

# Week 2 Exercises

List of all assignments for the second week together. Please find all DDL and all other needed queries for the exercises in following file on GitHub: [Week2/week2\\_exercises.sql](#)

## 07 Continuous data pipelines

We are going to practise the continuous data ingestion into Snowflake in this exercise. We will use snowpipe feature for this. It again requires configuration on the Cloud provider which is similar to settings which we have done for the batch data ingestion. There are required the bucket access permissions and IAM role in order to access Snowflake. Now we also need some kind of notification to be sent to snowpipe - when a new file will be landed in cloud storage. Let's deep dive into the setup:

### 1. Secure access to cloud storage configuration

This we already have in place. There are needed the following S3 bucket permissions

- s3:GetBucketLocation
- s3:GetObject
- s3:GetObjectVersion
- s3:ListBucket

We have bundled those into IAM policy and assign that policy to IAM role. Let's check in AWS console.

Once we have the IAM policy and IAM role we need to have storage integration on Snowflake side referencing this IAM role. This we have already done as well. Let's check our storage integration:

```
desc integration my_s3_access;
```

There we have the values `STORAGE_AWS_IAM_USER_ARN` and `STORAGE_AWS_EXTERNAL_ID` which we need for defining the trust relationship between Snowflake and AWS -> this we have done already.

### 2. Now let's move into defining the S3 Event Notification configuration

We will use the Amazon SQS for that. On Snowflake side we also need a stage. Let's reuse what we have defined for the batch processing - `my_s3_stage`.

We also need a landing table where we are going to ingest the data through snowpipe. You can find DDL script for a table in the file with all other queries for this exercise: [week2/week2\\_exercises.sql](#)

Now we can create a snowpipe which will be referencing our stage and copying data into our landing table:

```
create or replace pipe mypipe
  auto_ingest=true as
  copy into data_engineering.public.snowpipe_landing
  from @data_engineering.public.my_s3_stage/snowpipe/
  file_format = (type = 'CSV'
                 SKIP_HEADER = 1
                 FIELD_OPTIONALLY_ENCLOSED_BY='');;
```

Let's check the status of the pipe. If it is running. We can do that with following command:

```
select system$pipe_status('mypipe');
```

You can notice additional subdirectory after the stage name. Let's place the files for the snowpipe into separated directory to keep them separated from others.

Now, when we have the snowpipe object defined we can proceed with event notification configuration. We have to find out the SQS queue which Snowflake has assigned to our pipe.

Let's run the `show pipes` command and make a note of the ARN of the SQS queue. You can find it in `notification_channel`.

Now we need to configure that channel on S3 side:

- Let's go into the bucket properties, find an event notifications section and create event notification.
- We select all object create events in OBJECT creation section
- Scroll down to destination section where we select SQS queue and specify the SQS queue ARN which we got from show pipes command.

And that's it. We should have all the needed configuration in place. Let's try some data ingestion. Please download the sample data from GitHub. You can find them here:

- `Week1/source_files/snowpipe/MOCK_DATA.csv`
- `Week1/source_files/snowpipe/MOCK_DATA-2.csv`
- `Week1/source_files/snowpipe/MOCK_DATA-3.csv`

Let's upload the first file `MOCK_DATA.csv` into our S3 bucket, under the snowpipe subdirectory which we have defined in the Snowpipe definition.

We need to wait a little bit and then we can try to query the landing table. Soon there should be ingested the data. There might be up to 1 min latency so please keep trying to query the table for a while.

## 08 Streams and Task

Let's follow-up on previous session dedicated to snowpipes. Now we try to automatically process the data which have been ingested by snowpipe. We need two additional database objects for it - streams and task.

1. We will start by creating a stream on top of the landing table.
2. We can check if stream already have some data by querying one of the system functions
3. Let's upload the second file `MOCK_DATA-2.csv` into S3 bucket. This action should trigger following scenario:
  - SQS event notification will be sent
  - Snowpipe will fetch the file and ingest the data
  - Our newly created stream should have some data - exactly those from this file.
4. When done, we can check the count in landing table if it has now 2000 records.
5. It means our newly created stream has data now as well - check it by querying the same function again
6. Let's query the stream itself - same like table to check the stream content. We can find the stream metadata at the end of the column list.

That's it for the streams. Now we have an object which holds only freshly loaded data. We need to have an ability to process such data somehow (apply some transformations or aggregations). And for that we need a task.

7. Let's say we want to calculate how many entries for each gender are loaded for each ingested file. We will create a target table called `gender_summary` to save the result.
8. Next we need to grant a privilege to execute a task to our sysadmin role. Let's run the grant command under the account admin role. EXECUTE TASK is account level privilege which is by default granted only to account admin
9. Now we can create a task `t_gender_cnt` which will be running each minute and it will have a condition to start only when our `str_landing` stream has some data. Task will insert aggregated data into the `gender_summary` table together with actual timestamp so we know when the load was done.
10. Let's check the state of the task with show tasks command. It is suspended. We need to resume it.
11. Once task is resumed it should be triggered within the minute because we still have data in our stream. Let's wait for a while before checking the stream status and content of the `gender_summary` table.
12. We can also check the task history by querying the table function `task_history()` to see the loads. We can see many skipped instances - the condition was not fulfilled + there should be also one entry with SUCCEEDED status.
13. Let's upload the latest file into external stage to test out the complete integration. File will be ingested by snowpipe, data will be available in stream which triggers our task which will process those data.
14. We can check the `gender_summary` table and `task_history()` table function again to see that also the latest file has been loaded and processed.
15. As a last step we can again suspend our task not to be started every minute.

## 09 Monitoring the Data Pipeline

### Exercise #1

In the first exercise in this session we are going to build an error notification for task. When task fails, an SNS message will be sent and we can react on that message somehow - send a slack/teams notifications for instance. As we do not have a Slack environment available we will build everything up to that part when it should be send to slack.

Let's start with error notifications setup in AWS:

1. Create an SNS Topic - select a standard type and name the topic `sf_error_notification`
2. Record the ARN of the topic
3. Create an IAM policy which will define following action:  
`sns:publish`

You can copy the following text to the JSON tab of the policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": "<sns_topic_arn>"
    }
  ]
}
```

```
}  
]  
}
```

You need to replace the `sns_topic_arn` with the ARN of the SNS topic you created in step 1.

Click to next and skip the tags definition, then click on review of the policy and name it, e. g. `snowflake_sns_topic`.

4. Now we need an IAM role. But we already have a role linked to our Snowflake account so we can try to reuse it and just attach another policy to that one.

- Open IAM roles, locate your snowflake role and click on **add permissions** and **attach policies**.
- Find the `snowflake_sns_topic` policy and add it to that role.

5. Now we need to create a new object on Snowflake side. That object is called Notification integration. It is integration object that references the SNS topic we have created. Only ACCOUNTADMIN can create such object so do not forget to switch your role. You can find the DDL statement on GitHub in `week2/week2_exercises.sql` file.

6. Once we have the integration we need to grant Snowflake access to the SNS topic. Run the `desc notification integration my_error_notification;`

And make a note of following values:

- `SF_AWS_IAM_USER_ARN`
- `SF_AWS_EXTERNAL_ID`

7. Now we have to modify the trust relationship in the IAM role. Let's open our role in IAM and go to **Trust relationship** tab. We already have there the configuration for our storage integration. The retrieved value of `SF_AWS_IAM_USER_ARN` should be same like we have already defined. So nothing should be changed.

But you should get different value for `SF_AWS_EXTERNAL_ID`. Here we need to change the configuration. We have to use an array for all the values in `sts:ExternalId` field. It should look like this:

```
"Condition": {  
    "StringEquals": {  
        "sts:ExternalId": [  
            "<sf_aws_external_id_storage_integration>",  
            "<sf_aws_external_id_error_integration>"  
        ]  
    }  
}
```

That's it. We have configured the error notification. Now we need two additional things:

- Enable the error integration in tasks
- Some process which will process sent SNS messages

8. Let's modify our task from previous exercise and add this error notification to it.

9. As a process which will be automatically processing the SNS messages we can use lambda function. Let's create one even though we will not finish the integration as we do not have the slack environment. But if you have any slack environment feel free to use it and test the whole integration.

Go to Lambda and create a new function called `sfErrorProcess`:

- Select author from scratch
- Select Python 3.8. as runtime
- Select x86\_64 as architecture

Now we need to add a trigger to our function. It will be triggered automatically by new incoming SNS message. Let's click on add trigger:

- Search for SNS as a source
- And then paste ARN of our SNS topic or AWS console should offer it for you automatically.

Now we need to just write the processing code in Python. You can find such an example of such code in following file on github: `sfErrorProcess.py`

You can go and check the code or update it according your needs. The code is doing following:

- Format the message by using markdown
- Uses Slack API to send message into configured channel. It requires to have a slack integration configured.

## Exercise #2

We are going to try to improve our pipeline little bit and built also email notification for monitoring the task state. We will build a stored procedure which will be checking the state of the task. In case of the suspended state, an email will be sent to defined email addresses.

This solution will use another notification integration but this time the type of it will be email. In this case we do not need to create any integration towards cloud provider as the email is sent directly by Snowflake. But this feature currently works only for Snowflake accounts hosted on AWS. If your account uses another cloud provider this will not work for you.

1. Create a new notification integration called `my_email_int` where the TYPE will be EMAIL and allowed recipients will be your verified email address. This needs to be done as accountadmin

You must have verified email address assigned to your Snowflake profile in order to receive e-mails from Snowflake. Go to your profile settings, enter your email address and click on send verification Email. If you have already done this, then it is not needed.

2. Grant this new integration to our sysadmin role
3. Now we can write a stored procedure which will have one input parameter and it will be `task_name`. The procedure will do following:
  - Check the state of the task by running `show task` command
  - In case the state will be suspended it will call system stored procedure `SYSTEM$SEND_EMAIL` and send the email notification to verified email address.

## 10 Stored Procedures

We are going to improve the monitoring of our data pipeline based on snowpipe, stream and task. We will build another stored procedure which will be monitoring the stale state of our `STR_LANDING` stream.

Two input parameters:

- `stream_name` – name of the stream we want to check
- `staleness_limit` - how many days before stream become stale we want to be notified

Procedure will use a `stale_after` value from `show stream` command and calculate the `stale_after` value by doing date diff between this value and current timestamp

In case the calculated value of days is lower or equal to provided staleness limit then email should be sent with following message:

```
Please check stream. It might became stale in 5 and your defined
notification limit is: 6
```

And procedure will return following message:

```
Email has been sent. Stream will soon become stale.
```

In case the condition will not be met then procedure will just return this message:

```
Stream stale state is ok. It will become stale in 6 days and your
notification limit is: 4
```

## 11 User defined functions

Please use the stored procedure from previous session as a skeleton. This one will be very similar. You need to find out how to write that error message which should be send by email and it combines text with `input_parameters` stored in variables.

We are going to practice scalar and table UDF in this example. Firstly we will try to extract a domain name from user emails in our `snowpipe_landing` table.

Then we will try to improve the solution and write table function which will extract 3 values from single email and return them as a table. We are going to extract the username, first and second domain. This time we will use Python as language for our UDF and UDTF.

1. Let's write a UDF called `get_email_domain(email string)`:
  - It uses Python as a language
  - It extracts email domain part (everything after @ char) from given email
2. Let's write another function called `extract_email(email string)`:
  - It returns table with following schema: `username string, domain1 string, domain2 string`
  - It uses Python as language
  - It extracts following values from given email:
    - Username - everything before '@' char
    - Domain1 - everything after '@' and before '.'
    - Domain2 - everything after '.' In domain part

## 12 External tables





This exercise is dedicated to external tables. You can find the source data which we will need in following Github location:

- `Week2/source_files/external_tables`

Please download all 4 files to your local computer and then upload to your external stage location. Firstly we are going to upload only 3 files out of four.

1. Create a subdirectory `external_tables` in you external stage

2. Create four another directories inside the external\_stage directory to simulate we are loading data each month and they are organised in separated directories.
  - 01\_2023
  - 02\_2023
  - 03\_2023
  - 04\_2023

<input type="checkbox"/>	Name ▲	Type ▼	Last modified ▼
<input type="checkbox"/>	 01_2023/	Folder	-
<input type="checkbox"/>	 02_2023/	Folder	-
<input type="checkbox"/>	 03_2023/	Folder	-
<input type="checkbox"/>	 04_2023/	Folder	-

3. Upload the file ext\_table1.csv to folder 01\_2023 and accordingly also files ext\_table2.csv and ext\_table3.csv

Now we are ready to go and we can create external tables in Snowflake. To validate we have our files available in the stage we start with listing all the files in external stage under the external\_tables subdirectory.

4. Let's start with the easiest option - create and external table without knowing the file structure. We are not going to define the columns. Just the location, file format (csv).
5. Query the table to see how the data are presented
6. Try to query the table again, now by querying the individual columns by \$1:c1, \$2:c2, etc.
7. And query it for the last time now with proper data type and name:
  - \$1:c1::timestamp\_ntz created,
  - \$1:c2::number id,
  - \$1:c3::varchar first\_name,
  - \$1:c4::varchar last\_name,
  - \$1:c5::varchar email,
  - \$1:c6::varchar gender
8. Let's create a second version of the external table as now we know the structure and we can include it into external table definition.
9. Query the table to check it out
10. We can try to query it together with external table metadata: METADATA\$FILENAME, METADATA\$FILE\_ROW\_NUMBER
11. Create the last version of the external table - now with defined partitions. We are going to use the created column and extract the load year month from it. A partition column must evaluate as an expression that parses the path and/or filename information in the METADATA\$FILENAME pseudocolumn.
12. We can extract that info with following command:
 

```
select distinct to_date(split_part(METADATA$FILENAME, '/', 3), 'MM_YYYY')
from @my_s3_stage/external_tables;
```
13. Now we have a definition of the partition column so we can include it into table definition and create the third external table.
14. Query the table to verify it works
15. Query the distinct load year month to find out which partitions we have available in external table. We should get the partitions from 2023-01 up to 2023-03
16. Now, let's upload the last file into the external stage into the 04\_2023 directory
17. If we query the external table now we can see that data are still not available

18. We have to refresh the external table so let's do it
19. We will get an output of the command which will be saying the the last file which we have uploaded has been registered for our external table.
20. If we query the table now, the data from the latest file will be available.

## 13 Data Unloading

This exercise will be dedicated to practising unloading the data into internal and external stages. Let's start with internal stage and csv/parquet data. We are going to use `GPT_TWEETS` table as a source.

We can check the content of the table stage with command: `list @%gpt_tweets;`

Let's prepare the data for offloading. We want to have a user activity overview data set containing only users with at least two tweets and following attributes:

- total number of sent tweets
- total replies received
- total likes received
- ordered from the most active users

Once we have a query for such data set we can include it into `COPY` unloading command and unload the data into table stage, under `csv` subdirectory. As a file format we are going to use our `my_csv_format`.

Let's also try to offload the data into json format and external stage. This could be the use case when you need to share the data via some external system via API. This is also related to practising the semi structured data processing functions like `object_construct()` and `array_agg()`.

Let's do one example together and then you will try to do it yourself. We are going to turn our table into the json document and then unload it into our external stage under `json` subdirectory.

Another simple example is showing how to construct an array. And now you should know everything for creation the JSON file yourself.

### And now tasks for you:

1. Unload the same user activity overview dataset into the parquet format:
  - Use `parquet` subdirectory in table stage
  - Use custom file name `tweet_activity_summary`
2. Please create a JSON file with following structure:

```
{
  username: "abc",
  tweetSummary: [
    {
      tweet: "Random tweet text 1",
      tweetStats: {
        likeCount: 4,
        replyCount: 2,
        retweetCount: 3,
        quoteCount: 0
      }
    }
  ]
}
```



```
}
```

You can see that JSON file contains an array of objects with all the tweets per user, along with statistics for that tweet.

Note:

- You will need functions `OBJECT_CONSTRUCT()` and `ARRAY_AGG()`
- To create an array of objects you will need either CTE or join the dataset with this structure but without array back to base table in order to create an array
- So you need to create this JSON structure without array first and then do the aggregation for each user