories:
    - `db_objects/tables/V1.1__users.sql`

- db_objects/file_formats/V1.2__file_formats.sql
- db_objects/stages/V1.3__stages.sql

You can find all the scripts in repository dedicated to this ccourse: `week3/exercises/cicd/`

6. When we have db objects ready we also need to have our GitHub Actions workflow. You can find it again in `week3/cicd/.github`

It needs to be placed under directory called `.github/workflows/`

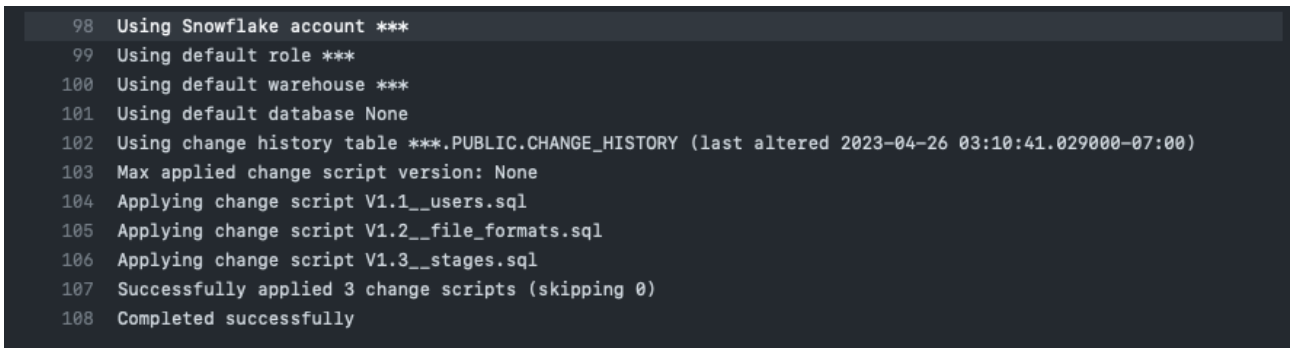
7. You could notice some variables in the workflow script. We need to create those variables as GitHub secrets. Please use details for your Snowflake account:
   - SNOWFLAKE_ACCOUNT
   - SNOWFLAKE_USER
   - SNOWFLAKE_ROLE
   - SNOWFLAKE_PASSWORD
   - SNOWFLAKE_WH
   - SNOWFLAKE_DB
   - SCHEMACHANGE_VAR: use a following structure
     {"target_db":"DB_NAME","env":"ENV_NAME"}
     - As env value use DEV

8. We need to create the CHANGE_HISTORY table which schema change will use for tracking the changes

```
CREATE TABLE IF NOT EXISTS CHANGE_HISTORY
(
    VERSION VARCHAR
   ,DESCRIPTION VARCHAR
   ,SCRIPT VARCHAR
   ,SCRIPT_TYPE VARCHAR
   ,CHECKSUM VARCHAR
   ,EXECUTION_TIME NUMBER
   ,STATUS VARCHAR
   ,INSTALLED_BY VARCHAR
   ,INSTALLED_ON TIMESTAMP_LTZ
)
```

9. Now we should have everything ready and we can try the whole process. So let's push the changes into git repository. That action will trigger our GitHub action and we should be able to see the progress in Actions tab.
   - Git add -A
   - Git commit -m "Initial commit"
   - Git push

You can see in GitHub Actions log what scripts have been applied to DB.



10. Now we can also check in Snowflake the newly created objects and the content of the CHANGE_HISTORY table.

11. Let's try to do some changes. We can test the repeatable scripts which are deployed every time the schema change run - if there is a change in the file. So let's create a view directory and new script inside for the view definition: `/views/R__view_users.sql` with simple view which selects only last_name and email from users table:

```
-- Set the database and schema context
USE DATABASE {{ target_db }};
USE SCHEMA PUBLIC;

CREATE OR REPLACE view users_view
as select
LAST_NAME,
EMAIL
from users;
```

12. As another change we are going to alter the users table to add the phone_number columns. We are going to create a new script with alter command: V1.4__alter_users.sql

You can put into the script the following code:

```
-- Set the database and schema context
USE DATABASE {{ target_db }};
USE SCHEMA PUBLIC;

alter table users add column phone_number varchar;
```

13. Let's add the changes in git again:

- `Git add -A`
- `Git commit -m "Changes"`
- `Git push`