Getting Started with Unstructured Data

Overview

This Quickstart is designed to help you understand the capabilities included in Snowflake's support for unstructured data. Sign up for a free 30-day trial of Snowflake and follow along with this lab exercise. After completing this lab, you'll be ready to start storing and managing your own unstructured data in Snowflake.

Prerequisites

- Snowflake account
- Basic knowledge of SQL, database concepts, and objects

What You'll Learn

- How to access and store unstructured data
- How to govern unstructured data
- How to search for unstructured data using directory tables
- How to securely share unstructured data
- How to process unstructured data

What You'll Build

- A stage for storing and accessing files in Snowflake
- A user-defined function using Snowflake's engine to process files
- A secure view to share in the Snowflake Marketplace

Notice and Terms of Use

The data provided for this lab is an extract from the Enron email database made available by Carnegie Mellon University (https://www.cs.cmu.edu/~enron/).

Use of the data provided is limited to this quickstart in connection with the Snowflake service and is subject to any additional terms and conditions on the Carnegie Mellon site.

By accessing this data, you acknowledge and agree to the limits and terms related to the use of the Enron email data.

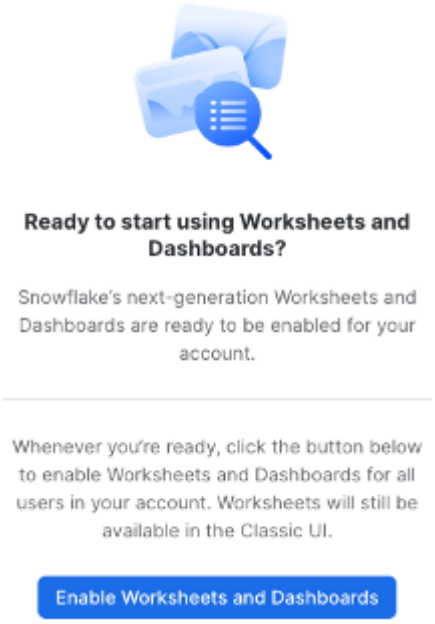
Prepare your lab environment

If you haven't already, register for a Snowflake free 30-day trial. The Snowflake edition (Standard, Enterprise, Business Critical, e.g.), cloud provider (AWS, Azure, e.g.), and Region (US East, EU, e.g.) do not matter for this lab. We suggest you select the region which is physically closest to you and the Enterprise Edition, our most popular offering. After registering, you will receive an email with an activation link and your Snowflake account URL.

Navigating to Snowsight

For this lab, you will use the latest Snowflake web interface, Snowsight.

1. Log into your Snowflake trial account
2. Click on **Snowsight** Worksheets tab. The new web interface opens in a separate tab or window.
3. Click **Worksheets** in the left-hand navigation bar. The **Ready to Start Using Worksheets and Dashboards** dialog opens.
4. Click the **Enable Worksheets and Dashboards button**.



Store & Access Unstructured Data

Let's start by preparing to load the unstructured data into Snowflake. Snowflake supports two types of stages for storing data files used for loading and unloading:

- [Internal stages] store the files internally within Snowflake.
- [External stages] store the files in an external location (i.e. S3 bucket) that is referenced by the stage. An external stage specifies location and credential information, if required, for the bucket.

Create a Database, Schema, and Warehouse

Before creating any stages, let's create a database and a schema that will be used for loading the unstructured data. We will use the UI within the Worksheets tab to run the DDL that creates the database and schema. Copy the commands below into your trial environment, and execute each individually.

```
use role sysadmin;

create or replace database emaildb comment = 'Enron Email Corpus Database';
create or replace schema raw;
create or replace warehouse quickstart;

use database emaildb;
use schema raw;
use warehouse quickstart;
```

Access Unstructured Data Stored in an S3 Bucket

The data we will be using in this lab is stored in an S3 bucket.

Create an External Stage

You are working with unstructured PDFs that have already been staged in a public, external S3 bucket. Before you can use this data, you first need to create a Stage that specifies the location of our external bucket.

For this lab we are using an AWS S3 bucket in us-east-1. To minimize data egress/transfer costs in the future, you should select a staging location from the same cloud provider and region as your Snowflake environment.
Grant the `PUBLIC` schema access to the database, schema, and warehouse just created (This will be relevant in the next section).

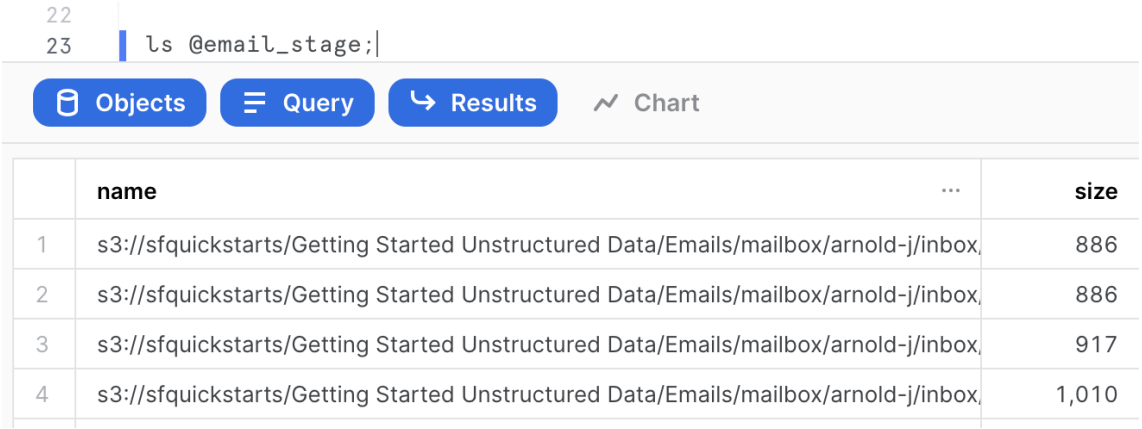
```
use role sysadmin;
grant usage on database emaildb to public;
grant usage on schema emaildb.raw to public;
grant usage on warehouse quickstart to public;
```

From the same worksheet you've been using, run this command to create an external stage called `email_stage`.

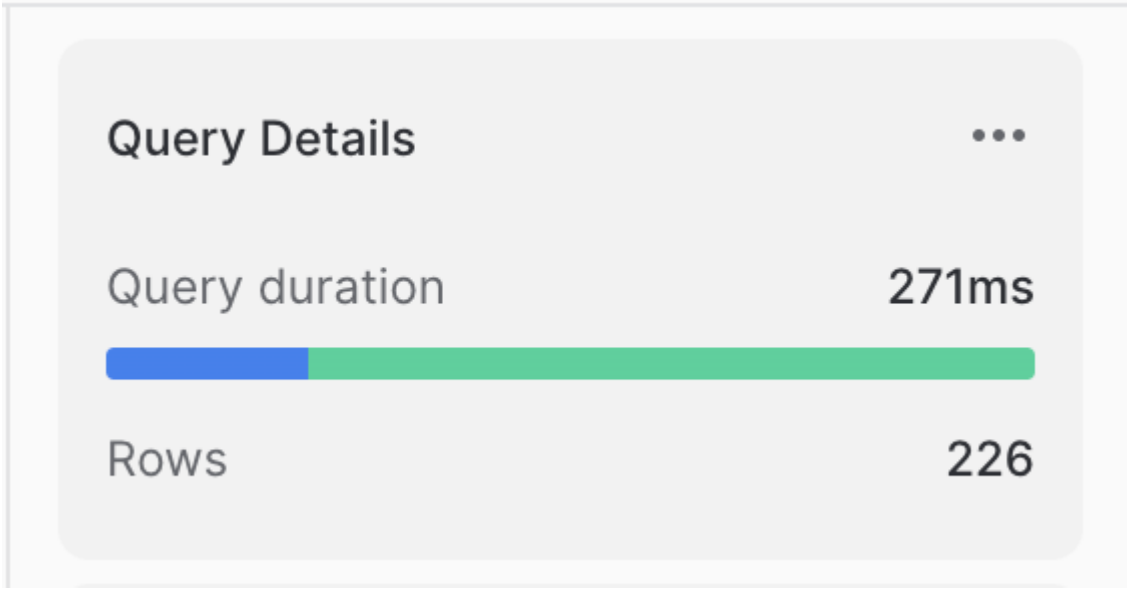
```
use schema emaildb.raw;
-- Create an external stage where files are stored.
create or replace stage email_stage
 url = "s3://sfquickstarts/Getting Started Unstructured Data/Emails/mailbox/"
 directory = (enable = true auto_refresh = true);
```

You can run this command to see a list of the files in your external stage.

```
ls @email_stage;
```



We can see that the stage contains various mailboxes from Enron users containing some files (which are text files) of various sizes. We can get a summary of the file corpus in the query results on the right-hand side. For the purpose of this quickstart, just a sample of 226 files out of 517,551 files has been extracted.



The size of the files: We can see that the file size ranges from 666 bytes to 1023 bytes, with the majority of the files closer to 1023 bytes. If we click on the size metric, we can get more detailed information. The total corpus size is 208,414 bytes, with an average size of 922 bytes. If we hover over the histogram, we can filter results based on file size.

Store Unstructured Data in an Internal Stage

Alternatively, you can store data directly in Snowflake with internal stages. Now, we want to create an internal stage and upload the same files while maintaining the directory structure of the various individual mailboxes on an internal stage.

Create an Internal Stage

Run this command to create an internal stage called `email_stage_internal` as follows.

```
use schema emaildb.raw;
create or replace stage email_stage_internal
directory = (enable = TRUE)
encryption = (type = 'SNOWFLAKE_SSE');
```

Note that we have to specify a server-side encryption on the internal stage. When using the default client-side encryption, the files will be returned encrypted and not readable when accessed through URLs (in Section 6).

Download Data and Scripts

We need to first download the following files to the local workstation by clicking on the hyperlinks below. The subsequent steps require [SnowSQL CLI] installed on the local workstation where the lab is ran:

- upload.snf: SnowSQL Script which will upload the files to the internal stage just created.
- mailbox.tar.gz: The actual email data.

Once downloaded, untar the contents of the files on your local workstation and note the full path including the mailbox parent directory of the tar archive. In the example below, this path is `/Users/znh/Downloads/quickstart/`. The file can be untarred as follows.

```
cd /Users/znh/Downloads/quickstart/
tar xzvf mailbox.tar.gz
```

Upload Files using SnowSQL

Before opening terminal, find out your account identifier which for the trial account will be `<account-locator>.<region-id>.<cloud>`. These fields can be retrieved from the URL of your Snowflake account.

For example, the URL to access the trial account is `https://xx74264.ca-central-1.aws.snowflakecomputing.com/`. These are the values for the account identifier:

- Account Locator: `xx74264`
- Region ID: `ca-central-1`
- Cloud: `aws`

There may be additional segments if you are using your own account part of an organization. You can find those from the URL of your Snowflake account. Please check the [Snowflake Documentation] for additional details on this topic.

Now open a terminal on your workstation and run the following SnowSQL command. You will be prompted for the password for the Snowflake user passed as a parameter.

```
snowsql -a <account-identifier> \
-u <user-id> -d emaildb -r sysadmin -s raw \
-D srcpath=<source-path> \
-D stagename=@email_stage_internal \
-o variable_substitution=true -f upload.snf
```

Using the examples above for the path and the account identifier, and the userid `myuser`, the command would be:

```
snowsql -a xx74264.ca-central-1.aws \
-u myuser -d emaildb -r sysadmin -s raw \
-D srcpath=/Users/znh/Downloads/quickstart/mailbox \
-D stagename=@email_stage_internal \
-o variable_substitution=true -f /Users/znh/Downloads/quickstart/upload.snf
```

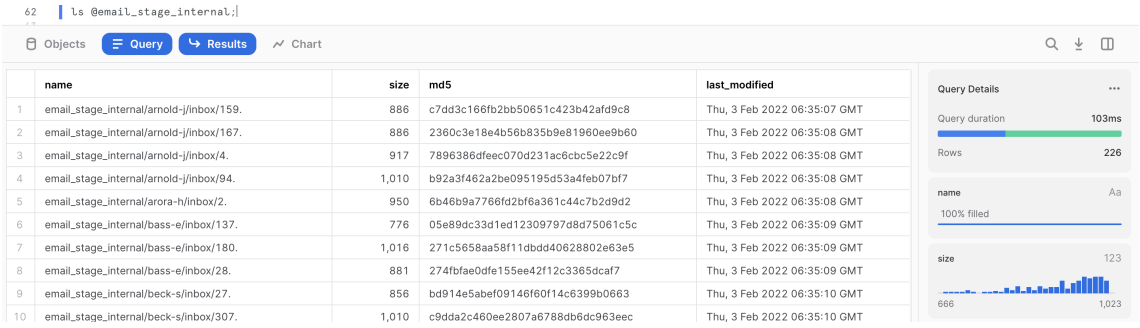
The upload may take a few seconds depending on the speed of your internet connection. If the upload is successful, you should see data being uploaded in each subfolder, and you may see an output like the following on your terminal.

```
| 172.    | 172.    |         1023 |         1023 | NONE               | NONE               | UPLOADED |         |
| 200.    | 200.    |          972 |          972 | NONE               | NONE               | UPLOADED |         |
| 27.     | 27.     |          971 |          971 | NONE               | NONE               | UPLOADED |         |
| 36.     | 36.     |          964 |          964 | NONE               | NONE               | UPLOADED |         |
| 5.      | 5.      |         1002 |         1002 | NONE               | NONE               | UPLOADED |         |
| 60.     | 60.     |          976 |          976 | NONE               | NONE               | UPLOADED |         |
| 83.     | 83.     |          879 |          879 | NONE               | NONE               | UPLOADED |         |
| 98.     | 98.     |          802 |          802 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
14 Row(s) produced. Time Elapsed: 0.978s
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
| source  | target  | source_size  | target_size  | source_compression | target_compression | status   | message |
|---------+---------+--------------+--------------+--------------------+--------------------+----------+---------|
| 53.     | 53.     |         1018 |         1018 | NONE               | NONE               | UPLOADED |         |
| 6.      | 6.      |          933 |          933 | NONE               | NONE               | UPLOADED |         |
| 813.    | 813.    |         1010 |         1010 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
3 Row(s) produced. Time Elapsed: 0.623s
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
| source  | target  | source_size  | target_size  | source_compression | target_compression | status   | message |
|---------+---------+--------------+--------------+--------------------+--------------------+----------+---------|
| 121.    | 121.    |          952 |          952 | NONE               | NONE               | UPLOADED |         |
| 131.    | 131.    |          800 |          800 | NONE               | NONE               | UPLOADED |         |
| 272.    | 272.    |          998 |          998 | NONE               | NONE               | UPLOADED |         |
| 38.     | 38.     |          974 |          974 | NONE               | NONE               | UPLOADED |         |
| 39.     | 39.     |          882 |          882 | NONE               | NONE               | UPLOADED |         |
| 53.     | 53.     |          912 |          912 | NONE               | NONE               | UPLOADED |         |
| 9.      | 9.      |          964 |          964 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
7 Row(s) produced. Time Elapsed: 0.840s
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
| source  | target  | source_size  | target_size  | source_compression | target_compression | status   | message |
|---------+---------+--------------+--------------+--------------------+--------------------+----------+---------|
| 233.    | 233.    |          948 |          948 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
1 Row(s) produced. Time Elapsed: 0.698s
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
| source  | target  | source_size  | target_size  | source_compression | target_compression | status   | message |
|---------+---------+--------------+--------------+--------------------+--------------------+----------+---------|
| 196.    | 196.    |          756 |          756 | NONE               | NONE               | UPLOADED |         |
| 540.    | 540.    |          884 |          884 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
2 Row(s) produced. Time Elapsed: 0.624s
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
| source  | target  | source_size  | target_size  | source_compression | target_compression | status   | message |
|---------+---------+--------------+--------------+--------------------+--------------------+----------+---------|
| 103.    | 103.    |          855 |          855 | NONE               | NONE               | UPLOADED |         |
+---------+---------+--------------+--------------+--------------------+--------------------+----------+---------+
1 Row(s) produced. Time Elapsed: 0.538s
Goodbye!
```

Verify if the files have been uploaded successfully by entering the following command on your Snowflake worksheet.

```
ls @email_stage_internal;
```

You should now see an identical list of files uploaded to the internal stage. Make sure you see 226 files uploaded



Govern Unstructured Data Access

Just like structured and semi-structured data, access permissions to unstructured data in Snowflake can be governed using role-based access control (RBAC).

Create Role and Grant Access

Let's create a role that will provide read access to the external stage we've already created that contains PDFs.

Let's first create the analyst role.

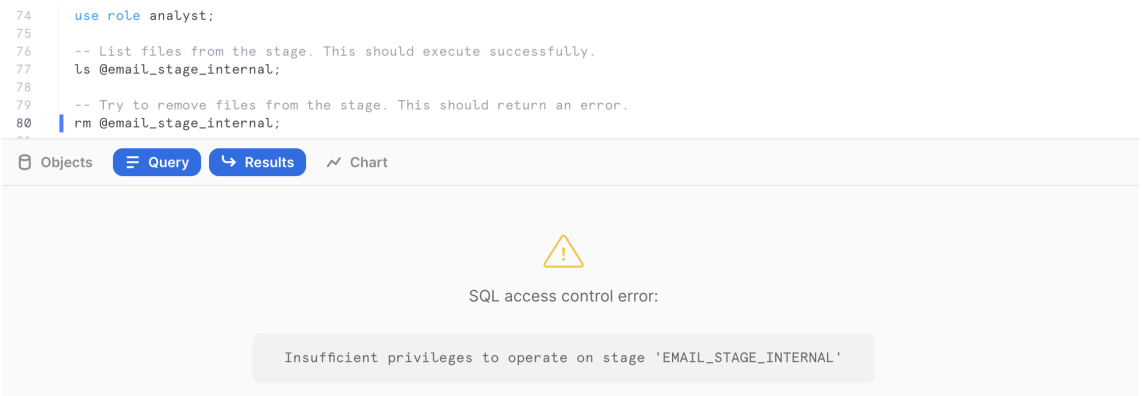
```
use role accountadmin;
create or replace role analyst;
grant role analyst to role sysadmin;
```

Then, switch back to `sysadmin` role, and `grant` the role `analyst` the rights to use the database `emaildb`, and the schema `raw` we just created earlier, as well as the ability to `read` from the stage.

```
grant usage on database emaildb to role analyst;
grant usage on schema emaildb.raw to role analyst;
grant usage on warehouse quickstart to role analyst;
grant read on stage email_stage_internal to role analyst;
```

You can verify the `analyst` role only has access to read by listing the files in the internal stage, then trying to remove files from the external stage. When trying to remove, this should result in an error message.

```
use role analyst;

-- List files from the stage. This should execute successfully.
ls @email_stage_internal;

-- Try to remove files from the stage. This should return an error.
rm @email_stage_internal;
```



In the subsequent sections, we will see a more fine-grained access control of the different unstructured files stored in Snowflake using scoped URLs.

Catalog Unstructured Data using Directory Tables

One of the main pain points in managing large repositories of unstructured data is the ability to access metadata easily on the numerous files, as well as retrieve files per some metadata attributes (last modified, file size, file patterns).

Directory Tables are built-in tables in Snowflake that provide an up-to-date, tabular file catalog for external and internal stages. Directory Tables make it easy to search for and query files using SQL.

We will first reset the session parameters to the correct role, virtual warehouse, database and schema:

```
use role sysadmin;
use warehouse quickstart;
use schema emaildb.raw;
```

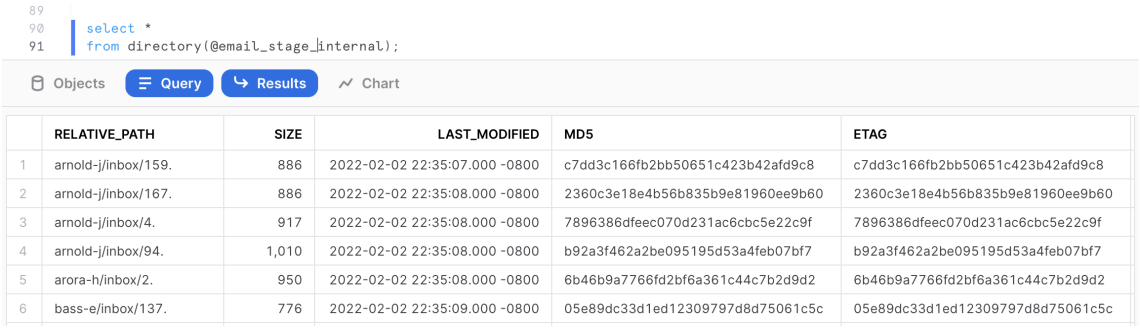
Prior to accessing the directory, it needs to be refreshed first for the files previously uploaded using the following command.

```
alter stage email_stage_internal refresh;
```

Run the following command to access the directory table.

```
select *
from directory(@email_stage_internal);
```

This will provide some detailed metadata information about the files stored in the stage including the `RELATIVE_PATH`, the `LAST MODIFIED` timestamp, the `SIZE`, the `ETAG` as well as the `FILE URL` (more information on this in the next section).



	RELATIVE_PATH	SIZE	LAST_MODIFIED	MD5	ETAG
1	arnold-j/inbox/159.	886	2022-02-02 22:35:07.000 -0800	c7dd3c166fb2bb50651c423b42afd9c8	c7dd3c166fb2bb50651c423b42afd9c8
2	arnold-j/inbox/167.	886	2022-02-02 22:35:08.000 -0800	2360c3e18e4b56b835b9e81960ee9b60	2360c3e18e4b56b835b9e81960ee9b60
3	arnold-j/inbox/4.	917	2022-02-02 22:35:08.000 -0800	7896386dfeec070d231ac6cbc5e22c9f	7896386dfeec070d231ac6cbc5e22c9f
4	arnold-j/inbox/94.	1,010	2022-02-02 22:35:08.000 -0800	b92a3f462a2be095195d53a4feb07bf7	b92a3f462a2be095195d53a4feb07bf7
5	arora-h/inbox/2.	950	2022-02-02 22:35:08.000 -0800	6b46b9a7766fd2bf6a361c44c7b2d9d2	6b46b9a7766fd2bf6a361c44c7b2d9d2
6	bass-e/inbox/137.	776	2022-02-02 22:35:09.000 -0800	05e89dc33d1ed12309797d8d75061c5c	05e89dc33d1ed12309797d8d75061c5c

Searching Directory Tables

We could now query these files using some SQL commands. For example, let's assume we want to identify all the emails from the mailbox belonging to the user `nemec-g`. We could easily do this with the following SQL query.

```
select *
from directory(@email_stage_internal)
where RELATIVE_PATH like '%nemec-g%';
```

```
select *
from directory(@email_stage_internal)
where RELATIVE_PATH like '%nemec-g%';
```

Objects | Query | Results | Chart

	RELATIVE_PATH	SIZE	LAST_MODIFIED	MD5	ETAG	FIL
1	nemec-g/inbox/106.	995	2022-02-02 22:35:37.000 -0800	e1395972e03e89741f1df457151b0c2e	e1395972e03e89741f1df457151b0c2e	htt
2	nemec-g/inbox/1138.	856	2022-02-02 22:35:35.000 -0800	9bd33a76387d83f58a9cf0916aba4a01	9bd33a76387d83f58a9cf0916aba4a01	htt
3	nemec-g/inbox/1222.	1,008	2022-02-02 22:35:35.000 -0800	8c3cf972b4ef94fe20f5dcbf436d3c96	8c3cf972b4ef94fe20f5dcbf436d3c96	htt
4	nemec-g/inbox/1225.	909	2022-02-02 22:35:35.000 -0800	8ed2ca537b3171fbda0c45a7302109e1	8ed2ca537b3171fbda0c45a7302109e1	htt
5	nemec-g/inbox/1263.	986	2022-02-02 22:35:36.000 -0800	ec58b5156d78d98da4baf24584503e0a	ec58b5156d78d98da4baf24584503e0a	htt
6	nemec-g/inbox/1316.	952	2022-02-02 22:35:37.000 -0800	577548b04b32916ec62fb53954858451	577548b04b32916ec62fb53954858451	htt
7	nemec-g/inbox/141.	919	2022-02-02 22:35:37.000 -0800	22cedf8a1d227002c22ae4ce8d0b80e1	22cedf8a1d227002c22ae4ce8d0b80e1	htt
8	nemec-g/inbox/303.	997	2022-02-02 22:35:36.000 -0800	55b9e5a68be1fe758eea1b56b510958a	55b9e5a68be1fe758eea1b56b510958a	htt

This query returns the 13 emails belonging to that user. Now let's try to identify the 5 largest email text in the dataset. We can do that with the following query.

```
select *
from directory(@email_stage_internal)
order by size desc
limit 5;
```

```
select *
from directory(@email_stage_internal)
order by size desc
limit 5;
```

Objects | Query | Results | Chart

	RELATIVE_PATH	SIZE	LAST_MODIFIED	MD5	ETAG	FILE_
1	white-s/inbox/172.	1,023	2022-02-02 22:35:54.000 -0800	071f9b499fd89f8ea9121d37255105c9	071f9b499fd89f8ea9121d37255105c9	https
2	crandell-s/inbox/198.	1,022	2022-02-02 22:35:15.000 -0800	5c0fbe9e02e84711820a2cdd32897220	5c0fbe9e02e84711820a2cdd32897220	https
3	mckay-b/inbox/24.	1,022	2022-02-02 22:35:34.000 -0800	a84e395e6bc0ea689bf969c9d176b349	a84e395e6bc0ea689bf969c9d176b349	https
4	lay-k/inbox/552.	1,021	2022-02-02 22:35:30.000 -0800	a7a3e933d1d472341a188516142cfe6b	a7a3e933d1d472341a188516142cfe6b	https
5	reitmeyer-j/inbox/51.	1,020	2022-02-02 22:35:40.000 -0800	83ff08f2477116c695826a16733bfcd6	83ff08f2477116c695826a16733bfcd6	https

Automatic Refresh

Say you want the directory table to refresh whenever a file is added to your S3 bucket. This can be accomplished by using event notifications in S3. When a new file is added to a bucket, S3 will send a notification to Snowflake, and a Stream can refresh the directory table.

In this quickstart, we won't setup notifications in S3, but the command below is what you would use to create a stream on the directory table for a stage. More detailed documentation for automatically refreshing directory tables can be found [here].

```
-- Create a table stream on directory table
create stream documents_stream on directory(<stage_name>);
```

URLs for Secure Access

In the previous sections, we have seen how to store unstructured data in Snowflake, as well as access metadata about the unstructured files, and build queries to retrieve files based on metadata filters.

In this section, we will look into how Snowflake offers access to the unstructured data through various types of URLs, as well as provide a more granular governance over unstructured data than at the stage level, as reviewed previously

in Section 2.

There are three different types of URLs that you can use to access unstructured data:

- [**Scoped URL**]: A scoped file URL can be generated for a user to give the user short-lived, scoped access to the file without giving privileges on the stage.
- [**File URL**]: A file URL requires a user to be authenticated with Snowflake and requires the user to have read privileges on the stage.
- [**Pre-signed URL**]: As the name suggests, pre-signed URLs are already authenticated. Users can simply download the files using pre-signed URLs.

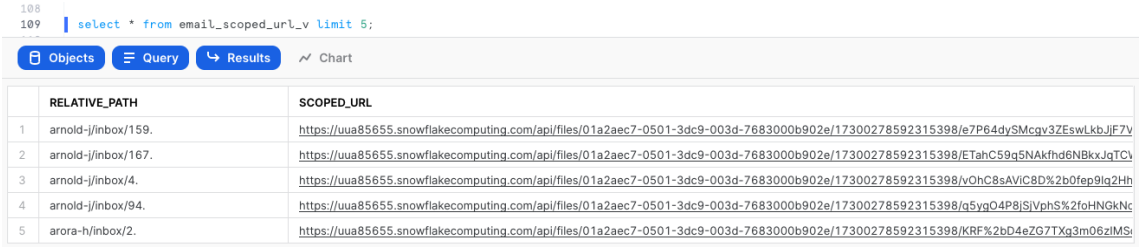
The URL format for files is https://.snowflakecomputing.com/api/files////.

Scoped URL

Scoped URLs are encoded URLs that permit temporary access to a staged file without granting privileges to the stage. The URL expires when the persisted query result period ends (i.e. the results cache expires), which is currently 24 hours.

We can generate a scoped URL of any file using the function `build_scoped_url()`. For example, let's create a view which provides the scoped URL for files in the stage `email_stage_internal`.

```
create or replace view email_scoped_url_v
as
select
    relative_path
    , build_scoped_file_url(@email_stage_internal,relative_path) as scoped_url
from directory(@email_stage_internal);

select * from email_scoped_url_v limit 5;
```



	RELATIVE_PATH	SCOPED_URL
1	arnold-j/inbox/159.	https://uua85655.snowflakecomputing.com/api/files/01a2aec7-0501-3dc9-003d-7683000b902e/17300278592315398/e7P64dySMcgv3ZEswLkbJjF7v
2	arnold-j/inbox/167.	https://uua85655.snowflakecomputing.com/api/files/01a2aec7-0501-3dc9-003d-7683000b902e/17300278592315398/ETahC59q5NAkfhd6NBkxJqTCv
3	arnold-j/inbox/4.	https://uua85655.snowflakecomputing.com/api/files/01a2aec7-0501-3dc9-003d-7683000b902e/17300278592315398/vOhC8sAViC8D%2b0fep9lq2Hh
4	arnold-j/inbox/94.	https://uua85655.snowflakecomputing.com/api/files/01a2aec7-0501-3dc9-003d-7683000b902e/17300278592315398/q5ygO4P8jSjVphS%2foHNGkNc
5	arora-h/inbox/2.	https://uua85655.snowflakecomputing.com/api/files/01a2aec7-0501-3dc9-003d-7683000b902e/17300278592315398/KRF%2bD4eZG7TXg3m06zIMS

As explained previously, this URL will be valid for 24 hours. Snowsight retrieves the file only for the user who generated the scoped URL.

Scoped URLs enable access to the files via a view that retrieves scoped URLs. Only roles that have privileges on the view can access the files. The scoped URL contents are all encrypted and doesn't give any information about the bucket, database or schema.

Secure Access with RBAC and Scoped URL

Let's assume a scenario where the `analyst` role needs access to all the files from the inbox of `NEMEC-G`. We can build a dynamic view which will filter the output based on the role of the view user.

First let's build an assignment table where various roles (or even users) can be assigned a given mailbox for further analysis.

```
create or replace table assignment (mailbox string, role string, filter string);

insert into assignment values ('NEMEC-G','ANALYST','%nemec-g%');
insert into assignment values ('*','SYSADMIN','%');

select * from assignment;
```

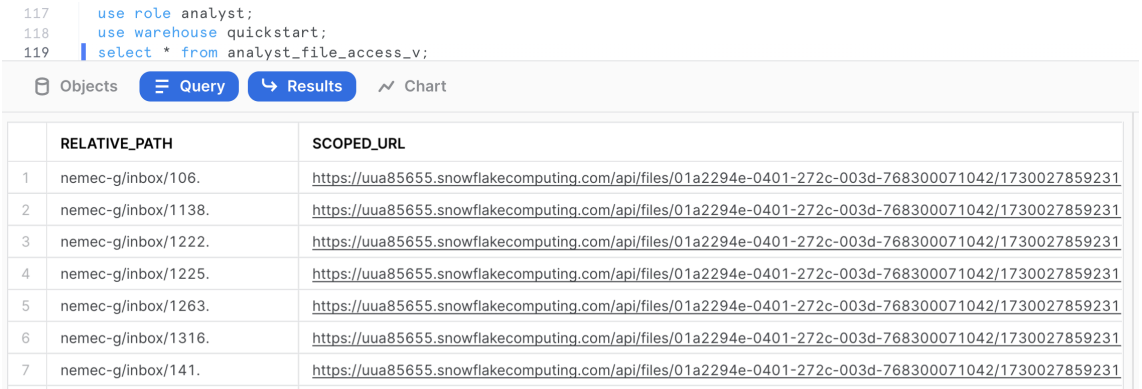
As we can see from the query output above, the assignment table controls a 1-to-1 role-to-mailbox access mapping.

We can now build a SQL view which will join with the assignment table to dynamically filter rows based on the role of the user executing the view and grant access to the view to the `analyst` role:

```
create or replace secure view analyst_file_access_v as
    select
        relative_path
        , build_scoped_file_url(@email_stage_internal,relative_path) as scoped_url
    from directory(@email_stage_internal)
        inner join assignment
    where relative_path like filter
          and role = current_role();

grant select on analyst_file_access_v to role analyst;
```

Let's switch to role `analyst` and query the view.

```
use role analyst;
use warehouse quickstart;
use schema raw;
select * from analyst_file_access_v;
```



	RELATIVE_PATH	SCOPED_URL
1	nemec-g/inbox/106.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
2	nemec-g/inbox/1138.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
3	nemec-g/inbox/1222.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
4	nemec-g/inbox/1225.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
5	nemec-g/inbox/1263.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
6	nemec-g/inbox/1316.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231
7	nemec-g/inbox/141.	https://uua85655.snowflakecomputing.com/api/files/01a2294e-0401-272c-003d-768300071042/1730027859231

Click the `scoped_url` corresponding to the file `1222.` to your workstation and review the file locally.

If we switch the role to `sysadmin` and run the same query:

```
use role sysadmin;
use warehouse quickstart;
use schema raw;
select * from analyst_file_access_v;
```

This returns scoped URLs to access all the files.

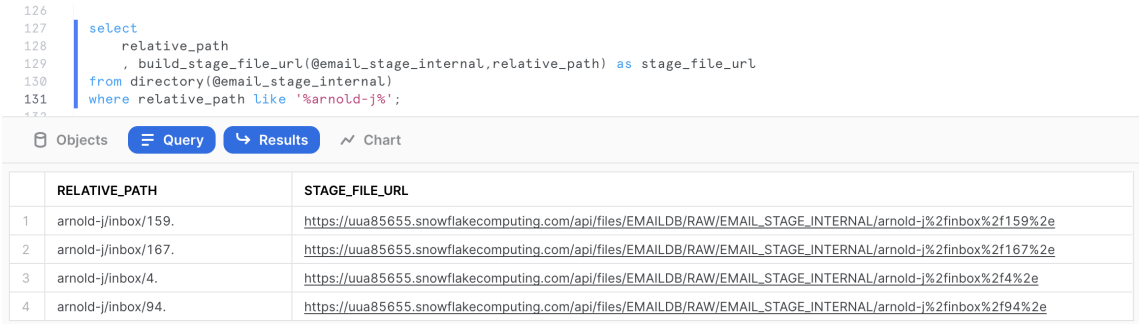
As we just experimented, scoped URLs are ideal for use in custom applications, providing unstructured data to other accounts via a share, or for downloading and ad-hoc analysis of unstructured data via Snowsight.

File URL

A file URL is a permanent URL that identifies the database, schema, stage, and file path to a set of files, as opposed to the previous scoped URL where all this information is encrypted. A role that has sufficient privileges on the stage can access the files. It does not contain any authentication token. Authentication needs to be done when connecting through REST API. However, it works from our current authenticated Snowsight session.

The following query will provide the file urls for the mailbox `arnold-j`.

```
select
    relative_path
    , build_stage_file_url(@email_stage_internal,relative_path) as stage_file_url
from directory(@email_stage_internal)
where relative_path like '%arnold-j%';
```



	RELATIVE_PATH	STAGE_FILE_URL
1	arnold-j/inbox/159.	https://uua85655.snowflakecomputing.com/api/files/EMAILDB/RAW/EMAIL_STAGE_INTERNAL/arnold-j%2finbox%2f159%2e
2	arnold-j/inbox/167.	https://uua85655.snowflakecomputing.com/api/files/EMAILDB/RAW/EMAIL_STAGE_INTERNAL/arnold-j%2finbox%2f167%2e
3	arnold-j/inbox/4.	https://uua85655.snowflakecomputing.com/api/files/EMAILDB/RAW/EMAIL_STAGE_INTERNAL/arnold-j%2finbox%2f4%2e
4	arnold-j/inbox/94.	https://uua85655.snowflakecomputing.com/api/files/EMAILDB/RAW/EMAIL_STAGE_INTERNAL/arnold-j%2finbox%2f94%2e

Click on any of the stage file URLs and download the file to your workstation.

Pre-signed URL

Pre-signed URLs are used to download or access files, via a web browser for example, without authenticating into Snowflake or passing an authorization token. These URLs are ideal for business intelligence applications or reporting tools that need to display the unstructured file contents.

Pre-signed URLs are open but temporary. The expiration time for the access token is configurable when generating the URL. Any user or application can directly access or download the files until the expiration time is reached.

The following query will generate the pre-signed URLs for the mailbox `beck-s`. In the `get_presigned_url()` system function, we pass the parameter `300` which represents the expiration time for the token in seconds (which is 300s, so 5 minutes in this case).

```
select
    relative_path
    , get_presigned_url(@email_stage_internal,relative_path,300) as presigned_url
from directory(@email_stage_internal)
where relative_path like '%beck-s%';
```

Click on any cell value in the `PRESIGNED_URL` column. This should give the actual full URL on the right hand side in the grey cell. (Click to Copy)

```
132
133 select
134     relative_path
135     , get_presigned_url(@email_stage_internal,relative_path,300) as presigned_url
136 from directory(@email_stage_internal)
137 where relative_path like '%beck-s%';
```

Objects | Query | Results | Chart

	RELATIVE_PATH	PRESIGNED_URL
1	beck-s/inbox/27.	https://sfc-prod2-ds1-20-customer-stage.s3.us-west-2.amazonaws.com/ihuu-s-p2sv2850/stages/211abb7d-241e-4a48-ba1a-d4f1c
2	beck-s/inbox/307.	https://sfc-prod2-ds1-20-customer-stage.s3.us-west-2.amazonaws.com/ihuu-s-p2sv2850/stages/211abb7d-241e-4a48-ba1a-d4f1c
3	beck-s/inbox/413.	https://sfc-prod2-ds1-20-customer-stage.s3.us-west-2.amazonaws.com/ihuu-s-p2sv2850/stages/211abb7d-241e-4a48-ba1a-d4f1c
4	beck-s/inbox/537.	https://sfc-prod2-ds1-20-customer-stage.s3.us-west-2.amazonaws.com/ihuu-s-p2sv2850/stages/211abb7d-241e-4a48-ba1a-d4f1c
5	beck-s/inbox/638.	https://sfc-prod2-ds1-20-customer-stage.s3.us-west-2.amazonaws.com/ihuu-s-p2sv2850/stages/211abb7d-241e-4a48-ba1a-d4f1c

Open a new tab in your web browser, and paste the copied URL. This should download the file on your workstation. As you can see, the URL is valid even when executed outside from the Snowsight UI.

Perform Natural Language Processing

We have so far reviewed how to store unstructured data files, retrieve them, provide granular access to the files through various URLs and through secure views. In this section, we want to extract additional attributes from the files.

The entities extracted are going to be person names mentioned in the emails, as well as locations. The goal is to have these additional attributes used to enrich the file-level metadata for analytics.

The Java Code

The Java for the user-defined function (UDF) has already been written and provided below. This code uses the open source Apache OpenNLP library to perform natural language processing on English text in this occurrence.

The Java code leverages pre-built machine learning models to perform named entity extraction for persons and locations. These models are packaged manually post-build in a Fat JAR.

At a high level, the code does the following:

- Parse the text file contents using an Apache Tika text parser.
- Tokenize the parsed contents using a model.
- From the tokens, perform a named entity extraction for persons and locations using pre-built ML models.
- Serialize the results in a JSON string returned as an output.

```
public String ParseText(String filePath) throws IOException, SAXException,
TikaException {

// Configure gson
GsonBuilder gsonBuilder = new GsonBuilder()
Gson gson = gsonBuilder.create();

//detecting the file type
BodyContentHandler handler = new BodyContentHandler();
Metadata metadata = new Metadata();
ParseContext pcontext=new ParseContext();

SnowflakeFile file = SnowflakeFile.newInstance(filePath);
```

```
InputStream ins = file.getInputStream();

//Text document parser
TXTParser  TexTParser = new TXTParser();
TexTParser.parse(ins, handler, metadata, pcontext);

String Contents = handler.toString();

String[] AllPersonEntities = null;
String[] AllLocationEntities = null;
String JsonResult = null;

NamedEntities NE = new NamedEntities(AllPersonEntities, AllLocationEntities);

try {
       for(int i=0;i<sentences.length;i++){
            String Tokens[] = new
NamedEntityExtraction().ParseTokens(Contents);
            String PersonEntities[] = new
NamedEntityExtraction().findName(Tokens);
            String LocationEntities[] = new
NamedEntityExtraction().findLocation(Tokens);

            AllPersonEntities = ArrayUtils.addAll(AllPersonEntities,
PersonEntities);
            AllLocationEntities = ArrayUtils.addAll(AllLocationEntities,
LocationEntities);

            NE.setPersons(AllPersonEntities);
            NE.setLocations(AllLocationEntities);

            JsonResult = gson.toJson(NE).toString();


          } catch (IOException e) {
              e.printStackTrace();
          }
           return(JsonResult);
}
```

A few elements relevant for this code:

- Notice the main class is **NamedEntityExtraction**.
- The method **ParseText** will be invoked by the UDF in the next section.
- The file path is passed as a parameter. It can be a URL to the file, or the path on the stage.

Creating a UDF in Snowflake

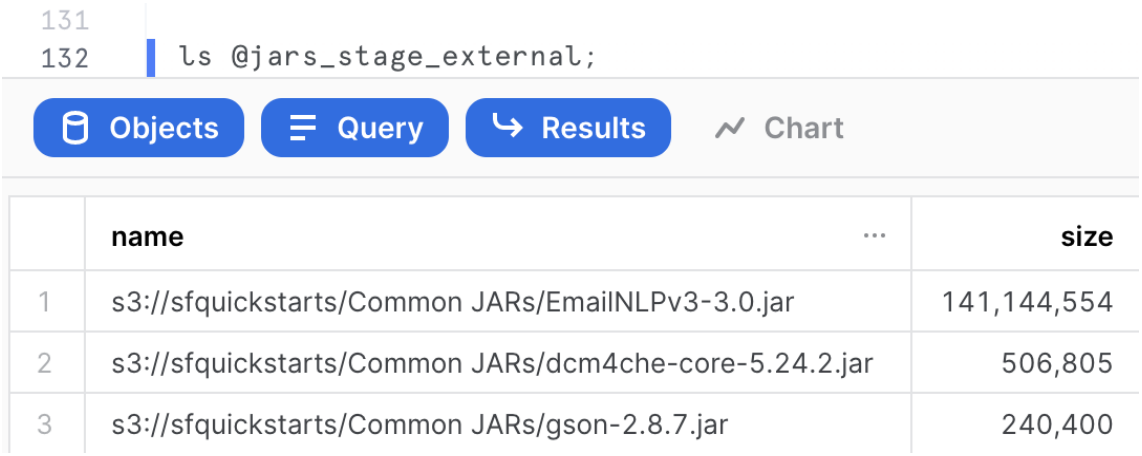
The precompiled jar file including all the dependencies has been uploaded and available on a Snowflake s3 public bucket. Creating the UDF involves a few steps in Snowflake.

1. Create the external stage mapping to the S3 bucket URI where the jar file is currently available. From the Snowflake worksheet, enter the following command:

```
use role sysadmin;
use schema emaildb.raw;

create or replace stage jars_stage_external
url = "s3://sfquickstarts/Common JARs/"
directory = (enable = true auto_refresh = true);
```

From the Snowflake worksheet, you can run the following command to confirm the jar file is listed in the external stage.

```
ls @jars_stage_external;
```



	name	size
1	s3://sfquickstarts/Common JARs/EmailNLPv3-3.0.jar	141,144,554
2	s3://sfquickstarts/Common JARs/dcm4che-core-5.24.2.jar	506,805
3	s3://sfquickstarts/Common JARs/gson-2.8.7.jar	240,400

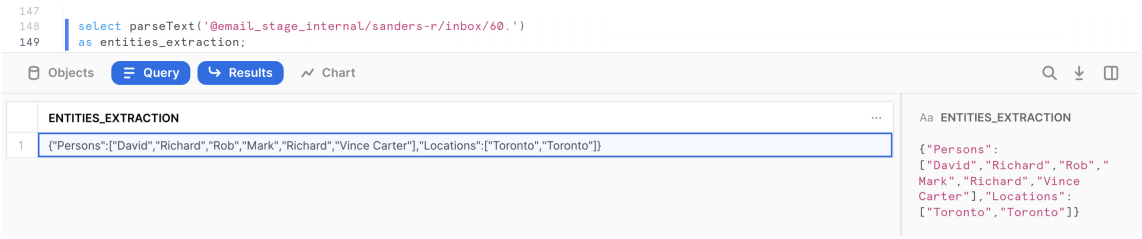
2. We can now create the UDF in Snowflake as the following.

```
create or replace function  parseText(file string)
returns string
language java
imports = ('@jars_stage_external/EmailNLPv3-3.0.jar')
handler = 'NamedEntityExtraction.ParseText'
;
```

Invoking the Java UDF

The UDF can be invoked on any text file containing readable english. From the email corpus stored on the internal stage, we can invoke the UDF as follows.

```
select parseText('@email_stage_internal/sanders-r/inbox/60.')
as entities_extraction;
```



The output is actually serialized as a valid JSON format by the UDF. It contains 2 arrays, one for each named entities extraction:

```
{
"Persons":["David","Richard","Rob","Mark","Richard","Vince Carter"],
"Locations":["Toronto","Toronto"]
}
```

Since we are using pre-built models which haven't been trained on this particular corpus, the entity extraction may not always be accurate. However, the models perform overall quite well for illustration purposes for this quickstart.

Extracting and Storing Named Entities

We want to store the named entities as additional attributes for analysts to be able to select and retrieve the files of interest in their analysis, as well as perform some analytics on the attributes found.

We first want to scale-up the default warehouse size to run the Java UDF at scale across all cores available on all nodes on a 2XL warehouse (64 nodes). This can be done easily and quickly because of Snowflake's instant elasticity:

```
alter warehouse quickstart set warehouse_size = xxlarge;
```

Now let's run the following query to store the named entities into a table.

```
create or replace table email_named_entities_base as
select
    relative_path
    , upper(replace(get(split(relative_path, '/'), 0), '\"', ''))    as mailbox
    , parseText('@email_stage_internal/' || relative_path)         as named_entities
from (
    select relative_path
    from directory(@email_stage_internal)
    group by relative_path
);
```

After running this query, we can set the `warehouse` size back to a smaller size.

THIS STEP IS VERY IMPORTANT NOT TO EXHAUST YOUR TRIAL CREDIT

```
alter warehouse quickstart set warehouse_size = xsmall;
```

Verify with the following command that the warehouse is back to an `xsmall` size.

```
show warehouses;
```

The output should show the `QUICKSTART` size as being `XSMALL`. Now let's query the base table containing the named entities.

```
select
    *
from email_named_entities_base
limit 5;
```

For each email, we now have created a `MAILBOX` attribute, as well as JSON string containing the named entities present in the file.

```
167
168 select
169     *
170 from email_named_entities_base
171 limit 5;
```

Objects | Query | Results | Chart

	RELATIVE_PATH	MAILBOX	NAMED_ENTITIES
1	dean-c/inbox/608.	DEAN-C	{"Persons":["Steiner","David","Dietrich","Dan","Dave Steiner"],"Locations":["Houston","Portland","Craigl
2	kaminski-v/inbox/637.	KAMINSKI-V	{"Persons":["Vince J","Vince JInbox","Tony Sent","Barrett","Misty Subject","Anthony Cox"],"Locations":[
3	jones-t/inbox/742.	JONES-T	{"Persons":["Sara","Jones","Tana","Koehler","Anne","Heard","Marie"],"Locations":["Houston"]}
4	tycholiz-b/inbox/144.	TYCHOLIZ-B	{"Persons":["Ward","Kim S","Anne","Richard","Barry","Anne","Kim"],"Locations":["Palo Alto","Anne","Palc
5	martin-t/inbox/140.	MARTIN-T	{"Persons":["Martin","Thomas","Brian","Bryan","Gary","Thomas","Gary"],"Locations":["Texas"]}

Exploring the Mailbox Corpus

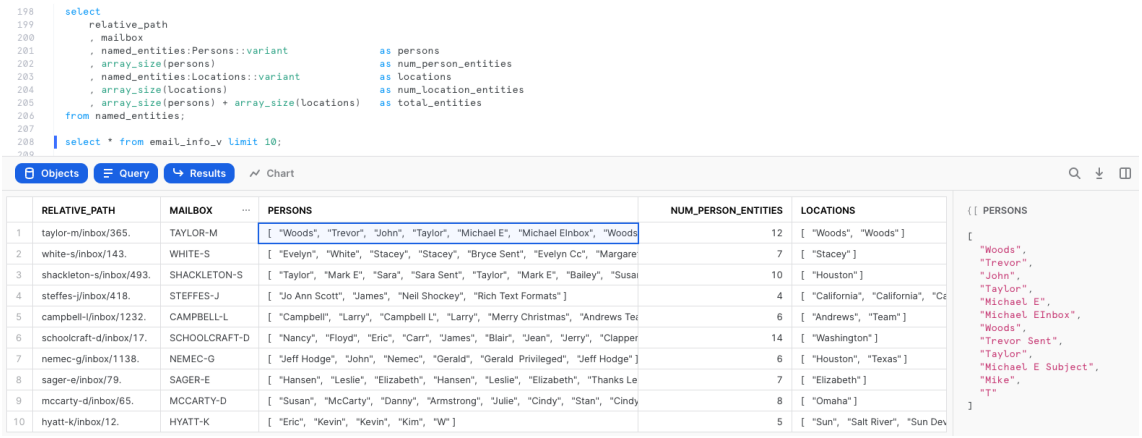
We have now extracted the named entities the analysts are interested in seeing to do some analytics on this email corpus. We can use Snowflake native capabilities to easily store and query semi-structured data, in this case JSON.

We will first create a view to parse the `NAMED_ENTITIES` JSON string and extract separately the persons and the locations entities, as well as count the number of entities identified in each email:

```
create or replace view email_info_v
as
with named_entities
as (
    select
        relative_path
        , mailbox
        , parse_json(named_entities) as named_entities
    from email_named_entities_base
)
select
    relative_path
    , mailbox
    , named_entities:Persons::variant                as persons
    , array_size(persons)                            as num_person_entities
    , named_entities:Locations::variant                as locations
    , array_size(locations)                            as num_location_entities
    , array_size(persons) + array_size(locations)    as total_entities
    , build_scoped_file_url(@email_stage_internal, relative_path)    as
scoped_email_url
from named_entities;
```

We can query the view and examine the output. Notice that we now have JSON arrays in separate columns for name and location entities, count of entities, and a scoped URL to access the file if we need to further examine its contents.

```
select * from email_info_v limit 10;
```



```
select
    relative_path
    , mailbox
    , named_entities:Persons::variant               as persons
    , array_size(persons)                           as num_person_entities
    , named_entities:Locations::variant             as locations
    , array_size(locations)                         as num_location_entities
    , array_size(persons) + array_size(locations)   as total_entities
from named_entities;

select * from email_info_v limit 10;
```

Objects | Query | Results | Chart

	RELATIVE_PATH	MAILBOX	PERSONS	NUM_PERSON_ENTITIES	LOCATIONS
1	taylor-m/inbox/365.	TAYLOR-M	["Woods", "Trevor", "John", "Taylor", "Michael E", "Michael EInbox", "Woods	12	["Woods", "Woods"]
2	white-s/inbox/143.	WHITE-S	["Evelyn", "White", "Stacey", "Stacey", "Bryce Sent", "Evelyn Cc", "Margare	7	["Stacey"]
3	shackleton-s/inbox/493.	SHACKLETON-S	["Taylor", "Mark E", "Sara", "Sara Sent", "Taylor", "Mark E", "Bailey", "Susa	10	["Houston"]
4	steffes-j/inbox/418.	STEFFES-J	["Jo Ann Scott", "James", "Neil Shockey", "Rich Text Formats"]	4	["California", "California", "Ca
5	campbell-l/inbox/1232.	CAMPBELL-L	["Campbell", "Larry", "Campbell L", "Larry", "Merry Christmas", "Andrews Tea	6	["Andrews", "Team"]
6	schoolcraft-d/inbox/17.	SCHOOLCRAFT-D	["Nancy", "Floyd", "Eric", "Carr", "James", "Blair", "Jean", "Jerry", "Clapper	14	["Washington"]
7	nemec-g/inbox/1138.	NEMEC-G	["Jeff Hodge", "John", "Nemec", "Gerald", "Gerald Privileged", "Jeff Hodge"]	6	["Houston", "Texas"]
8	sager-e/inbox/79.	SAGER-E	["Hansen", "Leslie", "Elizabeth", "Hansen", "Leslie", "Elizabeth", "Thanks Le	7	["Elizabeth"]
9	mccarty-d/inbox/65.	MCCARTY-D	["Susan", "McCarty", "Danny", "Armstrong", "Julie", "Cindy", "Stan", "Cindy	8	["Omaha"]
10	hyatt-k/inbox/12.	HYATT-K	["Eric", "Kevin", "Kevin", "Kim", "W"]	5	["Sun", "Salt River", "Sun Dev

{[PERSONS

```
[
  "Woods",
  "Trevor",
  "John",
  "Taylor",
  "Michael E",
  "Michael EInbox",
  "Woods",
  "Trevor Sent",
  "Taylor",
  "Michael E Subject",
  "Mike",
  "T"
]
```

We can now query the view to retrieve various entity metrics. For example, the following query identifies the top 5 emails in terms of total number of entities.

```
select
    relative_path
    , mailbox
    , total_entities
    , num_person_entities
    , num_location_entities
    , persons
    , locations
    , scoped_email_url
from email_info_v
order by total_entities desc
limit 10;
```

```
select
    relative_path
    , mailbox
    , total_entities
    , num_person_entities
    , num_location_entities
    , persons
    , locations
from email_info_v
order by total_entities desc
limit 5;
```

Objects | Query | Results | Chart

	RELATIVE_PATH	MAILBOX	TOTAL_ENTITIES	NUM_PERSON_ENTITIES	NUM_LOCATION_ENTITIES	PERSON
1	williams-j/inbox/38.	WILLIAMS-J	16	15	1	["Murra
2	nemec-g/inbox/106.	NEMEC-G	16	14	2	["Jeffre
3	schoolcraft-d/inbox/17.	SCHOOLCRAFT-D	15	14	1	["Nanc
4	white-s/inbox/36.	WHITE-S	15	14	1	["Andy
5	jones-t/inbox/388.	JONES-T	14	13	1	["Gray"

The following query aggregates statistics on the number of entities identified per mailbox and returns the top 5 mailbox by total number of entities identified in the mailbox.

```
select
    mailbox
```

```
    , count(relative_path)                as num_emails
    , sum(total_entities)                 as sum_entities
    , round(avg(total_entities))          as avg_entities
    , sum(num_person_entities)            as sum_persons
    , round(avg(num_person_entities))     as avg_persons
    , sum(num_location_entities)          as sum_locations
    , round(avg(num_location_entities))   as avg_locations
from email_info_v
group by mailbox
order by sum_entities desc
limit 5;
```

We know that the most prolific mailbox is `WHITE-S` for identified entities in this email corpus.

Performing Analytics on the Mailbox

Let's assume an analyst wants to identify all email correspondence where the name of 'Willman' (chosen completely randomly in this example) is mentioned. With Snowflake, you can easily flatten arrays to run this type of query.

```
select
    relative_path
    , mailbox
    , pers.value::string as person_entity
from
    email_info_v
    , lateral flatten(input => Persons) pers;
```

```
216
217  select
218      relative_path
219      , mailbox
220      , pers.value::string    as person_entity
221  from
222      email_info_v
223      , lateral flatten(input => Persons) pers;
```

Objects | Query | Results | Chart

	RELATIVE_PATH	MAILBOX	PERSON_ENTITY
1	farmer-d/inbox/55.	FARMER-D	Sally Beck Charity Golf Event
2	farmer-d/inbox/55.	FARMER-D	Robert
3	farmer-d/inbox/55.	FARMER-D	Charles
4	farmer-d/inbox/55.	FARMER-D	Daren
5	farmer-d/inbox/55.	FARMER-D	Smith
6	farmer-d/inbox/55.	FARMER-D	George

We can use the previous query as a CTE to retrieve all emails mentioning a specific person, including a scoped URL to access the actual email file.

```
with persons_flattened as (
    select
        relative_path
        , mailbox
        , pers.value::string as person_entity
    from
        email_info_v
```

```
        , lateral flatten(input => Persons) pers
)
select
    relative_path
    , mailbox
    , person_entity
    , build_scoped_file_url(@email_stage_internal,relative_path) as scoped_email_url
from persons_flattened
where person_entity like '%Willmann%';
```

```
224
225  with persons_flattened as (
226      select
227          relative_path
228          , mailbox
229          , pers.value::string as person_entity
230      from
231          email_info_v
232          , lateral flatten(input => Persons) pers
233  )
234  select
235      relative_path
236      , mailbox
237      , person_entity
238      , build_scoped_file_url(@email_stage_internal,relative_path)    as email_url
239  from persons_flattened
240  where person_entity like '%Willmann%';
```

Objects | Query | Results | Chart

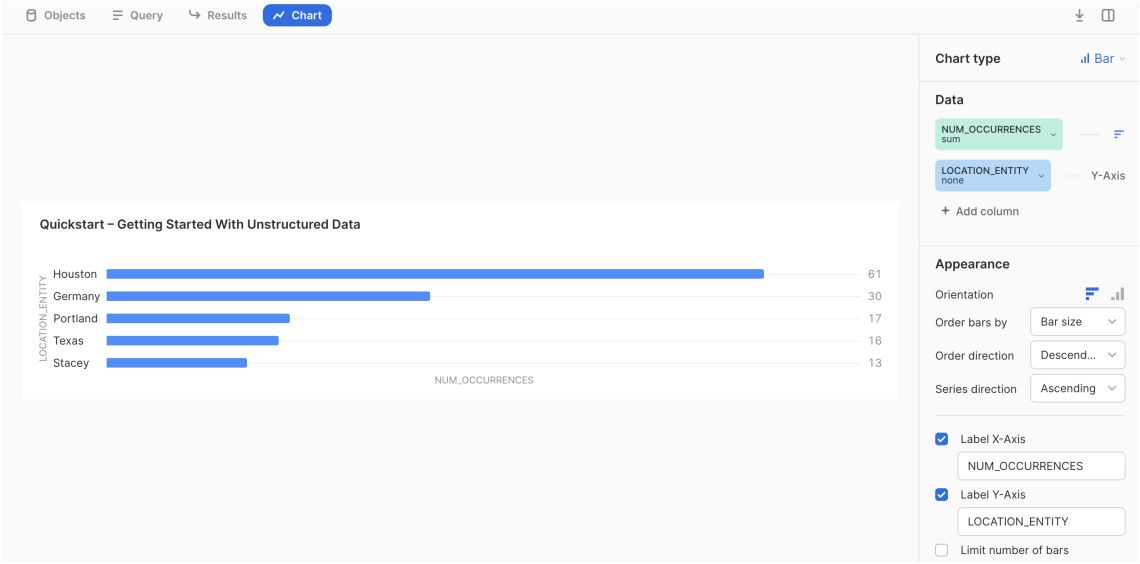
	RELATIVE_PATH	MAILBOX	PERSON_ENTITY	EMAIL_URL
1	may-l/inbox/503.	MAY-L	Donnie Willmann	https://uua85655.snowflakecomputing.com/api/files/01a22a34-0401-279d-003d-7
2	smith-m/inbox/148.	SMITH-M	Donnie Willmann	https://uua85655.snowflakecomputing.com/api/files/01a22a34-0401-279d-003d-7
3	keavey-p/inbox/82.	KEAVEY-P	Donnie Willmann	https://uua85655.snowflakecomputing.com/api/files/01a22a34-0401-279d-003d-7
4	donohoe-t/inbox/93.	DONOHOE-T	Donnie Willmann	https://uua85655.snowflakecomputing.com/api/files/01a22a34-0401-279d-003d-7
5	beck-s/inbox/307.	BECK-S	Donnie Willmann	https://uua85655.snowflakecomputing.com/api/files/01a22a34-0401-279d-003d-7

Click on any `MAY-L` mailbox email to download and review the email. This will allow you to learn a little bit more about that person's role and responsibilities in the Enron organization.

Let's assume we want to identify the top 5 locations mentioned in the email corpus.

```
with locations_flattened as (
    select
        relative_path
        , mailbox
        , loc.value::string    as location_entity
    from
        email_info_v
        , lateral flatten(input => Locations) loc
)
select
    location_entity
    , count(location_entity) as num_occurrences
from locations_flattened
group by location_entity
order by num_occurrences desc
limit 5;
```

We can use Snowsight to produce a visualization. Click on chart and set the settings as shown below.



As you can see, we can perform aggregations, and analytics on unstructured text data after extracting entities and information of interest. At this point, one could run more advanced data science use cases or visualizations using the numerous options available in Snowflake.

Share Unstructured Data

In this example, we want to share all the email corpus mentioning 'Willman' with another party using a Snowflake reader account.

Creating a Reader Account

The first step is to create a reader account as follows. Note that you will need to provide a secure password of your choice.

```
use role accountadmin;
create managed account if not exists emaildb_reader
admin_name='admin', admin_password='<password>',
type=reader, COMMENT='Emaildb Reader Account';
```

This command should return the account name, and the URL to access the account. Please copy and paste the output of the command as it provides you the login URL information for the account, as well as the user and password you chose

```
{
"accountName":"<account_name>",
"loginUrl":"https://<account>.snowflakecomputing.com"
}
```

You can run the following command to retrieve the managed account information at anytime.

```
show managed accounts;
```

You can connect to the previous reader account using the URL, and the userid/password credentials you passed as parameters.

Creating a Secure View to Share

Let's now create the secure view based on the query ran in the previous section that will be shared with the reader account.

```
use role sysadmin;
use schema emaildb.raw;

create or replace secure view email_corpus_willman_v as
with persons_flattened as (
    select
        relative_path
        , mailbox
        , pers.value::string as person_entity
    from
        email_info_v
        , lateral flatten(input => Persons) pers
)
select
    relative_path
    , mailbox
    , person_entity
    , build_scoped_file_url(@email_stage_internal,relative_path) as email_url
from persons_flattened
where person_entity like '%Willmann%';
```

Create the Share

We can now create the share as follows. You will need to provide the reader account name created earlier:

```
-- Create the share object
use role accountadmin;
create or replace share email_corpus
comment='Share in scope email corpus information';

--what are we sharing?
grant usage on database emaildb to share email_corpus;
grant usage on schema emaildb.raw to share email_corpus;
grant select on view emaildb.raw.email_corpus_willman_v to share email_corpus;

-- whom are we sharing with?
alter share email_corpus add accounts = <reader-account-locator>;
```

We can review the share we have just created. The following command provides all the shares in the account.

```
-- check the share
show shares like 'email_corpus';
```

We can get more details about the share, and the scope of the objects shared using the following command.

```
-- review share
describe share email_corpus;
```

```
314
315   -- review share
316   describe share email_corpus;
```

Objects | Query | Results | Chart

	kind	name	shared_on
1	DATABASE	EMAILDB	2022-02-07 14:01:06.307 -0800
2	SCHEMA	EMAILDB.RAW	2022-02-07 14:01:06.604 -0800
3	VIEW	EMAILDB.RAW.EMAIL_CORPUS_WILLMAN_V	2022-02-07 14:01:07.077 -0800

This command shows us that the view `EMAIL_CORPUS_WILLMAN_V` is shared from the database `EMAILDB` and schema `RAW` in the current account.

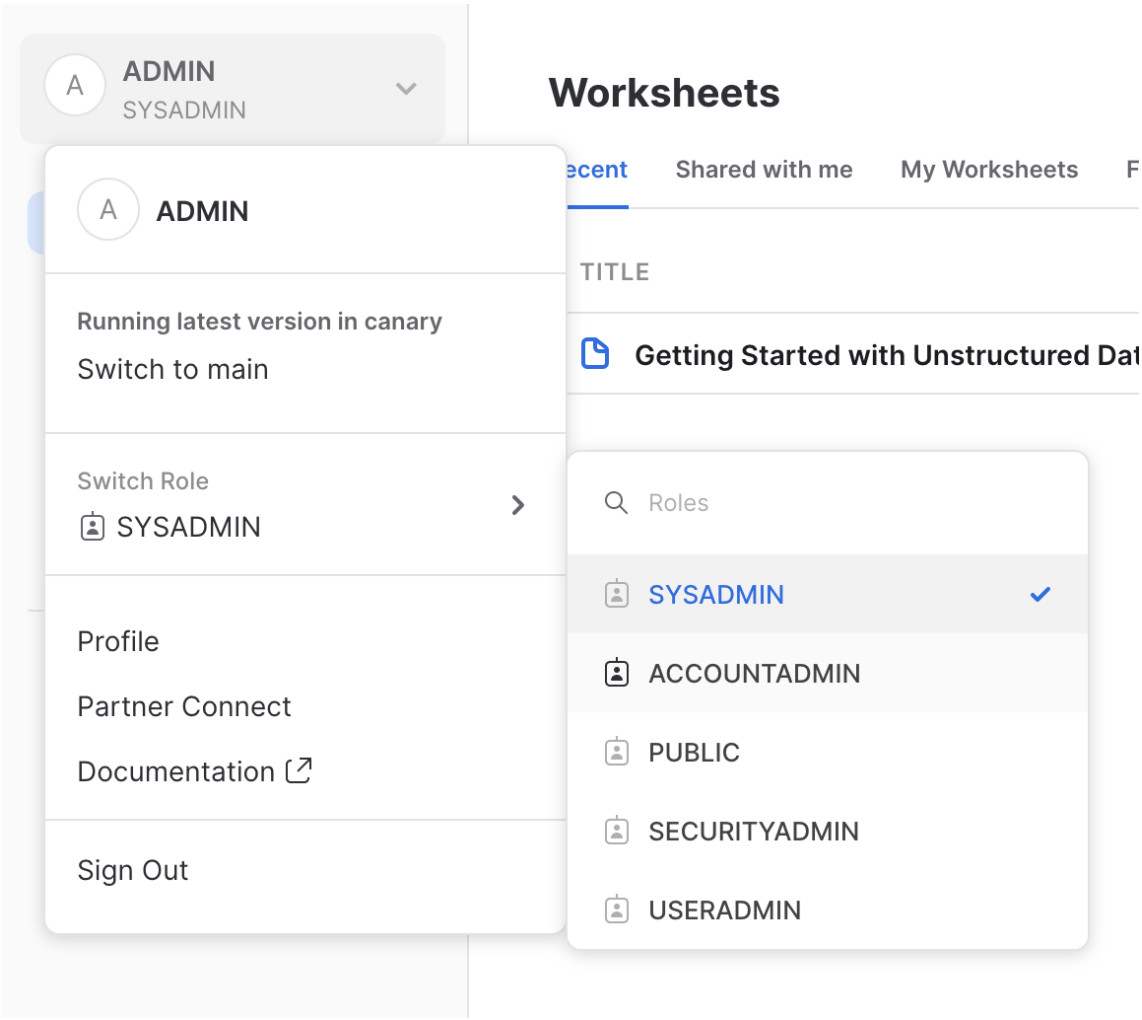
Accessing Shared Data

Switch to the web browser tab where you opened the session with the reader account created in step 6.1 or open a new session on the reader account. Now, click on the blue Snowsight UI button at the top and authenticate again.

After logging in, as this is a new account, click on the worksheet button at the top and create a new virtual warehouse.

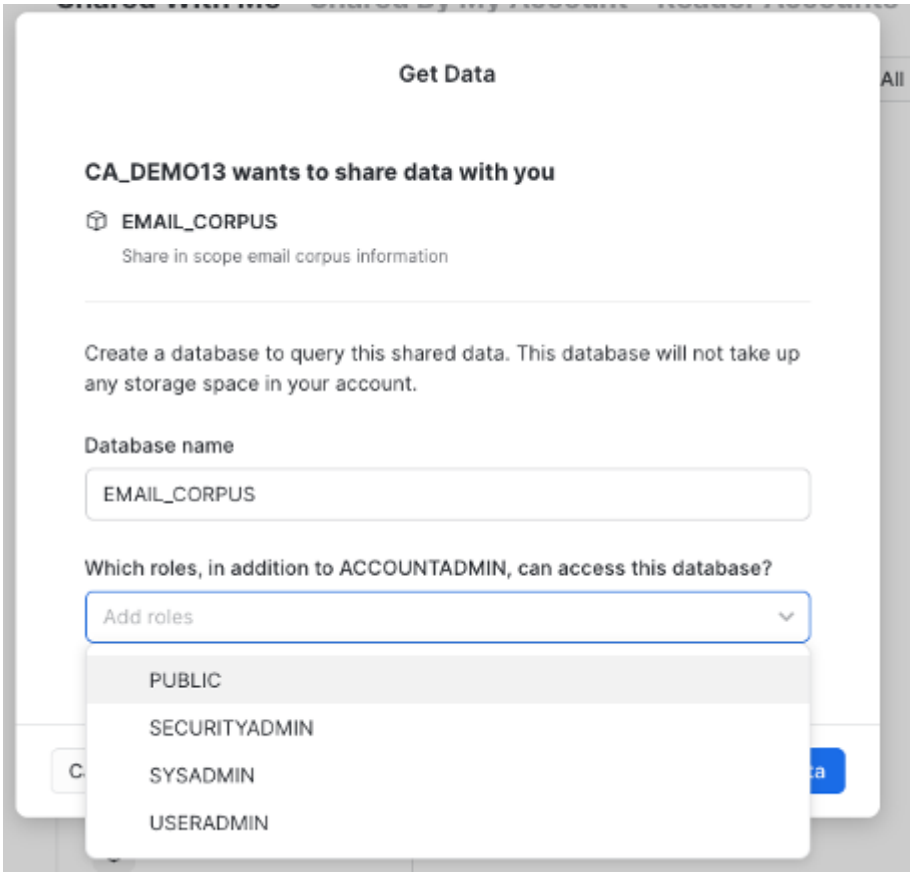
```
use role sysadmin;

create or replace warehouse compute_wh
warehouse_size=xsmall
auto_suspend=1
auto_resume=true
initially_suspended=true;

grant usage on warehouse compute_wh to public;
```

First, let's switch back to the `ACCOUNTADMIN` role. Click on the **Home** button in the top-left. Then in the top-left, click on **ADMIN**, then hover over **Switch Role**, and click on **ACCOUNTADMIN**.

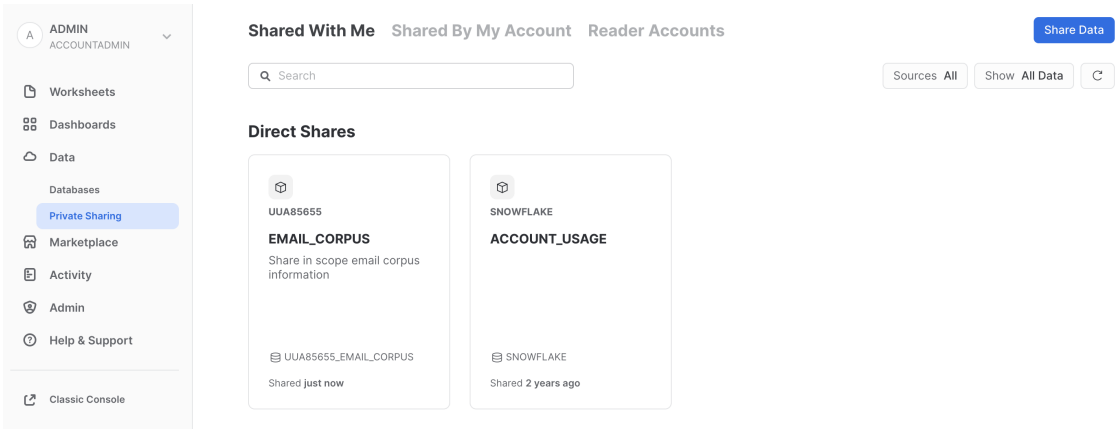


Now let's view the shared data. In the pane on the left, click on on **Data**, then **Private Sharing**. You will see the `EMAIL_CORPUS` database listed under **Ready to Get**. Select it and give the database name `EMAIL_CORPUS` and

make it available to `PUBLIC`, then click the **Get Data** button.



Click on **Databases**, then click on the **Refresh** button (round arrow button on the right side above the database list). You will now see the database `EMAIL_CORPUS`.



Select the worksheet created in this reader account. Add the following commands to set the worksheet session parameters and review the view objects available.

```
use role sysadmin;
use schema <account-locator>_email_corpus.raw;
use warehouse compute_wh;
```

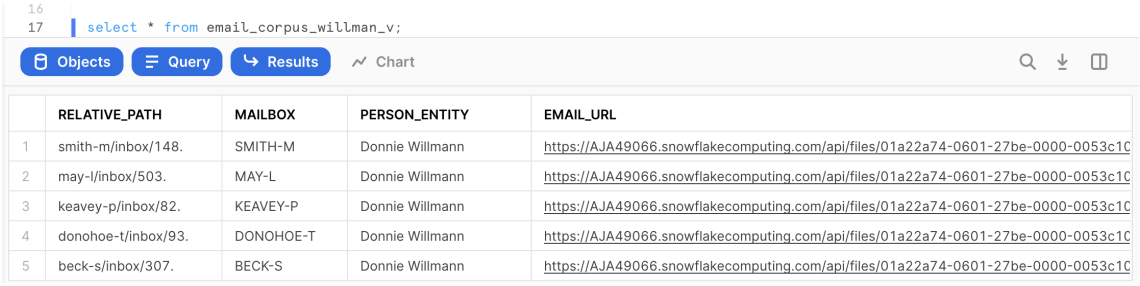
```
show views;
```

If you expand the database hierarchy on the left side of the window, you will see that the share appears from the consumer side as a database `EMAIL_CORPUS`, with a schema `RAW` and a single view object `EMAIL_CORPUS_WILLMAN_V`.

We can now query the shared data. The query below will display only the emails related to 'Willmann'.

```
select * from email_corpus_willman_v;
```

From the results, notice that all the URLs are encrypted, not revealing any information of the location where the shared data came from. Click on any `EMAIL_URL` and get access to the actual email text downloaded to your workstation. Download and review the email. Make sure it is valid.



	RELATIVE_PATH	MAILBOX	PERSON_ENTITY	EMAIL_URL
1	smith-m/inbox/148.	SMITH-M	Donnie Willmann	https://AJA49066.snowflakecomputing.com/api/files/01a22a74-0601-27be-0000-0053c1C
2	may-l/inbox/503.	MAY-L	Donnie Willmann	https://AJA49066.snowflakecomputing.com/api/files/01a22a74-0601-27be-0000-0053c1C
3	keavey-p/inbox/82.	KEAVEY-P	Donnie Willmann	https://AJA49066.snowflakecomputing.com/api/files/01a22a74-0601-27be-0000-0053c1C
4	donohoe-t/inbox/93.	DONOHOE-T	Donnie Willmann	https://AJA49066.snowflakecomputing.com/api/files/01a22a74-0601-27be-0000-0053c1C
5	beck-s/inbox/307.	BECK-S	Donnie Willmann	https://AJA49066.snowflakecomputing.com/api/files/01a22a74-0601-27be-0000-0053c1C

You have now completed this demonstration of how unstructured data can be securely shared in the Snowflake Data Cloud using Snowflake Data Sharing capabilities.

Conclusion

Congratulations! You used Snowflake to perform natural language processing on email files.

What we've covered

- Accessing external data with an **External Stage**
- Storing unstructured data with an **Internal Stage** and **SnowSQL**
- Governing unstructured data with **Role-Based Access Control**
- Catalog unstructured data with **Directory Tables**
- Securely access unstructured data with **Scoped, File, and Pre-signed URLs**
- Processing unstructured data with a **Java UDF**
- Sharing unstructured data in the **Data Cloud**

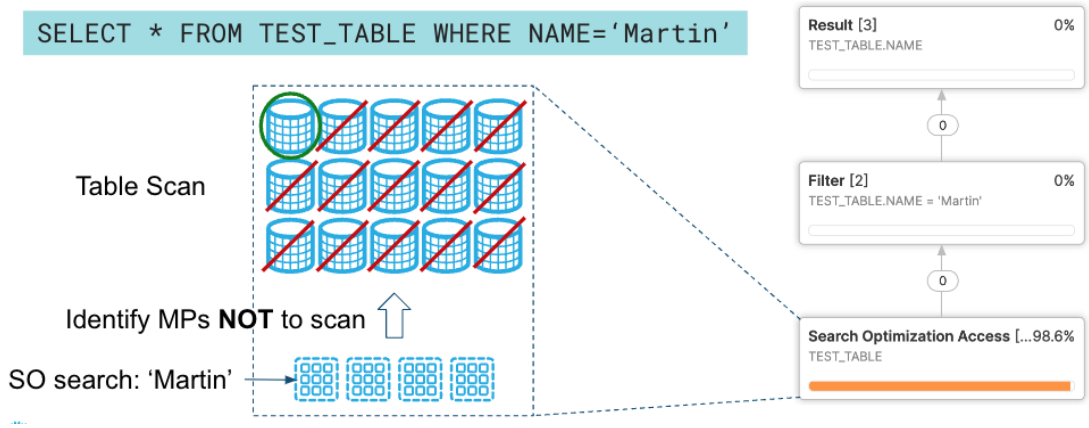
Getting started with Search Optimization

Overview

Are you looking to significantly improve the performance of point lookup queries and certain analytical queries that require fast access to records? Search Optimization Service will help you achieve exactly that.

We create optimized search access paths for the columns in your tables. We take advantage of those optimized paths in addition to other processing enhancements to reduce the number of micro partitions scanned and hence speed up the queries.

For example, in the picture below, we have a query that is trying to find all rows where the name is Martin in a table. If Search Optimization is enabled, it helps identify ***the micro partitions that don't contain 'Martin' in the name column*** and reduces the number of partitions to be scanned. In this particular example, it reduces the number of partitions to be scanned from 15 to 1.



Prerequisites

A basic knowledge of how to run and monitor queries in the Snowflake Web UI.

What you'll learn

- How to acquire a suitable dataset from Snowflake Marketplace
- How to enable Search Optimization
- What the performance impact of enabling Search Optimization on different queries is

What You'll Need

- A supported [browser]
- A Snowflake account with the Enterprise Edition
  - Sign-up using Snowflake Trial

  OR
  - Get access to an existing Snowflake Enterprise Edition account with the `ACCOUNTADMIN` role or the `IMPORT SHARE` privilege

What You'll Build

Performant queries that explore the data from wikidata datasource. Wikidata is a free, collaborative, multilingual knowledge graph. It is a document-oriented database, focused on items, which represent any kind of topic, concept, or object. More information can be found at https://en.wikipedia.org/wiki/Wikidata

Negative: The Marketplace data used in this guide changes from time-to-time, and your query results may be slightly different than indicated in this guide. Additionally, the Snowflake UI changes periodically as well, and instructions/screenshots may be out of date.

Setup Snowflake Account and Virtual Warehouse

The first step in the guide is to set up or log into Snowflake and set up a virtual warehouse if necessary.

Access Snowflake's Web UI

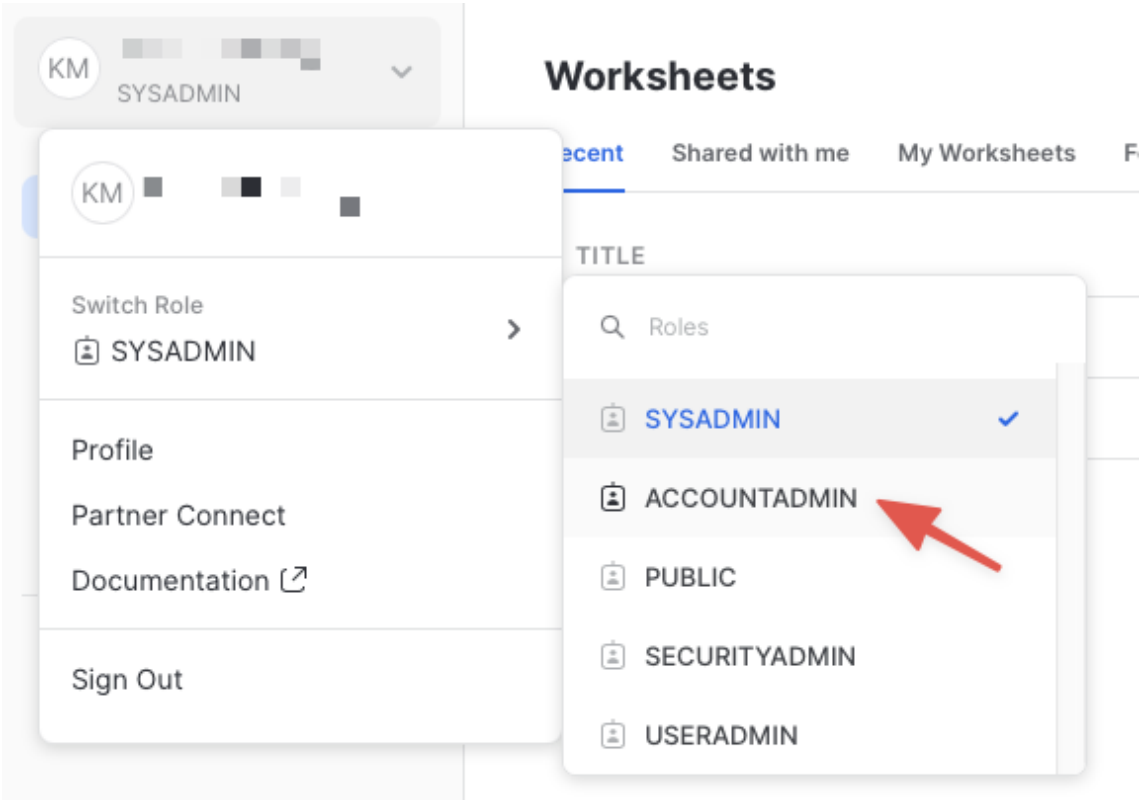
app.snowflake.com

If this is the first time you are logging into the Snowflake UI, you will be prompted to enter your account name or account URL that you were given when you acquired a trial. The account URL contains your account name and potentially the region. Click `Sign-in` and you will be prompted for your user name and password.

If this is not the first time you are logging into the Snowflake UI, you should see a `Select an account to sign into` prompt and a button for your account name listed below it. Click the account you wish to access and you will be prompted for your user name and password (or another authentication mechanism).

Switch to the appropriate role

The Snowflake web interface has a lot to offer, but for now, switch your current role from the default `SYSADMIN` to `ACCOUNTADMIN`.

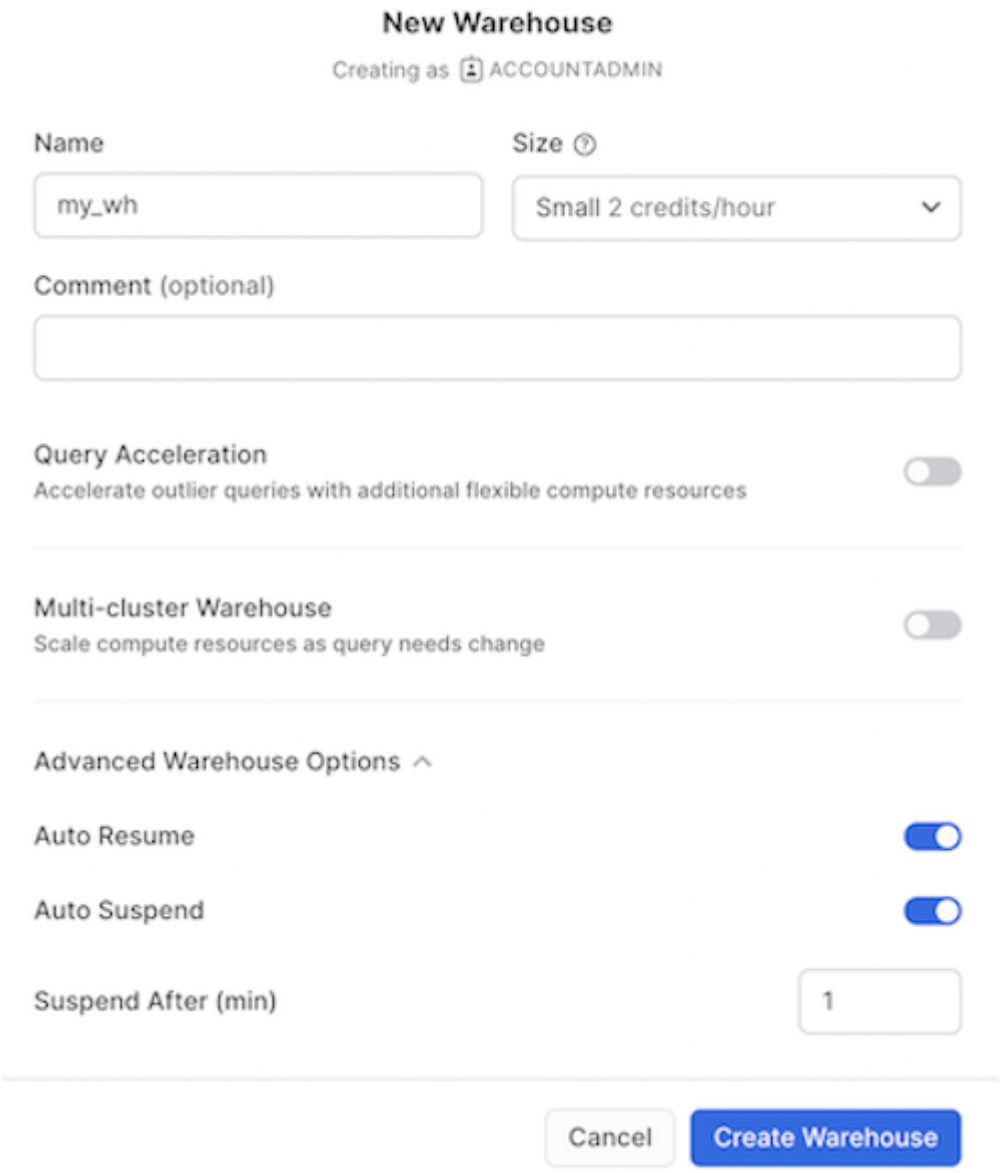


This will allow you to create shared databases from Snowflake Marketplace listings. If you don't have the `ACCOUNTADMIN` role, switch to a role with `IMPORT SHARE` privileges instead.

Create a Virtual Warehouse (if needed)

If you don't already have access to a Virtual Warehouse to run queries, you will need to create one.

- Navigate to the Compute > Warehouses screen using the menu on the left side of the window
- Click the big blue + Warehouse button in the upper right of the window
- Create a Small Warehouse as shown in the screen below



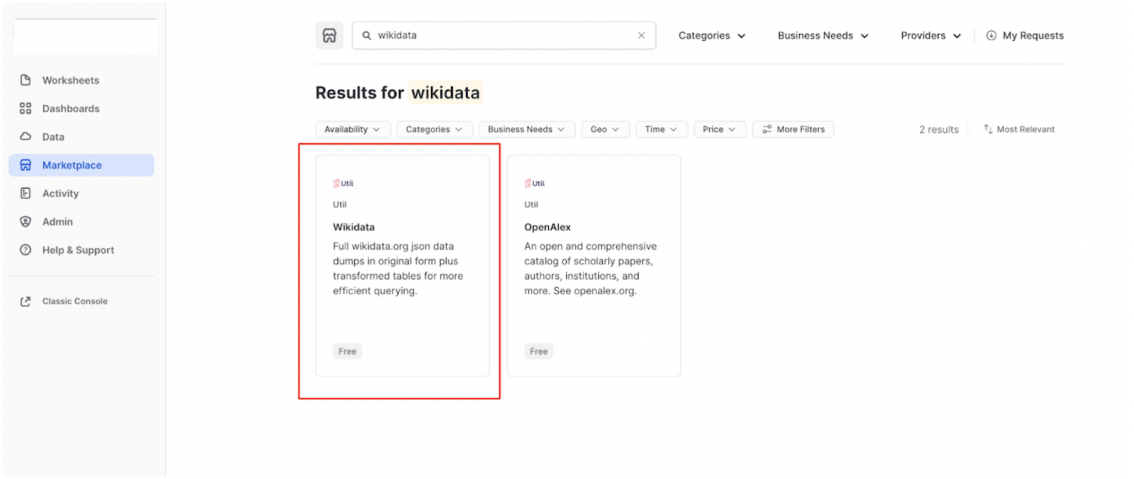
Be sure to change the `Suspend After (min) field` to 1 min to avoid wasting compute credits.

If you already have access to a Virtual Warehouse to run queries, make sure to scale it up or down to `Small` Warehouse for this guide.

Acquiring Data from Snowflake Marketplace

The next step is to acquire data that has all data types supported by Search Optimization. The best place to acquire this data is the Snowflake Marketplace.

- Navigate to the `Marketplace` screen using the menu on the left side of the window
- Search for `Wikidata` in the search bar
- Find and click the `Util Wikidata` tile



- Once in the listing, click the big blue Get Data button

On the `Get Data` screen, you may be prompted to complete your user profile if you have not done so before. Enter your name and email address into the profile screen and click the blue `Save` button. You will be returned to the `Get Data` screen.

Congratulations! You have just created a shared database named `WIKIDATA` from a listing on the Snowflake Marketplace. Click the big blue `Query Data` button and advance to the next step in the guide.

Data Setup

The prior section opened a worksheet editor in the new Snowflake UI with a few pre-populated queries that came from the sample queries defined in the Marketplace listing. You are not going to run any of these queries in this guide, but you are welcome to run them later.

Understanding the data

You are going to first copy over two of the tables from `WIKIDATA` (the database that you just imported) into a new database (we will call it `WIKI_SO`).

This is necessary as

- You wouldn't have the privileges to set up Search Optimization on the shared `WIKIDATA` database
- It will allow you to run the same query on both search optimized (in `WIKI_SO` database) and non search optimized tables (in `WIKIDATA` database) to compare the performance of Search Optimization.

The first table we will use is `wikidata_original` , it has information about the wikidata articles such as description, label etc. There are ***96.9 million rows*** in this table.

The second table is `entity_is_subclass_of`, which contains the information about subclass categories like subclass id and subclass name. It is a smaller table and has ***~3.3 million rows***.

To further understand the relationship between these two tables, consider the following query, where we are exploring an entity with Id `Q1968` (which is a Formula One article)

```
SELECT o.label, e.subclass_of_name
    FROM
        entity_is_subclass_of as e
    JOIN wikidata_original as o
    ON e.entity_id = o.id
WHERE
    e.entity_id = 'Q1968'; -- Formula One article
```

The result looks like this:

	LABEL	SUBCLASS_OF_NAME
1	Formula One	formula racing

So, Id `Q1968` is an article about `Formula One` (`LABEL`) and this entity rightly belongs to the subclass `Formula Racing` (`SUBCLASS_OF_NAME`).

Copy the required tables into a new database

Now let's copy over the above two tables into a new database before we enable Search Optimization on them.

Before we run the queries to do so, let's create a new worksheet named `Search Optimization Guide` by clicking the `+ icon` on the left navigation bar. Throughout this guide, we will run the queries on the search optimized tables in the `Search Optimization Guide` worksheet.

Run the query below in the `Search Optimization Guide` worksheet:

```
CREATE DATABASE wiki_so;
CREATE SCHEMA experiments;`

//Note: Substitute my_wh with your warehouse name if different
ALTER WAREHOUSE my_wh set warehouse_size=large;`

CREATE TABLE wiki_so.experiments.wikidata_original AS (SELECT * FROM
wikidata.wikidata.wikidata_original);

CREATE TABLE wiki_so.experiments.entity_is_subclass_of AS (SELECT * FROM
wikidata.wikidata.entity_is_subclass_of);

//Note: Substitute my_wh with your warehouse name if different
ALTER WAREHOUSE my_wh set warehouse_size=small;
```

You should find an output that indicates the number of rows that were copied into the tables. This will take a few minutes to complete.

Enable Search Optimization

Now let's enable Search Optimization for the `wikidata_original` table in the newly created `WIKI_SO` Database (`Search Optimization Guide` Worksheet). We can either enable Search Optimization on the whole table or enable it for a few columns depending on the queries we want to accelerate.

For this guide, let's selectively enable Search optimization for a few columns:

```
// Defining Search Optimization on VARCHAR fields
ALTER TABLE wikidata_original ADD SEARCH OPTIMIZATION ON EQUALITY(id, label,
description);

// Defining Search Optimization on VARCHAR fields optimized for Wildcard search
ALTER TABLE wikidata_original ADD SEARCH OPTIMIZATION ON SUBSTRING(description);

// Defining Search Optimization on VARIANT field
ALTER TABLE wikidata_original ADD SEARCH OPTIMIZATION ON EQUALITY(labels);
```

Ensure Search Optimization first time indexing is complete

Now, let's verify that Search Optimization is enabled and the backend process has finished indexing our data. It might take about 5 minutes for that to happen as the optimized search access paths are being built for these columns by Snowflake.

Run the below query against the newly created database (`WIKI_SO`)

```
DESCRIBE SEARCH OPTIMIZATION ON wikidata_original;
```

It would return a result like below:

expression_id	method	target	target_data_type ···	active
1	EQUALITY	ID	VARCHAR(16777216)	true
2	EQUALITY	LABEL	VARCHAR(16777216)	true
3	EQUALITY	DESCRIPTION	VARCHAR(16777216)	true
4	SUBSTRING	DESCRIPTION	VARCHAR(16777216)	true
5	EQUALITY	LABELS:	VARIANT	true

Make sure that all the rows have the `active` column set to `true` before proceeding further in this guide.

Now you are all set up to run some queries and dive deep into Search Optimization.

We have intentionally enabled Search Optimization for `wikidata_original` table and not `entity_is_subclass_of` table for this guide.

Negative: Note: Please note that the results, query time, partitions or bytes scanned might differ when you run the queries in comparison to the values noted below as the data gets refreshed monthly in the above two tables.

Equality and Wildcard Search

Now let's build some queries and observe how Search Optimization helps optimize them.

To start off, we have already enabled Search Optimization on the `LABEL` and `DESCRIPTION` fields for equality and substring predicates respectively in the previous section.

expression_id	method	target	target_data_type	...	active
1	EQUALITY	ID	VARCHAR(16777216)		true
2	EQUALITY	LABEL	VARCHAR(16777216)		true
3	EQUALITY	DESCRIPTION	VARCHAR(16777216)		true
4	SUBSTRING	DESCRIPTION	VARCHAR(16777216)		true
5	EQUALITY	LABELS:	VARIANT		true

Negative: Note: If you wish to run the queries below on both databases (`WIKIDATA` and `WIKI_SO`) to evaluate performance impact, please make sure to run the commands below before you switch from one database to another. This will ensure that no cached results (hot or warm) are used.

```
ALTER SESSION SET USE_CACHED_RESULT = false;
ALTER WAREHOUSE my_wh SUSPEND;
```

Now, let's say you want to find all the articles about the `iPhone` which have the words `wikimedia or page` in the description (in that order). The query would look like:

```
SELECT *
  FROM wikidata_original
  WHERE
    label= 'iPhone' AND
    description ILIKE '%wikimedia%page%';
```

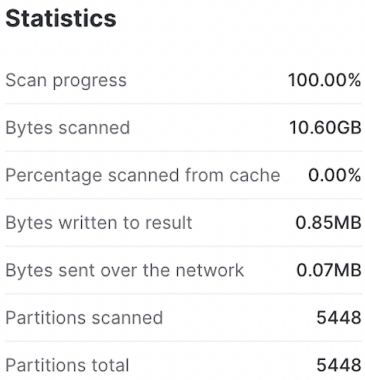
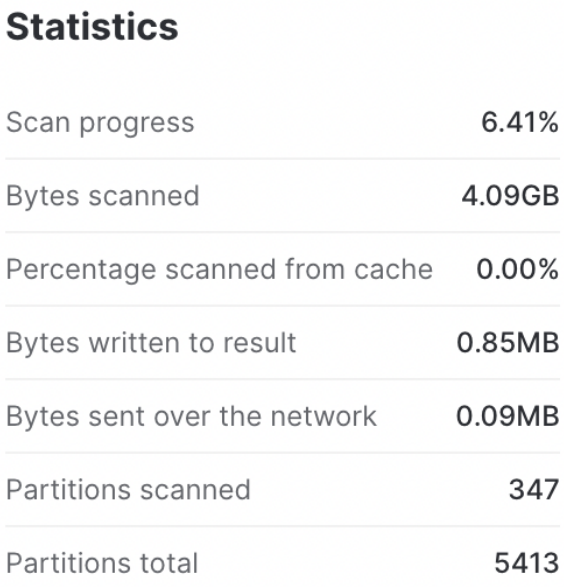
Without search optimization	With Search Optimization
It takes 28 seconds to run the query on the table without search optimization. the other interesting aspect is, ***almost all partitions*** need to be scanned. also you will note that ~ 23.01gb data is scanned. following are the full statistics **Statistics** Scan progress 100.00% Bytes scanned 23.01GB Percentage scanned from cache 0.00% Bytes sent over the network 0.02MB Partitions scanned 5443 Partitions total 5448	On the other hand, the query takes 5.7 seconds on the search optimized table. you will notice that ***only 7 partitions*** of the total 5413 partitions are scanned. in addition only 31.79mb of the data needs to be scanned. **Statistics** Scan progress 0.13% Bytes scanned 31.79MB Percentage scanned from cache 0.00% Bytes sent over the network 0.11MB Partitions scanned 7 Partitions total 5413

Looking at the numbers side by side, we know that Search Optimization has definitely improved the query performance.

	Without Search Optimization	With Search Optimization	Performance Impact
Query run time	28 seconds	5.7 seconds	***79.64% improvement*** in query speed
Percentage of partitions scanned	99.91%	0.13%	***99.78% less partitions*** scanned
Bytes scanned	23.01GB	31.79MB	***99.86% less data*** scanned

Let's look at another example. Say, you want to find all articles which have the words `blog post` in their description, following would be the query to do so:

```
SELECT *
  FROM wikidata_original
  WHERE
    description ILIKE '%blog post%';
```

Without search optimization	With Search Optimization
The query runs for 23 seconds and ***ALL partitions*** are scanned. Also, 10.60GB of data is scanned. See the picture below for full details. 	On the other hand, the query runs in 8.7 seconds on the Search Optimized table. You'll also notice that ***only 347 partitions*** of the total 5413 partitions are scanned. In addition 4.09GB of the data was scanned. See the picture below for full details. 

*As you can see from the **Performance Impact** column above, using Search Optimization allows us to make significant improvements in query performance.*

	Without Search Optimization	With Search Optimization	Performance Impact
Query run time	23 seconds	8.7 seconds	***62.17% improvement*** in

			query speed
Percentage of partitions scanned	100%	6.41%	***99.59% less partitions*** scanned
Bytes scanned	10.60GB	4.09GB	***61.42% less data*** scanned

Searching in Variant data

In this section, let's search in the variant data and analyze how Search Optimization helps in these cases.

Negative: Note: If you wish to run the queries below on both databases (`WIKIDATA` and `WIKI_SO`) to evaluate performance impact, please make sure to run the commands below before you switch from one database to another. This will ensure that no cached results (hot or warm) are used.

```
ALTER SESSION SET USE_CACHED_RESULT = false;
ALTER WAREHOUSE my_wh SUSPEND;
```

To start off, we have already enabled Search Optimization on the `Labels` field which is an unstructured JSON.

expression_id	method	target	target_data_type	...	active
1	EQUALITY	ID	VARCHAR(16777216)		true
2	EQUALITY	LABEL	VARCHAR(16777216)		true
3	EQUALITY	DESCRIPTION	VARCHAR(16777216)		true
4	SUBSTRING	DESCRIPTION	VARCHAR(16777216)		true
5	EQUALITY	LABELS:	VARIANT		true

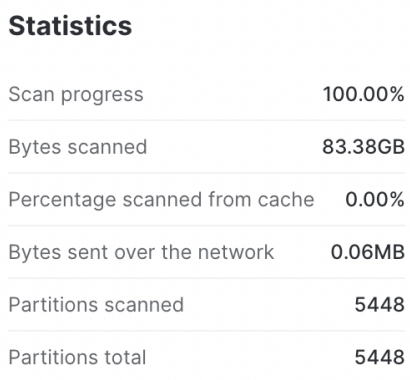
Let's say you want to find all entries where the label is set to `National Doughnut Day` in the `English version` of the article. To do so, you can run the following query:

```
SELECT *
  FROM wikidata_original
  WHERE labels:en:value = 'National Doughnut Day';
```

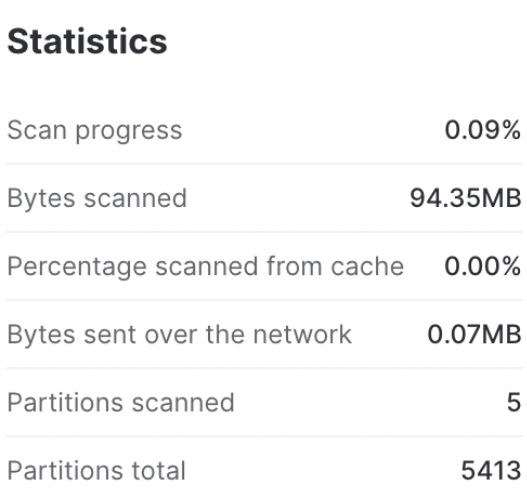
The above query returns ***2 rows out of 96.9 million rows***.

Without search optimization	With Search Optimization
The query runs for 42 seconds on the shared database. You will also see that ***ALL partitions*** need to be scanned. In addition, ~83.38GB of	On the other hand, it takes 5.2 seconds to run the same query on the search optimized table. You will also notice that ***only 5 partitions*** of the total 5413 partitions are

data was scanned.



scanned. In addition only 94.25MB of the data was scanned.



*From the **Performance Impact** column below, we see that using Search Optimization allows us to make significant improvements in query performance*

	Without Search Optimization	With Search Optimization	Performance Impact
Query run time	42 seconds	5.2 seconds	***87.62% improvement*** in query speed
Percentage of partitions scanned	100%	0.09%	***99.91% less partitions*** scanned
Bytes scanned	83.38GB	94.35MB	***99.80% less data*** scanned

Accelerating Joins

The search optimization service can improve the performance of queries that join a small table with a large table.

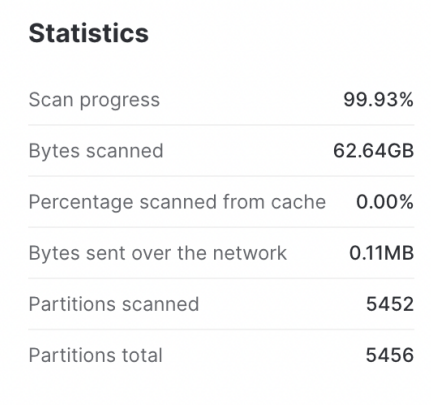
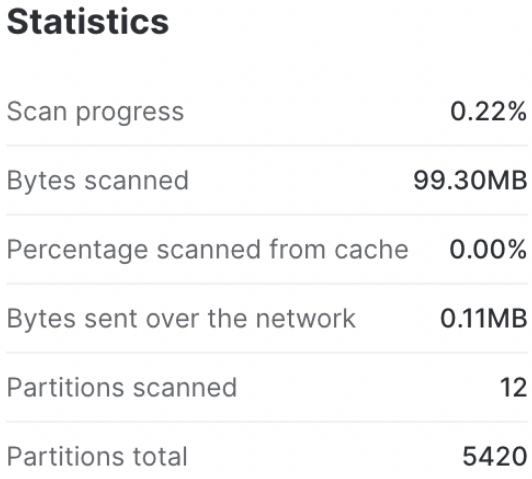
Note: In data warehousing, the large table is often referred to as the fact table. The small table is referred to as the dimension table. The rest of this topic uses these terms when referring to the large table and the small table in the join.

To enable the search optimization service to improve the performance of joins, you need to add Search Optimization to the fact table (the larger of the two tables). In addition, the dimension table (the smaller of the two tables) should have few distinct values. In our guide, `wikidata_original` is the fact table whereas `entity_is_subclass_of` is the dimension table.

Let's say you want to find out the Subclass ID of all articles related to `'Formula One'`. Say we know the following entity ids in the wikidata_original table mapping to 'Formula One' articles are => `'Q1437617','Q8564669','Q1968' and 'Q5470299'`

So, the query to find the `'Formula One'` subclass Ids would look like below:

```
SELECT *
  FROM entity_is_subclass_of AS e
  JOIN wikidata_original AS o ON (e.subclass_of_name = o.label)
  WHERE e.entity_id IN ('Q1437617','Q8564669','Q1968','Q5470299') ;
```

Without search optimization	With Search Optimization
It takes ~43 seconds and scans nearly ALL partitions and about 64.64GB of data to find the resulting Subclass ID. See the picture below for full details. 	On the other hand, the query on the search optimized table, returns the result (subclass_of_id => Q1199515) in 4.4 seconds. Also, only a small portion of data is scanned to find the answer (12 partitions and 99.30MB of data is scanned). 

If we compare the statistics side by side, we can observe that Search Optimization greatly optimized the JOIN query.

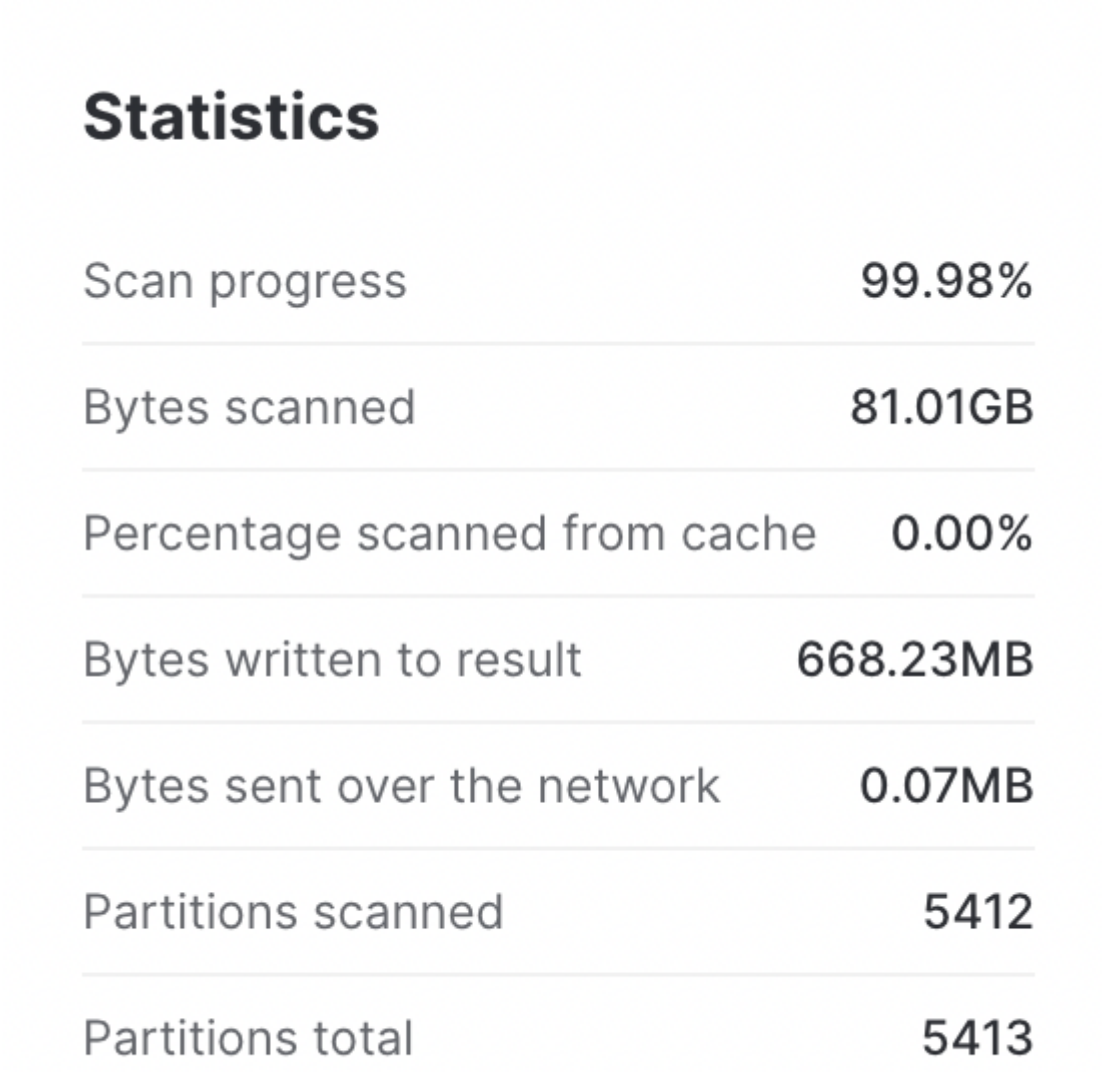
	Without Search Optimization	With Search Optimization	Performance Impact
Query run time	43 seconds	4.4 seconds	***92.09% improvement*** in query speed
Percentage of partitions scanned	99.92%	0.22%	***99.70% less partitions*** scanned
Bytes scanned	62.64GB	99.30MB	***99.84% less data*** scanned

Queries that are not benefitting from Search Optimization

Not all queries benefit from Search Optimization. One such example is the following query to get all entries that have description with the words wikimedia and page in that order. The query would look like:

```
SELECT *
  FROM WIKIDATA_ORIGINAL
  WHERE description ILIKE '%wikimedia%page%';
```

The following query returns ***1.4 Million rows***. As shown in the snapshot below, ***only 1 out of the 5413 partitions is skipped*** when you run the query on the search optimized `wikimedia_original` table in our newly created `WIKI_SO` database .



Such queries aren't benefitted from Search Optimization as the number of partitions that can be skipped by Search Optimization Service are very minimal.

Getting Started with Streams & Tasks

Overview

This guide will take you through a scenario of using Snowflake's Tasks and Streams capabilities to ingest a stream of data and prepare for analytics.

Streams provides a change tracking mechanism for your tables and views, enabling and ensuring "exactly once" semantics for new or changed data.

Tasks are Snowflake objects to execute a single command, which could be simple SQL command or calling an extensive stored procedure. Tasks can be scheduled or run on-demand, either within a Snowflake Virtual warehouse or serverless.

This Lab will also construct a Directed Acyclic Graph (DAG), which is a series of tasks composed of a single root task and additional tasks, organized by their dependencies. Tasks will be combined with table streams to create data pipelines, continuous ELT workflows to process recently received or changed table rows.

A simulated streaming datafeed will be used for this exercise, using a Snowpark-based Stored Procedure, to simplify your setup and focus on these two capabilities. The simulation will be high-volume, at 1 million transactions a minute (exceeding 15k/second), of credit card purchases and returns. This prerequisite streaming ingestion was modeled to mirror one created from Snowflake's Kafka Connector, without the distraction of having to setup a running Kafka instance.
While not covered in this exercise, one can use these building blocks to further enrich your data with Snowflake Data Marketplace data, train and deploy machine learning models, perform fraud detection, and other use cases by combining these skills with other Snowflake virtual hands-on labs.

Prerequisites

- Familiarity with Snowflake, basic SQL knowledge, Snowsight UI and Snowflake objects

What You'll Learn

- how to create a Snowflake Stream
- how to create and schedule a Snowflake Task
- how to orchestrate tasks into data pipelines
- how Snowpark can be used to build new types of user-defined functions and stored procedures
- multiple ways to manage and monitor your Snowflake tasks and data pipelines

What You'll Need

To participate in the virtual hands-on lab, attendees need the following:

- A Snowflake Enterprise Account on preferred AWS region with **ACCOUNTADMIN** access
- Be able to download a SQL File, or will have to copy/paste each command from this Guide

What You'll Build

- A Snowflake database that contains all data and objects built in this lab
- A Stage and Staging table to initially land your incoming data stream
- An Analytics-Ready Table
- A Stream to track recently-received credit card transactions
- Tasks orchestrated two ways to process those new transactions

Setting up Snowflake

Download

The first thing you will need to do is download the following .sql file that contains a series of SQL commands we will execute throughout this lab. **Click the green button to download the file**

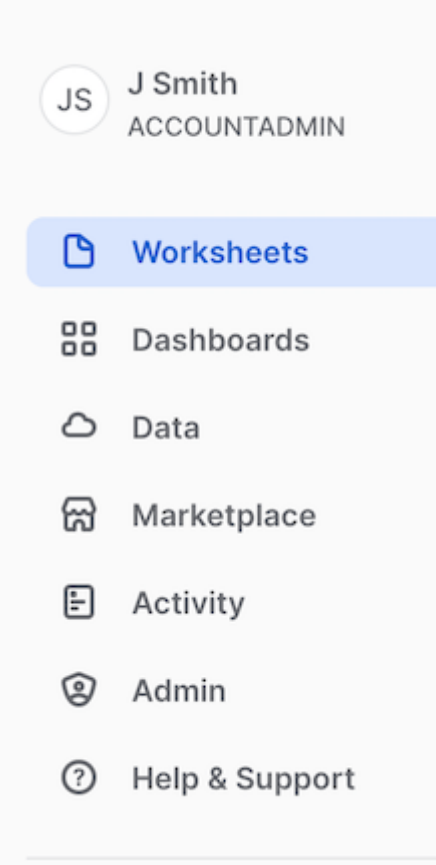
Data_Engineering_Streams_Tasks_VHOL.sql

Login

At this point log into your Snowflake. If you have just created a free trial account, feel free to minimize or close and hint boxes that are looking to help guide you. These will not be needed for this lab and most of the hints will be covered throughout the remainder of this exercise.

Create a Worksheet

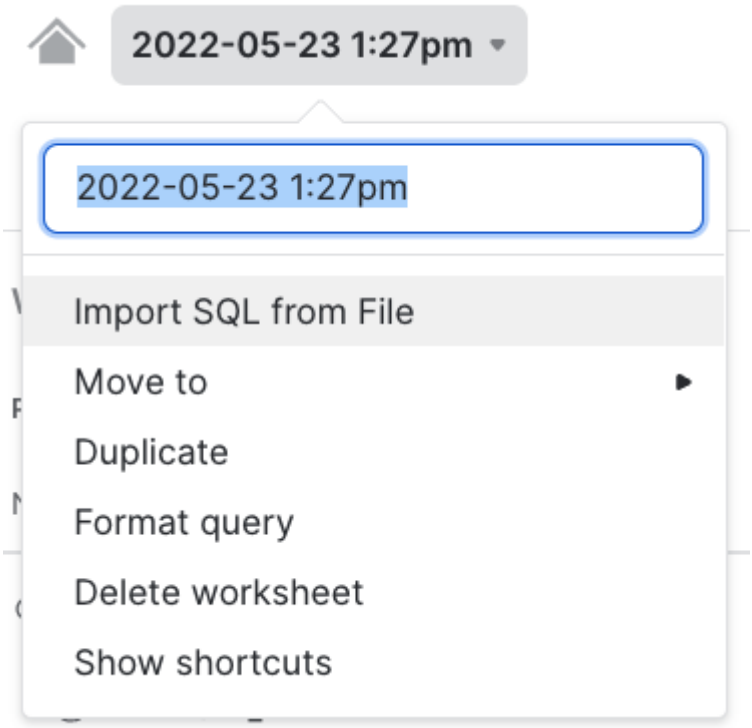
In the Snowflake UI click on **Worksheets** on the left side.



Create a new Worksheet by clicking on the new worksheet button (+ Worksheet) on the top right side.

To ingest SQL file in the Snowflake UI, navigate to the worksheet name (the worksheet name will be the date and time) on the top left hand side and click on the small down arrow next to it. This will give you the option to change

the worksheet name and also the option to import a SQL scrpt file. Click on **Import SQL from File**



You can now select the .sql file you downloaded and named earlier.

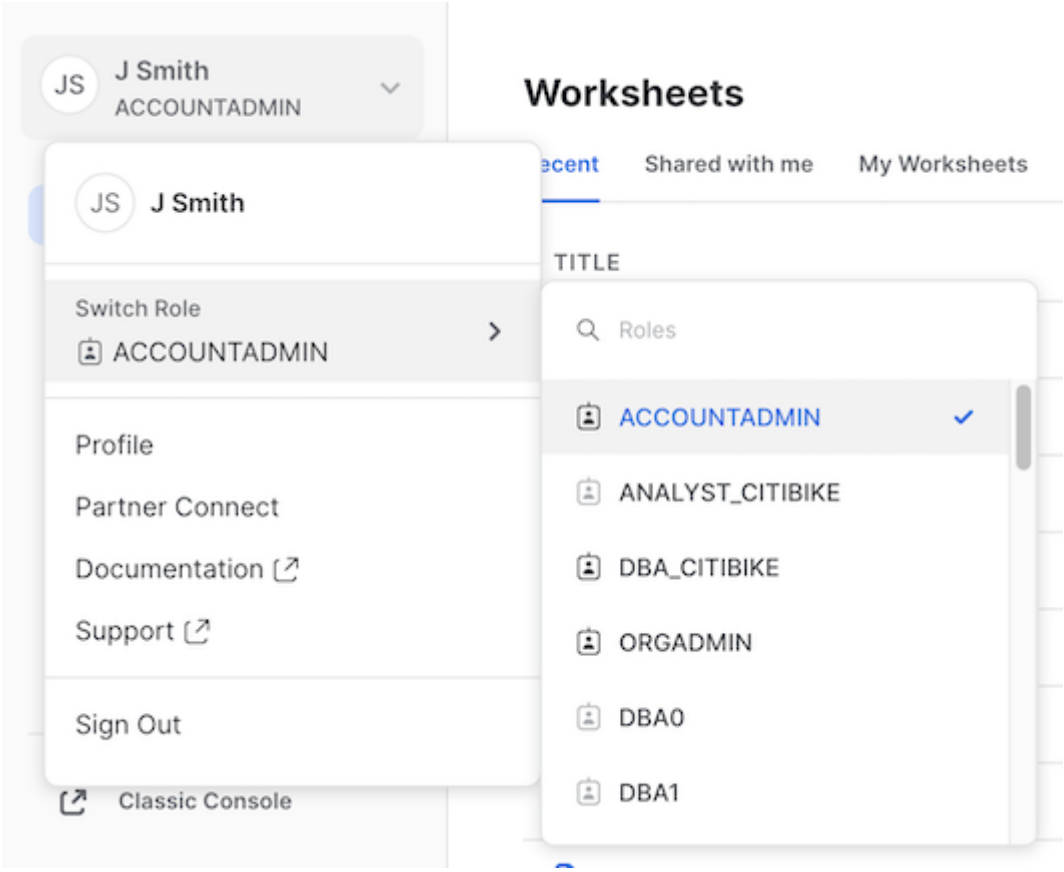
The SQL script file should show up as text in a new worksheet. You may need to scroll to the top of the file to start executing commands.

Each step throughout the Snowflake portion of the guide has an associated SQL command to perform the work we are looking to execute, and so feel free to step through each action running the code one command at-a-time as we walk through the lab.

Set your Role

Finally, switch to the ACCOUNTADMIN role. If you just created an evaluation account to go through this Lab, this should be easy. However, if you are using an established account and find this role missing from your list, you may need assistance to complete the next few steps. Creating a Role, Database, Stages, Tasks, and monitoring tasks

executed by 'System' requires higher-level permissions.



Begin Construction

Create Foundational Snowflake Objects for this Hands-on Lab

a) Create a new role for this Lab and grant permissions

```
use role ACCOUNTADMIN;
set myname = current_user();
create role if not exists VHOL;
grant role VHOL to user identifier($myname);
grant create database on account to role VHOL;
grant EXECUTE TASK, EXECUTE MANAGED TASK on ACCOUNT to role VHOL;
grant IMPORTED PRIVILEGES on DATABASE SNOWFLAKE to role VHOL;
```

b) Create a Dedicated Virtual Compute Warehouse

Size XS, dedicated for this Hands-on Lab

```
create or replace warehouse VHOL_WH WAREHOUSE_SIZE = XSMALL, AUTO_SUSPEND = 5,
AUTO_RESUME= TRUE;
grant all privileges on warehouse VHOL_WH to role VHOL;
```

c) Create Database used throughout this Lab

```
use role VHOL;
create or replace database VHOL_ST;
grant all privileges on database VHOL_ST to role VHOL;
use database VHOL_ST;
use schema PUBLIC;
use warehouse VHOL_WH;
```

d) Create an internal Stage

Dedicated for incoming streaming files (Typically real-time stream consumption would be automated from Kafka using Snowflake Kafka Connector, but for this exercise we are simulating data and focusing on Task and Stream usage.). Incoming data will JSON format, with many transactions within each file.

```
create or replace stage VHOL_STAGE
FILE_FORMAT = ( TYPE=JSON,STRIP_OUTER_ARRAY=TRUE );
```

d) Create a Staging/Landing Table

Where all incoming data will land initially. Each row will contain a transaction, but JSON will be stored as a VARIANT datatype within Snowflake.

```
create or replace table CC_TRANS_STAGING (RECORD_CONTENT variant);
```

Simulated Stream Source

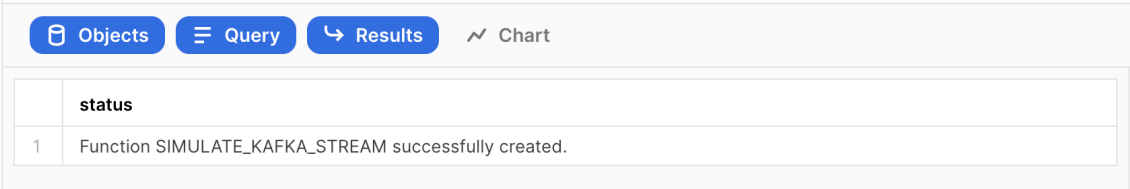
Create Simulation Data Generation Stored Procedure (Using Snowpark Java). We kept this as a separate step, as it is necessary for setup, but deep interrogation of how this Stored Procedure works to use Snowflake as a streaming ingestion process is not the focus of this Lab. Just copy, paste and run this to create the Stored Procedure:

```
create or replace procedure SIMULATE_KAFKA_STREAM(mystage STRING,prefix
STRING,numlines INTEGER)
  RETURNS STRING
  LANGUAGE JAVA
  PACKAGES = ('com.snowflake:snowpark:latest')
  HANDLER = 'StreamDemo.run'
  AS
  $$
    import com.snowflake.snowpark_java.Session;
    import java.io.*;
    import java.util.HashMap;
    public class StreamDemo {
      public String run(Session session, String mystage,String prefix,int numlines) {
        SampleData SD=new SampleData();
        BufferedWriter bw = null;
        File f=null;
        try {
            f = File.createTempFile(prefix, ".json");
            FileWriter fw = new FileWriter(f);
            bw = new BufferedWriter(fw);
            boolean first=true;
            bw.write("[");
            for(int i=1;i<=numlines;i++){
```

```
                if (first) first = false;
                else {bw.write(",");bw.newLine();}
                bw.write(SD.getDataLine(i));
            }
            bw.write("]");
            bw.close();
            return session.file().put(f.getAbsolutePath(),mystage,options)
[0].getStatus();
        }
        catch (Exception ex){
            return ex.getMessage();
        }
        finally {
            try{
                if(bw!=null) bw.close();
                if(f!=null && f.exists()) f.delete();
            }
            catch(Exception ex){
                return ("Error in closing:  "+ex);
            }
        }
      }

      private static final HashMap<String,String> options = new HashMap<String,
String>() {
        { put("AUTO_COMPRESS", "TRUE"); }
      };

      // sample data generator (credit card transactions)
    public static class SampleData {
      private static final java.util.Random R=new java.util.Random();
      private static final java.text.NumberFormat NF_AMT =
java.text.NumberFormat.getInstance();
      String[] transactionType=
{"PURCHASE","PURCHASE","PURCHASE","PURCHASE","PURCHASE","PURCHASE","PURCHASE","PURCHASE'

      String[] approved=
{"true","true","true","true","true","true","true","true","true","true","false"};
      static {
        NF_AMT.setMinimumFractionDigits(2);
        NF_AMT.setMaximumFractionDigits(2);
        NF_AMT.setGroupingUsed(false);
      }

      private static int randomQty(int low, int high){
        return R.nextInt(high-low) + low;
      }

      private static double randomAmount(int low, int high){
        return R.nextDouble()*(high-low) + low;
      }
```

```
      private String getDataLine(int rownum){
        StringBuilder sb = new StringBuilder()
             .append("{")
             .append("\"element\":"+rownum+",")
             .append("\"object\":\"basic-card\",")
             .append("\"transaction\":{")
             .append("\"id\":"+(1000000000 + R.nextInt(900000000))+",")

.append("\"type\":"+"\""+transactionType[R.nextInt(transactionType.length)]+"\",")
             .append("\"amount\":"+NF_AMT.format(randomAmount(1,5000)) +",")
             .append("\"currency\":"+"\"USD\",")
             .append("\"timestamp\":\""+java.time.Instant.now()+"\",")
             .append("\"approved\":"+approved[R.nextInt(approved.length)]+"")
             .append("},")
             .append("\"card\":{")
                 .append("\"number\":"+ java.lang.Math.abs(R.nextLong()) +"")
             .append("},")
             .append("\"merchant\":{")
             .append("\"id\":"+(100000000 + R.nextInt(90000000))+"")
             .append("}")
             .append("}");
        return sb.toString();
      }
    }
}
$$;
```

Which will return:



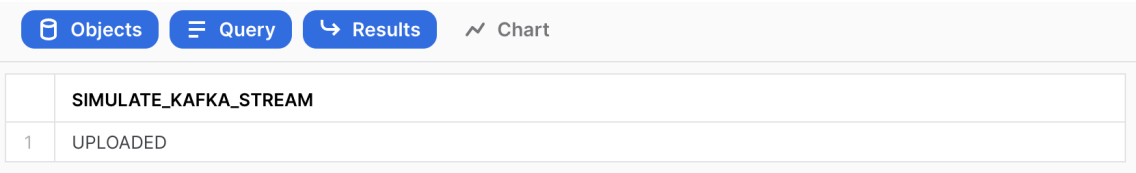
Develop and Testing

a) Call SP to generate the compressed JSON load file

Later, this will be setup to run repetitively on a schedule to simulate a real-time stream ingestion process. First, we run the stored procedure on-demand using:

```
call SIMULATE_KAFKA_STREAM('@VHOL_STAGE','SNOW_',1000000);
```

Which returns:



b) Verify file was created in the internal stage

```
list @VHOL_STAGE PATTERN='.*SNOW_.*';
```

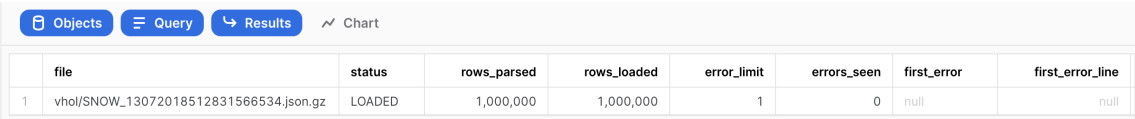
This file looks similar to this in your Stage:

	name	size	md5	last_modified
1	vhol/SNOW_13072018512831566534.json.gz	37,517,248	396ac114116ca4524d8cbb32a1a76bce	Tue, 23 Aug 2022 14:33:52 GMT

c) Load file into Staging Table (about 100Mb raw json data per file). Later, this will be setup to run every x minutes.

```
copy into CC_TRANS_STAGING from @VHOL_STAGE PATTERN='.*SNOW_.*';
```

Which will return this:

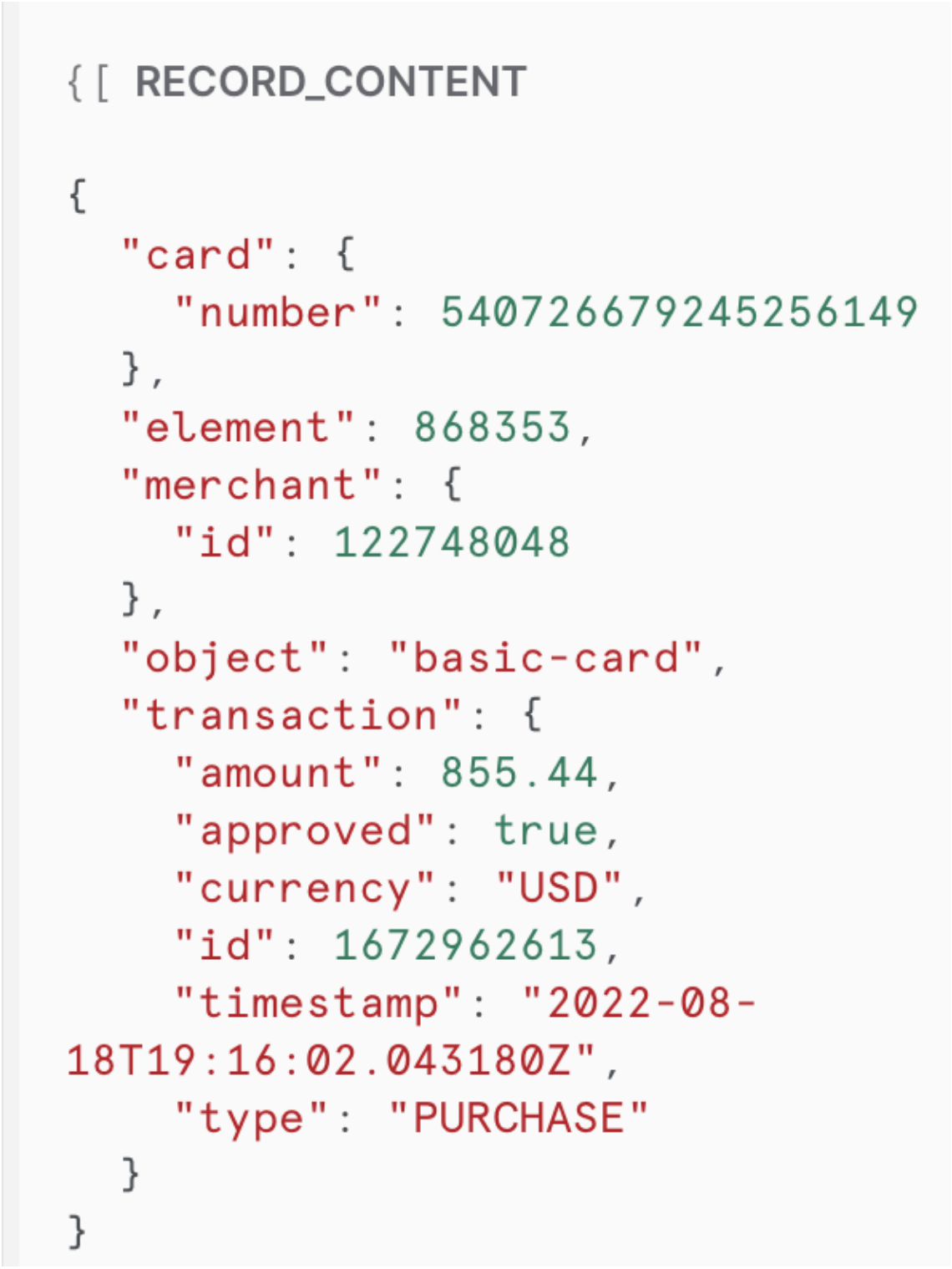


	file	status	rows_parsed	rows_loaded	error_limit	errors_seen	first_error	first_error_line
1	vhol/SNOW_13072018512831566534.json.gz	LOADED	1,000,000	1,000,000	1	0	null	null

d) Now, there should be raw source JSON data in our Landing/Staging Table, but now a VARIANT datatype.

```
select count(*) from CC_TRANS_STAGING;
select * from CC_TRANS_STAGING limit 10;
```

Select one of the rows, and you will see a better view of the contents, which will be similar to:

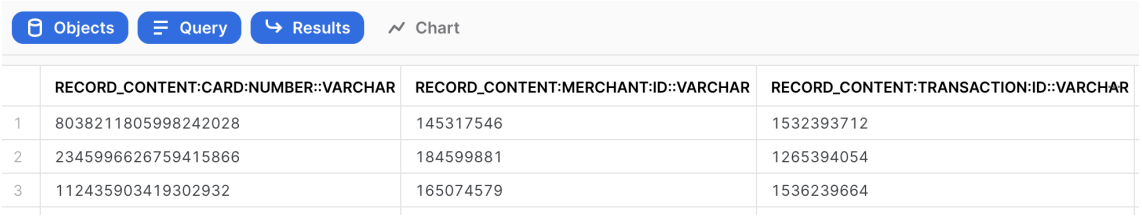


e) Run Test Queries. Now that it is a VARIANT datatype, the contents of the JSON is now understood.

```
select RECORD_CONTENT:card:number as card_id from CC_TRANS_STAGING limit 10;
```

```
select
RECORD_CONTENT:card:number::varchar,
RECORD_CONTENT:merchant:id::varchar,
RECORD_CONTENT:transaction:id::varchar,
RECORD_CONTENT:transaction:amount::float,
RECORD_CONTENT:transaction:currency::varchar,
RECORD_CONTENT:transaction:approved::boolean,
RECORD_CONTENT:transaction:type::varchar,
RECORD_CONTENT:transaction:timestamp::datetime
from CC_TRANS_STAGING
where RECORD_CONTENT:transaction:amount::float < 600 limit 10;
```

While the raw data was JSON, Snowflake has converted that during the COPY INTO step, creating VARIANT column values, making semi-structured data much easier



	RECORD_CONTENT:CARD:NUMBER::VARCHAR	RECORD_CONTENT:MERCHANT:ID::VARCHAR	RECORD_CONTENT:TRANSACTION:ID::VARCHAR
1	8038211805998242028	145317546	1532393712
2	2345996626759415866	184599881	1265394054
3	112435903419302932	165074579	1536239664

f) Create a View of Staging Table for real-time operational queries (normalize VARIANT column to a full tabular representation)

```
create or replace view CC_TRANS_STAGING_VIEW (card_id, merchant_id, transaction_id,
amount, currency, approved, type, timestamp ) as (
select
RECORD_CONTENT:card:number::varchar card_id,
RECORD_CONTENT:merchant:id::varchar merchant_id,
RECORD_CONTENT:transaction:id::varchar transaction_id,
RECORD_CONTENT:transaction:amount::float amount,
RECORD_CONTENT:transaction:currency::varchar currency,
RECORD_CONTENT:transaction:approved::boolean approved,
RECORD_CONTENT:transaction:type::varchar type,
RECORD_CONTENT:transaction:timestamp::datetime timestamp
from CC_TRANS_STAGING);
```

g) We will be creating a stream, so we need to enable change tracking

```
alter table CC_TRANS_STAGING set CHANGE_TRACKING = true;
alter view CC_TRANS_STAGING_VIEW set CHANGE_TRACKING = true;
```

h) Preview your View

```
select * from CC_TRANS_STAGING_VIEW limit 10;
select count(*) from CC_TRANS_STAGING_VIEW limit 10;
```

While data in staging table is JSON, one can now see traditional tablular view of the data

i) Create a Stream on the operational view

Note: While we could put a stream on the staging table, by using the view one can use the normalized view

```
create or replace stream CC_TRANS_STAGING_VIEW_STREAM on view CC_TRANS_STAGING_VIEW
SHOW_INITIAL_ROWS=true;
select count(*) from CC_TRANS_STAGING_VIEW_STREAM;
select * from CC_TRANS_STAGING_VIEW_STREAM limit 10;
```

Note: We are populating the Stream with the all current rows in the table. Querying the Stream does not consume it. But, something that is in a transaction (auto or manual committed) will consume the records in the stream for exactly-once change pattern.

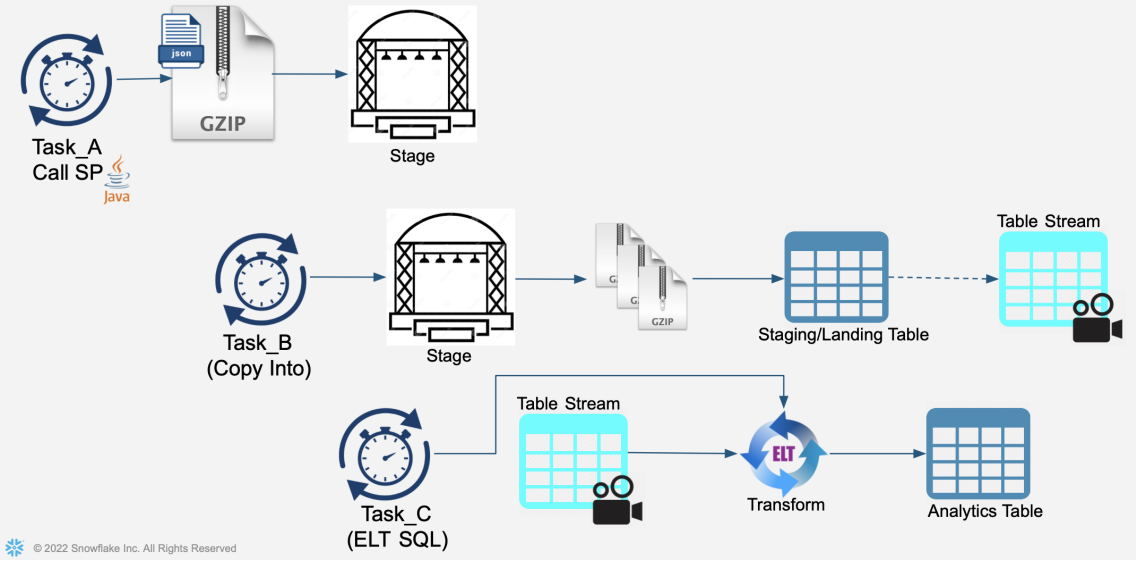
j) Create Analytical table (normalized) for transformation (ELT) and easier user consumption (Staging table for landing can then be purged after X days)

```
create or replace table CC_TRANS_ALL (
card_id varchar,
merchant_id varchar,
transaction_id varchar,
amount float,
currency varchar,
approved boolean,
type varchar,
timestamp datetime);
```

Create Data Pipeline #1

Create Tasks to orchestrate Processing for this Hands-on Lab. Each Task will be independent and separately scheduled.

See diagram:



- First task will create an incoming JSON file containing credit card transactions and store this file in the Stage.

- Second task will perform a COPY INTO operation, reading file(s) within the Stage and load into staging/landing table
- Third task will perform an ELT transformation operation, processing recently-added records to the staging/landing table and load into the "analytics-ready" table.

a) Create Task

Task be our real-time kafka streaming source (calling Stored Procedure to simulate incoming Kafka-provided credit card transactions). This task will be scheduled to run every 60 seconds, very similar to how Snowflake's Kafka Connector bundles and ingests data.

```
create or replace task GENERATE_TASK
WAREHOUSE=VHOL_WH
SCHEDULE = '1 minute'
COMMENT = 'Generates simulated real-time data for ingestion'
as
call SIMULATE_KAFKA_STREAM('@VHOL_STAGE','SNOW_',1000000);
```

b) View Definition of Task

Here are the details of your newly-created Task

```
describe task GENERATE_TASK;
```

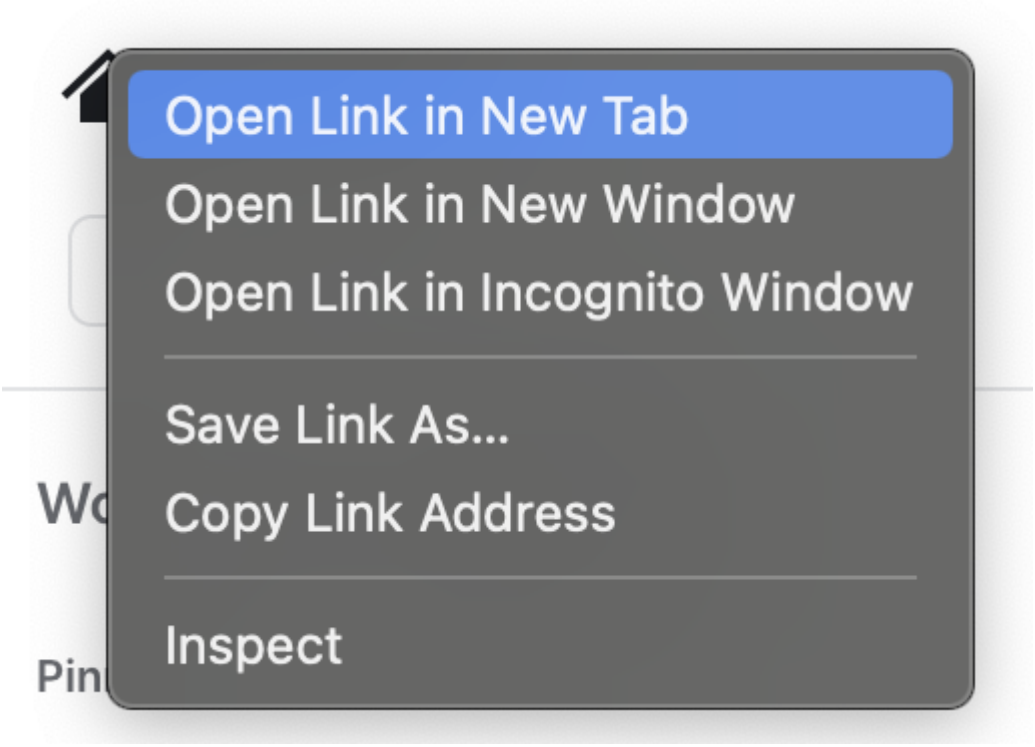
c) Manually Run Task

```
execute task GENERATE_TASK;
```

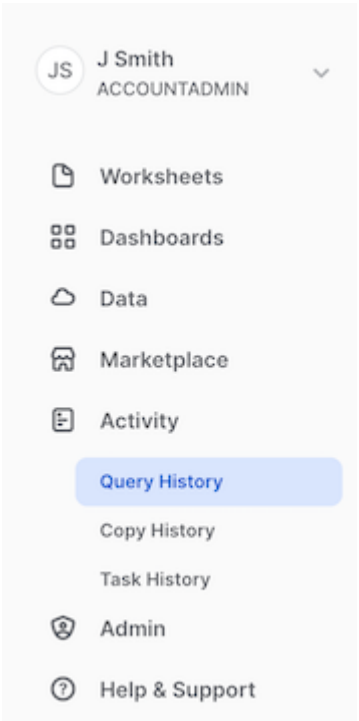
d) Monitor our Activities

- Right-Click on "House icon" at upper right and open in a new browser tab:

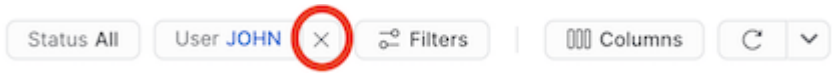
- and then



-
- From that new tab, select Activity" on the left menu, then "Query History":

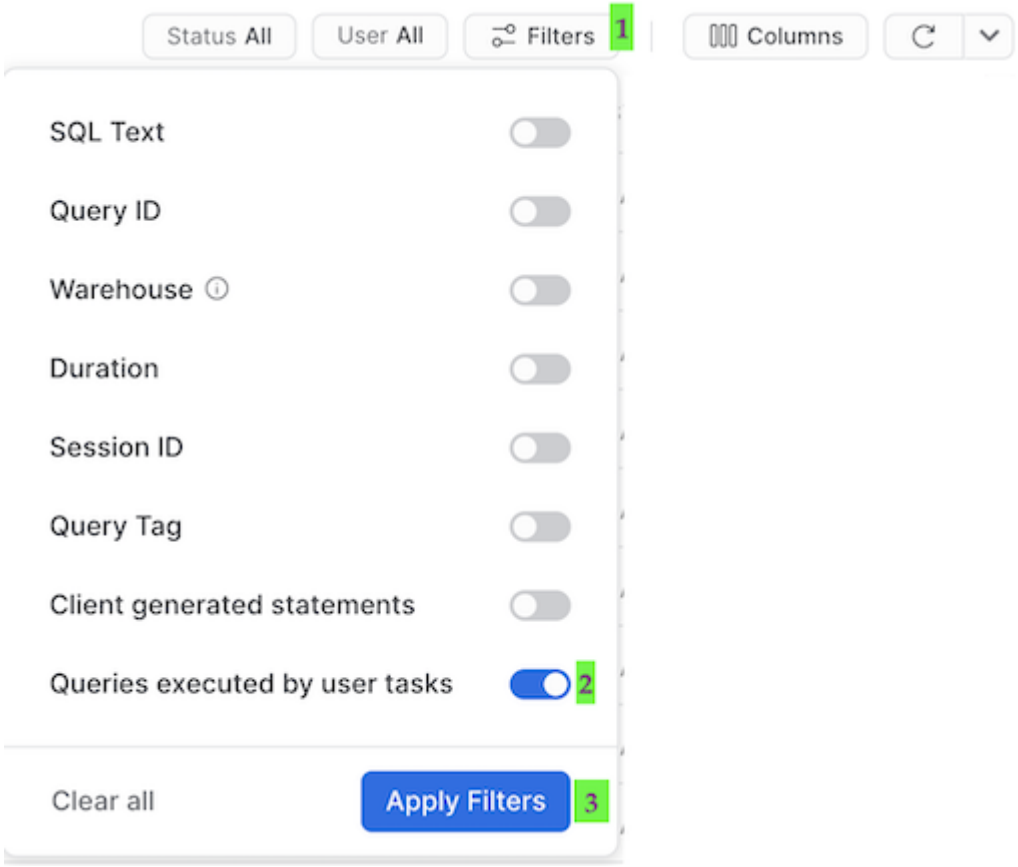


- Remove filters at the top of this table, including your username, as later scheduled tasks will run as

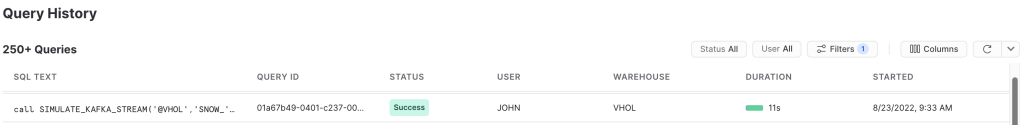


"System":

- Click "Filter", and add filter option 'Queries executed by user tasks' and click "Apply Filters":



- Now, you can see all executed SQL commands, including those executed by your running tasks:



If you don't see your tasks, make sure you are ACCOUNTADMIN in this tab/window too (or same role if not ACCOUNTADMIN but have rights to see SYSTEM-executed tasks). Finally, click on your Worksheet tab leaving your Query History tab available as you will want to return and jump between these two Snowsight views.

e) Enable Task to Run on its Schedule

Task scheduled to run every 1 minute

```
alter task GENERATE_TASK RESUME;
```

f) List Files in Stage

List Files in Stage now ready to be copied into Snowflake. NOTE: Task will not run instantly, it will wait a minute before running

```
list @VHOL_STAGE PATTERN='.*SNOW_.*';
```

g) Wait

Wait a couple minutes and then you should see that the Task is regularly generating and adding a new file to the Stage. Reminder: Remember to turn these Tasks off at end of lab or, if you take a break, jump to the last step of this section for the commands to suspend your tasks.

```
list @VHOL_STAGE PATTERN='.*SNOW_.*';
```

h) Create a Second Task

This Task will run every 3 minutes and utilizes the Staging Stream that identifies newly added records to the Staging file and loads into our analytical table. As there will be multiple files ready for ingestion, note each will be loaded independently and in parallel.

For this Task, we will utilize the option to run with a serverless compute, as a full compute warehouse is not really necessary to have running for these small continuous micro-batch tasks. In an actual implementation, the incoming simulated data stream will be external.

```
create or replace task PROCESS_FILES_TASK
USER_TASK_MANAGED_INITIAL_WAREHOUSE_SIZE = 'XSMALL'
SCHEDULE = '3 minute'
COMMENT = 'Ingests Incoming Staging Datafiles into Staging Table'
as
copy into CC_TRANS_STAGING from @VHOL_STAGE PATTERN='.*SNOW_.*';
```

Note: Snowflake does track what files in a stage it has already processed for a period of time, but, if you get thousands and thousands of files in your stage, you don't want this process to sort through them for each run of the Task. Using "PURGE" is a way to delete them during the COPY INTO step, or you can create a separate step to archive or delete these files on your preferred schedule.

i) Check Staging Table, View, and Stream

Query Staging Table and its Stream tracking changes, before running Process Task. Note before and after row counts.

```
select count(*) from CC_TRANS_STAGING;
select * from CC_TRANS_STAGING limit 10;
select count(*) from CC_TRANS_STAGING_VIEW_STREAM;
select * from CC_TRANS_STAGING_VIEW_STREAM limit 10;
```

j) Execute Task Manually

```
execute task PROCESS_FILES_TASK;
```

Wait here for processing (Can monitor from your Query History tab)

k) Query Staging View and its Stream

Streams make it so easy to track changes, after running Process Task:

```
select count(*) from CC_TRANS_STAGING_VIEW;
select count(*) from CC_TRANS_STAGING_VIEW_STREAM;
```

l) Begin task to run on its schedule

```
alter task PROCESS_FILES_TASK resume;
```

m) Create Third Task

This task will run every 4 minutes, leveraging the Staging Stream to process new records and load into the Analytical Table. Note that this first checks if there are records in the stream and then will process and load records if the stream has records.

```
create or replace task REFINE_TASK
USER_TASK_MANAGED_INITIAL_WAREHOUSE_SIZE = 'XSMALL'
SCHEDULE = '4 minute'
COMMENT = '2.  ELT Process New Transactions in Landing/Staging Table into a more
Normalized/Refined Table (flattens JSON payloads)'
when
SYSTEM$STREAM_HAS_DATA('CC_TRANS_STAGING_VIEW_STREAM')
as
insert into CC_TRANS_ALL (select
card_id, merchant_id, transaction_id, amount, currency, approved, type, timestamp
from CC_TRANS_STAGING_VIEW_STREAM);
```

Staging tables are typically periodically purged. This Table will be permanent, storing all transactions for their useful lifecycle. This ELT process would also typically include more transformations, validations, and refine/enrichment however that is not the focus of this hands-on Lab.

n) Query Analytical Table

You should see there are no records in this empty table

```
select count(*) from CC_TRANS_ALL;
```

o) Query Staging Table's Stream

Stream is showing there are records and has not yet been flushed/consumed.

```
select count(*) from CC_TRANS_STAGING_VIEW_STREAM;
```

p) Run Task Manually

```
execute task REFINE_TASK;
```

Wait here for task to complete, check Query History view to monitor.

q) Query Staging Table's Stream

Stream now showing stream after it has been flushed/consumed

```
select count(*) from CC_TRANS_ALL;
select count(*) from CC_TRANS_STAGING_VIEW_STREAM;
```

Note: The Stream will first show just the initial records (1,000,000). Once those are consumed for the initial rows, then the stream will show the rows added by the scheduled tasks.

r) Schedule Task

Resume Task so it will run on its schedule. Note it is fully independent of the other tasks we have running. You can also configure tasks to run at a certain time, using CRON-based syntax option.

```
alter task REFINE_TASK resume;
```

s) Reporting

One can report on LOAD metadata generated by Snowflake. This is available in two views:

1. For no latency, but only 14 days of retention:

   ```
   select * from VHOL_ST.INFORMATION_SCHEMA.LOAD_HISTORY where
   SCHEMA_NAME=current_schema() and TABLE_NAME='CC_TRANS_STAGING';
   ```

2. Up to 90 minutes latency with this view, but 365 days of retention:

   ```
   select * from SNOWFLAKE.ACCOUNT_USAGE.LOAD_HISTORY where
   SCHEMA_NAME=current_schema() and TABLE_NAME='CC_TRANS_STAGING';
   ```

t) Stop Tasks

```
alter task REFINE_TASK SUSPEND;
alter task PROCESS_FILES_TASK SUSPEND;
alter task GENERATE_TASK SUSPEND;
```

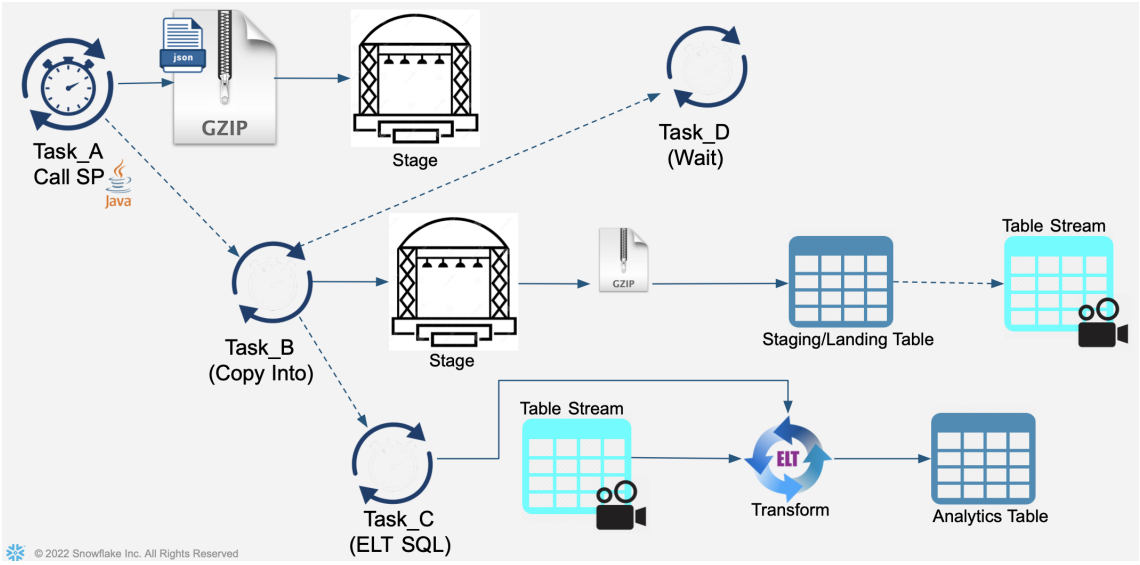
u) See how many transactions we have fully processed so far

```
select count(*) from CC_TRANS_ALL;
```

Create Data Pipeline #2

This section's Data Pipeline will be very similar to the first, except will orchestrate tasks with dependencies, rather than being independently scheduled and executed.

See diagram:



a) Create Two Tasks

Similar to the tasks created in the previous section, but ones that will become subtasks (rather than independent scheduling). Note: Define all of these to run within the same warehouse, but that is not a requirement.

- First Task

```
create or replace task REFINE_TASK2
WAREHOUSE=VHOL_WH
as
insert into CC_TRANS_ALL (select
card_id, merchant_id, transaction_id, amount, currency, approved, type,
timestamp
from CC_TRANS_STAGING_VIEW_STREAM);
```

- Second Task

```
create or replace task PROCESS_FILES_TASK2
WAREHOUSE=VHOL_WH
as
copy into CC_TRANS_STAGING from @VHOL_STAGE PATTERN='.*SNOW_.*';
```

b) Create Root Task

This is the Task where all others will be subprocesses of.

```
create or replace task LOAD_TASK
WAREHOUSE=VHOL_WH
SCHEDULE = '1 minute'
COMMENT = 'Full Sequential Orchestration'
as
call SIMULATE_KAFKA_STREAM('@VHOL_STAGE','SNOW_',1000000);
```

c) First Predecessor

-Have task REFINE_TASK2 be a predecessor of PROCESS_FILES_TASK2

```
alter task REFINE_TASK2 add after PROCESS_FILES_TASK2;
alter task REFINE_TASK2 RESUME;
```

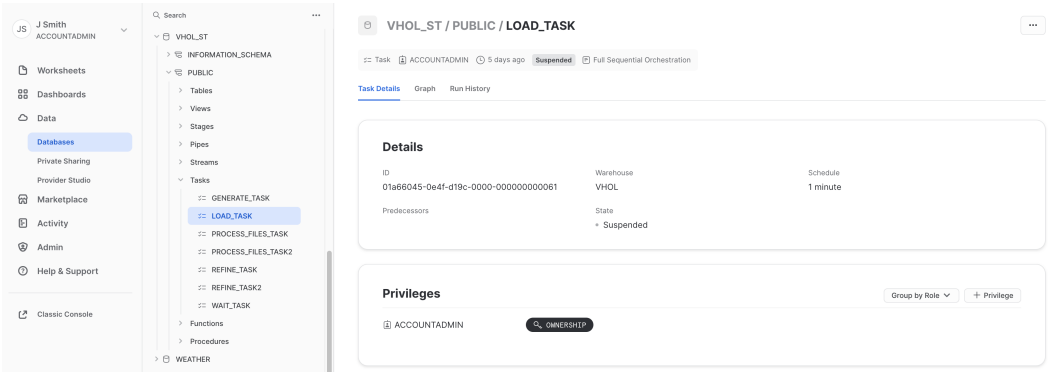
-Have task LOAD_TASK be a predecessor of PROCESS_FILES_TASK2

```
alter task PROCESS_FILES_TASK2 add after LOAD_TASK;
alter task PROCESS_FILES_TASK2 RESUME;
```

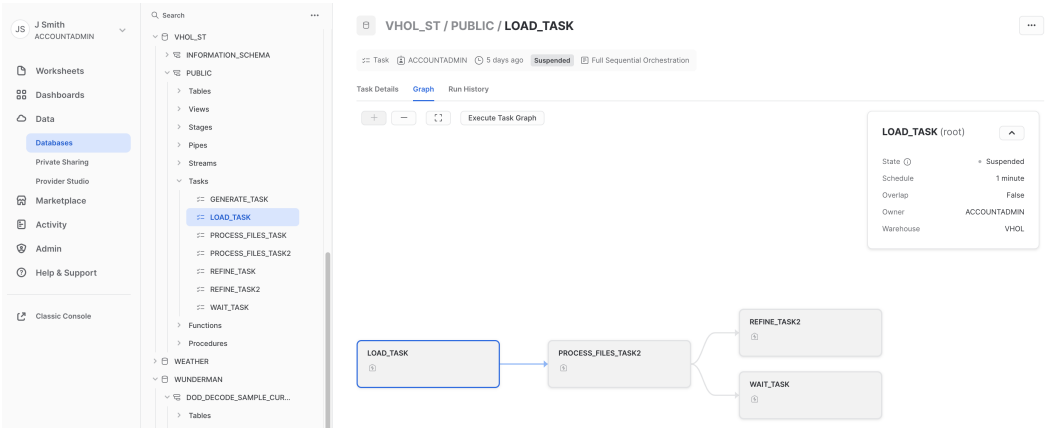
Note: One can also use the 'after' within the create task command

d) View DAG Orchestration

- In a new tab (use right-click on "home icon" at upper left), Navigate in Snowsight to: **Data>Databases>VHOL_ST>PUBLIC>Tasks>LOAD_TASK**
- Review "Task Details" tab:

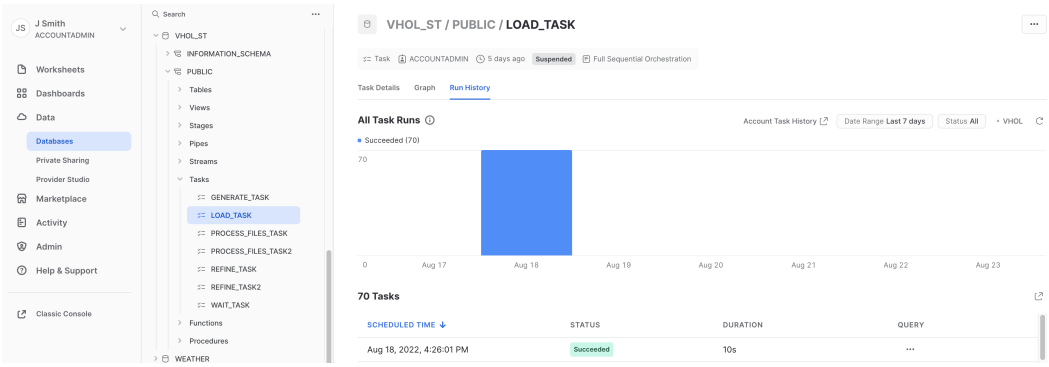


- Review "Graph" to see graphical representation of our simple flow:



Remember your warehouse is 'VHOL_WH'

- Review "Run History" to view Task Executions:



Note: Graph and Task History are recently added previews, so may need to ask on how to enable in your account if these menu options are missing.

e) Start LOAD Task

```
alter task LOAD_TASK RESUME;
```

Wait, as task will run after one minute. Then, see a file created in Stage

```
list @VHOL_STAGE PATTERN='.*SNOW_.*';
```

Go back to Activity tab, reviewing Query History and Task's Run History (refresh)

f) Tasks in Parallel

Tasks do not have to be sequential, they can also run in parallel. Let's create a simple task to demonstrate this. To add to the root task @ creation, you have to first suspend:

```
alter task LOAD_TASK SUSPEND;
create or replace task WAIT_TASK
  WAREHOUSE=VHOL_WH
  after PROCESS_FILES_TASK2
as
  call SYSTEM$wait(1);
```

g) Run Load Task

After this addition, resume/enable both tasks. Note we utilized the "AFTER" option during creation rather than using "ALTER" afterward to create this dependency to the DAG process.

```
alter task WAIT_TASK RESUME;
alter task LOAD_TASK RESUME;
```

h) Monitor

Go back to your Task tab, refreshing your Graph view and task's Run History.

i) Task Dependencies

Can also see task dependencies via SQL. As this is available via query, this can be collected for your data catalog and other analysis/reporting purposes.

```
select * from table(information_schema.task_dependents(task_name => 'LOAD_TASK',
recursive => true));
```

j) Section is Complete

Suspend your Tasks for this Section

```
alter task LOAD_TASK SUSPEND;
alter task REFINE_TASK2 SUSPEND;
alter task PROCESS_FILES_TASK2 SUSPEND;
alter task WAIT_TASK SUSPEND;
```

Final Steps & Cleanup

a) See how many transactions we have processed

```
select count(*) from CC_TRANS_ALL;
```

b) Look at our newest record now in your Analytical table:

```
select * from CC_TRANS_ALL order by TIMESTAMP desc limit 1;
```

c) See all tasks we have created, and to confirm their state is all "Suspended"

```
show tasks;
```

(suspend any still running)

d) Drop Database, removing all objects created by this Hands-on Lab (Optional)

```
drop database VHOL_ST;
```

e) Drop Warehouse (Optional)

```

use role ACCOUNTADMIN;
drop warehouse VHOL_WH;
```

e) Drop Role (Optional)

```
use role ACCOUNTADMIN;
drop role VHOL;
```

Conclusion

Congratulations, you have completed this Lab!

What We Covered

- Created a Snowflake Stream
- Created and Scheduled a Snowflake Task
- Assembled tasks into Data Pipelines
- Snowpark can be used to build new types of user-defined functions and stored procedures
- Managed and Monitored your Snowflake Tasks and Data Pipelines

Getting Started with Python

Overview

You can connect to Snowflake in many languages. If your language of choice is Python, you'll want to begin here to connect to Snowflake. We'll walk you through getting the Python Connector up and running, and then explore the basic operations you can do with it. You'll find the Python Connector to be quite robust, as it even supports integration with Pandas DataFrames.

Prerequisites

- Familiarity with Python

What You'll Learn

- how to set up the Python connector
- how to test your installation
- how to connect to Snowflake
- how to set session parameters
- how to create a warehouse
- how to create a database
- how to create a schema
- how to create a table
- how to insert data
- how to query data

What You'll Need

- A Snowflake Account
- A Text Editor

What You'll Build

- Examples of using the Snowflake Python Connector

Set up the Python Connector

For this guide, we'll be using Python 3. Let's check what version of Python you have on your system. Do this by opening a terminal and entering the following command:

```
python --version
```

If it outputs 3.5 or higher, you're good to go! If not, you'll need to install a newer version of Python. You can grab the latest release on the Python website.

Once you've got a recent version of Python, you can install the Snowflake Connector for Python. You'll do this via the Python package installer `pip` and by running the following command:

```
pip install --upgrade snowflake-connector-python
```

If you're on a Linux distribution, you'll also need to install a few packages from your distribution's repository. Specifically, you'll need the equivalent of:

- libm-devel
- openssl-devel

Once you have both Python and the Snowflake Connector installed, you're ready to go! Let's make sure that's the case.

Test Your Installation

Before we delve into *using* the Snowflake connector, let's ensure it's installed correctly. We can do this with the following script.

```
#!/usr/bin/env python
import snowflake.connector

# Gets the version
ctx = snowflake.connector.connect(
    user='<your_user_name>',
    password='<your_password>',
    account='<your_account_name>'
    )
cs = ctx.cursor()
try:
    cs.execute("SELECT current_version()")
    one_row = cs.fetchone()
    print(one_row[0])
finally:
    cs.close()
ctx.close()
```

Open up a text editor and copy that script in and save it as `validate.py`. You'll need to input your own user name, password, and account in the corresponding fields.

Open up a terminal in the location where the file is saved, and run the following command:

```
python validate.py
```

If everything is good to go, you'll see the installed Snowflake version. Otherwise, you'll get errors specific to your situation.

Connect to Snowflake

It's time to use the Snowflake Connector for Python. Open up your Python environment. The first thing you'll need to do is to import the Snowflake Connector module. Do this before using any Snowflake related commands.

```
import snowflake.connector
```

You may want to consider reading in your login information from external sources, such as environment variables. This will add some security to your scripts, and save you time in the long run. In this example, we're using `os.getenv` to pull the environment variable `SNOWSQL_PWD` for our variable `PASSWORD.`

```
PASSWORD = os.getenv('SNOWSQL_PWD')
```

You'll use those variables within the Python Connector like so:

```
conn = snowflake.connector.connect(
    user=USER,
```

```
    password=PASSWORD,
    account=ACCOUNT
    )
```

Nice! Now let's delve into the specifics.

Set Session Parameters

You can set session parameters to tweak your session and have it set up just the way you want it. There are two ways to accomplish this. First, you can set them when you initially connect, like so:

```
conn = snowflake.connector.connect(
    user='XXXX',
    password='XXXX',
    account='XXXX',
    session_parameters={
        'QUERY_TAG': 'EndOfMonthFinancials',
    }
)
```

Alternatively, you can set them after you connect by executing the SQL statement `ALTER SESSION SET`:

```
conn.cursor().execute("ALTER SESSION SET QUERY_TAG = 'EndOfMonthFinancials'")
```

You may have taken advantage of Snowflake's ability to use robust security methods. If so, you'll want to take extra steps to configure the Snowflake Python Connector.

- [Single Sign-on (SSO)]
- [Key Pair Authentication]
  - [Key Rotation]
- [OAuth]

At this point, you can begin manipulating objects within Snowflake.

Create a Warehouse

If you're familiar with the SQL commands to interact with Snowflake, then the commands within the Python connector will be familiar as well. Let's break down what a command looks like.

Remember that `conn` is the object that connects you to your Snowflake account. You can use that to execute SQL commands with the following format.

```
conn.cursor().execute("YOUR SQL COMMAND")
```

We first want to create a [virtual warehouse]. Virtual warehouses contain the servers that are required for you to perform queries and DML operations with Snowflake. Creating one can be done by incorporating the `CREATE WAREHOUSE` SQL command.

```
conn.cursor().execute("CREATE WAREHOUSE IF NOT EXISTS tiny_warehouse_mg")
```

The `CREATE WAREHOUSE` command creates a warehouse, as expected, but also implicitly sets that warehouse as the active warehouse for your session. If you've already created a warehouse, you can explicitly set it as the active warehouse with the `USE WAREHOUSE` command.

```
conn.cursor().execute("USE WAREHOUSE tiny_warehouse_mg")
```

Now that you have a warehouse, let's talk about databases.

Create a Database

The next step is to create a database. Databases contain your schemas, which contain your database objects. We can create one in a similar manner to creating a warehouse, but this time with the `CREATE DATABASE` command.

```
conn.cursor().execute("CREATE DATABASE IF NOT EXISTS testdb")
```

Again, creating a database also sets that database as the active one for the current session. If you need to set an already created database as the active database for the session, use the `USE DATABASE` command.

```
conn.cursor().execute("USE DATABASE testdb")
```

After that, you need to create a schema.

Create a Schema

Schemas are the grouping of your database objects. These include your tables, the data within them, and views. Schemas are found within databases. You can create a schema with the `CREATE SCHEMA` command.

```
conn.cursor().execute("CREATE SCHEMA IF NOT EXISTS testschema")
```

Since `CREATE SCHEMA` also sets it as the active schema for your session, you don't have to explicitly call `USE SCHEMA` as well. Only do so if you would like to use an already created schema.

```
conn.cursor().execute("USE SCHEMA testschema")
```

By default, the schema will be used in the current database. You can use it in another database by specifying that other database.

```
conn.cursor().execute("USE SCHEMA otherdb.testschema")
```

With a schema in use, it's time to move on to dealing with tables.

Create a Table

Alright, with a warehouse, database, and schema created you have everything you need to begin manipulating data in tables. First, you'll need to create the table. That's done with the `CREATE TABLE` command.

```
conn.cursor().execute(
    "CREATE OR REPLACE TABLE "
    "test_table(col1 integer, col2 string)")
```

Within `conn.cursor().execute` this example creates a table named `test_table` with two columns, one named `col1` which will contain integers and `col2` which will contain strings.

Insert Data

With the table `test_table` created, you can add data to it. You can do so with the command `INSERT`.

```
conn.cursor().execute(
    "INSERT INTO test_table(col1, col2) "
    "VALUES(123, 'test string1'),(456, 'test string2')")
```

This command inserts data into `test_table` row by row. The value found in the first column and first row is "123" and so on.

If you don't want to manually insert data row by row, however, you can load data instead. This is done with a combination of the `PUT` and `COPY INTO` commands.

```
conn.cursor().execute("PUT file:///tmp/data/file* @%test_table")
conn.cursor().execute("COPY INTO test_table")
```

The `PUT` command here is staging the file, and the `COPY INTO` command is copying that data from that file into the specified table. You can also use the `COPY INTO` command to copy data from an external location.

Query Data

Of course, you'll want to query your data at some point. It's easy to do that within the Python Connector too. To view values from the table, you can do so easily with the `print` command.

```
col1, col2 = conn.cursor().execute("SELECT col1, col2 FROM test_table").fetchone()
print('{0}, {1}'.format(col1, col2))
```

This code snippet is using the SQL command `SELECT col1, col2 FROM test_table` to select specific columns, and then is printing the first values in each, or in other words, the first row.

If you'd like to print entire columns, you can do so in a similar manner.

```
for (col1, col2) in conn.cursor().execute("SELECT col1, col2 FROM test_table"):
    print('{0}, {1}'.format(col1, col2))
```

To efficiently use your resources, you'll want to remember to explicitly close your connection to Snowflake after performing queries.

```
connection.close()
```

Look at that! You have now implemented all the steps needed to manipulate data within Snowflake.

Next Steps: Advanced Manipulation with Python

By now you have a grasp on the basics of using the Python Connector. As you may have noticed, it leans heavily on already-established Snowflake SQL commands. Of course, this isn't all you can do with the Connector. Here are some potential next steps that may be of interest to you depending on your use case.

- [Bind Data]
- [Use Pandas Dataframes]
- [Snowflake SQLAlchemy Toolkit]

This is just the surface of what you can do with Snowflake. To learn more, see the [Snowflake documentation].

Resource Optimization: Setup & Configuration

Setup & Configuration

Setup & Configuration queries provide more proactive insight into warehouses that are not utilizing key features that can prevent runaway resource and cost consumption. Leverage these key queries listed below to identify warehouses which should be re-configured to leverage the appropriate features.

What You'll Learn

- available features and settings to control Snowflake consumption
- the importance of configuring auto-resume and auto-suspend
- how to configure relevant statement timeouts
- the value and configuration of resource monitors
- how to analyze user activity
- how to analyze task behavior

What You'll Need

- A Snowflake Account
- Access to view [Account Usage Data Share]

Related Materials

- Resource Optimization: Usage Monitoring
- Resource Optimization: Billing Metrics
- Resource Optimization: Performance

Query Tiers

Each query within the Resource Optimization Snowflake Quickstarts will have a tier designation just to the right of its name as "(T*)". The following tier descriptions should help to better understand those designations.

Tier 1 Queries

At its core, Tier 1 queries are essential to Resource Optimization at Snowflake and should be used by each customer to help with their consumption monitoring - regardless of size, industry, location, etc.

Tier 2 Queries

Tier 2 queries, while still playing a vital role in the process, offer an extra level of depth around Resource Optimization and while they may not be essential to all customers and their workloads, it can offer further explanation as to any additional areas in which over-consumption may be identified.

Tier 3 Queries

Finally, Tier 3 queries are designed to be used by customers that are looking to leave no stone unturned when it comes to optimizing their consumption of Snowflake. While these queries are still very helpful in this process, they are not as critical as the queries in Tier 1 & 2.

Warehouses without Auto-Resume (T1)

TIER 1

Description:

Identifies all warehouses that do not have auto-resume enabled. Enabling this feature will automatically resume a warehouse any time a query is submitted against that specific warehouse. By default, all warehouses have auto-

resume enabled.

How to Interpret Results:

Make sure all warehouses are set to auto resume. If you are going to implement auto suspend and proper timeout limits, this is a must or users will not be able to query the system.

SQL

```
SHOW WAREHOUSES
;
SELECT "name" AS WAREHOUSE_NAME
      ,"size" AS WAREHOUSE_SIZE
  FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()))
 WHERE "auto_resume" = 'false'
;
```

Warehouses without Auto-Suspend (T1)

TIER 1

Description:

Identifies all warehouses that do not have auto-suspend enabled. Enabling this feature will ensure that warehouses become suspended after a specific amount of inactive time in order to prevent runaway costs. By default, all warehouses have auto-suspend enabled.

How to Interpret Results:

Make sure all warehouses are set to auto suspend. This way when they are not processing queries your compute footprint will shrink and thus your credit burn.

SQL

```
SHOW WAREHOUSES
;
SELECT "name" AS WAREHOUSE_NAME
      ,"size" AS WAREHOUSE_SIZE
  FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()))
 WHERE IFNULL("auto_suspend",0) = 0
;
```

Warehouses with Long Suspension (T1)

TIER 1

Description:

Identifies warehouses that have the longest setting for automatic suspension after a period of no activity on that warehouse.

How to Interpret Results:

All warehouses should have an appropriate setting for automatic suspension for the workload.

– For Tasks, Loading and ETL/ELT warehouses set to immediate suspension.

– For BI and SELECT query warehouses set to 10 minutes for suspension to keep data caches warm for end users

– For DevOps, DataOps and Data Science warehouses set to 5 minutes for suspension as warm cache is not as important to ad-hoc and highly unique queries.

SQL

```
SHOW WAREHOUSES
;
SELECT "name" AS WAREHOUSE_NAME
      ,"size" AS WAREHOUSE_SIZE
  FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()))
 WHERE "auto_suspend" >= 3600  // 3600 seconds = 1 hour
;
```

Warehouses without Resource Monitors (T1)

TIER 1

Description:

Identifies all warehouses without resource monitors in place. Resource monitors provide the ability to set limits on credits consumed against a warehouse during a specific time interval or date range. This can help prevent certain warehouses from unintentionally consuming more credits than typically expected.

How to Interpret Results:

Warehouses without resource monitors in place could be prone to excessive costs if a warehouse consumes more credits than anticipated. Leverage the results of this query to identify the warehouses that should have resource monitors in place to prevent future runaway costs.

SQL

```
SHOW WAREHOUSES
;
SELECT "name" AS WAREHOUSE_NAME
      ,"size" AS WAREHOUSE_SIZE
  FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()))
 WHERE "resource_monitor" = 'null'
;
```

User Segmentation (T1)

TIER 1

Description:

Lists out all warehouses that are used by multiple ROLEs in Snowflake and returns the average execution time and count of all queries executed by each ROLE in each warehouse.

How to Interpret Results:

If execution times or query counts across roles within a single warehouse are wildly different it might be worth segmenting those users into separate warehouses and configuring each warehouse to meet the specific needs of each workload

Primary Schema:

Account_Usage

SQL

```
SELECT *

FROM (
  SELECT

  WAREHOUSE_NAME
  ,ROLE_NAME
  ,AVG(EXECUTION_TIME) as AVERAGE_EXECUTION_TIME
  ,COUNT(QUERY_ID) as COUNT_OF_QUERIES
  ,COUNT(ROLE_NAME) OVER(PARTITION BY WAREHOUSE_NAME) AS ROLES_PER_WAREHOUSE


  FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
  where to_date(start_time) >= dateadd(month,-1,CURRENT_TIMESTAMP())
  group by 1,2
) A
WHERE A.ROLES_PER_WAREHOUSE > 1
order by 5 DESC,1,2
;
```

Screenshot

WAREHOUSE_NAME	ROLE_NAME	AVERAGE_EXECUTION_TIME	COUNT_OF_QUERIES	ROLES_PER_WAREHOUSE
AAA_DEMO_SUMMIT_THURS	ACCOUNTADMIN	557.400000	5	2
AAA_DEMO_SUMMIT_THURS	SYSADMIN	242.000000	17	2
ANDREW_WH	ACCOUNTADMIN	108.400000	10	2
ANDREW_WH	ANDREW_ROLE	62.533333	15	2
BMCNEELY_WH	ACCOUNTADMIN	320.772300	426	4
BMCNEELY_WH	BMCNEELY_HIVE	18.500000	2	4
BMCNEELY_WH	BMCNEELY_S1_S2	70.750000	4	4
BMCNEELY_WH	SYSADMIN	94.400000	25	4
CARLIN_TEST	ACCOUNTADMIN	228.550725	207	3
CARLIN_TEST	DBT	53.833333	6	3
CARLIN_TEST	SYSADMIN	8938.752672	655	3

Idle Users (T2)

TIER 2

Description:

Users in the Snowflake platform that have not logged in in the last 30 days

How to Interpret Results:

Should these users be removed or more formally onboarded?

Primary Schema:

Account_Usage

SQL

```
SELECT
    *
FROM SNOWFLAKE.ACCOUNT_USAGE.USERS
WHERE LAST_SUCCESS_LOGIN < DATEADD(month, -1, CURRENT_TIMESTAMP())
AND DELETED_ON IS NULL;
```

Users Never Logged In (T2)

TIER 2

Description:

Users that have never logged in to Snowflake

How to Interpret Results:

Should these users be removed or more formally onboarded?

Primary Schema:

Account_Usage

SQL

```
SELECT
    *
FROM SNOWFLAKE.ACCOUNT_USAGE.USERS
WHERE LAST_SUCCESS_LOGIN IS NULL;
```

Idle Roles (T2)

TIER 2

Description:

Roles that have not been used in the last 30 days

How to Interpret Results:

Are these roles necessary? Should these roles be cleaned up?

Primary Schema:

Account_Usage

SQL

```
SELECT
    R.*
FROM SNOWFLAKE.ACCOUNT_USAGE.ROLES R
LEFT JOIN (
    SELECT DISTINCT
        ROLE_NAME
    FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
    WHERE START_TIME > DATEADD(month,-1,CURRENT_TIMESTAMP())
        ) Q
```

```
                ON Q.ROLE_NAME = R.NAME
WHERE Q.ROLE_NAME IS NULL
and DELETED_ON IS NULL;
```

Idle Warehouses (T2)

TIER 2

Description:

Warehouses that have not been used in the last 30 days

How to Interpret Results:

Should these warehouses be removed? Should the users of these warehouses be enabled/onboarded?

SQL

```
SHOW WAREHOUSES;

select *
from table(result_scan(last_query_id())) a
left join (select distinct WAREHOUSE_NAME from
SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
    WHERE START_TIME > DATEADD(month,-1,CURRENT_TIMESTAMP())
) b on b.WAREHOUSE_NAME = a."name"

where b.WAREHOUSE_NAME is null;
```

Set Statement Timeouts (T2)

TIER 2

Description:

Statement timeouts provide additional controls around how long a query is able to run before cancelling it. Using this feature will ensure that any queries that get hung up for extended periods of time will not cause excessive consumption of credits.

Show parameter settings at the Account, Warehouse, and User Session levels.

SQL

```
SHOW PARAMETERS LIKE 'STATEMENT_TIMEOUT_IN_SECONDS' IN ACCOUNT;
SHOW PARAMETERS LIKE 'STATEMENT_TIMEOUT_IN_SECONDS' IN WAREHOUSE <warehouse-name>;
SHOW PARAMETERS LIKE 'STATEMENT_TIMEOUT_IN_SECONDS' IN USER <username>;
```

How to Interpret Results:

This parameter is set at the account level by default. When the parameter is also set for both a warehouse and a user session, the lowest non-zero value is enforced.

Stale Table Streams (T2)

TIER 2

Description:

Indicates whether the offset for the stream is positioned at a point earlier than the data retention period for the table (or 14 days, whichever period is longer). Change data capture (CDC) activity cannot be returned for the table.

How to Interpret Results:

To return CDC activity for the table, recreate the stream. To prevent a stream from becoming stale, consume the stream records within a transaction during the retention period for the table.

SQL

```
SHOW STREAMS;

select *
from table(result_scan(last_query_id()))
where "stale" = true;
```

Failed Tasks (T2)

TIER 2

Description:

Returns a list of task executions that failed.

How to Interpret Results:

Revisit these task executions to resolve the errors.

Primary Schema:

Account_Usage

SQL

```
select *
  from snowflake.account_usage.task_history
  WHERE STATE = 'FAILED'
  and query_start_time >= DATEADD (day, -7, CURRENT_TIMESTAMP())
  order by query_start_time DESC
  ;
```

Long Running Tasks (T2)

TIER 2

Description:

Returns an ordered list of the longest running tasks

How to Interpret Results:

revisit task execution frequency or the task code for optimization

Primary Schema:

Account_Usage

SQL

```
select DATEDIFF(seconds, QUERY_START_TIME,COMPLETED_TIME) as DURATION_SECONDS
               ,*
from snowflake.account_usage.task_history
WHERE STATE = 'SUCCEEDED'
and query_start_time >= DATEADD (day, -7, CURRENT_TIMESTAMP())
order by DURATION_SECONDS desc
  ;
```

Resource Optimization: Usage Monitoring

Usage Monitoring

Usage Monitoring queries are designed to identify the warehouses, queries, tools, and users that are responsible for consuming the most credits over a specified period of time. These queries can be used to determine which of those resources are consuming more credits than anticipated and take the necessary steps to reduce their consumption.

What You'll Learn

- how to analyze consumption trends and patterns
- how to identify consumption anomolies
- how to analyze partner tool consumption metrics

What You'll Need

- A Snowflake Account
- Access to view [Account Usage Data Share]

Query Tiers

Each query within the Resource Optimization Snowflake Quickstarts will have a tier designation just to the right of its name as "(T*)". The following tier descriptions should help to better understand those designations.

Tier 1 Queries

At its core, Tier 1 queries are essential to Resource Optimization at Snowflake and should be used by each customer to help with their consumption monitoring - regardless of size, industry, location, etc.

Tier 2 Queries

Tier 2 queries, while still playing a vital role in the process, offer an extra level of depth around Resource Optimization and while they may not be essential to all customers and their workloads, it can offer further explanation as to any additional areas in which over-consumption may be identified.

Tier 3 Queries

Finally, Tier 3 queries are designed to be used by customers that are looking to leave no stone unturned when it comes to optimizing their consumption of Snowflake. While these queries are still very helpful in this process, they are not as critical as the queries in Tier 1 & 2.

Credit Consumption by Warehouse (T1)

TIER 1

Description:

Shows the total credit consumption for each warehouse over a specific time period.

How to Interpret Results:

Are there specific warehouses that are consuming more credits than the others? Should they be? Are there specific warehouses that are consuming more credits than anticipated for that warehouse?

Primary Schema:

Account_Usage

SQL

```
// Credits used (all time = past year)
SELECT WAREHOUSE_NAME
      ,SUM(CREDITS_USED_COMPUTE) AS CREDITS_USED_COMPUTE_SUM
  FROM ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
 GROUP BY 1
 ORDER BY 2 DESC
;

// Credits used (past N days/weeks/months)
SELECT WAREHOUSE_NAME
      ,SUM(CREDITS_USED_COMPUTE) AS CREDITS_USED_COMPUTE_SUM
  FROM ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
 WHERE START_TIME >= DATEADD(DAY, -7, CURRENT_TIMESTAMP())  // Past 7 days
 GROUP BY 1
 ORDER BY 2 DESC
;
```

Screenshot

WAREHOUSE_NAME	CREDITS_USED_COMPUTE_SUM
RESET_WH	11.865555556
DGARDNER_WH	8.536111111
BI_LARGE_WH	6.804444444
DARREN_LOAD_WH	4.508055555
LOAD_WH	1.755833334
BI_MEDIUM_WH	1.074444445
DEMO_WH	0.999444444
DEV_WH	0.193333333
CLOUD_SERVICES_ONLY	0.000000000
SE_HOL_DATA_ENG_WH	0.000000000

Average Hour-by-Hour Consumption Over the Past 7 Days (T1)

TIER 1

Description:

Shows the total credit consumption on an hourly basis to help understand consumption trends (peaks, valleys) over the past 7 days.

How to Interpret Results:

At which points of the day are we seeing spikes in our consumption? Is that expected?

Primary Schema:

Account_Usage

SQL (by hour, warehouse)

```
// Credits used by [hour, warehouse] (past 7 days)
SELECT START_TIME
      ,WAREHOUSE_NAME
      ,CREDITS_USED_COMPUTE
  FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
 WHERE START_TIME >= DATEADD(DAY, -7, CURRENT_TIMESTAMP())
   AND WAREHOUSE_ID > 0  // Skip pseudo-VWs such as "CLOUD_SERVICES_ONLY"
 ORDER BY 1 DESC,2
;
```

Screenshot

START_TIME	WAREHOUSE_NAME	CREDITS_USED_COMPUTE
2020-10-20 09:00:00.000 -0700	BI_LARGE_WH	0.000000000
2020-10-20 09:00:00.000 -0700	BI_MEDIUM_WH	0.356666667
2020-10-20 09:00:00.000 -0700	LOAD_WH	0.540000000
2020-10-20 09:00:00.000 -0700	RESET_WH	1.281111111
2020-10-20 08:00:00.000 -0700	RESET_WH	0.025555556
2020-10-16 17:00:00.000 -0700	DGARDNER_WH	0.000000000
2020-10-16 09:00:00.000 -0700	DGARDNER_WH	0.553055556
2020-10-15 16:00:00.000 -0700	DGARDNER_WH	0.000000000
2020-10-15 14:00:00.000 -0700	DGARDNER_WH	0.183333333
2020-10-14 17:00:00.000 -0700	DGARDNER_WH	0.567222222
2020-10-14 16:00:00.000 -0700	DGARDNER_WH	0.166666667
2020-10-14 11:00:00.000 -0700	DGARDNER_WH	0.350000000
2020-10-13 17:00:00.000 -0700	DGARDNER_WH	0.260833333
2020-10-13 16:00:00.000 -0700	DGARDNER_WH	0.414722222
2020-10-13 15:00:00.000 -0700	DGARDNER_WH	0.516666667
2020-10-09 16:00:00.000 -0700	DGARDNER_WH	0.227222222
2020-10-09 15:00:00.000 -0700	DGARDNER_WH	0.405277778
2020-10-09 14:00:00.000 -0700	DGARDNER_WH	0.688055556
2020-10-08 08:00:00.000 -0700	DGARDNER_WH	0.280277778
2020-10-07 18:00:00.000 -0700	DGARDNER_WH	0.381944444
2020-10-07 17:00:00.000 -0700	DGARDNER_WH	0.166666667
2020-10-07 16:00:00.000 -0700	DEMO_WH	0.000000000

####SQL (by hour)

```
SELECT DATE_PART('HOUR', START_TIME) AS START_HOUR
      ,WAREHOUSE_NAME
      ,AVG(CREDITS_USED_COMPUTE) AS CREDITS_USED_COMPUTE_AVG
  FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
 WHERE START_TIME >= DATEADD(DAY, -7, CURRENT_TIMESTAMP())
   AND WAREHOUSE_ID > 0  // Skip pseudo-VWs such as "CLOUD_SERVICES_ONLY"
 GROUP BY 1, 2
```

```
 ORDER BY 1, 2
;
```

Screenshot

START_HOUR	WAREHOUSE_NAME	CREDITS_USED_COMPUTE_AVG
0	MDONOVAN_WH	0.044166667000
1	JRYAN_VWH	0.373888889000
2	JRYAN_VWH	0.352222222000
2	MDONOVAN_WH	0.043888889000
2	MHENSON_WH	0.166666667000
3	ARCH_DEV_EXEC_WH	0.063888889000
3	MHENSON_WH	0.217777778000
4	ARCH_DEV_EXEC_WH	0.055694444500
4	JB_VWH	1.803055555750
4	JRYAN_VWH	0.733333333000
4	MDONOVAN_WH	0.000000000000
5	ARCH_DEV_EXEC_WH	0.191111111000
5	JB_VWH	2.016388889000
5	JRYAN_VWH	0.222222222333
6	ARCH_DEV_EXEC_WH	0.030555555500
6	DSILVA_WH	0.056388889000
6	JB_VWH	0.000000000000
6	JRYAN_VWH	0.179444444500
6	MHENSON_WH	0.674444444000
7	ADHOC	0.000000000000

Average Query Volume by Hour (Past 7 Days) (T1)

TIER 1

Description:

Shows average number of queries run on an hourly basis to help better understand typical query activity.

How to Interpret Results:

How many queries are being run on an hourly basis? Is this more or less than we anticipated? What could be causing this?

Primary Schema:

Account_Usage

SQL

```
SELECT DATE_TRUNC('HOUR', START_TIME) AS QUERY_START_HOUR
      ,WAREHOUSE_NAME
      ,COUNT(*) AS NUM_QUERIES
  FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
 WHERE START_TIME >= DATEADD(DAY, -7, CURRENT_TIMESTAMP())  // Past 7 days
 GROUP BY 1, 2
 ORDER BY 1 DESC, 2
;
```

Screenshot

QUERY_START_HOUR	NUM_QUERIES
2020-10-20 13:00:00.000 -0700	97
2020-10-20 11:00:00.000 -0700	1
2020-10-20 10:00:00.000 -0700	67
2020-10-20 09:00:00.000 -0700	306
2020-10-20 08:00:00.000 -0700	215
2020-10-19 14:00:00.000 -0700	18
2020-10-18 11:00:00.000 -0700	18
2020-10-16 17:00:00.000 -0700	75
2020-10-16 00:00:00.000 -0700	48

2020-10-16 09:00:00.000 -0700	48
2020-10-16 07:00:00.000 -0700	18
2020-10-15 16:00:00.000 -0700	18
2020-10-15 14:00:00.000 -0700	31
2020-10-14 17:00:00.000 -0700	38
2020-10-14 16:00:00.000 -0700	19
2020-10-14 11:00:00.000 -0700	12
2020-10-13 17:00:00.000 -0700	6
2020-10-13 16:00:00.000 -0700	54
2020-10-13 15:00:00.000 -0700	34
2020-10-12 11:00:00.000 -0700	18
2020-10-09 16:00:00.000 -0700	17
2020-10-09 15:00:00.000 -0700	21

Warehouse Utilization Over 7 Day Average (T1)

TIER 1

Description:

This query returns the daily average of credit consumption grouped by week and warehouse.

How to Interpret Results:

Use this to identify anomolies in credit consumption for warehouses across weeks from the past year.

Primary Schema:

Account_Usage

SQL

```
WITH CTE_DATE_WH AS(
  SELECT TO_DATE(START_TIME) AS START_DATE
        ,WAREHOUSE_NAME
        ,SUM(CREDITS_USED) AS CREDITS_USED_DATE_WH
    FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
   GROUP BY START_DATE
           ,WAREHOUSE_NAME
)
SELECT START_DATE
      ,WAREHOUSE_NAME
      ,CREDITS_USED_DATE_WH
      ,AVG(CREDITS_USED_DATE_WH) OVER (PARTITION BY WAREHOUSE_NAME ORDER BY START_DATE
ROWS 7 PRECEDING) AS CREDITS_USED_7_DAY_AVG
      ,100.0*((CREDITS_USED_DATE_WH / CREDITS_USED_7_DAY_AVG) - 1) AS
PCT_OVER_TO_7_DAY_AVERAGE
  FROM CTE_DATE_WH
QUALIFY CREDITS_USED_DATE_WH > 100  // Minimum N=100 credits
    AND PCT_OVER_TO_7_DAY_AVERAGE >= 0.5  // Minimum 50% increase over past 7 day
average
 ORDER BY PCT_OVER_TO_7_DAY_AVERAGE DESC
;
```

Screenshot

START_DATE	WAREHOUSE_NAME	CREDITS_USED_DATE_WH	CREDITS_USED_7_DAY_AVG	PCT_OVER_TO_7_DAY_AVERAGE
2020-09-08	IF_WH_SMALL	214.050029444	30.382002847250	604.529028320400
2020-06-15	MDONOVAN_WH	192.260606389	46.090388784625	317.138174484500
2020-06-16	MDONOVAN_WH	167.147202777	66.185671145875	152.542884106400
2020-05-26	AF_DW_XXL	170.247457777	93.518240833000	82.047327088900
2020-05-27	AF_DW_XXL	223.435996667	136.824159444333	63.301567189900
2020-05-28	AF_DW_XXL	220.893701667	157.841545000000	39.946489795800

Forecasting Usage/Billing (T1)

TIER 1

Description:

This query provides three distinct consumption metrics for each day of the contract term. (1) the contracted consumption is the dollar amount consumed if usage was flat for the entire term. (2) the actual consumption pulls from the various usage views and aggregates dollars at a day level. (3) the forecasted consumption creates a straight line regression from the actuals to project go-forward consumption.

How to Interpret Results:

This data should be mapped as line graphs with a running total calculation to estimate future forecast against the contract amount.

Primary Schema:

Account_Usage

SQL

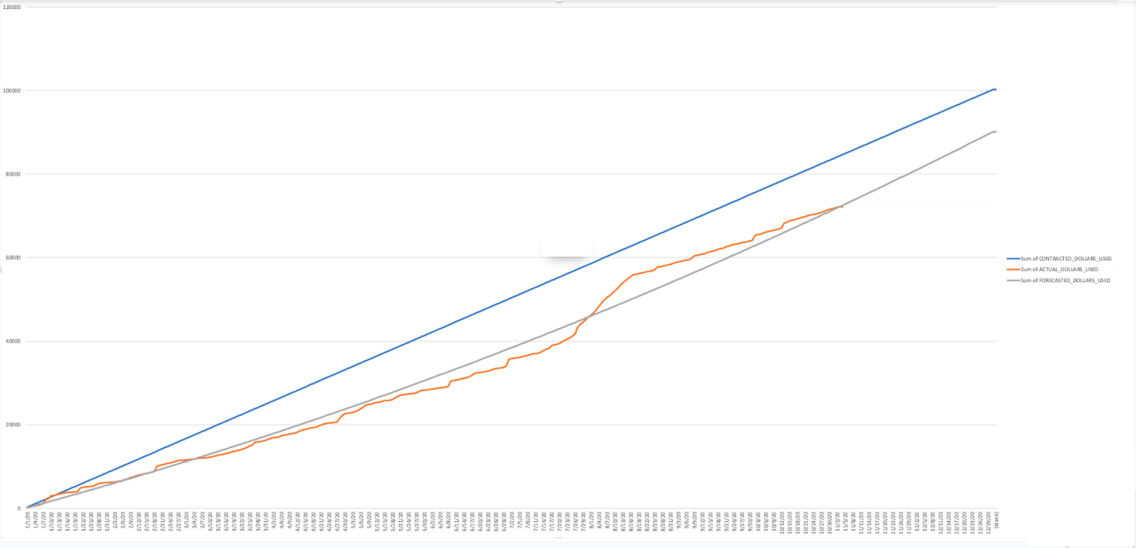
```
SET CREDIT_PRICE = 4.00; --edit this number to reflect credit price
SET TERM_LENGTH = 12; --integer value in months
SET TERM_START_DATE = '2020-01-01';
SET TERM_AMOUNT = 100000.00; --number(10,2) value in dollars
WITH CONTRACT_VALUES AS (
     SELECT
               $CREDIT_PRICE::decimal(10,2) as CREDIT_PRICE
              ,$TERM_AMOUNT::decimal(38,0) as TOTAL_CONTRACT_VALUE
              ,$TERM_START_DATE::timestamp as CONTRACT_START_DATE
              ,DATEADD(day,-1,DATEADD(month,$TERM_LENGTH,$TERM_START_DATE))::timestamp
as CONTRACT_END_DATE
),
PROJECTED_USAGE AS (
     SELECT
                CREDIT_PRICE
               ,TOTAL_CONTRACT_VALUE
               ,CONTRACT_START_DATE
               ,CONTRACT_END_DATE
               ,(TOTAL_CONTRACT_VALUE)
                   /
                   DATEDIFF(day,CONTRACT_START_DATE,CONTRACT_END_DATE)  AS
DOLLARS_PER_DAY
               , (TOTAL_CONTRACT_VALUE/CREDIT_PRICE)
                   /
               DATEDIFF(day,CONTRACT_START_DATE,CONTRACT_END_DATE) AS CREDITS_PER_DAY
     FROM      CONTRACT_VALUES
),
ACTUAL_USAGE AS (
 SELECT TO_DATE(START_TIME) AS CONSUMPTION_DATE
   ,SUM(DOLLARS_USED) as ACTUAL_DOLLARS_USED
 FROM (
   --COMPUTE FROM WAREHOUSES
   SELECT
             'WH Compute' as WAREHOUSE_GROUP_NAME
            ,WMH.WAREHOUSE_NAME
            ,NULL AS GROUP_CONTACT
            ,NULL AS GROUP_COST_CENTER
            ,NULL AS GROUP_COMMENT
            ,WMH.START_TIME
            ,WMH.END_TIME
            ,WMH.CREDITS_USED
            ,$CREDIT_PRICE
            ,($CREDIT_PRICE*WMH.CREDITS_USED) AS DOLLARS_USED
            ,'ACTUAL COMPUTE' AS MEASURE_TYPE
   from     SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY WMH
   UNION ALL
   --COMPUTE FROM SNOWPIPE
   SELECT
             'Snowpipe' AS WAREHOUSE_GROUP_NAME
            ,PUH.PIPE_NAME AS WAREHOUSE_NAME
```

```
        ,NULL AS GROUP_CONTACT
        ,NULL AS GROUP_COST_CENTER
        ,NULL AS GROUP_COMMENT
        ,PUH.START_TIME
        ,PUH.END_TIME
        ,PUH.CREDITS_USED
        ,$CREDIT_PRICE
        ,($CREDIT_PRICE*PUH.CREDITS_USED) AS DOLLARS_USED
        ,'ACTUAL COMPUTE' AS MEASURE_TYPE
from    SNOWFLAKE.ACCOUNT_USAGE.PIPE_USAGE_HISTORY PUH
UNION ALL
--COMPUTE FROM CLUSTERING
SELECT
         'Auto Clustering' AS WAREHOUSE_GROUP_NAME
        ,DATABASE_NAME || '.' || SCHEMA_NAME || '.' || TABLE_NAME AS WAREHOUSE_NAME
        ,NULL AS GROUP_CONTACT
        ,NULL AS GROUP_COST_CENTER
        ,NULL AS GROUP_COMMENT
        ,ACH.START_TIME
        ,ACH.END_TIME
        ,ACH.CREDITS_USED
        ,$CREDIT_PRICE
        ,($CREDIT_PRICE*ACH.CREDITS_USED) AS DOLLARS_USED
        ,'ACTUAL COMPUTE' AS MEASURE_TYPE
from    SNOWFLAKE.ACCOUNT_USAGE.AUTOMATIC_CLUSTERING_HISTORY ACH
UNION ALL
--COMPUTE FROM MATERIALIZED VIEWS
SELECT
         'Materialized Views' AS WAREHOUSE_GROUP_NAME
        ,DATABASE_NAME || '.' || SCHEMA_NAME || '.' || TABLE_NAME AS WAREHOUSE_NAME
        ,NULL AS GROUP_CONTACT
        ,NULL AS GROUP_COST_CENTER
        ,NULL AS GROUP_COMMENT
        ,MVH.START_TIME
        ,MVH.END_TIME
        ,MVH.CREDITS_USED
        ,$CREDIT_PRICE
        ,($CREDIT_PRICE*MVH.CREDITS_USED) AS DOLLARS_USED
        ,'ACTUAL COMPUTE' AS MEASURE_TYPE
from    SNOWFLAKE.ACCOUNT_USAGE.MATERIALIZED_VIEW_REFRESH_HISTORY MVH
UNION ALL
--COMPUTE FROM SEARCH OPTIMIZATION
SELECT
         'Search Optimization' AS WAREHOUSE_GROUP_NAME
        ,DATABASE_NAME || '.' || SCHEMA_NAME || '.' || TABLE_NAME AS WAREHOUSE_NAME
        ,NULL AS GROUP_CONTACT
        ,NULL AS GROUP_COST_CENTER
        ,NULL AS GROUP_COMMENT
        ,SOH.START_TIME
        ,SOH.END_TIME
        ,SOH.CREDITS_USED
        ,$CREDIT_PRICE
```

```
           ,($CREDIT_PRICE*SOH.CREDITS_USED) AS DOLLARS_USED
           ,'ACTUAL COMPUTE' AS MEASURE_TYPE
   from    SNOWFLAKE.ACCOUNT_USAGE.SEARCH_OPTIMIZATION_HISTORY SOH
   UNION ALL
   --COMPUTE FROM REPLICATION
   SELECT
            'Replication' AS WAREHOUSE_GROUP_NAME
           ,DATABASE_NAME AS WAREHOUSE_NAME
           ,NULL AS GROUP_CONTACT
           ,NULL AS GROUP_COST_CENTER
           ,NULL AS GROUP_COMMENT
           ,RUH.START_TIME
           ,RUH.END_TIME
           ,RUH.CREDITS_USED
           ,$CREDIT_PRICE
           ,($CREDIT_PRICE*RUH.CREDITS_USED) AS DOLLARS_USED
           ,'ACTUAL COMPUTE' AS MEASURE_TYPE
   from    SNOWFLAKE.ACCOUNT_USAGE.REPLICATION_USAGE_HISTORY RUH
   UNION ALL

   --STORAGE COSTS
   SELECT
            'Storage' AS WAREHOUSE_GROUP_NAME
           ,'Storage' AS WAREHOUSE_NAME
           ,NULL AS GROUP_CONTACT
           ,NULL AS GROUP_COST_CENTER
           ,NULL AS GROUP_COMMENT
           ,SU.USAGE_DATE
           ,SU.USAGE_DATE
           ,NULL AS CREDITS_USED
           ,$CREDIT_PRICE
           ,((STORAGE_BYTES + STAGE_BYTES +
FAILSAFE_BYTES)/(1024*1024*1024*1024)*23)/DA.DAYS_IN_MONTH AS DOLLARS_USED
           ,'ACTUAL COMPUTE' AS MEASURE_TYPE
   from    SNOWFLAKE.ACCOUNT_USAGE.STORAGE_USAGE SU
   JOIN    (SELECT COUNT(*) AS DAYS_IN_MONTH,TO_DATE(DATE_PART('year',D_DATE)||'-
'||DATE_PART('month',D_DATE)||'-01') as DATE_MONTH FROM
SNOWFLAKE_SAMPLE_DATA.TPCDS_SF10TCL.DATE_DIM GROUP BY
TO_DATE(DATE_PART('year',D_DATE)||'-'||DATE_PART('month',D_DATE)||'-01')) DA ON
DA.DATE_MONTH = TO_DATE(DATE_PART('year',USAGE_DATE)||'-
'||DATE_PART('month',USAGE_DATE)||'-01')
) A
 group by 1
),
FORECASTED_USAGE_SLOPE_INTERCEPT as (
 SELECT

REGR_SLOPE(AU.ACTUAL_DOLLARS_USED,DATEDIFF(day,CONTRACT_START_DATE,AU.CONSUMPTION_DATE))
 as SLOPE

,REGR_INTERCEPT(AU.ACTUAL_DOLLARS_USED,DATEDIFF(day,CONTRACT_START_DATE,AU.CONSUMPTION_I
 as INTERCEPT
```

```
 FROM        PROJECTED_USAGE PU
 JOIN        SNOWFLAKE_SAMPLE_DATA.TPCDS_SF10TCL.DATE_DIM DA ON DA.D_DATE BETWEEN
PU.CONTRACT_START_DATE AND PU.CONTRACT_END_DATE
 LEFT JOIN   ACTUAL_USAGE AU ON AU.CONSUMPTION_DATE = TO_DATE(DA.D_DATE)
)
SELECT
        DA.D_DATE::date as CONSUMPTION_DATE
       ,PU.DOLLARS_PER_DAY AS CONTRACTED_DOLLARS_USED
       ,AU.ACTUAL_DOLLARS_USED
       --the below is the mx+b equation to get the forecasted linear slope
       ,DATEDIFF(day,CONTRACT_START_DATE,DA.D_DATE)*FU.SLOPE + FU.INTERCEPT AS
FORECASTED_DOLLARS_USED
FROM        PROJECTED_USAGE PU
JOIN        SNOWFLAKE_SAMPLE_DATA.TPCDS_SF10TCL.DATE_DIM    DA ON DA.D_DATE BETWEEN
PU.CONTRACT_START_DATE AND PU.CONTRACT_END_DATE
LEFT JOIN   ACTUAL_USAGE                                    AU ON AU.CONSUMPTION_DATE
= TO_DATE(DA.D_DATE)
JOIN        FORECASTED_USAGE_SLOPE_INTERCEPT                FU ON 1 = 1
;
```

Screenshot



Partner Tools Consuming Credits (T1)

TIER 1

Description:

Identifies which of Snowflake's partner tools/solutions (BI, ETL, etc.) are consuming the most credits.

How to Interpret Results:

Are there certain partner solutions that are consuming more credits than anticipated? What is the reasoning for this?

Primary Schema:

Account_Usage

SQL

```
--THIS IS APPROXIMATE CREDIT CONSUMPTION BY CLIENT APPLICATION
WITH CLIENT_HOUR_EXECUTION_CTE AS (
    SELECT  CASE
         WHEN CLIENT_APPLICATION_ID LIKE 'Go %' THEN 'Go'
         WHEN CLIENT_APPLICATION_ID LIKE 'Snowflake UI %' THEN 'Snowflake UI'
         WHEN CLIENT_APPLICATION_ID LIKE 'SnowSQL %' THEN 'SnowSQL'
         WHEN CLIENT_APPLICATION_ID LIKE 'JDBC %' THEN 'JDBC'
         WHEN CLIENT_APPLICATION_ID LIKE 'PythonConnector %' THEN 'Python'
         WHEN CLIENT_APPLICATION_ID LIKE 'ODBC %' THEN 'ODBC'
         ELSE 'NOT YET MAPPED: ' || CLIENT_APPLICATION_ID
       END AS CLIENT_APPLICATION_NAME
    ,WAREHOUSE_NAME
    ,DATE_TRUNC('hour',START_TIME) as START_TIME_HOUR
    ,SUM(EXECUTION_TIME)  as CLIENT_HOUR_EXECUTION_TIME
    FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY" QH
    JOIN "SNOWFLAKE"."ACCOUNT_USAGE"."SESSIONS" SE ON SE.SESSION_ID = QH.SESSION_ID
    WHERE WAREHOUSE_NAME IS NOT NULL
    AND EXECUTION_TIME > 0

 --Change the below filter if you want to look at a longer range than the last 1 month
    AND START_TIME > DATEADD(Month,-1,CURRENT_TIMESTAMP())
    group by 1,2,3
    )
, HOUR_EXECUTION_CTE AS (
    SELECT  START_TIME_HOUR
    ,WAREHOUSE_NAME
    ,SUM(CLIENT_HOUR_EXECUTION_TIME) AS HOUR_EXECUTION_TIME
    FROM CLIENT_HOUR_EXECUTION_CTE
    group by 1,2
)
, APPROXIMATE_CREDITS AS (
    SELECT
    A.CLIENT_APPLICATION_NAME
    ,C.WAREHOUSE_NAME
    ,(A.CLIENT_HOUR_EXECUTION_TIME/B.HOUR_EXECUTION_TIME)*C.CREDITS_USED AS
APPROXIMATE_CREDITS_USED

    FROM CLIENT_HOUR_EXECUTION_CTE A
    JOIN HOUR_EXECUTION_CTE B  ON A.START_TIME_HOUR = B.START_TIME_HOUR and
B.WAREHOUSE_NAME = A.WAREHOUSE_NAME
    JOIN "SNOWFLAKE"."ACCOUNT_USAGE"."WAREHOUSE_METERING_HISTORY" C ON
C.WAREHOUSE_NAME = A.WAREHOUSE_NAME AND C.START_TIME = A.START_TIME_HOUR
)

SELECT
 CLIENT_APPLICATION_NAME
,WAREHOUSE_NAME
,SUM(APPROXIMATE_CREDITS_USED) AS APPROXIMATE_CREDITS_USED
FROM APPROXIMATE_CREDITS
GROUP BY 1,2
```

```
ORDER BY 3 DESC
;
```

Screenshot

CLIENT_APPLICATION_NAME	WAREHOUSE_NAME	APPROXIMATE_CREDITS_USED
Snowflake UI	CARLIN_TEST	58.450691666766
Snowflake UI	DJZ_WH	57.789665776081
Snowflake UI	CENG_MANUAL_CLUSTER	53.324313888000
Snowflake UI	JCRAMER_WH	43.348983612000
Snowflake UI	VLAD	39.626438141602
Go	JFIELDING_WH	37.928385557000
Snowflake UI	SNOWFLAKE_WH	24.983436389000
Snowflake UI	CHECHANI_WH	21.905146390000
Snowflake UI	JKGENOMICS_WH	16.159712193157
Snowflake UI	ADW_WH	10.795769675182
Snowflake UI	JC1	9.255301946000
Snowflake UI	SGURSOY_LOAD_WH	7.336536945000
ODBC	JKGENOMICS_WH	6.679740861843
Snowflake UI	TNORTELL_WH	6.108621388000
Go	AP_WAREHOUSE	5.868022778000
ODBC	STOM_WH	5.309183949844
Snowflake UI	RKANN_WH	4.706433055000
Snowflake UI	NEL_XL	4.659713055000

Credit Consumption by User (T1)

TIER 1

Description:

Identifies which users are consuming the most credits within your Snowflake environment.

How to Interpret Results:

Are there certain users that are consuming more credits than they should? What is the purpose behind this additional usage?

Primary Schema:

Account_Usage

SQL

```
--THIS IS APPROXIMATE CREDIT CONSUMPTION BY USER
WITH USER_HOUR_EXECUTION_CTE AS (
    SELECT  USER_NAME
    ,WAREHOUSE_NAME
```

```
    ,DATE_TRUNC('hour',START_TIME) as START_TIME_HOUR
    ,SUM(EXECUTION_TIME)  as USER_HOUR_EXECUTION_TIME
    FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
    WHERE WAREHOUSE_NAME IS NOT NULL
    AND EXECUTION_TIME > 0

 --Change the below filter if you want to look at a longer range than the last 1 month
    AND START_TIME > DATEADD(Month,-1,CURRENT_TIMESTAMP())
    group by 1,2,3
    )
, HOUR_EXECUTION_CTE AS (
    SELECT  START_TIME_HOUR
    ,WAREHOUSE_NAME
    ,SUM(USER_HOUR_EXECUTION_TIME) AS HOUR_EXECUTION_TIME
    FROM USER_HOUR_EXECUTION_CTE
    group by 1,2
)
, APPROXIMATE_CREDITS AS (
    SELECT
    A.USER_NAME
    ,C.WAREHOUSE_NAME
    ,(A.USER_HOUR_EXECUTION_TIME/B.HOUR_EXECUTION_TIME)*C.CREDITS_USED AS
APPROXIMATE_CREDITS_USED

    FROM USER_HOUR_EXECUTION_CTE A
    JOIN HOUR_EXECUTION_CTE B  ON A.START_TIME_HOUR = B.START_TIME_HOUR and
B.WAREHOUSE_NAME = A.WAREHOUSE_NAME
    JOIN "SNOWFLAKE"."ACCOUNT_USAGE"."WAREHOUSE_METERING_HISTORY" C ON
C.WAREHOUSE_NAME = A.WAREHOUSE_NAME AND C.START_TIME = A.START_TIME_HOUR
)

SELECT
 USER_NAME
,WAREHOUSE_NAME
,SUM(APPROXIMATE_CREDITS_USED) AS APPROXIMATE_CREDITS_USED
FROM APPROXIMATE_CREDITS
GROUP BY 1,2
ORDER BY 3 DESC
;
```

Screenshot

USER_NAME	WAREHOUSE_NAME	APPROXIMATE_CREDITS_USED
JOHN	COMPUTE_WH	5.710443333000
JOHN	ANALYTICS_WH	3.794843889000
JOHN	LOAD_WH	3.651798056000
JOHN	DATALAKE_WH	2.666681944000
JOHN	RESET_WH	0.705689722000
JOHN	BI_LARGE_WH	0.457556112000
JOHN	TEST_WH1	0.166753889000
SYSTEM	TASK_WH	0.078918473061
JOHN	TASK_WH	0.000748470939
JOHN	BI_MEDIUM_WH	0.000078889000

Queries by # of Times Executed and Execution Time (T2)

TIER 2

Description:

Are there any queries that get executed a ton?? how much execution time do they take up?

How to Interpret Results:

Opportunity to materialize the result set as a table?

Primary Schema:

Account_Usage

SQL

```
SELECT
QUERY_TEXT
,count(*) as number_of_queries
,sum(TOTAL_ELAPSED_TIME)/1000 as execution_seconds
,sum(TOTAL_ELAPSED_TIME)/(1000*60) as execution_minutes
,sum(TOTAL_ELAPSED_TIME)/(1000*60*60) as execution_hours

  from SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY Q
  where 1=1
  and TO_DATE(Q.START_TIME) >     DATEADD(month,-1,TO_DATE(CURRENT_TIMESTAMP()))
 and TOTAL_ELAPSED_TIME > 0 --only get queries that actually used compute
  group by 1
  having count(*) >= 10 --configurable/minimal threshold
  order by 2 desc
  limit 100 --configurable upper bound threshold
  ;
```

Top 50 Longest Running Queries (T2)

TIER 2

Description:

Looks at the top 50 longest running queries to see if there are patterns

How to Interpret Results:

Is there an opportunity to optimize with clustering or upsize the warehouse?

Primary Schema:

Account_Usage

SQL

```
select

          QUERY_ID
         --reconfigure the url if your account is not in AWS US-West

,'https://'||CURRENT_ACCOUNT()||'.snowflakecomputing.com/console#/monitoring/queries/det
queryId='||Q.QUERY_ID as QUERY_PROFILE_URL
         ,ROW_NUMBER() OVER(ORDER BY PARTITIONS_SCANNED DESC) as QUERY_ID_INT
         ,QUERY_TEXT
         ,TOTAL_ELAPSED_TIME/1000 AS QUERY_EXECUTION_TIME_SECONDS
         ,PARTITIONS_SCANNED
         ,PARTITIONS_TOTAL

from SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY Q
 where 1=1
  and TO_DATE(Q.START_TIME) >     DATEADD(month,-1,TO_DATE(CURRENT_TIMESTAMP()))
    and TOTAL_ELAPSED_TIME > 0 --only get queries that actually used compute
    and ERROR_CODE iS NULL
    and PARTITIONS_SCANNED is not null

  order by  TOTAL_ELAPSED_TIME desc

   LIMIT 50
   ;
```

Top 50 Queries that Scanned the Most Data (T2)

TIER 2

Description:

Looks at the top 50 queries that scan the largest number of micro partitions

How to Interpret Results:

Is there an opportunity to optimize with clustering or upsize the warehouse?

Primary Schema:

Account_Usage

SQL

```
select

          QUERY_ID
          --reconfigure the url if your account is not in AWS US-West

,'https://'||CURRENT_ACCOUNT()||'.snowflakecomputing.com/console#/monitoring/queries/det
queryId='||Q.QUERY_ID as QUERY_PROFILE_URL
         ,ROW_NUMBER() OVER(ORDER BY PARTITIONS_SCANNED DESC) as QUERY_ID_INT
         ,QUERY_TEXT
         ,TOTAL_ELAPSED_TIME/1000 AS QUERY_EXECUTION_TIME_SECONDS
         ,PARTITIONS_SCANNED
         ,PARTITIONS_TOTAL

from SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY Q
 where 1=1
  and TO_DATE(Q.START_TIME) >     DATEADD(month,-1,TO_DATE(CURRENT_TIMESTAMP()))
    and TOTAL_ELAPSED_TIME > 0 --only get queries that actually used compute
    and ERROR_CODE iS NULL
    and PARTITIONS_SCANNED is not null

  order by  PARTITIONS_SCANNED desc

   LIMIT 50
   ;
```

Queries by Execution Buckets over the Past 7 Days (T2)

TIER 2

Description:

Group the queries for a given warehouse by execution time buckets

How to Interpret Results:

This is an opportunity to identify query SLA trends and make a decision to downsize a warehouse, upsize a warehouse, or separate out some queries to another warehouse

Primary Schema:

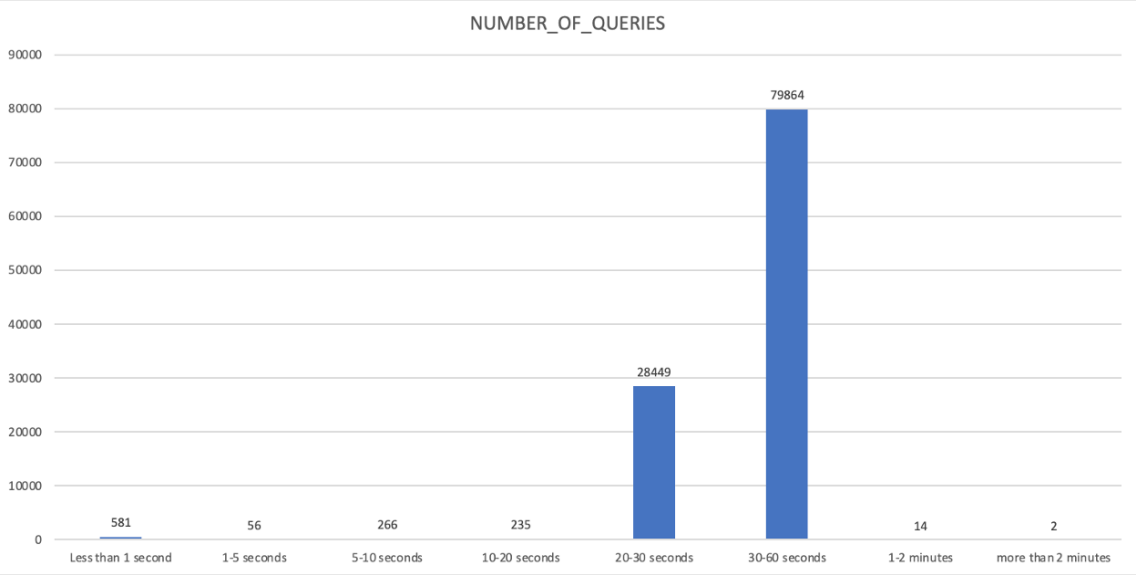
Account_Usage

SQL

```
WITH BUCKETS AS (

SELECT 'Less than 1 second' as execution_time_bucket, 0 as execution_time_lower_bound,
1000 as execution_time_upper_bound
UNION ALL
SELECT '1-5 seconds' as execution_time_bucket, 1000 as execution_time_lower_bound,
5000 as execution_time_upper_bound
UNION ALL
SELECT '5-10 seconds' as execution_time_bucket, 5000 as execution_time_lower_bound,
```

```
10000 as execution_time_upper_bound
UNION ALL
SELECT '10-20 seconds' as execution_time_bucket, 10000 as execution_time_lower_bound,
20000 as execution_time_upper_bound
UNION ALL
SELECT '20-30 seconds' as execution_time_bucket, 20000 as execution_time_lower_bound,
30000 as execution_time_upper_bound
UNION ALL
SELECT '30-60 seconds' as execution_time_bucket, 30000 as execution_time_lower_bound,
60000 as execution_time_upper_bound
UNION ALL
SELECT '1-2 minutes' as execution_time_bucket, 60000 as execution_time_lower_bound,
120000 as execution_time_upper_bound
UNION ALL
SELECT 'more than 2 minutes' as execution_time_bucket, 120000 as
execution_time_lower_bound, NULL as execution_time_upper_bound
)

SELECT
 COALESCE(execution_time_bucket,'more than 2 minutes')
,count(Query_ID) as number_of_queries

from SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY Q
FULL OUTER JOIN BUCKETS B ON (Q.TOTAL_ELAPSED_TIME) >= B.execution_time_lower_bound
and (Q.TOTAL_ELAPSED_TIME) < B.execution_time_upper_bound
where Q.Query_ID is null
OR (
TO_DATE(Q.START_TIME) >= DATEADD(week,-1,TO_DATE(CURRENT_TIMESTAMP()))
and warehouse_name = <WAREHOUSE_NAME>
and TOTAL_ELAPSED_TIME > 0
  )
group by 1,COALESCE(b.execution_time_lower_bound,120000)
order by COALESCE(b.execution_time_lower_bound,120000)
  ;
```

Screenshot



Warehouses with High Cloud Services Usage (T2)

TIER 2

Description:

Shows the warehouses that are not using enough compute to cover the cloud services portion of compute, ordered by the ratio of cloud services to total compute

How to Interpret Results:

Focus on Warehouses that are using a high volume and ratio of cloud services compute. Investigate why this is the case to reduce overall cost (might be cloning, listing files in S3, partner tools setting session parameters, etc.). The goal to reduce cloud services credit consumption is to aim for cloud services credit to be less than 10% of overall credits. ####Primary Schema: Account_Usage

SQL

```
select
    WAREHOUSE_NAME
    ,SUM(CREDITS_USED) as CREDITS_USED
    ,SUM(CREDITS_USED_CLOUD_SERVICES) as CREDITS_USED_CLOUD_SERVICES
    ,SUM(CREDITS_USED_CLOUD_SERVICES)/SUM(CREDITS_USED) as PERCENT_CLOUD_SERVICES
from "SNOWFLAKE"."ACCOUNT_USAGE"."WAREHOUSE_METERING_HISTORY"
where TO_DATE(START_TIME) >= DATEADD(month,-1,CURRENT_TIMESTAMP())
and CREDITS_USED_CLOUD_SERVICES > 0
group by 1
order by 4 desc
;
```

Screenshot

WAREHOUSE_NAME	CREDITS_USED	CREDITS_USED_CLOUD_SERVICES	PERCENT_CLOUD_SERVICES
BI_MEDIUM_WH	0.000078889	0.000078889	1.000000000000
BI_LARGE_WH	0.438058612	0.438058612	1.000000000000
CLOUD_SERVICES_ONLY	0.002304444	0.002304444	1.000000000000
ANALYTICS_WH	3.794843889	0.084288334	0.022211278373
DATALAKE_WH	13.049561666	0.222895000	0.017080650347
LOAD_WH	3.651798056	0.030131388	0.008251110148
RESET_WH	0.705689722	0.003467500	0.004913632567
TASK_WH	0.079666944	0.000222500	0.002792877307
COMPUTE_WH	5.710443333	0.003776667	0.000661361435
TEST_WH1	0.166753889	0.000087222	0.000523058266

Warehouse Utilization (T2)

TIER 2

Description:

This query is designed to give a rough idea of how busy Warehouses are compared to the credit consumption per hour. It will show the end user the number of credits consumed, the number of queries executed and the total execution time of those queries in each hour window.

How to Interpret Results:

This data can be used to draw correlations between credit consumption and the #/duration of query executions. The more queries or higher query duration for the fewest number of credits may help drive more value per credit.

Primary Schema:

Account_Usage

SQL

```
SELECT
       WMH.WAREHOUSE_NAME
      ,WMH.START_TIME
      ,WMH.CREDITS_USED
      ,SUM(COALESCE(B.EXECUTION_TIME_SECONDS,0)) as TOTAL_EXECUTION_TIME_SECONDS
      ,SUM(COALESCE(QUERY_COUNT,0)) AS QUERY_COUNT

FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY WMH
LEFT JOIN (

      --QUERIES FULLY EXECUTED WITHIN THE HOUR
      SELECT
         WMH.WAREHOUSE_NAME
        ,WMH.START_TIME
        ,SUM(COALESCE(QH.EXECUTION_TIME,0))/(1000) AS EXECUTION_TIME_SECONDS
        ,COUNT(DISTINCT QH.QUERY_ID) AS QUERY_COUNT
      FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY     WMH
      JOIN SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY             QH ON QH.WAREHOUSE_NAME =
WMH.WAREHOUSE_NAME
                                                                    AND
```

```
QH.START_TIME BETWEEN WMH.START_TIME AND WMH.END_TIME
                                                                    AND
QH.END_TIME BETWEEN WMH.START_TIME AND WMH.END_TIME
      WHERE TO_DATE(WMH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      AND TO_DATE(QH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      GROUP BY
      WMH.WAREHOUSE_NAME
      ,WMH.START_TIME

      UNION ALL

      --FRONT part OF QUERIES Executed longer than 1 Hour
      SELECT
         WMH.WAREHOUSE_NAME
        ,WMH.START_TIME
        ,SUM(COALESCE(DATEDIFF(seconds,QH.START_TIME,WMH.END_TIME),0)) AS
EXECUTION_TIME_SECONDS
        ,COUNT(DISTINCT QUERY_ID) AS QUERY_COUNT
      FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY     WMH
      JOIN SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY             QH ON QH.WAREHOUSE_NAME =
WMH.WAREHOUSE_NAME
                                                                    AND
QH.START_TIME BETWEEN WMH.START_TIME AND WMH.END_TIME
                                                                    AND
QH.END_TIME > WMH.END_TIME
      WHERE TO_DATE(WMH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      AND TO_DATE(QH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      GROUP BY
      WMH.WAREHOUSE_NAME
      ,WMH.START_TIME

      UNION ALL

      --Back part OF QUERIES Executed longer than 1 Hour
      SELECT
         WMH.WAREHOUSE_NAME
        ,WMH.START_TIME
        ,SUM(COALESCE(DATEDIFF(seconds,WMH.START_TIME,QH.END_TIME),0)) AS
EXECUTION_TIME_SECONDS
        ,COUNT(DISTINCT QUERY_ID) AS QUERY_COUNT
      FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY     WMH
      JOIN SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY             QH ON QH.WAREHOUSE_NAME =
WMH.WAREHOUSE_NAME
                                                                    AND
QH.END_TIME BETWEEN WMH.START_TIME AND WMH.END_TIME
                                                                    AND
QH.START_TIME < WMH.START_TIME
      WHERE TO_DATE(WMH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      AND TO_DATE(QH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      GROUP BY
      WMH.WAREHOUSE_NAME
      ,WMH.START_TIME
```

```
      UNION ALL

      --Middle part OF QUERIES Executed longer than 1 Hour
      SELECT
         WMH.WAREHOUSE_NAME
        ,WMH.START_TIME
        ,SUM(COALESCE(DATEDIFF(seconds,WMH.START_TIME,WMH.END_TIME),0)) AS
EXECUTION_TIME_SECONDS
        ,COUNT(DISTINCT QUERY_ID) AS QUERY_COUNT
      FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY     WMH
      JOIN SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY             QH ON QH.WAREHOUSE_NAME =
WMH.WAREHOUSE_NAME
                                                                    AND
WMH.START_TIME > QH.START_TIME
                                                                    AND
WMH.END_TIME < QH.END_TIME
      WHERE TO_DATE(WMH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      AND TO_DATE(QH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
      GROUP BY
      WMH.WAREHOUSE_NAME
      ,WMH.START_TIME

) B ON B.WAREHOUSE_NAME = WMH.WAREHOUSE_NAME AND B.START_TIME = WMH.START_TIME

WHERE TO_DATE(WMH.START_TIME) >= DATEADD(week,-1,CURRENT_TIMESTAMP())
GROUP BY

      WMH.WAREHOUSE_NAME
      ,WMH.START_TIME
      ,WMH.CREDITS_USED
;
```

Screenshot

WAREHOUSE_NAME	START_TIME	CREDITS_USED	TOTAL_EXECUTION_TIME_SECO	QUERY_COUNT
ANALYTICS_WH	2020-10-28 11:00:00.000 -0...	0.699688889	75.129000	23
DATALAKE_WH	2020-10-28 11:00:00.000 -0...	2.666681944	55.043000	2

Snowflake for Data Lake

Overview

This Quickstart is intended to help you use a sample of features available in Snowflake for your cloud data lake. This lab does not require that you have an existing data lake. All data will be provided in a publicly available cloud storage location. The datasets used in this Quickstart contain trip data in Apache Parquet format from the Citibike transportation company in New York. To show Snowflake's support for unstructured data, we'll also use research papers (PDF) from the 2020 Conference on Neural Information Processing Systems.

Prerequisites

- Use of the Snowflake free 30-day trial
- Basic knowledge of SQL, database concepts, and objects
- Familiarity with JSON and Apache Parquet semi-structured data

What You'll Learn

- How to query partitioned semi-structured data stored in files in external cloud object storage with **External Tables**
- How to use **Schema Detection** to automatically determine and return the schema of staged files
- How to improve performance of queries over external object stores with **Materialized Views** over External Tables
- Process and analyze **unstructured data** with **Snowpark**

What You'll Build

- An External Function to fetch JSON data from an API and store in a table
- An External Stage and External Table for querying Apache Parquet files stored in Amazon S3
- A Materialized View over an External Table to improve performance
- A Java UDF for processing PDF files

Prepare Your Lab Environment

If you haven't already, register for a Snowflake free 30-day trial. The cloud provider (AWS, Azure, Google Cloud), and Region (US East, EU, e.g.) do *not* matter for this lab. However, we suggest you select the region which is physically closest to you. Select the **Enterprise** edition so you can use some advanced capabilities that are not available in the Standard edition. After registering, you will receive an email with an activation link and your Snowflake account URL. Bookmark this URL for easy, future access. After activation, you will create a username and password. Write down these credentials.

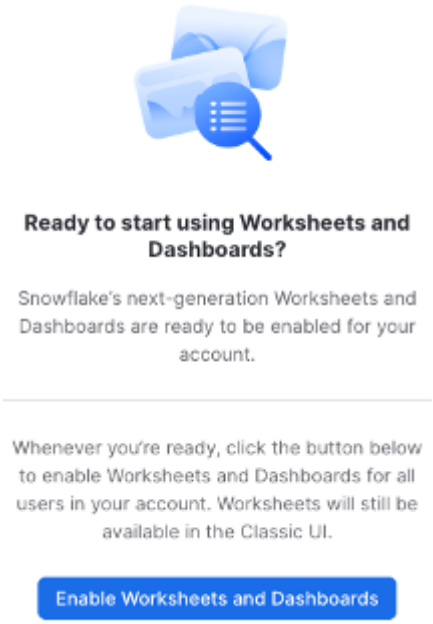
To easily follow the instructions, resize your browser windows so you can view this Quickstart and your Snowflake environment side-by-side. If possible, even better is to use a secondary display dedicated to the Quickstart.

While commands from this Quickstart can be copy/pasted to a worksheet in your Snowflake account, you can optionally download the script as a SQL file to your local machine.

Navigating to Snowsight

For this lab, you will use the latest Snowflake web interface, Snowsight.

1. Log into your Snowflake trial account
2. Click on **Snowsight** Worksheets tab. The new web interface opens in a separate tab or window.
3. Click **Worksheets** in the left-hand navigation bar. The **Ready to Start Using Worksheets and Dashboards** dialog opens.
4. Click the **Enable Worksheets and Dashboards button**.



Create a Warehouse, Database, Schemas

Create a warehouse, database, and schema that will be used for loading, storing, processing, and querying data for this Quickstart. We will use the UI within the Worksheets tab to run the DDL that creates these objects.

Copy the commands below into your trial environment. To execute a single statement, just position the cursor anywhere within the statement and click the Run button. To execute several statements, they must be highlighted through the final semi-colon prior to clicking the Run button.

```
use role SYSADMIN;
create or replace warehouse LOAD_WH with
  warehouse_size = 'xlarge'
  auto_suspend = 300
  initially_suspended = true;

use warehouse LOAD_WH;

create or replace database CITIBIKE_LAB;
create or replace schema DEMO;
create or replace schema UTILS;
```

Load the Data

Create an API integration to support creating an external function call to a REST API.

```
use schema UTILS;
use role accountadmin;

create or replace api integration fetch_http_data
  api_provider = aws_api_gateway
  api_aws_role_arn = 'arn:aws:iam::148887191972:role/ExecuteLambdaFunction'
```

```
  enabled = true
  api_allowed_prefixes = ('https://dr14z5kz5d.execute-api.us-east-
1.amazonaws.com/prod/fetchhttpdata');

grant usage on integration fetch_http_data to role sysadmin;
```

Now create the external function that uses the API integration object.

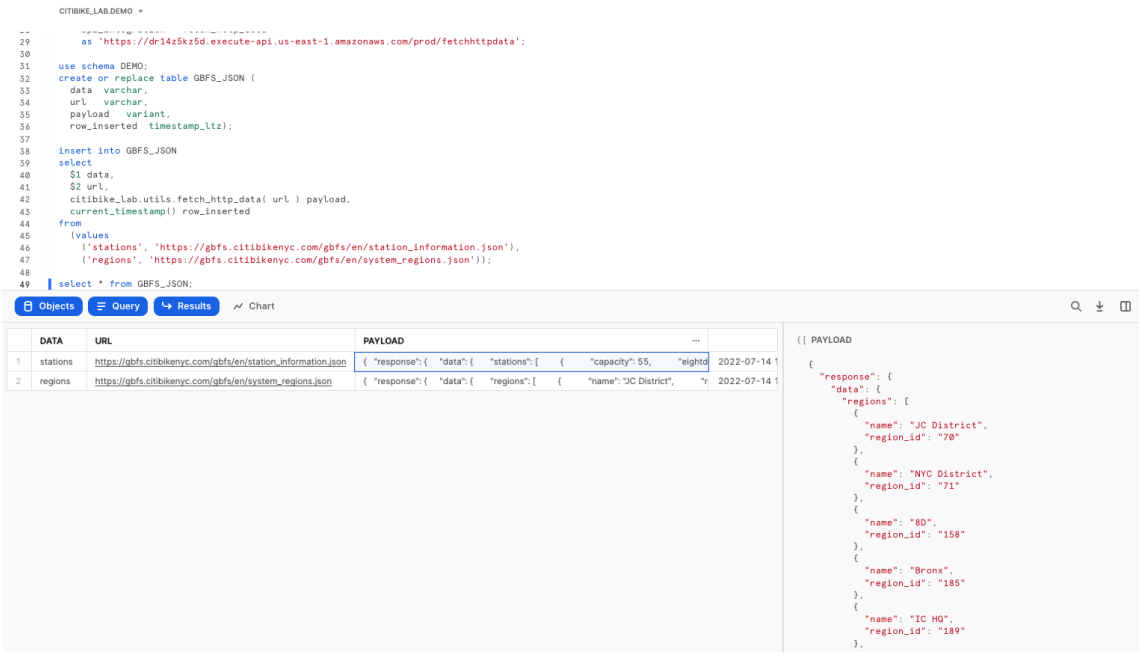
```
use role sysadmin;

create or replace external function utils.fetch_http_data(v varchar)
    returns variant
    api_integration = fetch_http_data
    as 'https://dr14z5kz5d.execute-api.us-east-1.amazonaws.com/prod/fetchhttpdata';
```

Create a few reference tables in which the data will be stored.

```
use schema DEMO;
create or replace table GBFS_JSON (
  data    varchar,
  url    varchar,
  payload    variant,
  row_inserted    timestamp_ltz);
```

Now populate the table with raw JSON data through the external function call, then preview the contents of the table `GBFS_JSON`.

```
insert into GBFS_JSON
select
  $1 data,
  $2 url,
  citibike_lab.utils.fetch_http_data( url ) payload,
  current_timestamp() row_inserted
from
  (values
    ('stations', 'https://gbfs.citibikenyc.com/gbfs/en/station_information.json'),
    ('regions', 'https://gbfs.citibikenyc.com/gbfs/en/system_regions.json'));

select * from GBFS_JSON;
```



Now refine that raw JSON data by extracting out the `STATIONS` nodes, storing the results in a separate table.

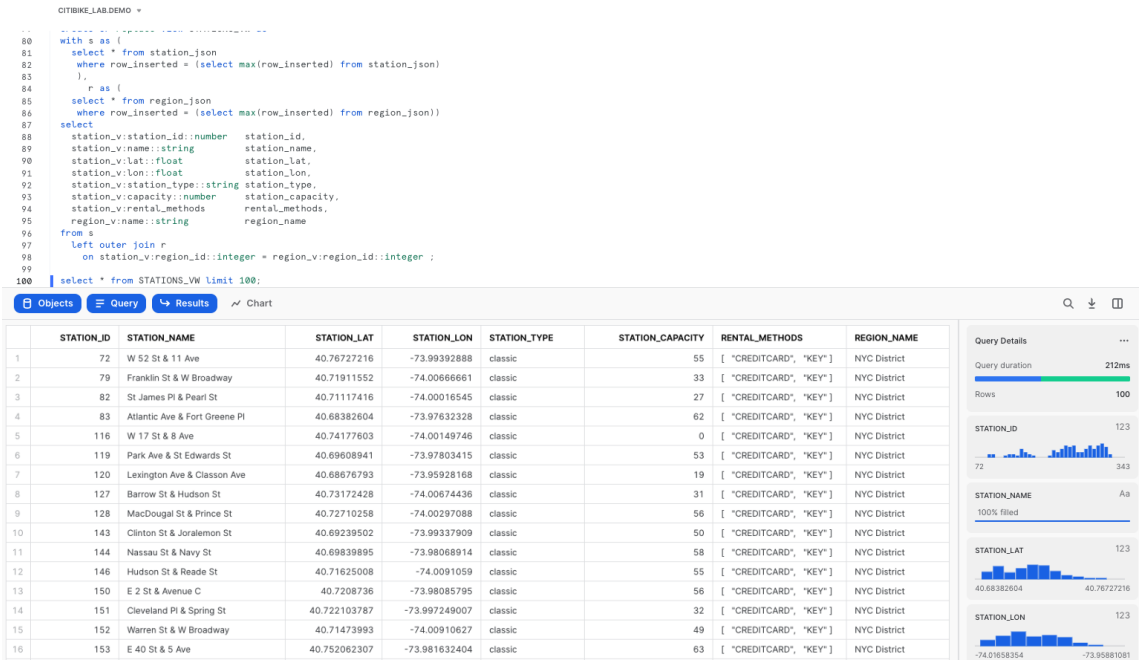
```
create or replace table STATION_JSON as
with s as (
  select payload, row_inserted
    from gbfs_json
   where data = 'stations'
     and row_inserted = (select max(row_inserted) from gbfs_json)
  )
select
  value station_v,
  payload:response.last_updated::timestamp last_updated,
  row_inserted
from s,
  lateral flatten (input => payload:response.data.stations) ;
```

Extract the individual region records, storing the results in a separate table.

```
create or replace table REGION_JSON as
with r as (
  select payload, row_inserted
    from gbfs_json
   where data = 'regions'
     and row_inserted = (select max(row_inserted) from gbfs_json)
  )
select
  value region_v,
  payload:response.last_updated::timestamp last_updated,
  row_inserted
from r,
  lateral flatten (input => payload:response.data.regions);
```

Lastly, create a view that flattens and joins the two tables a standard tabular structure.

```
create or replace view STATIONS_VW as
with s as (
  select * from station_json
   where row_inserted = (select max(row_inserted) from station_json)
   ),
     r as (
  select * from region_json
   where row_inserted = (select max(row_inserted) from region_json))
select
  station_v:station_id::number   station_id,
  station_v:name::string         station_name,
  station_v:lat::float           station_lat,
  station_v:lon::float           station_lon,
  station_v:station_type::string station_type,
  station_v:capacity::number     station_capacity,
  station_v:rental_methods       rental_methods,
  region_v:name::string          region_name
from s
  left outer join r
    on station_v:region_id::integer = region_v:region_id::integer ;

select * from STATIONS_VW limit 100;
```



	STATION_ID	STATION_NAME	STATION_LAT	STATION_LON	STATION_TYPE	STATION_CAPACITY	RENTAL_METHODS	REGION_NAME
1	72	W 52 St & 11 Ave	40.76727216	-73.99392888	classic	55	["CREDITCARD", "KEY"]	NYC District
2	79	Franklin St & W Broadway	40.71911552	-74.00666661	classic	33	["CREDITCARD", "KEY"]	NYC District
3	82	St James Pl & Pearl St	40.71117416	-74.00016545	classic	27	["CREDITCARD", "KEY"]	NYC District
4	83	Atlantic Ave & Fort Greene Pl	40.68382604	-73.97632328	classic	62	["CREDITCARD", "KEY"]	NYC District
5	116	W 17 St & 8 Ave	40.74177603	-74.00149746	classic	0	["CREDITCARD", "KEY"]	NYC District
6	119	Park Ave & St Edwards St	40.69608941	-73.97803415	classic	53	["CREDITCARD", "KEY"]	NYC District
7	120	Lexington Ave & Classon Ave	40.68676793	-73.95928168	classic	19	["CREDITCARD", "KEY"]	NYC District
8	127	Barrow St & Hudson St	40.73172428	-74.00674436	classic	31	["CREDITCARD", "KEY"]	NYC District
9	128	MacDougal St & Prince St	40.72710258	-74.00297088	classic	56	["CREDITCARD", "KEY"]	NYC District
10	143	Clinton St & Joralemon St	40.69239502	-73.99337909	classic	50	["CREDITCARD", "KEY"]	NYC District
11	144	Nassau St & Navy St	40.69839895	-73.98068914	classic	58	["CREDITCARD", "KEY"]	NYC District
12	146	Hudson St & Reade St	40.71625008	-74.0091059	classic	55	["CREDITCARD", "KEY"]	NYC District
13	150	E 2 St & Avenue C	40.7208736	-73.98085795	classic	56	["CREDITCARD", "KEY"]	NYC District
14	151	Cleveland Pl & Spring St	40.722103787	-73.997249007	classic	32	["CREDITCARD", "KEY"]	NYC District
15	152	Warren St & W Broadway	40.71473993	-74.00910627	classic	49	["CREDITCARD", "KEY"]	NYC District
16	153	E 40 St & 5 Ave	40.752062307	-73.981632404	classic	63	["CREDITCARD", "KEY"]	NYC District

External Tables

With Snowflake, you have options for various storage patterns. You can load semi-structured and unstructured data directly into Snowflake for the best security, performance, and automatic management, or you can read data from

external object storage. Say you already have data in cloud object storage, you can start processing and querying this data from Snowflake in minutes.

For this lab, Snowflake has provided the Citibike TRIPS data in an Amazon S3 bucket. The data files are in Apache Parquet format and are partitioned into folders by year. The bucket URL is: `"s3://sfquickstarts/VHOL Snowflake for Data Lake/Data/"`

Create an External Table linked to an S3 bucket

To create an external table over the files stored in that Amazon S3 bucket, first an external stage needs to be created by specifying the URL of the bucket and optionally the file format.

```
use schema demo;
use role SYSADMIN;
create or replace stage CITIBIKE_STAGE
  url = "s3://sfquickstarts/VHOL Snowflake for Data Lake/Data/"
  file_format=(type=parquet);
```

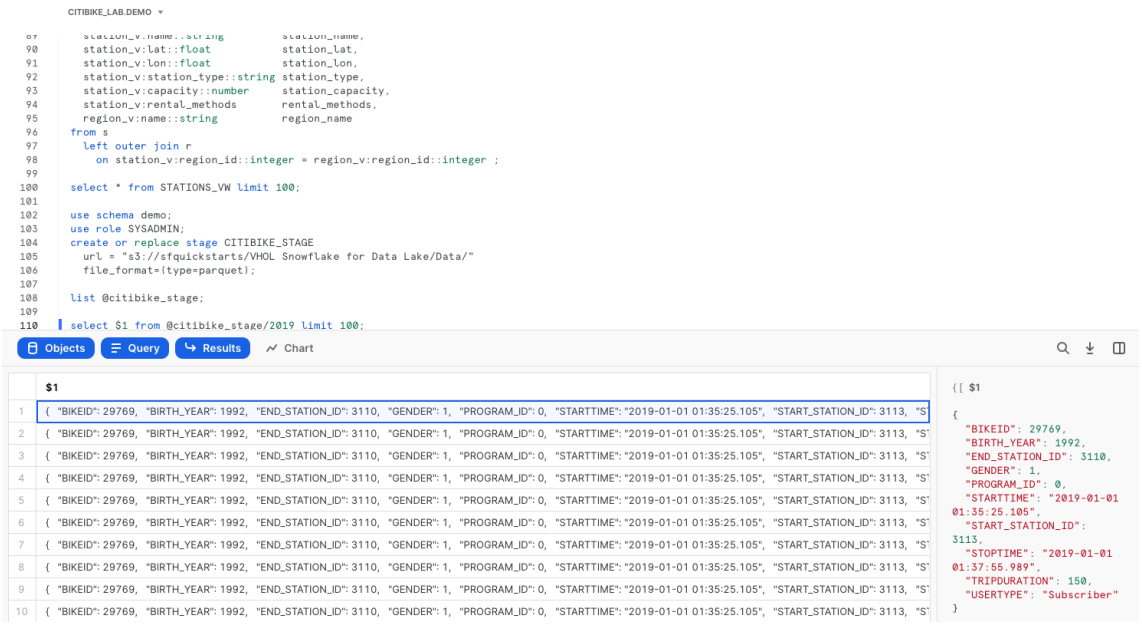
Now let's see what data is available in the Amazon S3 bucket by listing the files in the external stage.

```
list @citibike_stage;
```

With the right permissions, you can even look at the contents of individual files.

```
select $1 from @citibike_stage/2019 limit 100;
```

Click on row 1 in the results pane. On the right side of your screen, the contents of a single row from the Parquet file are displayed.



Let's create an external table using the external stage.

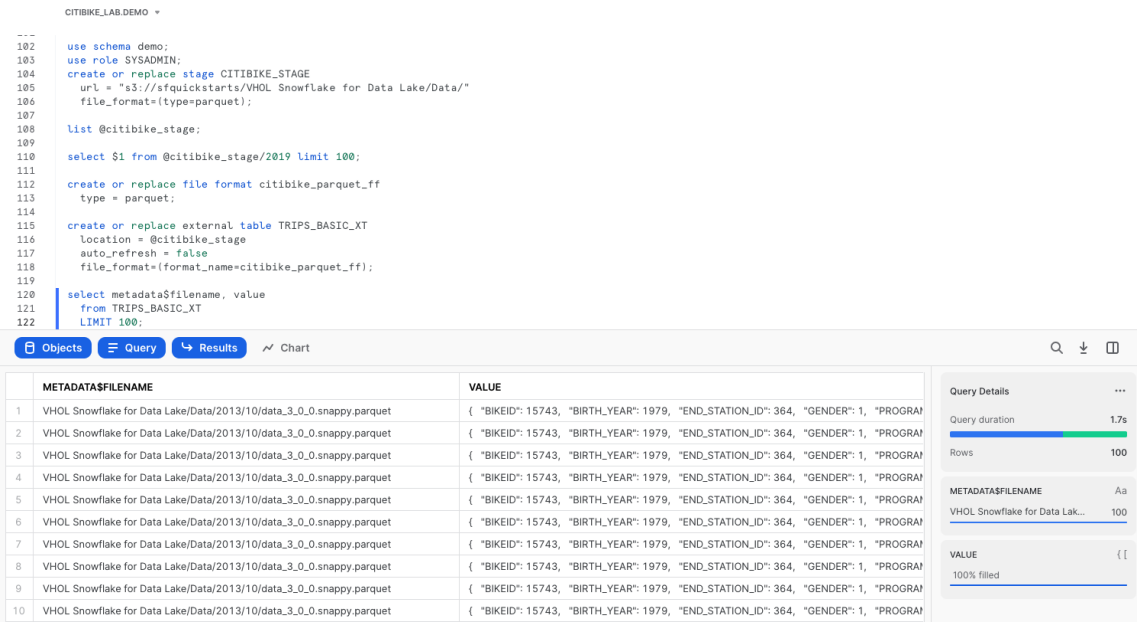
```
create or replace file format citibike_parquet_ff
  type = parquet;
```

```
create or replace external table TRIPS_BASIC_XT
  location = @citibike_stage
  auto_refresh = false
  file_format=(format_name=citibike_parquet_ff);
```

In this external table definition, there is only one column available: a VARIANT named `VALUE` that contains the file data. Using the VARIANT data type preserves the raw structure, allowing the flexibility to define schema later on.

We can also add a reference to a pseudocolumn called `metadata$filename`, to see which file the data came from.

```
select metadata$filename, value
  from TRIPS_BASIC_XT
  LIMIT 100;
```



Using Schema Detection to Define External Table

A single column named `VALUE` is not going to be very user-friendly. It would be much more useful to break out the individual fields into separate columns. We can use `INFER_SCHEMA` for Snowflake to determine the schema of the parquet files and then more easily define the schema for the table. Additionally, we can define a partitioning expression for the table that matches the underlying layout of the source data files in S3.

Now let's create a new external table on the same set of Parquet files, but this time define each column separately, and partition the files on the date portion of their folder names. We'll use Snowflake's `INFER_SCHEMA` and `GENERATE_COLUMN_DESCRIPTION` functionality to help create the table definition.

```
select *
from table(
  infer_schema(
    location=>'@citibike_stage'
    , file_format=>'citibike_parquet_ff'
```

```
    )
  );

SELECT
  $$ CREATE OR REPLACE EXTERNAL TABLE FOO ($$ || (
      SELECT
      GENERATE_COLUMN_DESCRIPTION(ARRAY_AGG(OBJECT_CONSTRUCT(*))
        , 'EXTERNAL_TABLE') ||
        $$) LOCATION = @citibike_stage FILE_FORMAT = my_parquet_format; $$
      FROM
        TABLE (
          INFER_SCHEMA(
            LOCATION => '@citibike_stage',
            FILE_FORMAT => 'citibike_parquet_ff'
          )
        )
    );
```

Below is the output of the command above.

```
-- CREATE OR REPLACE EXTERNAL TABLE FOO ("BIRTH_YEAR" NUMBER(4, 0) AS
($1:BIRTH_YEAR::NUMBER(4, 0)),
-- "PROGRAM_ID" NUMBER(4, 0) AS ($1:PROGRAM_ID::NUMBER(4, 0)),
-- "TRIPDURATION" NUMBER(9, 0) AS ($1:TRIPDURATION::NUMBER(9, 0)),
-- "START_STATION_ID" NUMBER(4, 0) AS ($1:START_STATION_ID::NUMBER(4, 0)),
-- "STOPTIME" TIMESTAMP_NTZ AS ($1:STOPTIME::TIMESTAMP_NTZ),
-- "END_STATION_ID" NUMBER(4, 0) AS ($1:END_STATION_ID::NUMBER(4, 0)),
-- "GENDER" NUMBER(2, 0) AS ($1:GENDER::NUMBER(2, 0)),
-- "USERTYPE" TEXT AS ($1:USERTYPE::TEXT),
-- "STARTTIME" TIMESTAMP_NTZ AS ($1:STARTTIME::TIMESTAMP_NTZ),
-- "BIKEID" NUMBER(9, 0) AS ($1:BIKEID::NUMBER(9, 0)))
-- LOCATION=@citibike_stage/Data
-- FILE_FORMAT='citibike_parquet_ff';
```

We can use the `CREATE EXTERNAL TABLE` statement that `INFER_SCHEMA` provided as our template, and then create a calculated startdate column that reflects how the table is partitioned in the underlying bucket.

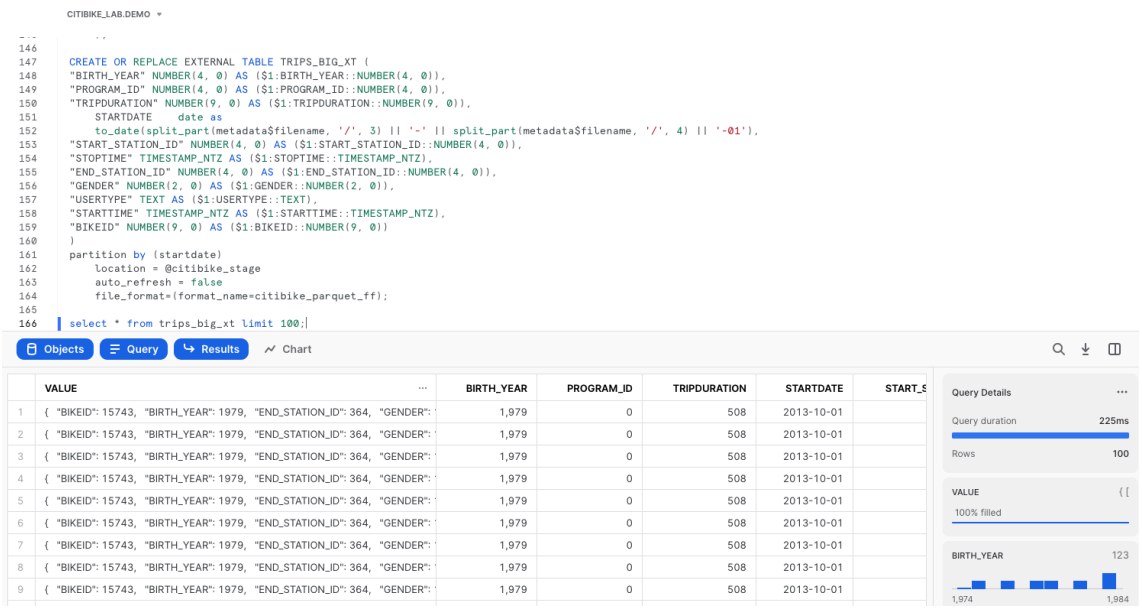
```
CREATE OR REPLACE EXTERNAL TABLE TRIPS_BIG_XT (
"BIRTH_YEAR" NUMBER(4, 0) AS ($1:BIRTH_YEAR::NUMBER(4, 0)),
"PROGRAM_ID" NUMBER(4, 0) AS ($1:PROGRAM_ID::NUMBER(4, 0)),
"TRIPDURATION" NUMBER(9, 0) AS ($1:TRIPDURATION::NUMBER(9, 0)),
    STARTDATE    date as
    to_date(split_part(metadata$filename, '/', 3) || '-' ||
split_part(metadata$filename, '/', 4) || '-01'),
"START_STATION_ID" NUMBER(4, 0) AS ($1:START_STATION_ID::NUMBER(4, 0)),
"STOPTIME" TIMESTAMP_NTZ AS ($1:STOPTIME::TIMESTAMP_NTZ),
"END_STATION_ID" NUMBER(4, 0) AS ($1:END_STATION_ID::NUMBER(4, 0)),
"GENDER" NUMBER(2, 0) AS ($1:GENDER::NUMBER(2, 0)),
"USERTYPE" TEXT AS ($1:USERTYPE::TEXT),
"STARTTIME" TIMESTAMP_NTZ AS ($1:STARTTIME::TIMESTAMP_NTZ),
"BIKEID" NUMBER(9, 0) AS ($1:BIKEID::NUMBER(9, 0))
)
partition by (startdate)
```

```
    location = @citibike_stage
    auto_refresh = false
    file_format=(format_name=citibike_parquet_ff);
```

Notice that every column definition consists of three parts: the column name, its datatype, and the transformation clause following the "as" keyword. The most basic transformation clause is just a reference to an element in the file as `value:"elementName"`, followed by an explicit datatype casting as `::datatype`.

Let's see what the data looks like in the new external table.

```
select * from trips_big_xt limit 100;
```

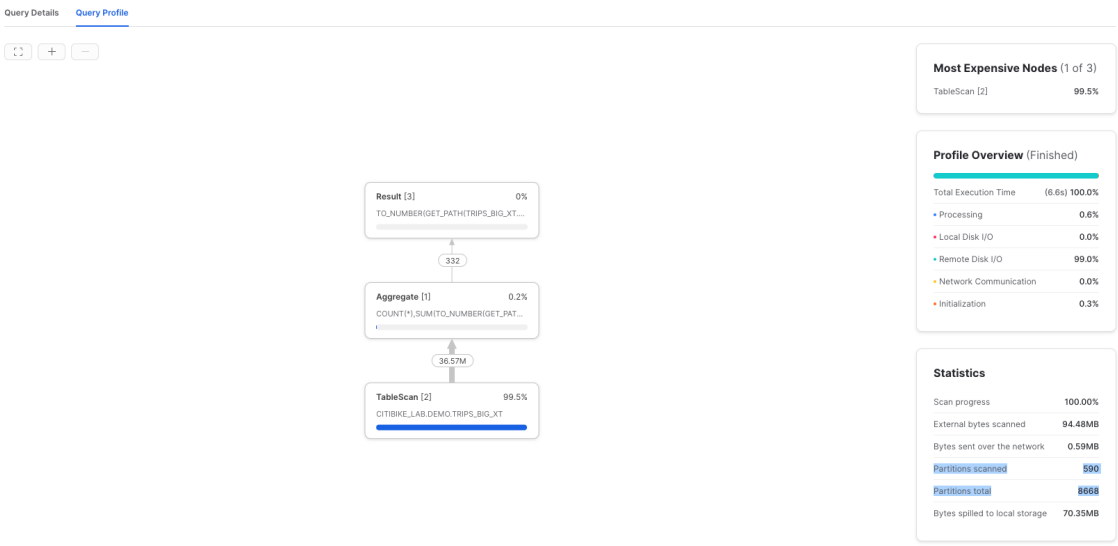


Querying External Tables

Now it's time to start querying. To effectively prune out files from the scan list, a partition column can be added to the filter.

```
select
  start_station_id,
  count(*) num_trips,
  avg(tripduration) avg_duration
from trips_big_xt
where startdate between to_date('2014-01-01') and to_date('2014-06-30')
group by 1;
```

In the Query Details of the Results pane, click on **View Query Profile** for more details in a new browser tab. Compare the query profile for the two queries we just executed. Looking at partitions scanned compared to partitions total, the query was able to effectively prune out over 90% of the file reads.



Without any ingestion, external table can be joined with other tables and views in Snowflake. Here's an example query joining an external table with a view created earlier.

```
with t as (
  select
     start_station_id,
     end_station_id,
     count(*) num_trips
  from trips_big_xt
  where startdate between to_date('2014-01-01') and to_date('2014-12-30')
    group by 1, 2)

select
   ss.station_name start_station,
   es.station_name end_station,
   num_trips
 from t inner join stations_vw ss on t.start_station_id = ss.station_id
        inner join stations_vw es on t.end_station_id = es.station_id
order by 3 desc;
```

Additional Options for External Tables

Although not covered hands-on in this Quickstart because they require access to a cloud console, there are two important optional parameters to note for External Tables.

REFRESH_ON_CREATE = { TRUE | FALSE }

`TRUE` by default, this parameter specifies whether to automatically refresh the external table metadata once, immediately after the external table is created. The metadata for an external table is the list of files that exist in the specified storage location. Setting this option to FALSE essentially creates an "empty" external table definition. To refresh the metadata requires execution of the command `ALTER EXTERNAL TABLE refresh;`

AUTO_REFRESH = { TRUE | FALSE }

Also `TRUE` by default, this parameter specifies whether Snowflake should enable triggering automatic refreshes of the external table metadata when new or updated data files are available in the named external stage specified. Setting this option to `TRUE` will keep external tables in sync with the contents of the related storage location.

Materialized Views over External Tables

External Tables offer the ability to have a SQL interface on top of object storage, without having to maintain an additional copy of the data in the Snowflake storage layer. Automatic refresh can keep the external metadata in sync with the contents of the storage location, eliminating many complex data engineering workflows. Defining the external table with an effective partitioning scheme can greatly improve query performance against external tables.

[Materialized Views] are pre-computed data sets derived from a query specification (the SELECT in the view definition) and stored for later use. Because the data is pre-computed, querying a materialized view is faster than executing the original query.

Combining these two techniques, i.e., creating a Materialized View over an External Table, provides better query performance, with the benefit of maintaining the original data source in external storage.

Create a Materialized View over an External Table

Run the commands below to create a materialized view over the external table created earlier in this Quickstart.

```
use role SYSADMIN;
use schema CITIBIKE_LAB.DEMO;

create or replace materialized view TRIPS_MV as
select
  startdate,
  start_station_id,
  end_station_id,
  count(*) num_trips
from trips_big_xt
group by 1, 2, 3;
```

Confirm that the number of rows in the materialized view matches the number of rows in the external table.

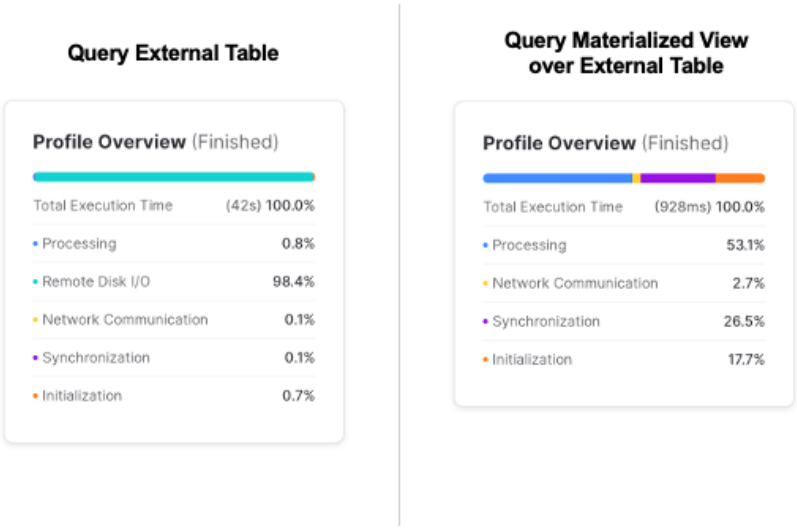
```
select
  count(*)         num_rows,
  sum(num_trips)     num_trips
from trips_mv;
```

Re-run the join query earlier, but replacing the external table with the new materialized view.

```
with t as (
  select
    start_station_id,
    end_station_id,
    sum(num_trips) num_trips
  from trips_mv
  where startdate between to_date('2014-01-01') and to_date('2014-12-30')
  group by 1, 2)

select
  ss.station_name start_station,
```

```
  es.station_name end_station,
  num_trips
from t
  inner join stations_vw ss
     on t.start_station_id = ss.station_id
  inner join stations_vw es
     on t.end_station_id = es.station_id
order by 3 desc;
```

In the Results panel, view the Query Profile. Notice how much faster this was than the same query against the external table.



Unstructured Data

So far you've seen how easy it is to query semi-structured data in Snowflake. Snowflake also supports unstructured data, which allows you to store and access files, natively process files using Snowpark, process files by calling out to external services using External Functions, and use Snowflake's role-based access controls to govern unstructured data.

Create an External Stage and Directory Table linked to an S3 bucket

Snowflake supports two types of stages for storing data files used for loading/unloading:

- **Internal stages** store the files internally within Snowflake, automatically encrypted and compressed.
- **External stages** access metadata of files in external storage that is referenced by the stage. An external stage specifies location and credential information, if required, for the object storage.

In this Quickstart, we are working with PDFs that have already been staged in a public, external S3 bucket. Before you can use this data, you first need to create a external stage that specifies the location of the storage bucket.

```
create or replace stage documents
 url = "s3://sfquickstarts/VHOL Snowflake for Data Lake/PDF/"
 directory = (enable = true auto_refresh = false);
```

List of all the files in the external stage.

```
ls @documents;
```

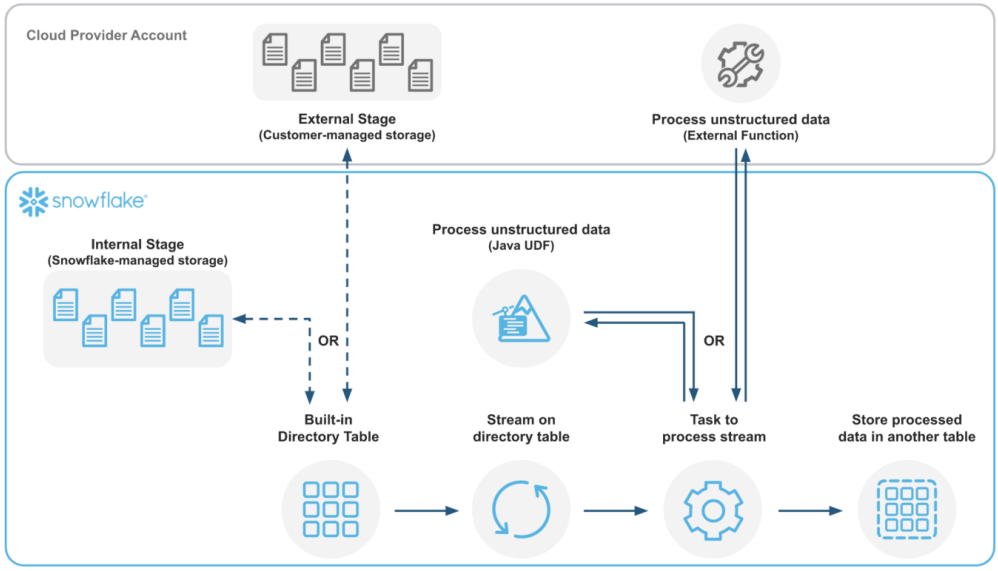
Directory tables are built-in tables in Snowflake that provide an up-to-date, tabular file catalog for external and internal stages. Directory Tables make it easy to search for files using a catalog, which can be challenging in cloud object storage.

When you created the external stage documents, you specified that a directory table be created with `directory = (enable = true)`. Refresh the stage metadata in the directory table, then search for files where the URL contains the substring 'algorithm%pdf'.

```
alter stage documents refresh;

select *
from directory(@documents)
where file_url ilike '%algorithm%pdf';
```

Processing Unstructured Data with Snowpark

Unstructured data can be processed natively in Snowflake using Snowpark, currently in public preview. In this example, you will see how a Java User-Defined Function (UDF) can extract text from PDF files into a structured table for analytical use. Here's an illustration for how the files are processed:



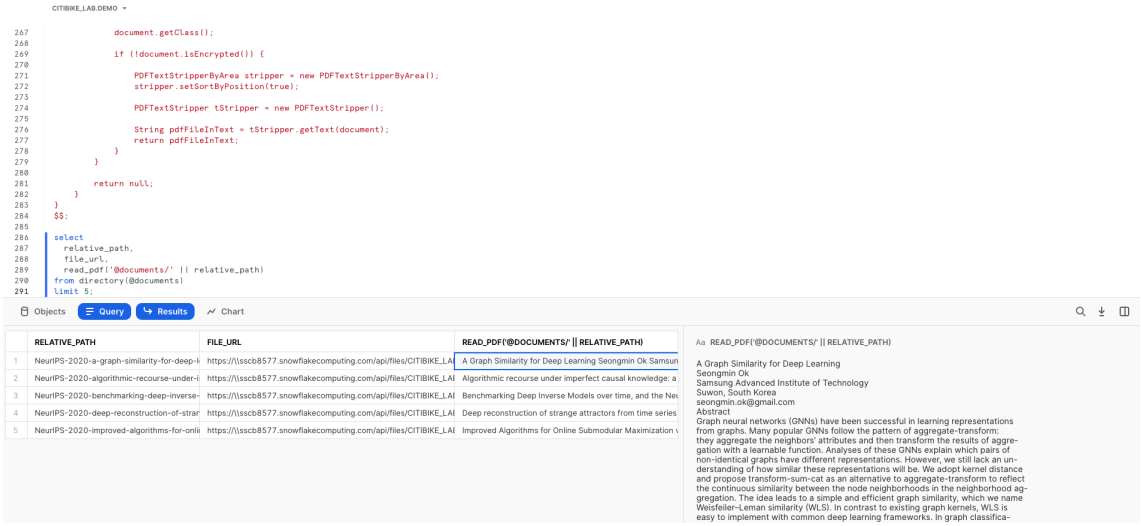
A Java UDF can be created either in-line or pre-compiled. In this example, you will see how to create an in-line UDF. Snowflake will compile the source code and store the compiled code in a JAR file, and you have the option to specify a location for the JAR file. A pre-compiled JAR file containing the Java code for parsing the PDFs has already been uploaded to an S3 bucket. The JAR file can be copied into a Snowflake stage, and from there a function can be created.

```
create or replace stage jars_stage
 url = "s3://sfquickstarts/VHOL Snowflake for Data Lake/JARs/"
```

```
 directory = (enable = true auto_refresh = false);

create or replace function read_pdf(file string)
returns string
language java
imports = ('@jars_stage/pdfbox-app-2.0.24.jar')
HANDLER = 'PdfParser.ReadFile'
as
$$
import org.apache.pdfbox.pdmodel.PDDocument;
import org.apache.pdfbox.text.PDFTextStripper;
import org.apache.pdfbox.text.PDFTextStripperByArea;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class PdfParser {

    public static String ReadFile(InputStream stream) throws IOException {
        try (PDDocument document = PDDocument.load(stream)) {

            document.getClass();

            if (!document.isEncrypted()) {

                PDFTextStripperByArea stripper = new PDFTextStripperByArea();
                stripper.setSortByPosition(true);

                PDFTextStripper tStripper = new PDFTextStripper();

                String pdfFileInText = tStripper.getText(document);
                return pdfFileInText;
            }
        }

        return null;
    }
}
$$;
```

Now this function can be called in queries to use the parsed results of PDFs.

```
select
  relative_path,
  file_url,
  read_pdf('@documents/' || relative_path)
from directory(@documents)
limit 5;
```



Conclusion

Congratulations, you have completed this Quickstart for a quick overview of capabilities Snowflake for data lakes.

What We've Covered

- How to query partitioned semi-structured data stored in files in external cloud object storage with **External Tables**
- How to use **Schema Detection** to automatically determine and return the schema of staged files
- How to improve performance of queries over external object stores with **Materialized Views** over External Tables
- Process and analyze **unstructured data** with **Snowpark**

Building a data application with Marketplace, Snowpark and Streamlit

Overview

In this hands-on lab, you will build a data application that leverages Economical Data Atlas published by Knoema on the Snowflake Marketplace.

You will process data with Snowpark, develop a simple ML model and create a Python User Defined Function (UDF) in Snowflake, then visualize the data with Streamlit.

Key Features & Technology

- Snowflake Marketplace
- Snowpark for Python
- Python libraries
- Python User Defined Functions (UDF)
- Streamlit

Prerequisites

- Accountadmin role access in Snowflake or a Snowflake trial account: https://signup.snowflake.com/
- Basic knowledge of SQL, database concepts, and objects
- Familiarity with Python. All code for the lab is provided.
- Ability to install and run software on your computer
- VSCode installed

What You'll Learn:

- How to consume datasets in the Snowflake Data Marketplace.
- How to perform queries on data in Python using Dataframes
- How to leverage existing Python libraries
- How to create a Snowpark Python User Defined Function in Snowflake
- How to create a data application with Streamlit to visualize data

What You'll Build

- A Python notebook that connects to Snowflake with Snowpark for Python and prepares features for a Linear Regression model training.
- A Snowflake User Defined Function (UDF) based on a Python trained model
- A Streamlit dashboard data application

Prepare your lab environment

1. Install conda to manage a separate environment by running pip install conda. NOTE: The other option is to use Miniconda
2. Open the terminal or command prompt
3. Create environment by running `conda create --name snowpark -c https://repo.anaconda.com/pkgs/snowflake python=3.8`
4. Activate conda environment by running `conda activate snowpark`
5. Install pandas by running `conda install pandas`
6. Install Streamlit by running `pip install streamlit` or `pip3 install streamlit`
7. Install scikit-learn by running `pip install -U scikit-learn`
8. Install Snowpark for Python by running `conda install snowflake-snowpark-python`

9. Create folder, e.g. "Summit HOL PCE" and download/save the Lab files in that folder.
    1. Link to required files: https://drive.google.com/drive/folders/1CN6Ljj59XWv2B3Epqxk4DtfDmCH1co_Q?usp=sharing

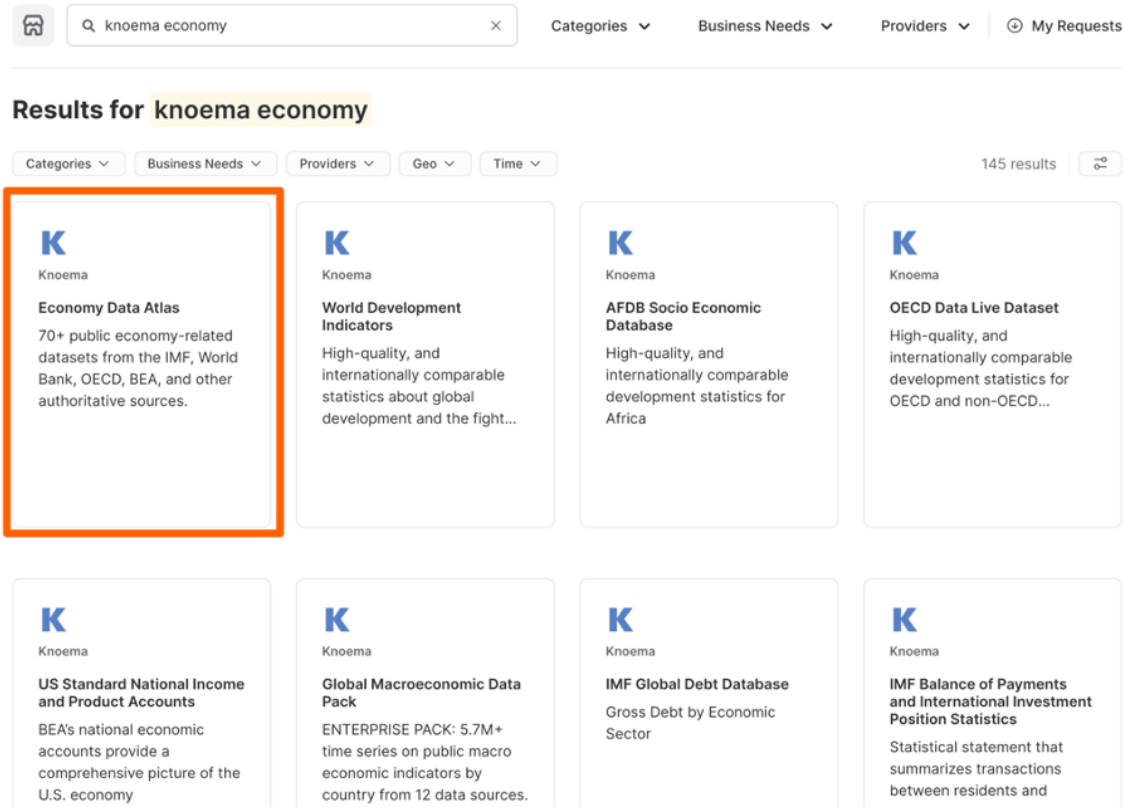
Prepare the Snowflake environment

Working with the Snowflake Marketplace

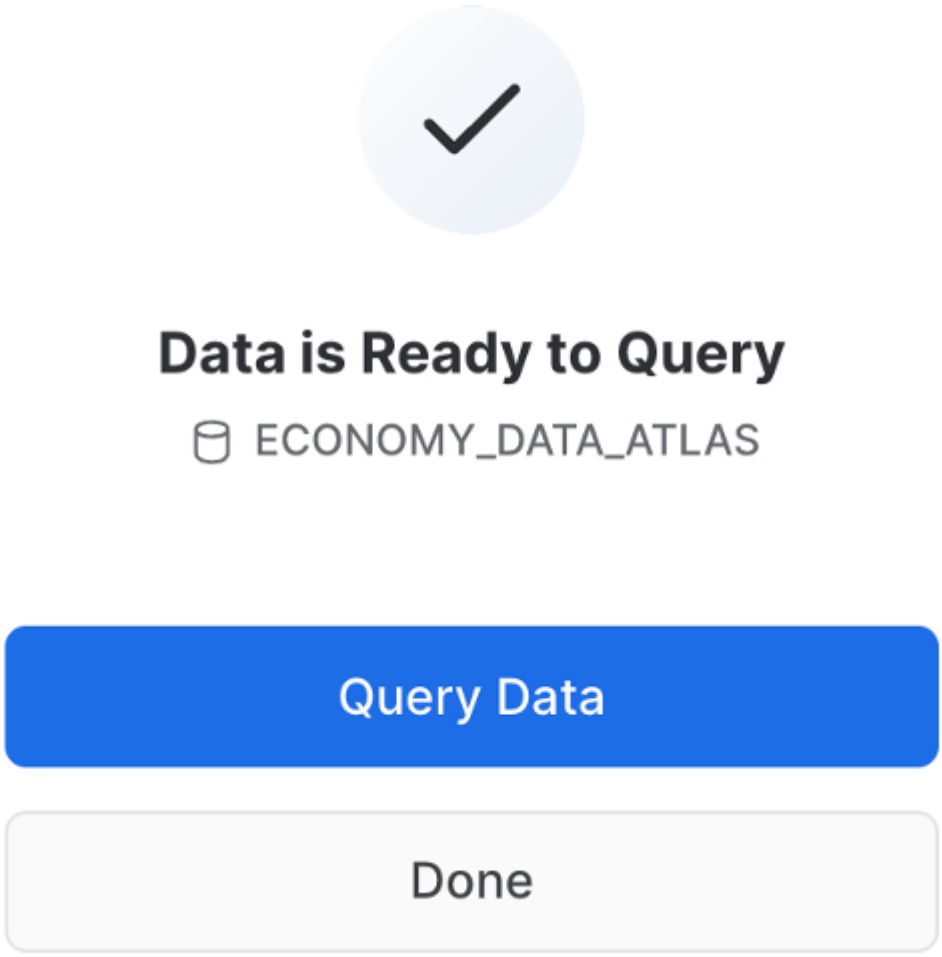
Snowflake's Marketplace provides visibility to a wide variety of datasets from third party data stewards which broaden access to data points used to transform business processes. The Marketplace also removes the need to integrate and model data by providing secure access to data sets fully maintained by the data provider.

Before we begin to review working with Marketplace data sets, verify you have installed a trial version of Snowflake. If not, click Install Snowflake Trial. Now that you have a working trial account, and you are logged into the Snowflake Console, follow the following steps.

- At the top left corner, make sure you are logged in as ACCOUNTADMIN, switch role if not
- Click on Marketplace
- At the Search bar, type: Knoema Economy then click on the Tile Box labeled: Economy Data Atlas.



- At the top right corner, Select Get Data
- Select the appropriate roles to access the Database being created and accept the Snowflake consumer terms and Knoema's terms of use.
- Create Database



- At this point you can select Query Data, this will open a worksheet with example queries.

ECONOMY_DATA_ATLAS

```
// What is the quarterly real GDP growth rate of the US over time?
SELECT * FROM "ECONOMY"."BEANIPA" WHERE "Table Name" = 'Percent Change From Preceding Period In Real Gross Domestic Product' AND "Indicator Name" = 'Gross domestic product (GDP)' AND "Frequency" = 'Q' ORDER BY "Date"

// What is the US unemployment rate over time?
SELECT * FROM "ECONOMY"."BLSUSLFSCPS2019" WHERE "Series Name"= 'Unemployment Rate - (Seas)' AND "Frequency" = 'M' ORDER BY "Date"

// What is the US inflation over time?
SELECT * FROM "ECONOMY"."BEANIPA" WHERE "Table Name" = 'Price Indexes For Personal Consumption Expenditures By Major Type Of Product' AND "Indicator Name" = 'Personal consumption expenditures (PCE)' AND "Frequency" = 'A' ORDER BY "Date"

// What is the EUR/USD exchange rate?
SELECT * FROM "ECONOMY"."EXRATESCC2018" WHERE "Currency" = 'EUR/USD' AND "Indicator Name" = 'Close' AND "Frequency" = 'D' AND "Date" > '2000-01-01' ORDER BY "Date"

// What is the US manufacturing PMI over time?
SELECT * FROM "ECONOMY"."ISMMNF2017" WHERE "Indicator Name" = 'PMI' ORDER BY "Date"
```

- We are interested in the US Inflation data, so we will use this query to explore the data for the application: `What is the US inflation over time?`

```
SELECT * FROM "ECONOMY"."BEANIPA" WHERE "Table Name" = 'Price Indexes For Personal Consumption Expenditures By Major Type Of Product' AND "Indicator Name" = 'Personal consumption expenditures (PCE)' AND "Frequency" = 'A' ORDER BY "Date"
```

Create a new database

Now that we have created a database with the Economy Data Atlas, we need to create a database for our application that will store the User Defined Function.

Select "Worksheets" from the Home menu of Snowflake. Create a new worksheet by selecting the

button.

In the worksheet copy the following script:

```
-- First create database using the Knoema Economical Data Atlas
-- Go to Marketplace to get database

-- Setup database, need to be logged in as accountadmin role */
--Set role and warehouse (compute)
USE ROLE accountadmin;
USE WAREHOUSE compute_wh;

--Create database and stage for the Snowpark Python UDF
CREATE DATABASE IF NOT EXISTS summit_hol;
CREATE STAGE IF NOT EXISTS udf_stage;

--Test the data
```

```
-- What's the size?
SELECT COUNT(*) FROM ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA;

-- What is the US inflation over time?
SELECT * FROM ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA
   WHERE "Table Name" = 'Price Indexes For Personal Consumption Expenditures By Major
Type Of Product'
     AND "Indicator Name" = 'Personal consumption expenditures (PCE)'
     AND "Frequency" = 'A'
ORDER BY "Date"
;

-- Now create UDF in VS Code / Notebook
-- Once we created the UDF with the Python Notebook we can test the UDF
SELECT predict_pce_udf(2021);
```

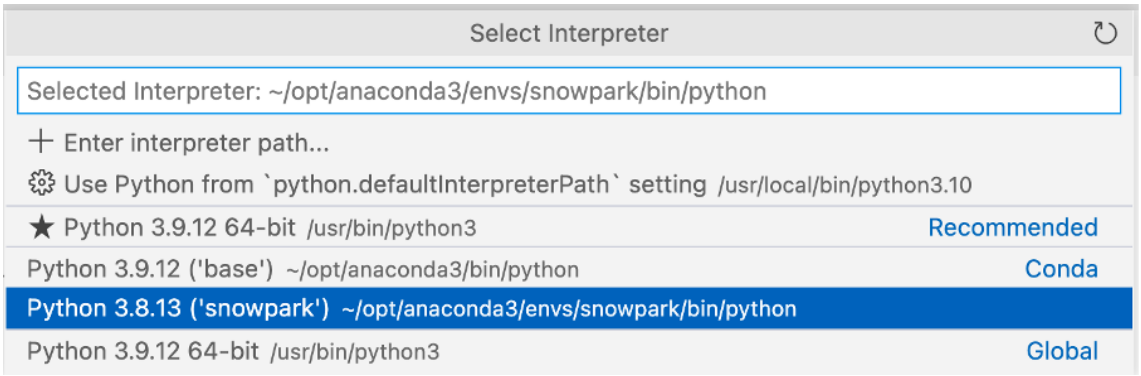
Exploring the Data with a (Jupyter) Notebook

Now that we have a database that we can use for the Application, we want to explore the data and create a ML model in a User Defined Function (UDF) that can be used by our application.

Open VS code and open the folder with the Python scripts that we created earlier.

You can open the Python notebook (my_snowpark_pce.ipynb) and Streamlit application script (my_snowpark_streamlit_app_pce.py). We will be walking through the sections of the code.

VS Code might ask for the Python environment: \



Make sure you select the 'snowpark' Conda environment that was created earlier.

You can select the Interpreter by clicking in the lower right corner: \

Initialize Notebook, import libraries and create Snowflake connection

Let's start by creating a Python script and adding the import statements to include the required libraries.

```
from snowflake.snowpark.session import Session
from snowflake.snowpark.types import IntegerType, FloatType
from snowflake.snowpark.functions import avg, sum, col, udf, call_udf, call_builtin,
year
import streamlit as st
import pandas as pd
from datetime import date

# scikit-learn (install: pip install -U scikit-learn)
from sklearn.linear_model import LinearRegression
```

Connect to Snowflake

In this step, you'll create a [Session object] to connect to your Snowflake. We will be using the database that we created in the Snowflake setup section.

```
# Session
connection_parameters = {
   "account": "<account_identifier>",
   "user": "<username>",
   "password": "<password>",
   "warehouse": "compute_wh",
   "role": "accountadmin",
   "database": "summit_hol",
   "schema": "public"
}
session = Session.builder.configs(connection_parameters).create()
# test if we have a connection
session.sql("select current_warehouse() wh, current_database() db, current_schema()
schema, current_version() v").show()
```

In the above code snippet, replace variables enclosed in "<>" with your values.

Query the data using a SQL statement and with the Snowpark Dataframe

In this step we will query the data using the traditional method of executing a SQL statement in the Session object, similar to querying data with the Snowflake for Python connector.

```
# SQL query to explore the data
session.sql("SELECT * FROM ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA WHERE \"Table Name\" =
'Price Indexes For Personal Consumption Expenditures By Major Type Of Product' AND
\"Indicator Name\" = 'Personal consumption expenditures (PCE)' AND \"Frequency\" = 'A'
ORDER BY \"Date\"").show()
```

Now we will query the data using a Snowpark DataFrame. As Snowpark uses lazy evaluation, the query and filter conditions are created and the *show()* method will push this to the Snowflake server where the query will be executed. This reduces the amount of data exchanged between Snowflake and the client/application.

```
# Now use Snowpark dataframe
snow_df_pce = (session.table("ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA")
                           .filter(col('Table Name') == 'Price Indexes For Personal
```

```
Consumption Expenditures By Major Type Of Product')
                            .filter(col('Indicator Name') == 'Personal consumption
expenditures (PCE)')
                            .filter(col('"Frequency"') == 'A')
                            .filter(col('"Date"') >= '1972-01-01'))
snow_df_pce.show()
```

Creating features for ML training

As part of the application we would like to have some predictions of the Personal consumption expenditures price index. So we will create a Pandas dataframe that can be used for training the model with the scikit-learn Linear Regression model. The Snowpark API for Python exposes a method to convert Snowpark DataFrames to Pandas. Again with the Snowpark lazy evaluation we can construct the dataframe query and the *to_pandas()* function will push the query to Snowflake and return the results as a Pandas dataframe.

```
# Let Snowflake perform filtering using the Snowpark pushdown and display results in a
Pandas dataframe
snow_df_pce = (session.table("ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA")
                        .filter(col('"Table Name"') == 'Price Indexes For Personal
Consumption Expenditures By Major Type Of Product')
                        .filter(col('"Indicator Name"') == 'Personal consumption
expenditures (PCE)')
                        .filter(col('"Frequency"') == 'A')
                        .filter(col('"Date"') >= '1972-01-01'))
pd_df_pce_year = snow_df_pce.select(year(col('"Date"')).alias('"Year"'),
col('"Value"').alias('PCE') ).to_pandas()
pd_df_pce_year
```

Train the Linear Regression model

Now that we have created the features, we can train the model. In this step we will transform the Pandas dataframe with the features to arrays using the NumPy library. Once trained we can display a prediction.

```
# train model with PCE index
x = pd_df_pce_year["Year"].to_numpy().reshape(-1,1)
y = pd_df_pce_year["PCE"].to_numpy()

model = LinearRegression().fit(x, y)

# test model for 2021
predictYear = 2021
pce_pred = model.predict([[predictYear]])
# print the last 5 years
print (pd_df_pce_year.tail() )
# run the prediction for 2021
print ('Prediction for '+str(predictYear)+': '+ str(round(pce_pred[0],2)))
```

Creating a User Defined Function in Snowflake with the trained model

In this step we will create a Python function that will use the trained model to predict a PCE index based on the function input. We will then use the Snowpark API to create an UDF. The Snowpark library uploads the code (and trained model) for your function to an internal stage. When you call the UDF, the Snowpark library executes your

function on the server, where the data is. As a result, the data doesn't need to be transferred to the client in order for the function to process the data.

```
def predict_pce(predictYear: int) -> float:
   return model.predict([[predictYear]])[0].round(2).astype(float)

_ = session.udf.register(predict_pce,
                         return_type=FloatType(),
                         input_type=IntegerType(),
                         packages= ["pandas","scikit-learn"],
                         is_permanent=True,
                         name="predict_pce_udf",
                         replace=True,
                         stage_location="@udf_stage")
```

Now we can test the UDF using a SQL command in Python.

```
session.sql("select predict_pce_udf(2021)").show()
```

Creating the Streamlit application

Import the required libraries

Now that we have a trained ML model and created a UDF to do predictions we can create the Streamlit application.

Similar to the notebook we create a Python script and add import statements to include the required libraries.

```
# Import required libraries
# Snowpark
from snowflake.snowpark.session import Session
from snowflake.snowpark.types import IntegerType
from snowflake.snowpark.functions import avg, sum, col, call_udf, lit, call_builtin,
year
# Pandas
import pandas as pd
#Streamlit
import streamlit as st
```

Set the application page context

We need to set the context for the application page.

```
#Set page context
st.set_page_config(
    page_title="Economical Data Atlas",
    page_icon="",
    layout="wide",
    initial_sidebar_state="expanded",
    menu_items={
        'Get Help': 'https://developers.snowflake.com',
        'About': "This is an *extremely* cool app powered by Snowpark for Python,
Streamlit, and Snowflake Marketplace"
```

```
    }
)
```

Connect to Snowflake

In this step, you'll create a [Session object], for example.

We will be using the database that we created in the Snowflake setup section.

```
# Create Session object
def create_session_object():
   connection_parameters = {
   "account": "<account_identifier>",
   "user": "<username>",
   "password": "<password>",
   "warehouse": "compute_wh",
   "role": "accountadmin",
   "database": "SUMMIT_HOL",
   "schema": "PUBLIC"
  }
   session = Session.builder.configs(connection_parameters).create()
   print(session.sql('select current_warehouse(), current_database(),
current_schema()').collect())
   return session
```

In the above code snippet, replace variables enclosed in "<>" with your values.

Load Data in Snowpark DataFrames

In this step, we'll create a dataframe with US Inflation (Personal consumption expenditures - PCE) data per year. We will be using the BEANIPA table (BEA NIPA: Bureau of Economic Analysis - National Income and Product Accounts data). This table contains around 1.6 million rows, using the Snowpark lazy evaluation this data is processed in Snowflake.

We'll create a dataframe with actual and predicted PCE values based on the UDF with a trained ML model we created in the Notebook section.

And we will combine the actual and predicted dataframes in a new dataframe so we can display the data in a single chart.

Note that when working with Streamlit we need Pandas DataFrames and Snowpark API for Python exposes a method to convert Snowpark DataFrames to Pandas.

We also want to show some key metrics, so we will extract metrics from the dataframes.

As a bonus we want to show the PCE data per quarter for a selected year and the breakdown per major type of product. We will create 2 data frames for this data.

```
  #US Inflation, Personal consumption expenditures (PCE) per year
   #Prepare data frame, set query parameters
   snow_df_pce = (session.table("ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA")
                          .filter(col('Table Name') == 'Price Indexes For Personal
Consumption Expenditures By Major Type Of Product')
                          .filter(col('Indicator Name') == 'Personal consumption
expenditures (PCE)')
```

```
                            .filter(col('"Frequency"') == 'A')
                            .filter(col('"Date"') >= '1972-01-01'))
   #Select columns, substract 100 from value column to reference baseline
   snow_df_pce_year = snow_df_pce.select(year(col('"Date"')).alias('"Year"'),
(col('"Value"')-100).alias('PCE')).sort('"Year"', ascending=False)
   #convert to pandas dataframe
   pd_df_pce_year = snow_df_pce_year.to_pandas()
   #round the PCE series
   pd_df_pce_year["PCE"] = pd_df_pce_year["PCE"].round(2)
    #create metrics
   latest_pce_year = pd_df_pce_year.loc[0]["Year"].astype('int')
   latest_pce_value = pd_df_pce_year.loc[0]["PCE"]
   delta_pce_value = latest_pce_value - pd_df_pce_year.loc[1]["PCE"]

   #Use Snowflake UDF for Model Inference
   snow_df_predict_years = session.create_dataframe([[int(latest_pce_year+1)],
[int(latest_pce_year+2)],[int(latest_pce_year+3)]], schema=["Year"])
   pd_df_pce_predictions = snow_df_predict_years.select(col("year"),
call_udf("predict_pce_udf", col("year")).as_("pce")).sort(col("year")).to_pandas()
   pd_df_pce_predictions.rename(columns={"YEAR": "Year"}, inplace=True)
   #round the PCE prediction series
   pd_df_pce_predictions["PCE"] =
pd_df_pce_predictions["PCE"].round(2).astype(float)-100


   #Combine actual and predictions dataframes
   pd_df_pce_all = (
       pd_df_pce_year.set_index('Year').sort_index().rename(columns={"PCE": "Actual"})
       .append(pd_df_pce_predictions.set_index('Year').sort_index().rename(columns=
{"PCE": "Prediction"}))
   )


   #Data per quarter
   snow_df_pce_q = (session.table("ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA")
                            .filter(col('Table Name') == 'Price Indexes For Personal
Consumption Expenditures By Major Type Of Product')
                            .filter(col('Indicator Name') == 'Personal consumption
expenditures (PCE)')
                            .filter(col('"Frequency"') == 'Q')
                            .select(year(col('"Date"')).alias('Year'),
                                    call_builtin("date_part", 'quarter',
col('"Date"')).alias('"Quarter"') ,
                                    (col('"Value"')-100).alias('PCE'))
                            .sort('Year', ascending=False))


   # by Major Type Of Product
   snow_df_pce_all = (session.table("ECONOMY_DATA_ATLAS.ECONOMY.BEANIPA")
                        .filter(col('"Table Name"') == 'Price Indexes For Personal
Consumption Expenditures By Major Type Of Product')
                        .filter(col('"Indicator Name"') != 'Personal consumption
```

```
expenditures (PCE)')
                        .filter(col('"Frequency"') == 'A')
                        .filter(col('"Date"') >= '1972-01-01')
                        .select('"Indicator Name"', year(col('"Date"')).alias('Year'),
(col('"Value"')-100).alias('PCE') ))
```

Add Web Page Components

In this step, you'll add...

1. A header and sub-header and also use containers and columns to organize the application content using Streamlit's *columns()* and *container()*
2. Display of metrics with delta using Streamlit's metric function.
3. Interactive bar chart using Streamlit's selectbox_()_ and *bar_chart()*

```
   # Add header and a subheader
   st.title("Knoema: Economical Data Atlas")
   st.header("Powered by Snowpark for Python and Snowflake Marketplace | Made with
Streamlit")
   st.subheader("Personal consumption expenditures (PCE) over the last 25 years,
baseline is 2012")
   # Add an explanation on the PCE Price Index that can be expanded
   with st.expander("What is the Personal Consumption Expenditures Price Index?"):
       st.write("""
        The prices you pay for goods and services change all the time - moving at
different rates and even in different directions. Some prices may drop while others
are going up. A price index is a way of looking beyond individual price tags to
measure overall inflation (or deflation) for a group of goods and services over time.


        The Personal Consumption Expenditures Price Index is a measure of the prices
that people living in the United States, or those buying on their behalf, pay for
goods and services.The PCE price index is known for capturing inflation (or deflation)
across a wide range of consumer expenses and reflecting changes in consumer behavior.
       """)
   # Use columns to display metrics for global value and predictions
   col11, col12, col13 = st.columns(3)
   with st.container():
       with col11:
           st.metric("PCE in " + str(latest_pce_year), round(latest_pce_value),
round(delta_pce_value), delta_color=("inverse"))
       with col12:
           st.metric("Predicted PCE for " + str(int(pd_df_pce_predictions.loc[0]
["Year"])), round(pd_df_pce_predictions.loc[0]["PCE"]),
               round((pd_df_pce_predictions.loc[0]["PCE"] - latest_pce_value)),
delta_color=("inverse"))
       with col13:
           st.metric("Predicted PCE for " + str(int(pd_df_pce_predictions.loc[1]
["Year"])), round(pd_df_pce_predictions.loc[1]["PCE"]),
               round((pd_df_pce_predictions.loc[1]["PCE"] - latest_pce_value)),
delta_color=("inverse"))

   # Barchart with actual and predicted PCE
```

```
   st.bar_chart(data=pd_df_pce_all.tail(25), width=0, height=0,
use_container_width=True)

   # Display interactive chart to visualize PCE per quarter and per major type of
product.
   with st.container():


       year_selection = st.selectbox('Select year',
pd_df_pce_year['Year'].head(25),index=0 )
       pd_df_pce_q = snow_df_pce_q.filter(col('Year') ==
year_selection).sort(col('"Quarter"')).to_pandas().set_index('Quarter')
       with st.expander("Price Indexes For Personal Consumption Expenditures per
Quarter"):
            st.bar_chart(data=pd_df_pce_q['PCE'], width=0, height=500,
use_container_width=True)
       pd_df_pce_all = snow_df_pce_all.filter(col('Year') ==
year_selection).sort(col('"Indicator Name"')).to_pandas().set_index('Indicator Name')
       st.write("Price Indexes For Personal Consumption Expenditures By Major Type Of
Product")
       st.bar_chart(data=pd_df_pce_all['PCE'], width=0, height=500,
use_container_width=True)
```

In the above code snippet, a bar chart is constructed using Streamlit's *bar_chart()* which takes a dataframe as one of the parameters. In our case, that is a subset (25 years) of the Personal consumption expenditures (PCE) price index dataframe filtered by date via Snowpark DataFrame's _filter() _combined with predicted PCE values leveraging the Snowflake User Defined Functions that contains a trained ML model. Key metrics like the last PCE value and the next 2 predictions including a delta with the last year are displayed using the Streamlit _metric() _function.

More details can be shown by using a year selection (Streamlit_ selectbox() _ function) and a chart with the quarterly values for the selected year and a detailed chart of the PCE values of the major product types for the selected year. Every time a year is selected, the query will be run on Snowflake and the results are displayed by Snowpark and Streamlit.

Run Web Application

The fun part! Assuming your Python script is free of syntax and connection errors, you're ready to run the application.

You can run this by executing:
`streamlit run my_snowpark_streamlit_app_pce.py` at the command line, or in the terminal section of VS code. (Replace *my_snowpark_streamlit_app_pce.py* with the name of your Python script.)

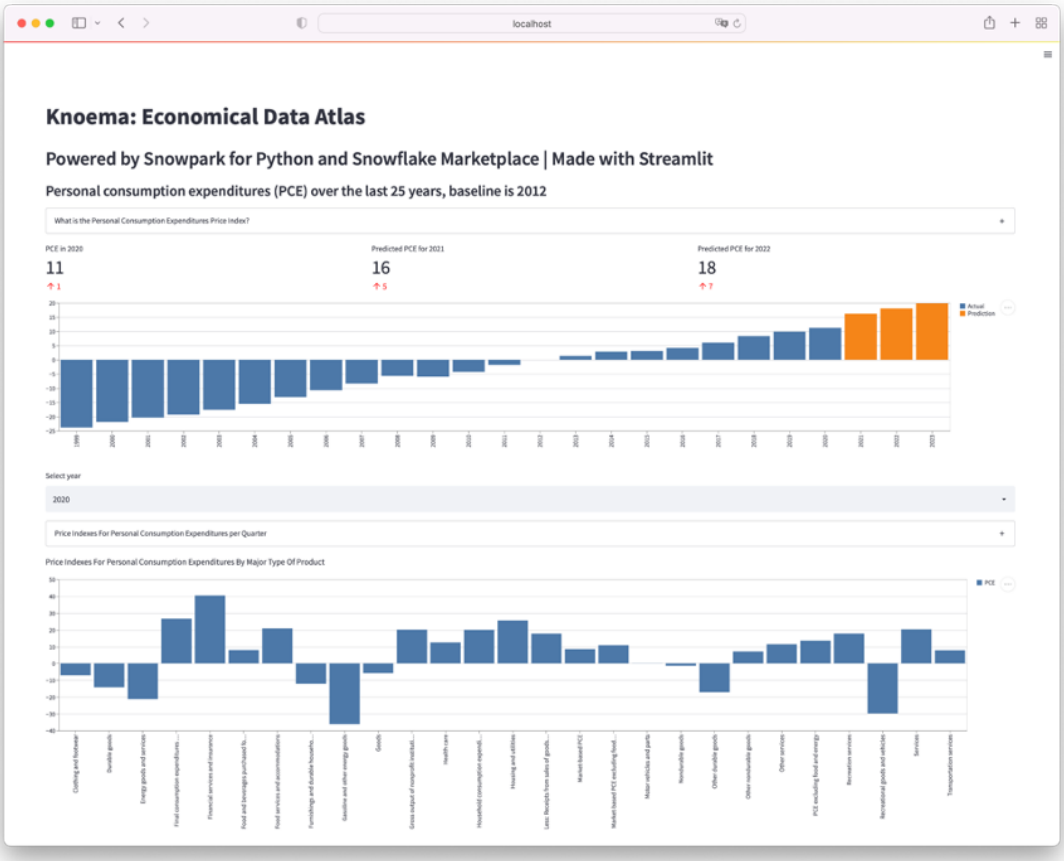
Make sure you have activated the 'snowpark' Conda environment by using this terminal command: `conda activate snowpark`

You will see a terminal prompt indicating that you have selected the right Conda environment:

```
(base) user SummitHOL % conda activate snowpark
(snowpark) user SummitHOL %
```

In the application:

1. You can click on expansion sections indicated with a "+"
2. You can select a year to show detailed information
3. The quarterly PCE values are collapsed by default, you can expand them by clicking the "+"



Exploratory Data Analysis with Snowflake and Deepnote

Overview

Deepnote brings Python and SQL into a collaborative data science notebook, along with a suite of low-code tools and tight integration with Snowflake. Altogether, Deepnote and Snowflake reduce the "time to insight" for teams as they explore their data.

In this quick-start guide, we will build an accelerated EDA workflow with Deepnote and Snowflake. Specifically, we look at various methods for effectively cleaning and visualizing weather readings; however, the techniques used can be applied more generally to any dataset used within Deepnote.

Prerequisites

- Familiarity with basic Python and SQL
- Familiarity with data science notebooks

What You'll Learn

This guide will walk you through a generalizable workflow for exploratory data analysis (EDA) using Deepnote and Snowflake. You will learn how to combine Python, SQL, and low-code solutions to complete common EDA tasks—including data wrangling and interactive data visualization.

What You'll Need

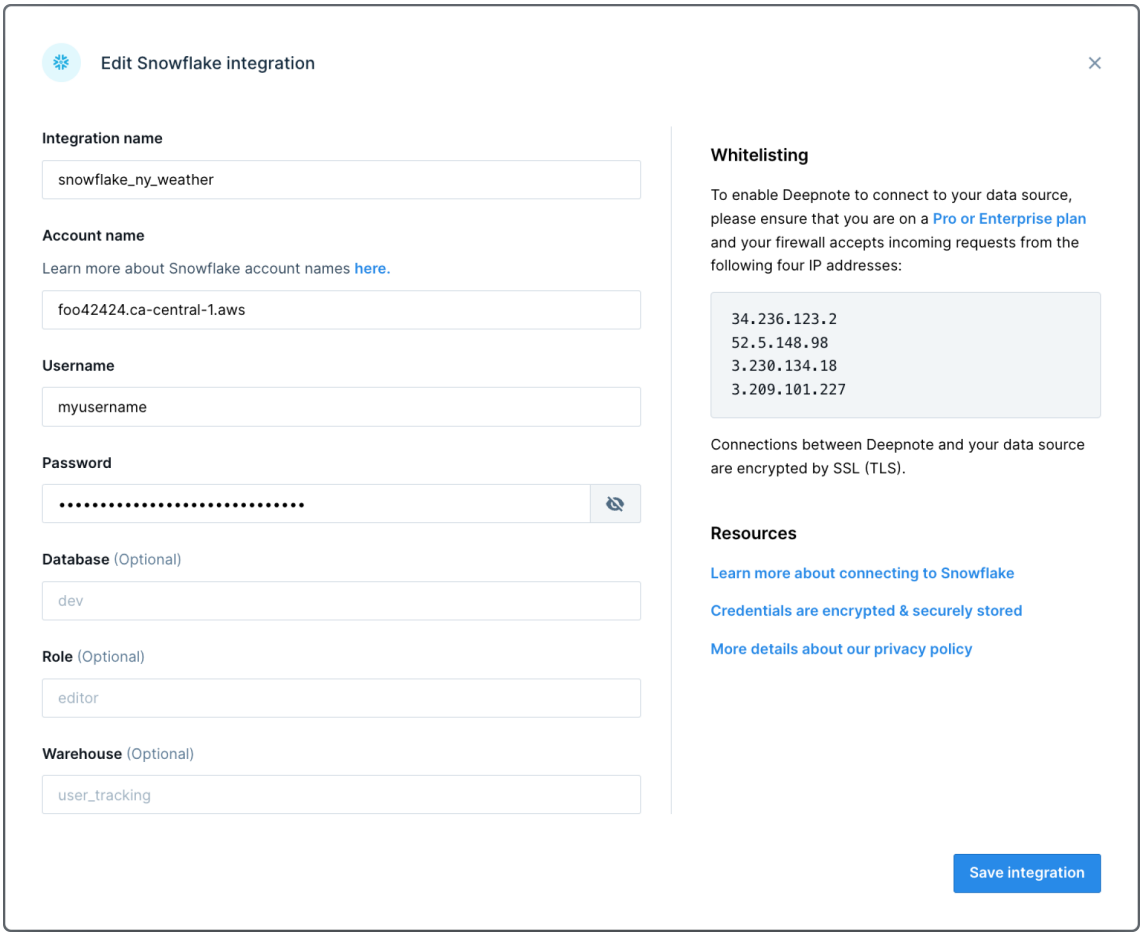
- A free Deepnote account
- A Snowflake account with admin access
- The weather dataset uploaded to your Snowflake database (uploading instructions [here]

What You'll Build

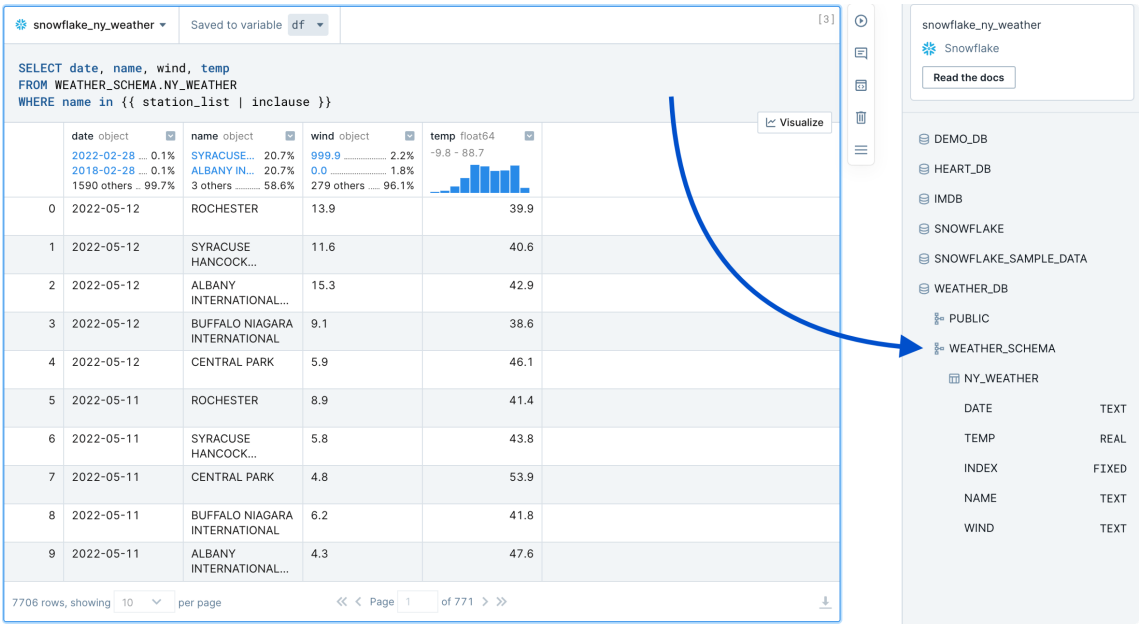
You will use the EDA tools in Deepnote to explore weather patterns in New York city. By the end of the guide, you will have created a notebook that contains generalizable techniques—and one that demonstrates how Snowflake and Deepnote work together to solve the hardest data analysis problems.

Setup a Snowflake integration inside Deepnote

To connect a Snowflake database to a Deepnote project, open the Snowflake integration modal and supply the associated Snowflake credentials (i.e., account, username, password). Note that the connection parameters to Snowflake are stored in the notebook as Python environment variables but can only be viewed by the workspace Admin.



Once you are connected, you will be able to browse your schema directly from Deepnote and query your tables with SQL blocks (described below).



Query Snowflake with Deepnote's SQL blocks

Similar to Python code cells used in Jupyter notebooks, Deepnote also includes native SQL cells which include syntax highlighting and autocomplete based on the Snowflake schema. Hover your mouse on the border of block, click the ➕ sign, and select your Snowflake SQL block from the list. Now you can write SQL as you would in a standard SQL editor as shown below.

Notice that in the example below we can already start exploring the weather data via the rich DataFrame display. This includes being able to use filters, sorting, pagination, as well as examining histograms, ratios of values, and data types. Importantly, the results set is saved to a Pandas DataFrame (in this case `df`)—meaning that you can continue using Pandas or any other tool in the Python library ecosystem to further explore the data.

snowflake_ny_weather | Saved to variable df [1]

```
SELECT date, name, wind, temp
FROM WEATHER_SCHEMA.NY_WEATHER
WHERE name in ('BUFFALO NIAGARA INTERNATIONAL','ROCHESTER')
```

Visualize

	date object	name object	wind object	temp float64
	2022-02-28 0.1% 2018-02-28 0.1% 1590 others 99.7%	BUFFALO N... 54.5% ROCHESTER 45.5%	0.0 4.5% 9.4 1.5% 229 others 94%	-1.0 - 83.2
0	2022-05-12	ROCHESTER	13.9	39.9
1	2022-05-12	BUFFALO NIAGARA INTERNATIONAL	9.1	38.6
2	2022-05-11	ROCHESTER	8.9	41.4
3	2022-05-11	BUFFALO NIAGARA INTERNATIONAL	6.2	41.8
4	2022-05-10	ROCHESTER	8.9	41.2
5	2022-05-10	BUFFALO NIAGARA INTERNATIONAL	11.3	44.6
6	2022-05-09	ROCHESTER	19.2	35.4
7	2022-05-09	BUFFALO NIAGARA INTERNATIONAL	12	33.9
8	2022-05-08	ROCHESTER	10.8	40.2
9	2022-05-08	BUFFALO NIAGARA INTERNATIONAL	7.2	39.7

2924 rows, showing 10 per page Page 1 of 293

Mix-and-match Python and SQL

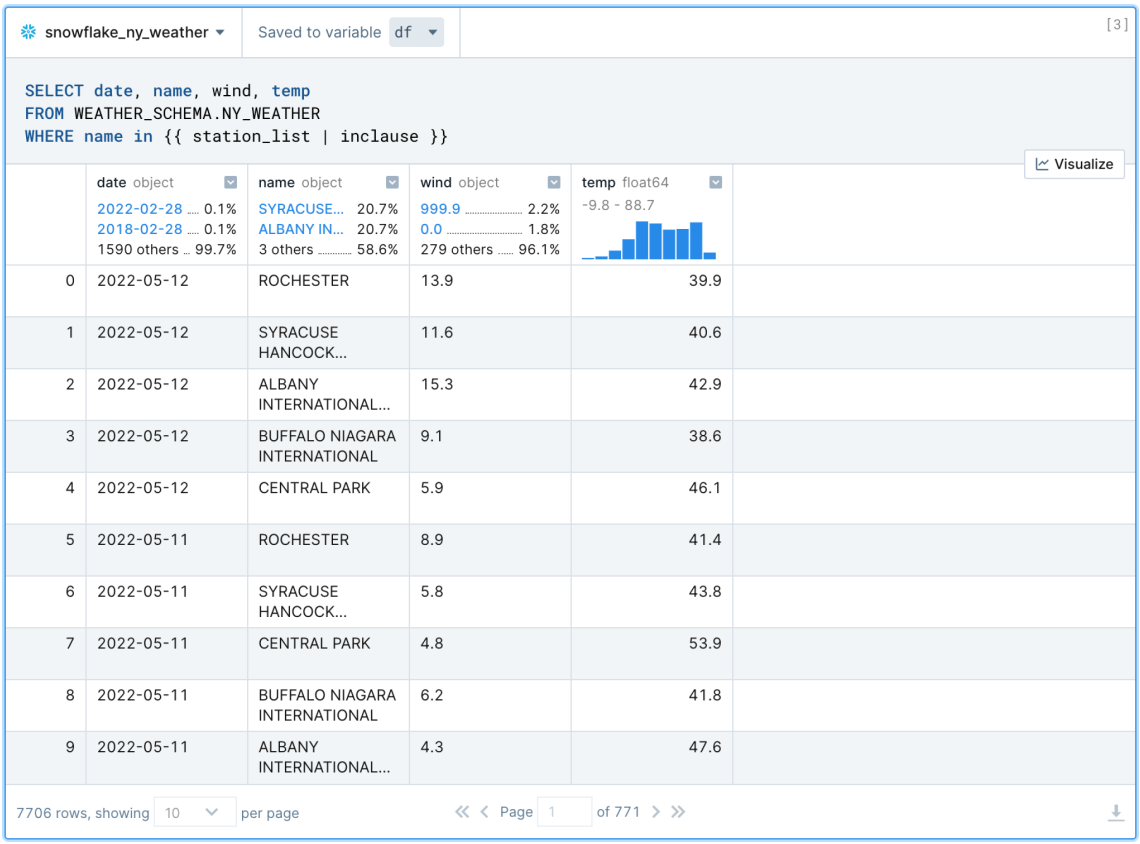
SQL queries in Deepnote also support the JinjaSQL templating language. This allows users to pass Python variables **directly into SQL queries** as well as use jinja-style control structures (conditionals and loops) for supplementing standard SQL.

In the example below, weather stations are filtered in the `WHERE` clause by passing in the python list called `station_list` rather than having to list them all out manually.

This is just one way to parameterize your SQL queries; we will see later how you can build a UI around your SQL blocks, allowing for interactive data exploration without having to repeat similar blocks of code, over and over again.

[2]

```
import pandas as pd
import numpy as np

station_list=['BUFFALO NIAGARA INTERNATIONAL',
              'ROCHESTER',
              'SYRACUSE HANCOCK INTERNATIONA',
              'ALBANY INTERNATIONAL AIRPORT',
              'CENTRAL PARK']
```

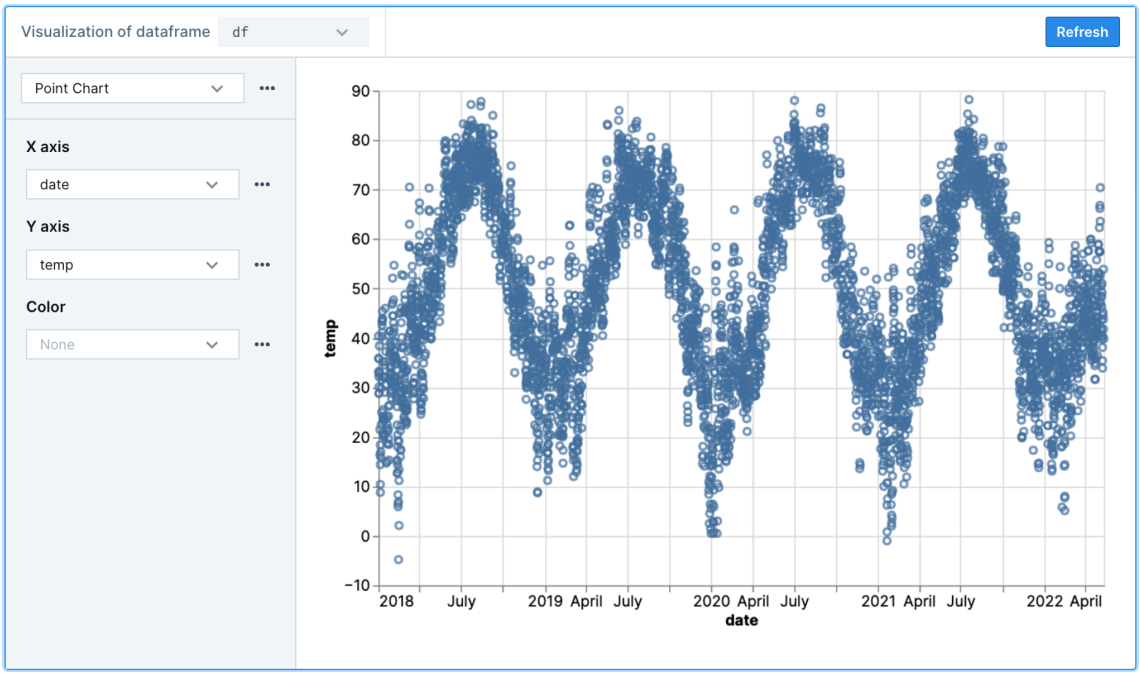


Visualize the results of a query

Deepnote provides no-code data visualization tools for streamlining EDA. Given the results of the query above, let's build a data visualization to quickly explore the expected weather patterns (again, click the ➕ sign to choose a chart block type). We should expect temperature to get warmer in the summer months and colder in the winter months.

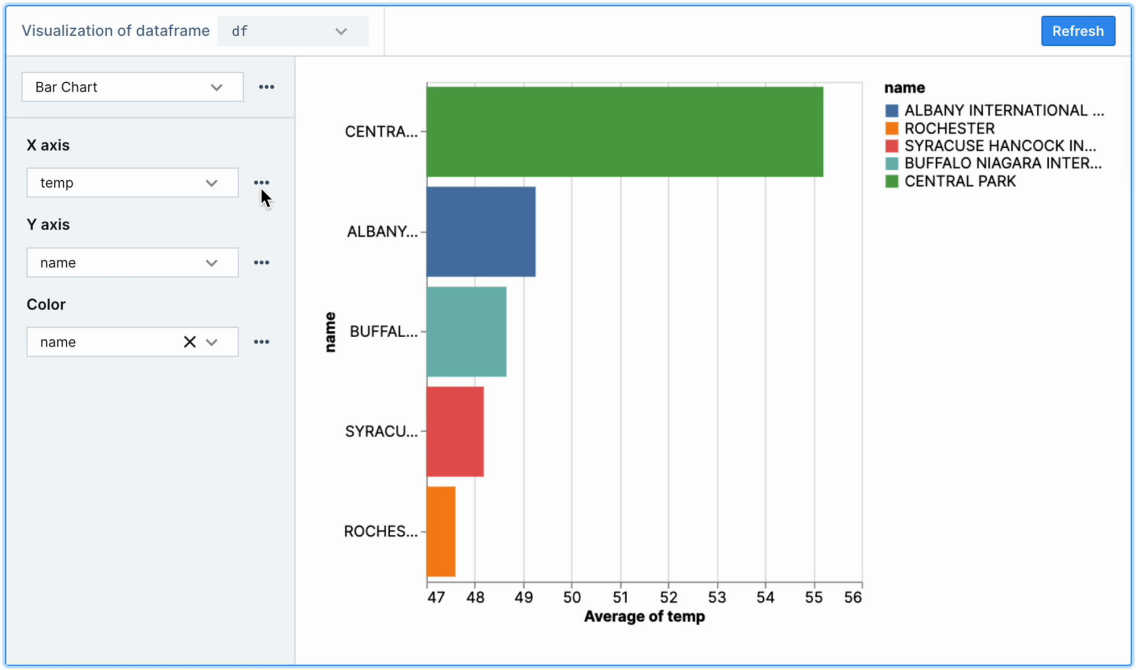
Let's choose `df` as the DataFrame to visualize by selecting it from the dropdown at the top of the Chart block. Select "point chart" and specify how you would like to map columns of your data to the X/Y coordinates on the chart. Since we want to see temperature as a function of time, let's select `date` for the X axis and `temp` for the Y axis (example below).

Sure enough, we can verify the seasonality in the weather patterns—and all without a single line of code.



While the chart above confirms the expected seasonality, we can't see the data broken down by station (i.e., the location/city of the weather sensor). Since the `name` column contains this information, we can again use no-code charts to look at the average `temp` by `name` to get a sense of how locations/cities differ in terms of temperature.

As shown below, aggregations and other options can be selected by clicking the ellipsis next to the encoding channels (X, Y, and Color). Here, we can visualize average temperature by station name by selecting `Average` from the aggregation options on the X axis (which is set to `temp`), and setting the Y and Color channels to `name`.



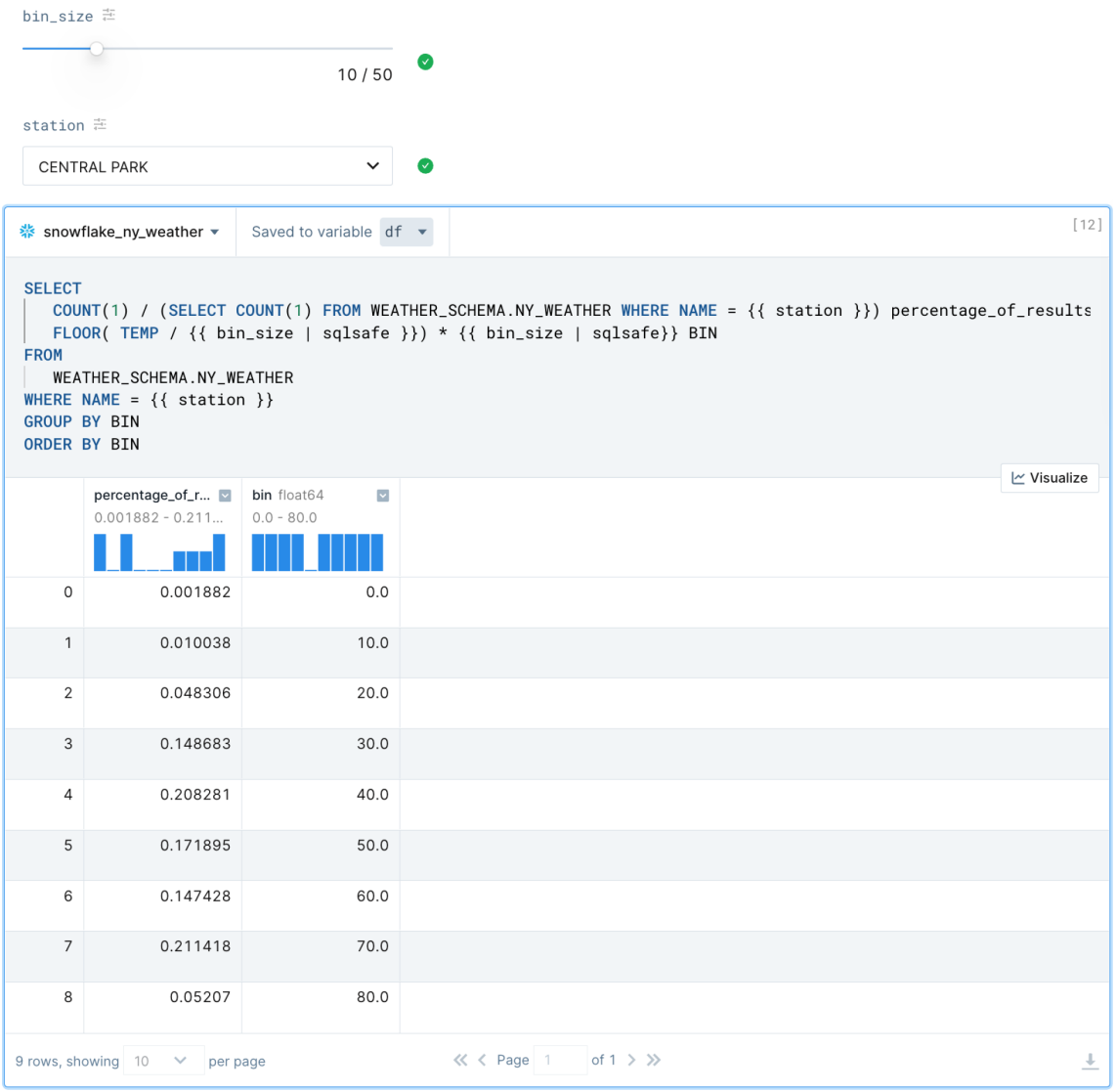
Many chart types and encoding combinations can used to create helpful data visualizations. This can be helpful not only in terms of developer speed but also for non-technical team members who may not be as familiar with plotting via code. For more complex needs, nothing is stopping us from using Altair, Plotly, or any other Python visualization library in Deepnote.

Build a UI around the SQL query

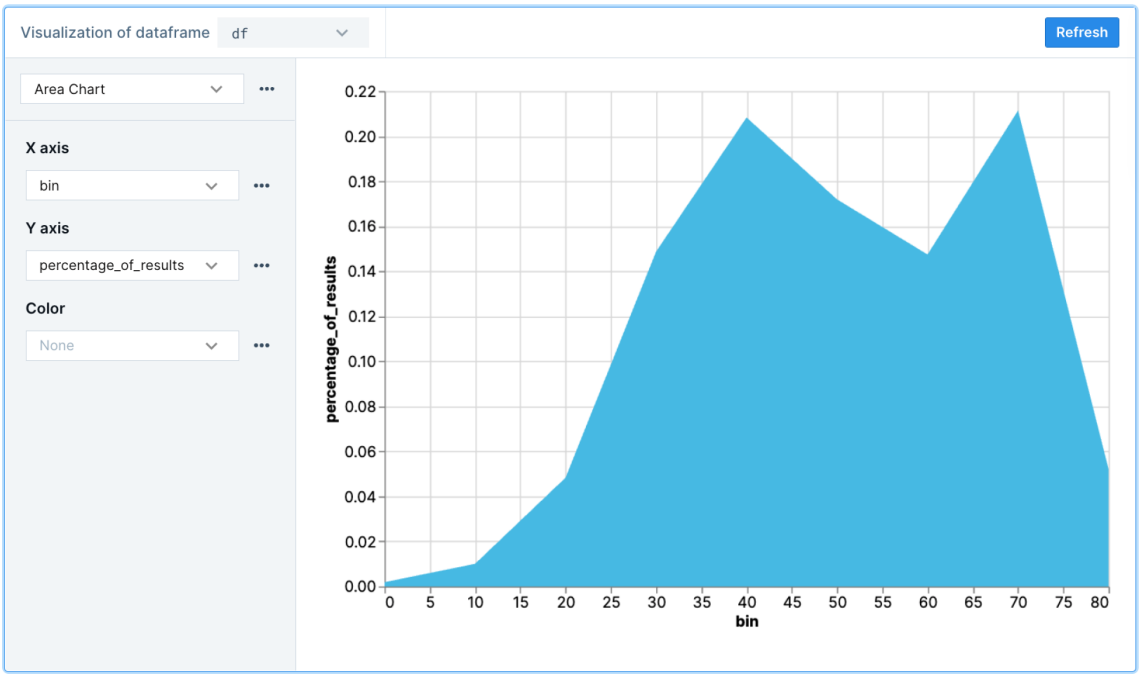
Data exploration is iterative. Put another way—data exploration is repetitive. While analysts strive to follow the DRY principle (do not repeat yourself), EDA code and behaviour is often at odds with that principle (in practice). For example, an SQL query that neatly displays one aspect of the data is likely going to be copied and pasted, then slightly changed, in order to display another aspect of the data.

Deepnote understands this tendency and provides a set of rich, interactive widgets (called Input blocks) that can be used to parameterize your SQL queries (or Python code). That is, instead of copying and pasting around similar queries, quickly wire up a slider bar (or some other UI element) to scroll through dimensions of your data. This approach is great for readability, reproducibility, and ease of use (especially for those non-technical users). As usual, click the ➕ button above to choose from the available input blocks.

The SQL query below constructs binned temperature readings (i.e., similar to a histogram) so that we can examine a distribution of results. As you can see, the `bin_size` slider bar and `name` dropdown selector provide a small UI for controlling the variables passed into the SQL query. This way, we can look at binned results for any station we'd like (including tweaking the bin size) without having to write additional code.



As a bit of a bonus, and to bring things full circle, remember that the output of this SQL query is still a Pandas DataFrame; therefore, we can again use a chart block to visualize it further (and we should get an actual histogram). Perfect.



Conclusion

With Deepnote and Snowflake you get the best in class for secure data governance and rapid exploratory programming, together in the same place. Snowflake provides a single unified view of data and ability execute diverse analytics workloads—with near-unlimited scale, concurrency, and performance. Snowflake dovetails perfectly with Deepnote's collaborative data science notebook and suite of tools built for exploratory data analysis.

If you want to learn more about Deepnote's data science notebook platform, visit deepnote.com.

What we've covered

We have built a generalizable workflow for exploratory data analysis. Deepnote's Python, SQL, and low-code solutions make analyzing data in Snowflake straightforward and time efficient—even for team members without advanced technical knowledge. Apply these approaches to your own datasets to make exploratory data analysis more productive.

How to Connect to Snowflake Using Spark

Spark processes large volumes of data and theSnowflake Data Cloud is a modern data platform, together they help enterprises make more data-driven decisions. But how does one go about connecting these two platforms?

Whether you're interested in using Spark to execute SQL queries on a Snowflake table or if you just want to read data from Snowflake and explore it using the Spark framework, this lab will walk you through:

- The installation and configuration of the Spark Snowflake Connector
- How to establish a Snowflake connection using PySpark
- How to establish a Snowflake connection using Scala

How to Install and Configure The Spark Snowflake Connector

To use the Spark Snowflake connector, you will need to make sure that you have the Spark environment configured with all of the necessary dependencies. The dependencies in question are the Snowflake JDBC driver, Spark Snowflake Connector (SSC), and the Spark framework itself.

Step 1

The first thing you need to do is decide which version of the SSC you would like to use and then go find the Scala and Spark version that is compatible with it. The SSC can be downloaded from Maven (an online package repository). In the repository, there are different package artifacts for each supported version of Scala, and within the Scala versions, there are different versions of the SSC.

In addition, there are separate artifacts that support different versions of the Spark framework. The SSC packages have the following naming conventions, "X.X.X-spark_Y.Y"

The three X's represent the version of Snowflake and the two Y's represent the version of Spark. For example, Snowflake version 2.9.1 or Spark version 3.1.

GroupId	ArtifactId	Latest Version	Updated	Download
net.snowflake	snowflake-jdbc	3.13.8 all (126)	13-Sep-2021	pom jar javadoc.jar sources.jar
net.snowflake	snowflake-jdbc-fips	3.13.8 all (24)	13-Sep-2021	pom jar javadoc.jar sources.jar
net.snowflake	spark-snowflake_2.11	2.9.1-spark_2.4 all (137)	23-Jul-2021	pom jar javadoc.jar sources.jar
net.snowflake	spark-snowflake_2.12	2.9.1-spark_3.1 all (35)	23-Jul-2021	pom jar javadoc.jar sources.jar

Step 2

Once you have found the version of the SSC you would like to use, the next step would be to download and install its corresponding jar files and the jar files for the dependencies mentioned above in your Spark cluster.

The [spark-shell --packages]command can be used to install both the Spark Snowflake Connector and the Snowflake JDBC driver in your local Spark cluster. It can also be used to install any other missing packages required for SSC.

Example command:

```
spark-shell --packages net.snowflake:snowflake-jdbc:3.12.17,net.snowflake:spark-snowflake_2.12:2.8.4-spark_3.0
```

How to Connect to Snowflake using PySpark

Install PySpark:

`pip install pyspark`

To enable Spark in Python, the PySpark script in the Spark distribution is required. What this means is that as we needed to install the SSC and Snowflake JDBC driver in the Spark shell script, we will have to do the same for the PySpark script using the command given below.

Example command:

```
pyspark --packages net.snowflake:snowflake-jdbc:3.8.0,net.snowflake:spark-
snowflake_2.11:2.4.14-spark_2.4
```

[PRO TIP:] Be sure to include the SSC and Spark JDBC driver in your Class Path environment variables so that the PySpark distribution recognizes the installation.

Now that we have our environment setup, the next step is to run our Python code that will allow us to `connect to Snowflake` using PySpark. The first setup in our code would be to import the necessary libraries and establish our Spark context for a local Spark cluster.

Example Code:

```
#import required libraries
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext

mySparkContext = SparkContext("local", "example App")
spark = SQLContext(mySparkContext )
spark_conf = SparkConf().setMaster('local')
```

Once the Spark context is set up, the next step is to create a python dictionary that contains the required Snowflake parameters to establish the Snowflake connection.

Required Parameters:

- `Account Identifier` -- Unique identifier for your Snowflake account, either created by your organization or automatically generated by Snowflake upon account creation
- `User Name` -- User name for your Snowflake account, created during account creation
- `Password` -- Password for your account, created during account creation
- `Database` -- Database in the Snowflake account that you are attempting to access
- `Schema` -- Schema within the database that you are attempting to access
- `Warehouse` -- Snowflake compute resources that are being used for your operations

Example Code:

```
# Snowflake connection parameters
sfparams = {
  "sfURL" : "<account_identifier>.snowflakecomputing.com",
  "sfUser" : "<user_name>",
  "sfPassword" : "<password>",
  "sfDatabase" : "<database>",
```

```
  "sfSchema" : "<schema>",
  "sfWarehouse" : "<warehouse>"
}
```

The final step would be to read in a [Snowflake table] into a Spark dataframe to verify that the connection has been established. To do this, the [dbtable]option has to be specified on the Spark dataframe. If you would like to execute a custom query, then the [query] option would need to be specified instead.

Example Code:

```
#read full table
df = spark.read.format("snowflake") \
  .options(**sfparams) \
  .option("dbtable",  "Employee") \
  .load()

#run custom query
df = spark.read.format("snowflake") \
  .options(**sfparams) \
  .option("query",  "SELECT * FROM Employee") \
  .load()
```

Zero to Snowflake: Simple SQL Stored Procedures

One question we often get when a customer is considering moving to Snowflake from another platform, like Microsoft SQL Server for instance, is what they can do about migrating their SQL stored procedures to Snowflake.

TL;DR: Below is a basic procedure template you can customize and extend for your use case:

```
CREATE OR REPLACE PROCEDURE simple_stored_procedure_example()
returns string not null
language javascript
as
$$
var cmd = `
<SOME SUPER SWEET SQL STATEMENT>
`
var sql = snowflake.createStatement({sqlText: cmd});
var result = sql.execute();
return '';
$$;
```

Drop your SQL query into the `<SOME SUPER SWEET SQL STATEMENT>` space and have fun. Note that you can use backticks around the SQL statement to keep it nicely formatted for readability.

Let's break down the example above:

1. Define the procedure without arguments.
2. Tell the procedure to return a string.
3. Make sure the runtime language is javascript ... duh.
4. Copy some SQL to the `cmd` variable.
5. Add the `cmd` variable to the `snowflake.createStatement()` function.
6. Execute the prepared statement in the `sql` variable, and store the results in a new variable called `result`.
7. Return a string (see step 2) on successful execution.
8. Profit. Note the moneybag.

This is another example where we pass in an argument to the procedure:

```
CREATE OR REPLACE PROCEDURE
simple_stored_procedure_example_with_arguments(awesome_argument VARCHAR)
returns string not null
language javascript
as
$$
var your_awesome_arg = AWESOME_ARGUMENT;
var result = `${your_awesome_arg}`
return result;
$$;
```

Here's how you call the procedure with any text you want in the argument:

```
CALL simple_stored_procedure_example_with_arguments('Howdy! ');
```

Giddy up!

Building a Snowflake Data Pipeline with a Stored Procedure, Task and Stream

Let's build a *slightly* more realistic scenario with a Snowflake task and stream. Snowflake's tasks are simply a way to run some SQL on a schedule or when triggered by other tasks. One problem with defining raw SQL in a task is that if you have to update the SQL definition---let's say by adding a column to a select statement---you must also remember to run an alter statement to resume the task when created or modified tasks are suspended by default.

By using a stored procedure like the template above, you can modify the SQL without having to remember to resume the task. So from the task's perspective, it's a transparent change, and if we combine a stream with a task, we can ensure that our procedure only executes when new data is added to the table.

Here's a more complete example that will only insert new records from a source table into a target table on a 15-minute schedule using a procedure, task and table stream for flagging new rows in the source table. First, let's create some tables and a stream for our data:

```
CREATE OR REPLACE TABLE source_table (
  uuid varchar
  ,inserted_at timestamp
);

CREATE OR REPLACE STREAM source_table_stream
ON TABLE source_table APPEND_ONLY=TRUE;

CREATE OR REPLACE TABLE target_table (
  uuid varchar
  ,source_inserted_at timestamp
  ,target_inserted_at timestamp
);
```

Next, we'll create a stored procedure that will insert a record into our source table and then select and insert that record into our target table from the stream:

```
CREATE OR REPLACE PROCEDURE source_table_stream_procedure()
returns string not null
language javascript
as
$$
var insert_cmd = `
INSERT INTO source_table
SELECT uuid_string(), current_timestamp::timestamp_ntz;
`
var sql_insert = snowflake.createStatement({sqlText: insert_cmd});
var insert_result = sql_insert.execute();

var stream_select_cmd = `
INSERT INTO target_table
SELECT
  uuid
  ,inserted_at
  ,current_timestamp::timestamp_ntz
FROM
```

```
  source_table_stream
WHERE
  metadata$action = 'INSERT';
`
var sql_select_stream = snowflake.createStatement({sqlText: stream_select_cmd});
var select_stream_result = sql_select_stream.execute();
return '';
$$;
```

We'll create the task:

```
CREATE OR REPLACE TASK source_table_stream_procedure_task
  WAREHOUSE = my_task_warehouse
  SCHEDULE = '15 MINUTE'
WHEN
  system$stream_has_data('source_table_stream')
AS
  CALL source_table_stream_procedure();
```

And, finally, set it to run:

```
ALTER TASK source_table_stream_procedure_task RESUME;
```

Let's check when the task is scheduled to run by looking at the `SCHEDULED_TIME` column by querying the task history:

```
SELECT *
FROM table(
  information_schema.task_history(
    task_name=>'source_table_stream_procedure_task'
    ,scheduled_time_range_start=>dateadd('hour',-1,current_timestamp())
  )
);
```

Congrats! You've just written a Snowflake data pipeline.

Getting Started with Geospatial - Geography

Overview

Geospatial query capabilities in Snowflake are built upon a combination of data types and specialized query functions that can be used to parse, construct, and run calculations over geospatial objects. This guide will introduce you to the `GEOGRAPHY` data type, help you understand geospatial formats supported by Snowflake, and walk you through the use of a variety of functions on a sample geospatial data set from the Snowflake Marketplace.

What You'll Learn

- how to acquire geospatial data from the Snowflake Marketplace
- how to interpret the `GEOGRAPHY` data type
- how to understand the different formats that `GEOGRAPHY` can be expressed in
- how to unload/load geospatial data
- how to use parser, constructor, and calculation geospatial functions in queries
- how to perform geospatial joins

What You'll Need

- A supported Snowflake [Browser]
- Sign-up for a Snowflake Trial
  - OR, have access to an existing Snowflake account with the `ACCOUNTADMIN` role or the `IMPORT SHARE` privilege
- Access to the geojson.io or WKT Playground website

What You'll Build

- A sample use case that involves points-of-interest in New York City.

Negative : The Marketplace data used in this QuickStart changes from time-to-time, and as such, your query results may be slightly different than indicated in this guide. Additionally, the Snowflake UI changes periodically as well, and instructions/screenshots may be out of date.

Acquire Marketplace Data

The first step in the guide is to acquire a geospatial data set that you can freely use to explore the basics of Snowflake's geospatial functionality. The best place to acquire this data is the Snowflake Marketplace!

Access Snowflake's Preview Web UI

app.snowflake.com

If this is the first time you are logging into the new Preview Snowflake UI, you will be prompted to enter your account name or account URL that you were given when you acquired a trial. The account URL contains your [account name] and potentially the region.

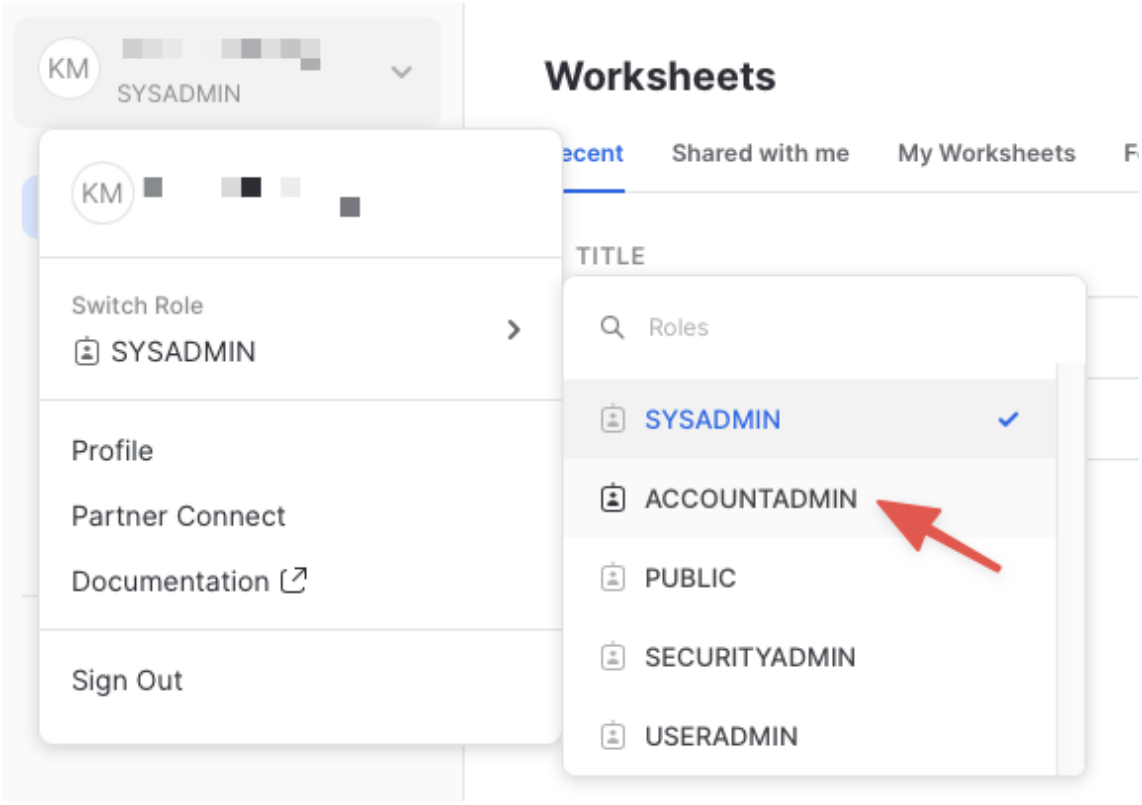
Click `Sign-in` and you will be prompted for your user name and password.

Positive : If this is not the first time you are logging into the new Preview Snowflake UI, you should see a "Select an account to sign into" prompt and a button for your account name listed below it. Click the account you wish to access and you will be prompted for your user name and password (or another authentication mechanism).

Increase Your Account Permission

The new Preview Snowflake web interface has a lot to offer, but for now, switch your current role from the default `SYSADMIN` to `ACCOUNTADMIN`. This increase in permissions will allow you to create shared databases from Snowflake Marketplace listings.

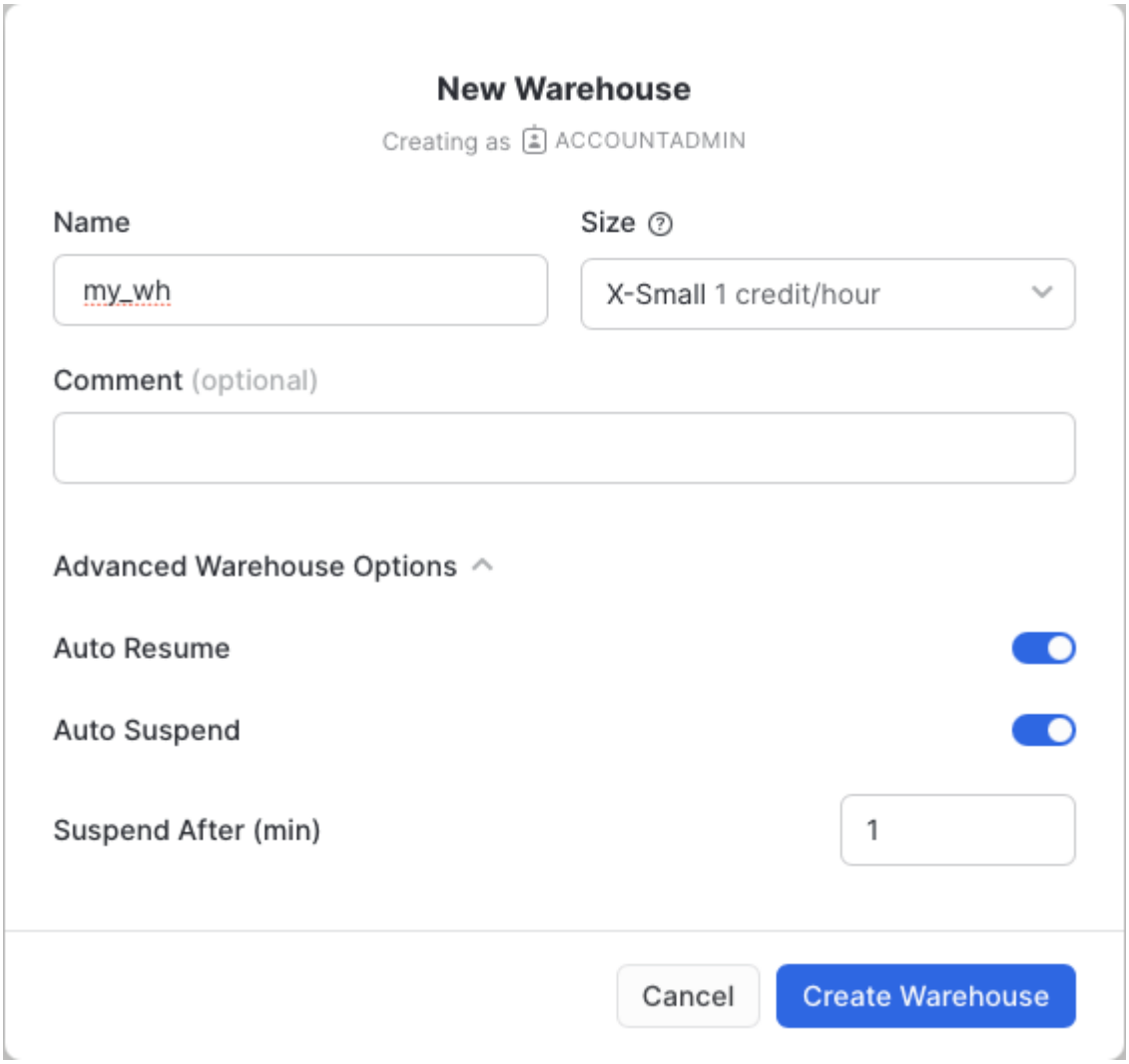
Positive : If you don't have the `ACCOUNTADMIN` role, switch to a role with `IMPORT SHARE` privileges instead.



Create a Virtual Warehouse (if needed)

If you don't already have access to a Virtual Warehouse to run queries, you will need to create one.

- Navigate to the `Compute > Warehouses` screen using the menu on the left side of the window
- Click the big blue `+ Warehouse` button in the upper right of the window
- Create an X-Small Warehouse as shown in the screen below

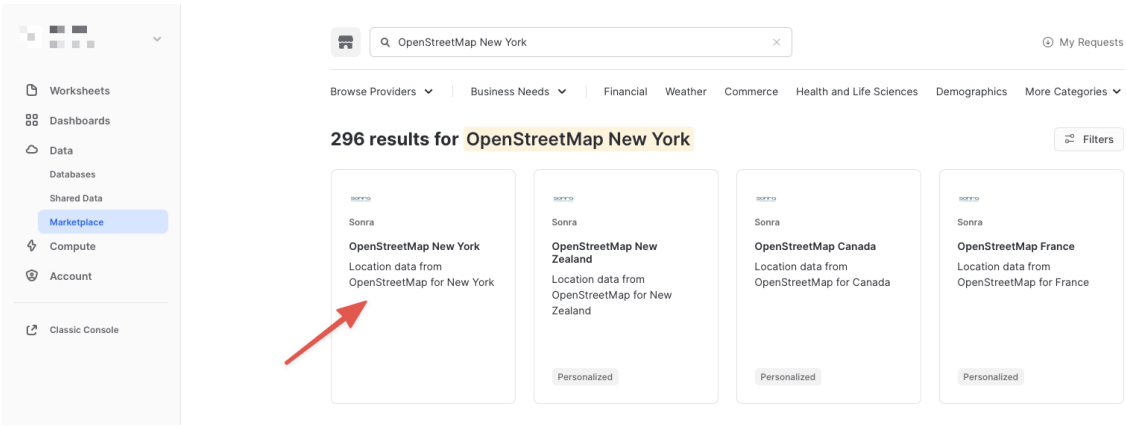


Be sure to change the `Suspend After (min)` field to 1 min to avoid wasting compute credits.

Acquire Data from the Snowflake Marketplace

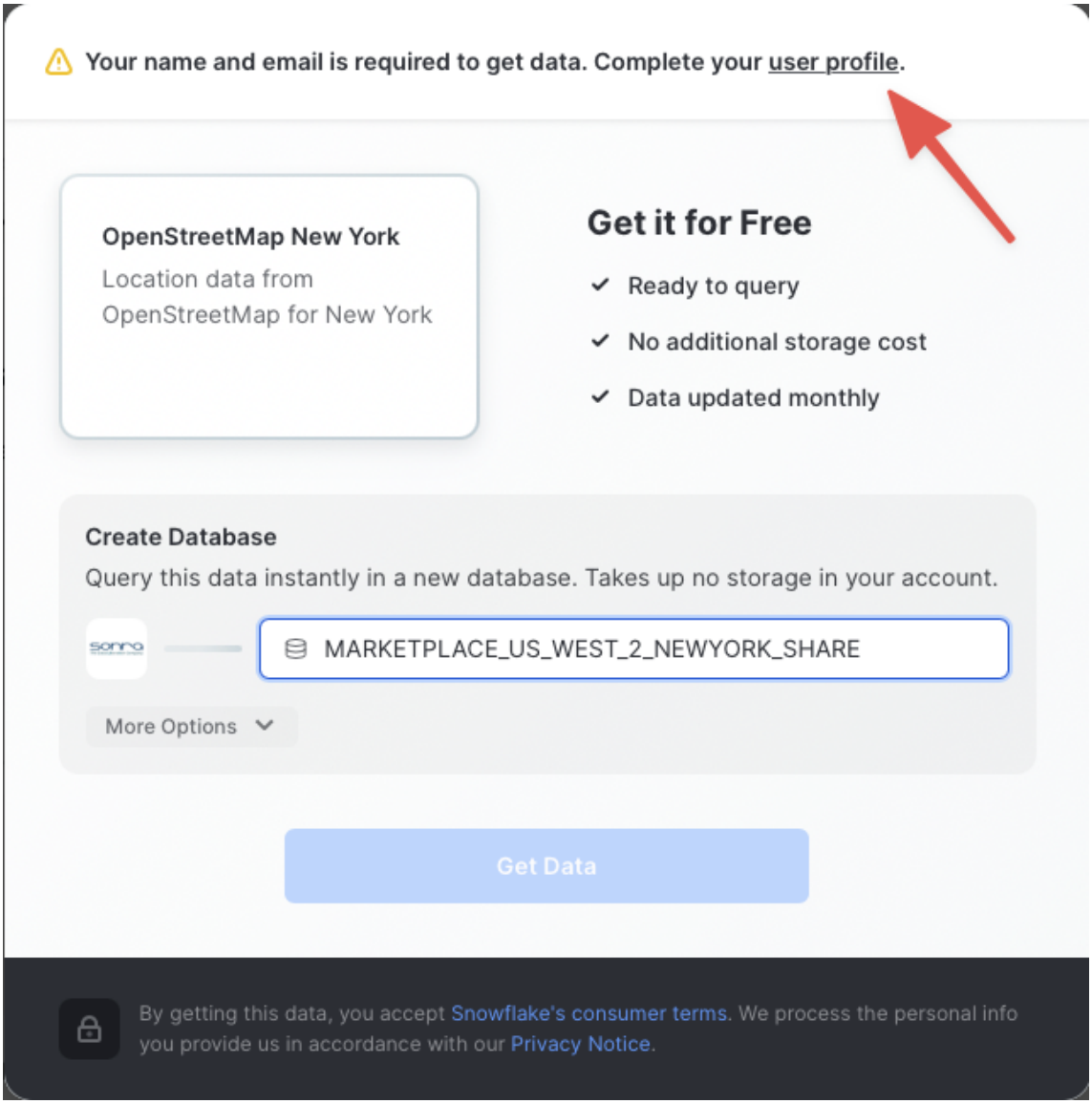
Now you can acquire sample geospatial data from the Snowflake Marketplace.

- Navigate to the `Marketplace` screen using the menu on the left side of the window
- Search for `OpenStreetMap New York` in the search bar
- Find and click the `Sonra OpenStreetMap New York` tile

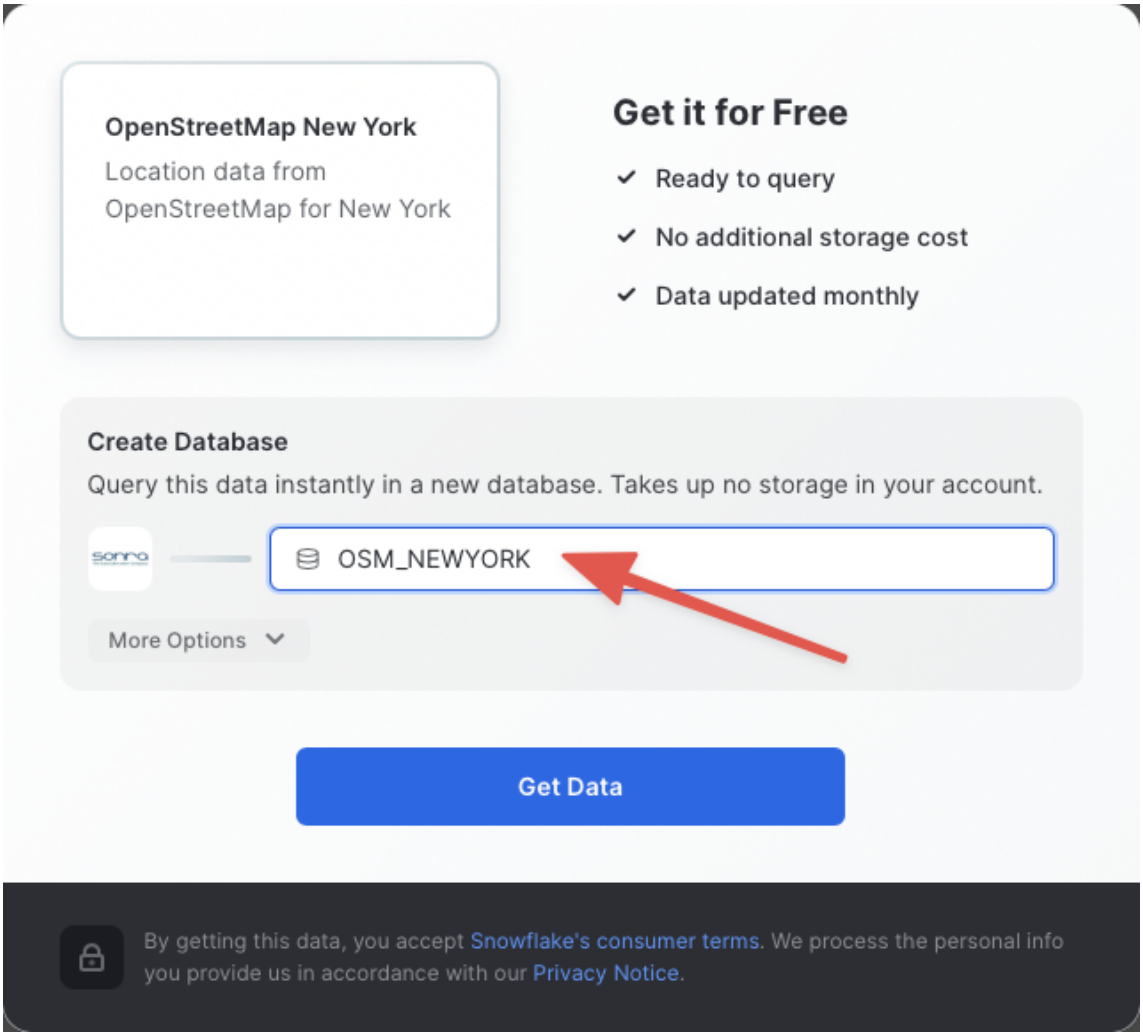


- Once in the listing, click the big blue `Get Data` button

Negative : On the `Get Data` screen, you may be prompted to complete your `user profile` if you have not done so before. Click the link as shown in the screenshot below. Enter your name and email address into the profile screen and click the blue `Save` button. You will be returned to the `Get Data` screen.



- On the `Get Data` screen, change the name of the database from the default to `OSM_NEWYORK`, as this name is shorter and all of the future instructions will assume this name for the database.



Congratulations! You have just created a shared database from a listing on the Snowflake Marketplace. Click the big blue `Query Data` button and advance to the next step in the guide.

Understand Geospatial Formats

The final step in the prior section opened a worksheet editor in the new Snowflake UI with a few pre-populated queries that came from the sample queries defined in the Marketplace listing. You are not going to run any of these queries in this guide, but you are welcome to run them later. Instead, you are going to open a new worksheet editor and run different queries to understand how the `GEOGRAPHY` data type works in Snowflake.

Open a New Worksheet and Choose Your Warehouse

- Click the big plus (+) button in the upper right of your browser window. This will open a new window or a new tab, or you may need to allow your browser to open the window/tab depending on your browser pop-up blocker settings.
- In the new Window, make sure `ACCOUNTADMIN` and `MY_WH` (or whatever your warehouse is named) are selected in the box beside the big plus (+) button in the upper right of your browser window.
- In the object browser on the left, expand the `OSM_NEWYORK` database, the `NEW_YORK` schema, and the `Views` grouping to see the various views that you have access to in this shared database. The data

provider has chosen to share only database views in this listing. You will use some of these views throughout the guide.

Now you are ready to run some queries.

The GEOGRAPHY data type

Snowflake's `GEOGRAPHY` data type is similar to the `GEOGRAPHY` data type in other geospatial databases in that it treats all points as longitude and latitude on a spherical earth instead of a flat plane. This is an important distinction from other geospatial types (such as `GEOMETRY`), but this guide won't be exploring those distinctions. More information about Snowflake's specification can be found [here].

Look at one of the views in the shared database which has a `GEOGRAPHY` column by running the following queries. Copy & paste the SQL below into your worksheet editor, put your cursor somewhere in the text of the query you want to run (usually the beginning or end), and either click the blue "Play" button in the upper right of your browser window, or press `CTRL+Enter` or `CMD+Enter` (Windows or Mac) to run the query.

```
// Set the working database schema
use schema osm_newyork.new_york;
```

The [use schema] command sets the active database.schema for your future queries so you do not have to fully qualify your objects.

```
// Describe the v_osm_ny_shop_electronics view
desc view v_osm_ny_shop_electronics;
```

The [desc or describe] command shows you the definition of the view, including the columns, their data type, and other relevant details. Notice the `coordinates` column is defined of `GEOGRAPHY` type. This is the column you will focus on in the next steps.

View GEOGRAPHY Output Formats

Snowflake supports 3 primary geospatial formats and 2 additional variations on those formats. They are:

- **GeoJSON**: a JSON-based standard for representing geospatial data
- **WKT & EWKT**: a "Well Known Text" string format for representing geospatial data and the "Extended" variation of that format
- **WKB & EWKB**: a "Well Known Binary" format for representing geospatial data in binary and the "Extended" variation of that format

These formats are supported for ingestion (files containing those formats can be loaded into a `GEOGRAPHY` typed column), query result display, and data unloading to new files. You don't need to worry about how Snowflake stores the data under the covers, but rather how the data is displayed to you or unloaded to files through the value of a session variable called `GEOGRAPHY_OUTPUT_FORMAT`.

Run the query below to make sure the current format is GeoJSON.

```
// Set the output format to GeoJSON
alter session set geography_output_format = 'GEOJSON';
```

The [alter session] command lets you set a parameter for your current user session, which in this case is the `GEOGRAPHY_OUTPUT_FORMAT`. The default value for this parameter is `'GEOJSON'`, so normally you wouldn't have to run this command if you want that format, but this guide wants to be certain the next queries are run with the `'GEOJSON'` output.

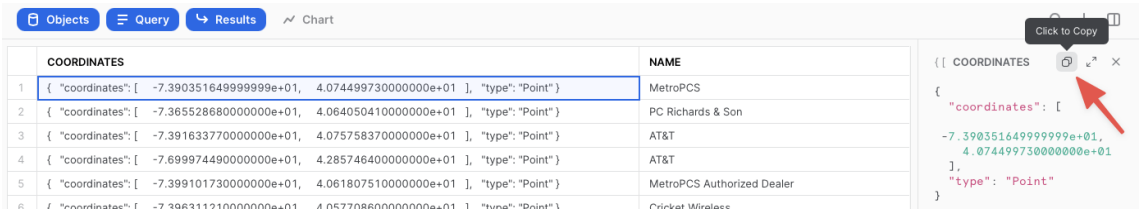
Now run the following query against the `V_OSM_NY_SHOP_ELECTRONICS` view.

```
// Query the v_osm_ny_shop_electronics view for rows of type 'node' (long/lat points)
select coordinates, name from v_osm_ny_shop_electronics where type = 'node' limit 25;
```

In the result set, notice the `coordinates` column and how it displays a JSON representation of a point. It should look something like this:

```
{   "coordinates": [     -7.390351649999999e+01,     4.074499730000000e+01    ],
"type": "Point" }
```

If you click on a cell in the `coordinates` column of the query result, the JSON representation will also show in the cell panel on the right side of the query window, and it includes a button that allows you to copy that JSON text (see screenshot below). You will use this capability in later exercises.



Now run the next query.

```
// Query the v_osm_ny_shop_electronics view for rows of type 'way' (a collection of
many points)
select coordinates, name from v_osm_ny_shop_electronics where type = 'way' limit 25;
```

Click on a cell in the `coordinates` column of the query result. Notice in the cell panel how the JSON is expanded with many more points in the JSON array. This shows you the difference between a geospatial representation of a single point, vs a representation of many points.

Now look at the same queries but in a different format. Run the following query:

```
// Set the output format to WKT
alter session set geography_output_format = 'WKT';
```

Run the previous two queries again. With each run, click on a cell in the `coordinates` column and examine the output.

```
select coordinates, name from v_osm_ny_shop_electronics where type = 'node' limit 25;
select coordinates, name from v_osm_ny_shop_electronics where type = 'way' limit 25;
```

WKT looks different than GeoJSON, and is arguably more readable. Here you can more clearly see the [geospatial object types] which are represented below in the example output:

```
// An example of a POINT
POINT(-74.0266511 40.6346599)
// An example of a POLYGON
POLYGON((-74.339971 43.0631175,-74.3397734 43.0631363,-74.3397902
43.0632306,-74.3399878 43.0632117,-74.339971 43.0631175))
```

Positive : You will use several different geospatial object types in this guide, and the guide will explain them more in later sections as you use them.

Lastly, look at WKB output. Run the following query:

```
// Set the output format to WKB
alter session set geography_output_format = 'WKB';
```

And run the two queries again, click on a cell in the `coordinates` column each time.

```
select coordinates, name from v_osm_ny_shop_electronics where type = 'node' limit 25;
select coordinates, name from v_osm_ny_shop_electronics where type = 'way' limit 25;
```

Notice how WKB is incomprehensible to a human reader. Other than the length of the binary value, it's hard to tell the difference between the `POINT` and the `POLYGON`. However, this format is handy in data loading/unloading, as you'll see in the next section.

Unload/Load Data

Now that you understand the different output formats, you can create new files from the electronics view, then load those files into new tables with the `GEOGRAPHY` data type. You will also encounter your first examples of geospatial *parsers* and *constructors*.

Create New WKB Files From Queries

In this step we're going to use Snowflake's [COPY into location] feature to take the output of a query and create a file in your local [user stage]. Because your output format is set to WKB, the geospatial column in that table will be represented in the WKB format in the new files.

Negative : The WKB format is being chosen here for its simplicity within a file. Since WKB is a single alpha-numeric string with no delimiters, spaces, or other difficult characters, it is excellent for storing geospatial data in a file. That doesn't mean other formats are to be avoided in real world use cases, but WKB will make your work easier in this guide.

Make sure we're using the WKB output format by running this query again:

```
alter session set geography_output_format = 'WKB';
```

If you're not familiar with the anatomy of a `COPY` command, the code comments below will break down the code of the first query, which copies a few columns and all rows from the electronics view:

```
// Define the write location (@~/ = my user stage) and file name for the file
copy into @~/osm_ny_shop_electronics_all.csv
// Define the query that represents the data output
from (select id,coordinates,name,type from v_osm_ny_shop_electronics)
// Indicate the comma-delimited file format and tell it to double-quote strings
file_format=(type=csv field_optionally_enclosed_by='"')
// Tell Snowflake to write one file and overwrite it if it already exists
single=true overwrite=true;
```

Run the query above and you should see an output that indicates the number of rows that were unloaded.

Run the second unload query below, which adds some filtering to the output query and a parser:

```
copy into @~/osm_ny_shop_electronics_points.csv
from (
  select id,coordinates,name,type,st_x(coordinates),st_y(coordinates)
  from v_osm_ny_shop_electronics where type='node'
) file_format=(type=csv field_optionally_enclosed_by='"')
single=true overwrite=true;
```

In this query, the parsers `ST_X` and `ST_Y` are extracting the longitude and latitude from a `GEOGRAPHY POINT` object. These parsers only accept single points as an input, so you had to filter the query on `type = 'node'`. In Snowflake, the 'x' coordinate is always the longitude and the 'y' coordinate is always the latitude, and as you will see in a future constructor, the longitude is always listed first.

LIST and Query User Staged Files

You should now have 2 files in your user stage. Verify they are there by running the [list] command. The 'osm' string will act as a filter to tell the command to show only the files beginning with 'osm'.

```
list @~/osm;
```

You can query a simple file directly in the stage by using the '$' notation below to represent each delimited column in the file, which in this case Snowflake assumes to be a comma-delimited CSV. Run this query:

```
select $1,$2,$3,$4 from @~/osm_ny_shop_electronics_all.csv;
```

Notice how the second column displays the WKB geospatial data in double-quotes because of how you created the file. This will not load directly into a `GEOGRAPHY` data type, so you need to further define the file format. Run each query below to create a local database and a new file format in that database. You will also switch your `GEOGRAPHY` output format back to WKT to improve readability of future queries.

```
// Create a new local database
create or replace database geocodelab;
// Change your working schema to the public schema in that database
use schema geocodelab.public;
// Create a new file format in that schema
create or replace file format geocsv type = 'csv' field_optionally_enclosed_by='"';
// Set the output format back to WKT
alter session set geography_output_format = 'WKT';
```

Now query the 'all' files in the stage using the file format:

```
select $1,TO_GEOGRAPHY($2),$3,$4
from @~/osm_ny_shop_electronics_all.csv
(file_format => 'geocsv');
```

Notice the use of the `TO_GEOGRAPHY` constructor which tells Snowflake to interpret the WKB binary string as geospatial data and construct a `GEOGRAPHY` type. The WKT output format allows you to see this representation in a more readable form. You can now load this file into a table that includes a `GEOGRAPHY` typed column by running the two queries below:

```
// Create a new 'all' table in the current schema
create or replace table electronics_all
(id number, coordinates geography, name string, type string);
// Load the 'all' file into the table
```

```
copy into electronics_all from @~/osm_ny_shop_electronics_all.csv
file_format=(format_name='geocsv');
```

You should see all rows loaded successfully into the table with 0 errors seen.

Now turn your attention to the other 'points' file. If you recall, you used `ST_X` and `ST_Y` to make discrete longitude and latitude columns in this file. It is not uncommon to receive data which contains these values in different columns, and you can use the `ST_MAKEPOINT` constructor to combine two discrete longitude and latitude columns into one `GEOGRAPHY` typed column. Run this query:

```
select $1,ST_MAKEPOINT($5,$6),$3,$4,$5,$6
from @~/osm_ny_shop_electronics_points.csv
(file_format => 'geocsv');
```

Negative : Notice in `ST_MAKEPOINT` that the longitude column is listed first. Despite the common verbal phrase "lat long," you always put longitude before latitude to represent a geospatial POINT object in Snowflake.

Now create a table and load the 'points' file into that table. Run these two queries.

```
// Create a new 'points' table in the current schema
create or replace table electronics_points
(id number, coordinates geography, name string, type string,
long number(38,7), lat number(38,7));
// Load the 'points' file into the table
copy into electronics_points from (
  select $1,ST_MAKEPOINT($5,$6),$3,$4,$5,$6
  from @~/osm_ny_shop_electronics_points.csv
) file_format=(format_name='geocsv');
```

You should see all rows loaded successfully into the table with 0 errors seen.

Positive : In the 'all' file load statement, you didn't have to specify a query to load the file because when you have a column in a file that is already in a Snowflake supported geospatial format, and load that value into a `GEOGRAPHY` typed column, Snowflake automatically does the geospatial construction for you. In the 'points' file, however, you must use a transform query to construct two discrete columns into a single `GEOGRAPHY` column using a geospatial constructor function.

To conclude this section, you can query your recently loaded tables using the two queries below:

```
select * from electronics_all;
select * from electronics_points;
```

Calculations and More Constructors

Now that you have the basic understand of how the `GEOGRAPHY` data type works and what a geospatial representation of data looks like in various output formats, it's time to walkthrough a scenario that requires you to run some geospatial queries to answer some questions.

Positive : It's worth noting here that the scenario in the next three sections is more akin to what a person would do with a map application on their mobile phone, rather than how geospatial data would be used in fictional business setting. This was chosen intentionally to make this guide and these queries more relatable to the person doing the guide, rather than trying to create a realistic business scenario that is relatable to all industries, since geospatial data is used very differently across industries.

Before you begin the scenario, switch the active schema back to the shared database and make sure the output format is either GeoJSON or WKT, as you will be using another website to visualize the query results. Which output you choose will be based on your personal preference - WKT is easier for the casual person to read, while GeoJSON is arguably more common. The GeoJSON visualization tool is easier to see the points, lines, and shapes, so this guide will be showing the output for GeoJSON.

Also note that from here on out, SQL statements and functions that have been previously covered will no longer have their usage explained in the code comments or the text of the guide. Run the two queries below:

```
use schema osm_newyork.new_york;
// Run just one of the below queries based on your preference
alter session set geography_output_format = 'GEOJSON';
alter session set geography_output_format = 'WKT';
```

The Scenario

Pretend that you are currently living in your apartment near Times Square in New York City. You need to make a shopping run to Best Buy and the liquor store, as well as grab a coffee at a coffee shop. Based on your current location, what are the closest stores or shops to do these errands, and are they the most optimal locations to go to collectively? Are there other shops you could stop at along the way?

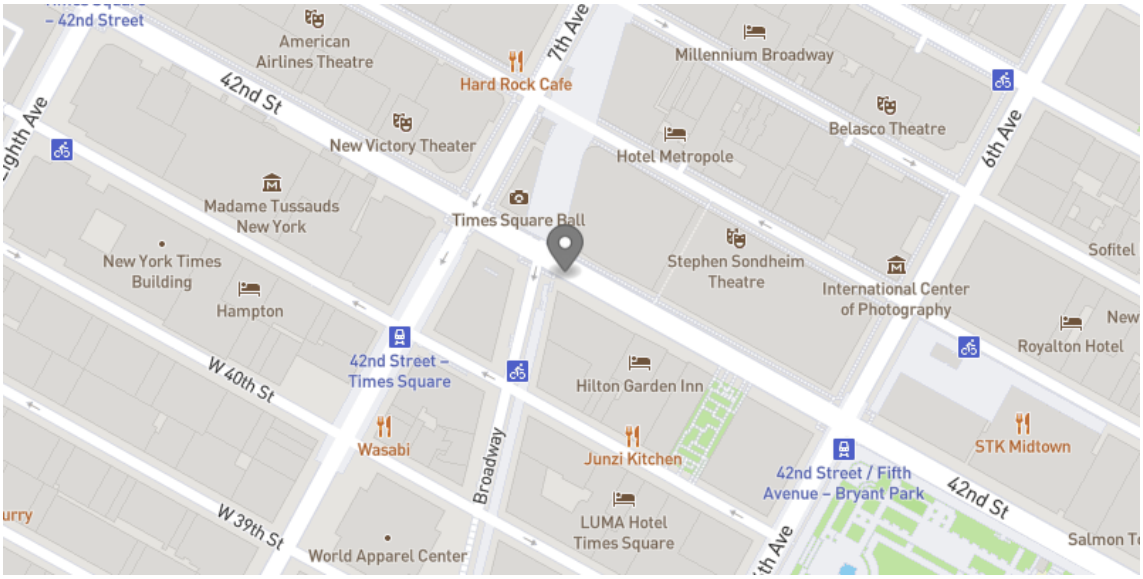
Start with running a query that represents your current location. This location has been preselected for the guide using a website that returns longitude and latitude when you click on a location on a map. Run this query:

```
select to_geography('POINT(-73.986226 40.755702)');
```

Notice there is no `from` clause in this query, which allows you to construct a `GEOGRAPHY` object in a simple `select` statement.

Negative : `POINT(-73.986226 40.755702)` is already a geography object in WKT format, so there was no real need to convert it again, but it was important to show the most basic way to use `TO_GEOGRAPHY` to construct a simple geography object.

- Click on the data cell that was returned and copy the `POINT` object using the method demonstrated earlier by clicking on the copy button in the cell panel on the right.
- Navigate to the geojson.io or WKT Playground website and clear the contents of the text box.
- Paste your `POINT` object into the text box (and click `PLOT SHAPE` for the WKT Playground).
- Use the map zoom controls (+/- buttons) and click the zoom out (-) button until you can see more of the New York City surroundings. You should see something like the screenshot below, though you may see more depending on your browser window size and how far you zoomed out.



In the image above, the dark grey map location icon represents the `POINT` object location. Now you know where you are!

Find the Closest Locations

In the next step, you are going to run queries to find the closest Best Buy, liquor store, and coffee shop to your current location from above. These queries are very similar and will do several things:

- One will query the electronics view, the other two will query the food & beverages view, applying appropriate filters to find the thing we're looking for.
- All queries will use the `ST_DWITHIN` function in the `where` clause to filter out stores that aren't within the stated distance. The function takes two points and a distance to determine whether those two points are less than or equal to the stated distance from each other, returning `true` if they are and `false` if they are not. In this function, you will use the `coordinates` column from each view to scan through all of the Best Buys, liquor stores, or coffee shops and compare them to your current location `POINT`, which you will construct using the previously used `ST_MAKEPOINT`. You will then use 1600 meters for the distance value, which is roughly equivalent to a US mile.
    - Note that in the queries below, the syntax `ST_DWITHIN(...) = true` is used for readability, but the `= true` is not required for the filter to work. It is required if you were to need an `= false` condition.
- All queries will also use the `ST_DISTANCE` function, which actually gives you a value in meters representing the distance between the two points. When combined with `order by` and `limit` clauses, this will help you return only the row that is the smallest distance, or closest.
    - Also note in `ST_DISTANCE` that you use the constructor `TO_GEOGRAPHY` for your current location point instead of the `ST_MAKEPOINT` constructor that you used earlier in `ST_DWITHIN`. This is to show you that that `TO_GEOGRAPHY` is a general purpose constructor where `ST_MAKEPOINT` specifically makes a `POINT` object, but in this situation they resolve to the same output. Sometimes there is more than one valid approach to construct a geospatial object.

  Run the following queries (the first one has comments similar to above):

```
// Find the closest Best Buy
select id, coordinates, name, addr_housenumber, addr_street,
// Use st_distance to calculate the distance between your location and Best Buy
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
// Filter just for Best Buys
where name = 'Best Buy' and
// Filter for Best Buys that are within about a US mile (1600 meters)
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
// Order the results by the calculated distance and only return the lowest
order by 6 limit 1;

// Find the closest liquor store
select id, coordinates, name, addr_housenumber, addr_street,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'alcohol' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 6 limit 1;

// Find the closest coffee shop
select id, coordinates, name, addr_housenumber, addr_street,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 6 limit 1;
```

In each case, the query returns a `POINT` object, which you aren't going to do anything with just yet, but now you have the queries that return the desired results. It would be really nice, however, if you could easily visualize how these points relate to each other.

Collect Points Into a Line

In the next step of this section, you're going to 'collect' the points using `ST_COLLECT` and make a `LINESTRING` object with the `ST_MAKELINE` constructor. You will then be able to visualize this line on geojson.io.

- The first step in the query to is create a common table expression (CTE) query that unions together the queries you ran in the above step (keeping just the `coordinates` and `distance_meters` columns). This CTE will result in a 4 row output - 1 row for your current location, 1 row for the Best Buy location, 1 row for the liquor store, and 1 row for the coffee shop.
- You will then use `ST_COLLECT` to aggregate those 4 rows in the `coordinates` column into a single geospatial object, a `MULTIPOINT`. This object type is a collection of `POINT` objects that are interpreted as having no connection to each other other than they are grouped. A visualization tool will not connect these points, just plot them, so in the next step you'll turn these points into a line.

Run this query and examine the output:

```
// Create the CTE 'locations'
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'alcohol' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
// Query the CTE result set, aggregating the coordinates into one object
select st_collect(coordinates) as multipoint from locations;
```

The next thing you need to do is convert that `MULTIPOINT` object into a `LINESTRING` object using `ST_MAKELINE`, which takes a set of points as an input and turns them into a `LINESTRING` object. Whereas a `MULTIPOINT` has points with no assumed connection, the points in a `LINESTRING` will be interpreted as connected in the order they appear. Needing a collection of points to feed into `ST_MAKELINE` is the reason why you did the `ST_COLLECT` step above, and the only thing you need to do to the query above is wrap the `ST_COLLECT` in an `ST_LINESTRING` like so:

```
select st_makeline(st_collect(coordinates),to_geography('POINT(-73.986226
40.755702)'))
```

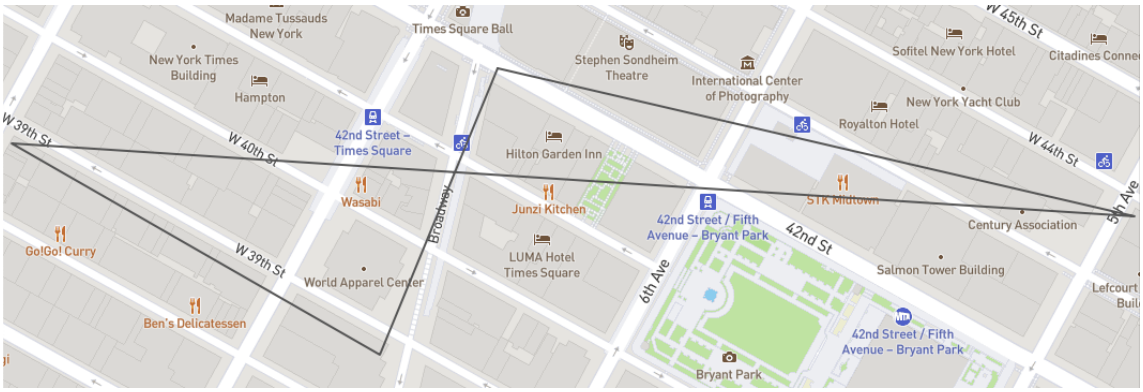
Positive : You may be wondering why your current position point was added as an additional point in the line when you already included it as the first point in the `MULTIPOINT` collection above? Stay tuned for why you need this later, but logically it makes sense that you plan to go back to your New York City apartment at the end of your shopping trip.

Here is the full query for you to run (without comments):

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_electronics
```

```
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'alcohol' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_makeline(st_collect(coordinates),to_geography('POINT(-73.986226
40.755702)'))
as linestring from locations;
```

Copy the result cell from the above query and paste it into geojson.io. You should get this:



Yikes! You can see in the image above that the various shops are in three different directions from your original location. That could be a long walk. Fortunately, you can find out just how long by wrapping a `ST_DISTANCE` function around the `LINESTRING` object, which will calculate the length of the line in meters. Run the query below:

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
```

```
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'alcohol' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
// Feed the linestring into an st_length calculation
select st_length(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)')))
as length_meters from locations;
```

Wow! Almost 2120 meters!

Negative : It is correct to note that this distance represents a path based on how a bird would fly, rather than how a human would navigate the streets. The point of this exercise is not to generate walking directions, but rather to give you a feel of the various things you can parse, construct, and calculate with geospatial data and functions in Snowflake.

Now move to the next section to see how you can optimize your shopping trip.

Joins

In the previous section, all of your queries to find the closest Best Buy, liquor store, and coffee shop were based on proximity to your Times Square apartment. But wouldn't it make more sense to see, for example, if there was a liquor store and/or coffee shop closer to Best Buy? You can use geospatial functions in a table join to find out.

Is There Anything Closer to Best Buy?

You have been using two views in your queries so far: `v_osm_ny_shop_electronics`, where stores like Best Buy are catalogued, and `v_osm_ny_shop_food_beverage`, where liquor stores and coffee shops are catalogued. To find the latter near the former, you'll join these two tables. The new queries introduce a few changes:

- The electronics view will serve as the primary view in the query, where you'll put a filter on the known Best Buy store using its id value from the view.
- Instead of the `JOIN` clause using a common `a.key = b.key` foreign key condition, the `ST_DWITHIN` function will serve as the join condition (remember before the note about not needing to include the `= true` part).
- The `ST_DISTANCE` calculation is now using the Best Buy coordinate and all of the other coordinates in the food & beverage view to determine the closest liquor store and coffee shop location to Best Buy.

Run the two queries below (only the first is commented):

```
// Join to electronics to find a liquor store closer to Best Buy
select fb.id,fb.coordinates,fb.name,fb.addr_housenumber,fb.addr_street,
// The st_distance calculation uses coordinates from both views
st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
// The join is based on being within a certain distance
```

```
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
// Hard-coding the known Best Buy id below
where e.id = 1428036403 and fb.shop = 'alcohol'
// Ordering by distance and only showing the lowest
order by 6 limit 1;

// Join to electronics to find a coffee shop closer to Best Buy
select fb.id,fb.coordinates,fb.name,fb.addr_housenumber,fb.addr_street,
st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'coffee'
order by 6 limit 1;
```

If you note in the result of each query, the first query found a different liquor store closer to Best Buy, whereas the second query returned the same coffee shop from your original search, so you've optimized as much as you can.

Negative : The id of the selected Best Buy was hard coded into the above queries to keep them easier to read and to keep you focused on the join clause of these queries, rather than introducing sub queries to dynamically calculate the nearest Best Buy. Those sub queries would have created longer queries that were harder to read.

Positive : If you're feeling adventurous, go read about other possible relationship functions that could be used in the join for this scenario [here].

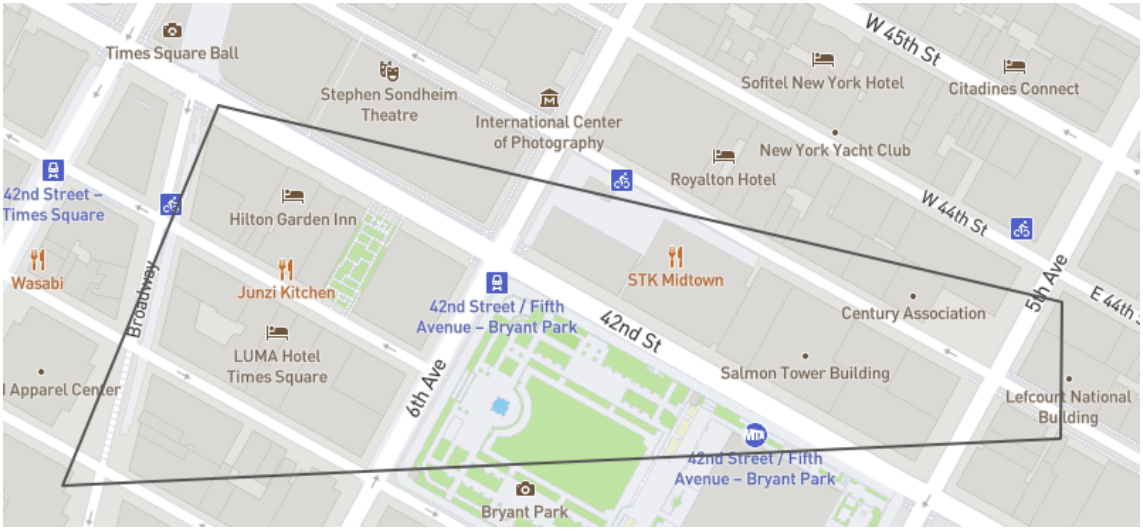
Calculate a New Linestring

Now that you know that there is a better option for the liquor store, substitute the above liquor store query into the original linestring query to produce a different object. For visualization sake, the order of the statements in the unions have been changed, which affects the order of the points in the object.

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates, st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
```

```
order by 2 limit 1))
select st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)')) as linestring from locations;
```

Copy the result cell from the above query and paste it into geojson.io. You should get this:



Much better! This looks like a more efficient shopping path. Check the new distance by running this query:

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates, st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_length(st_makeline(st_collect(coordinates),
```

```
to_geography('POINT(-73.986226 40.755702)')))
as length_meters from locations;
```

Nice! 1537 meters, which is a savings of about 583 meters, or a third of a mile. By joining the two shop views together, you were able to find an object in one table that is closest to an object from another table to optimize your route. Now that you have a more optimized route, can you stop at any other shops along the way? Advance to the next section to find out.

Additional Calculations and Constructors

The `LINESTRING` object that was created in the previous section looks like a nice, clean, four-sided polygon. As it turns out, a `POLYGON` is another geospatial object type that you can construct and work with. Where you can think of a `LINESTRING` as a border of a shape, a `POLYGON` is the filled version of the shape itself. The key thing about a `POLYGON` is that it must end at its beginning, where a `LINESTRING` does not need to return to the starting point.

Positive : Remember in a previous section when you added your Times Square Apartment location to both the beginning and the end of the `LINESTRING`? In addition to the logical explanation of returning home after your shopping trip, that point was duplicated at the beginning and end so you can construct a `POLYGON` in this section!

Construct a Polygon

Constructing a `POLYGON` is done with the `ST_MAKEPOLYGON` function, just like the `ST_MAKELINE`. The only difference is where `ST_MAKELINE` makes a line out of points, `ST_MAKEPOLYGON` makes a polygon out of lines. Therefore, the only thing you need to do to the previous query that constructed the line is to wrap that construction with `ST_MAKEPOLYGON` like this:

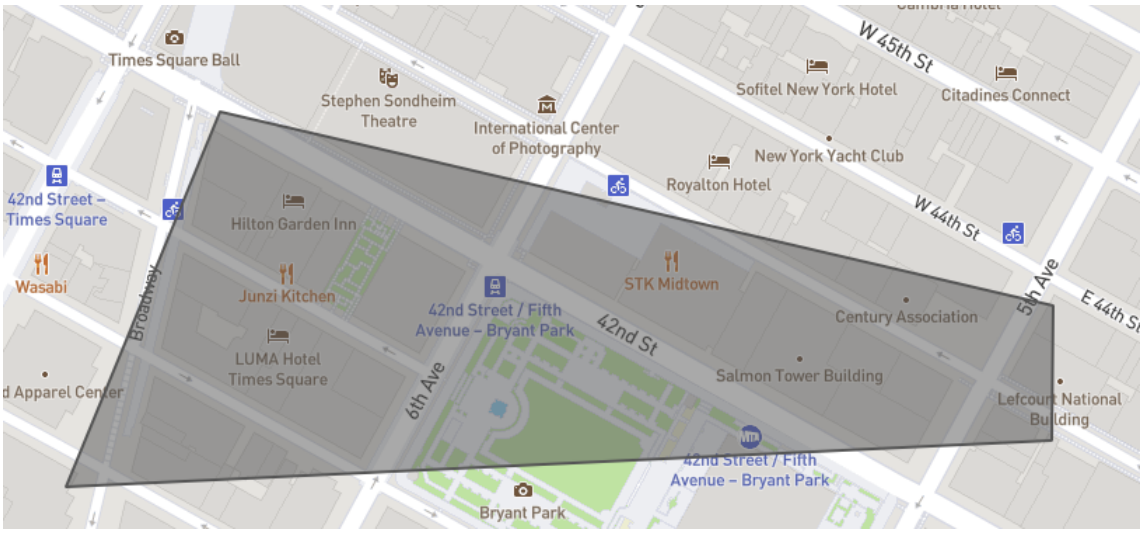
```
select st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)')))
```

This really helps illustrate the construction progression: from individual points, to a collection of points, to a line, to a polygon. Run this query to create your polygon:

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates, st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
```

```
as distance_meters
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)'))) as polygon from locations;
```

Copy the result cell from the above query and paste it into geosjon.io. You should get this:



And just like before where you could calculate the distance of a `LINESTRING` using `ST_DISTANCE`, you can calculate the perimeter of a `POLYGON` using `ST_PERIMETER`, which you wrap around the polygon construction in the same way you wrapped around the line construction. Run this query to calculate the perimeter:

```
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates, st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
```

```
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_perimeter(st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)')))) as perimeter_meters from locations;
```

Nice! That query returned the same 1537 meters you got before as the distance of the `LINESTRING`, which makes sense, because the perimeter of a `POLYGON` is the same distance as a `LINESTRING` that constructs a `POLYGON`.

Find Shops Inside The Polygon

The final activity you will do in this guide is to find any type of shop within the `v_osm_ny_shop` view that exists inside of the `POLYGON` you just created in the previous step. This will reveal to you all of the stores you can stop at along your path to your core stops. To accomplish this, here is what you will do to the query that builds the `POLYGON`:

- The `POLYGON` is a result set in its own right, so you are going to wrap this query in another CTE. This will allow you to refer back to the polygon as a singular entity more cleanly in a join. You will call this CTE the `search_area`.
- Then you will join the `v_osm_ny_shop` to the `search area` CTE using the `ST_WITHIN` function, which is different than `ST_DWITHIN`. The `ST_WITHIN` function takes one geospatial object and determines if it is completely inside another geospatial object, returning `true` if it is and `false` if it isn't. In the query, it will determine if any row in `v_osm_ny_shop` is completely inside the `search_area` CTE.

Run this query to see what shops are inside the polygon:

```
// Define the outer CTE 'search_area'
with search_area as (
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates,
st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
```

```
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)'))) as polygon from locations)
select sh.id,sh.coordinates,sh.name,sh.shop,sh.addr_housenumber,sh.addr_street
from v_osm_ny_shop sh
// Join v_osm_ny_shop to the 'search_area' CTE using st_within
join search_area sa on st_within(sh.coordinates,sa.polygon);
```

You should see similar results as below, though the number of rows may be different (WKT output is shown below for readability):

	ID	COORDINATES	NAME	SHOP	ADDR_HOUSENUMBER	ADDR_STREET
1	2,709,702,328	POINT(-73.9863047 40.7554639)	JD Sports	sports	1466	Broadway
2	4,665,840,837	POINT(-73.9845292 40.7546279)	Whole Foods Market	supermarket	1095	Avenue of the Americas
3	5,859,226,170	POINT(-73.9814677 40.7544852)	Allen Edmonds	shoes	20	West 43rd Street
4	6,288,552,743	POINT(-73.9836957 40.7548753)	Sprint	mobile_phone		
5	4,553,345,925	POINT(-73.9804303 40.7544315)	Sayki	clothes	340	Madison Avenue
6	1,428,036,380	POINT(-73.9800487 40.7542617)	Urban Outfitters	clothes		
7	5,982,230,986	POINT(-73.9816908 40.754007)	Mets Clubhouse Shop	clothes		
8	6,067,214,134	POINT(-73.9861387 40.7541518)	MT News	convenience	1440	Broadway
9	5,715,737,678	POINT(-73.9868232 40.7537775)	Ricky's NYC	cosmetics	1412	Broadway
10	2,501,550,492	POINT(-73.9849066 40.7540609)	Kinokuniya	books	1073	6th Avenue
11	4,915,842,122	POINT(-73.9855487 40.7552586)	Modell's Sporting Goods	sports	136	West 42nd Street
12	4,623,422,789	POINT(-73.9811804 40.7539199)	The North Face	clothes		
13	6,308,405,188	POINT(-73.9863365 40.7551246)	Foot Locker	shoes		

And your final step will be to construct a single geospatial object that includes both the `POLYGON` you created as well as a `POINT` for every shop inside the `POLYGON`. This single object is known as a `GEOMETRYCOLLECTION`, which a geospatial type that can hold any combination of geospatial objects as one grouping. To create this object, you will do the following:

- Create a CTE that unions the `POLYGON` query with the above query that finds shops inside the polygon, keeping only the necessary `coordinates` column in the latter query for simplicity. This CTE will produce 1 row for the `POLYGON` and rows for each individual shop `POINT` inside the `POLYGON`.
- Use `ST_COLLECT` to aggregate the rows above (1 `POLYGON`, all the `POINTS`) into a single `GEOMETRYCOLLECTION`.

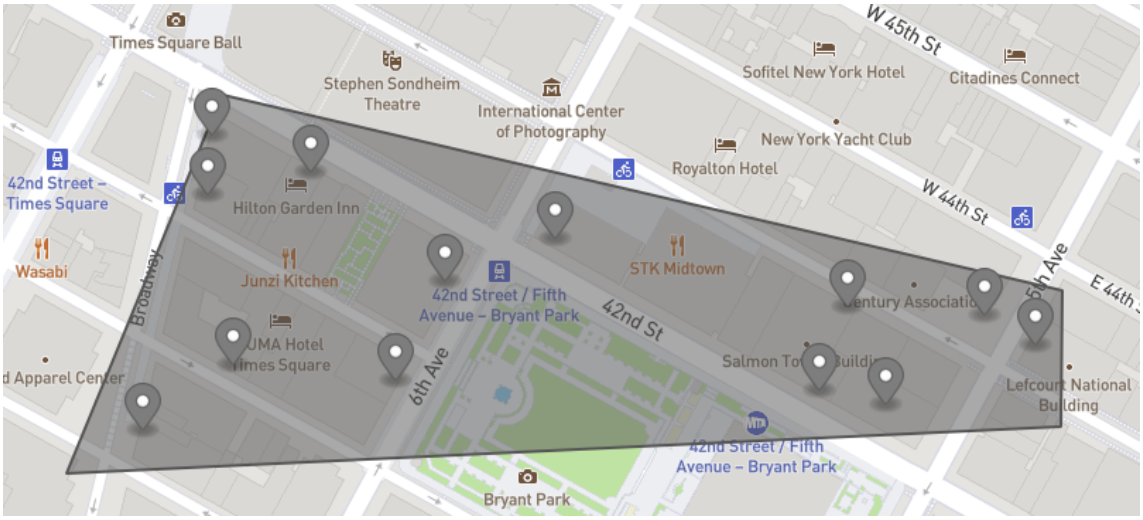
Run the query below:

```
// Define the outer CTE 'final_plot'
with final_plot as (
// Get the original polygon
(with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
```

```
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates,
st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)'))) as polygon from locations)
union all
// Find the shops inside the polygon
(with search_area as (
with locations as (
(select to_geography('POINT(-73.986226 40.755702)') as coordinates,
0 as distance_meters)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_food_beverages
where shop = 'coffee' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1)
union all
(select fb.coordinates,
st_distance(e.coordinates,fb.coordinates) as distance_meters
from v_osm_ny_shop_electronics e
join v_osm_ny_shop_food_beverages fb on st_dwithin(e.coordinates,fb.coordinates,1600)
where e.id = 1428036403 and fb.shop = 'alcohol'
order by 2 limit 1)
union all
(select coordinates,
st_distance(coordinates,to_geography('POINT(-73.986226 40.755702)'))::number(6,2)
as distance_meters
from v_osm_ny_shop_electronics
where name = 'Best Buy' and
st_dwithin(coordinates,st_makepoint(-73.986226, 40.755702),1600) = true
order by 2 limit 1))
select st_makepolygon(st_makeline(st_collect(coordinates),
to_geography('POINT(-73.986226 40.755702)'))) as polygon from locations)
select sh.coordinates
from v_osm_ny_shop sh
```

```
join search_area sa on st_within(sh.coordinates,sa.polygon)))
// Collect the polygon and shop points into a geometrycollection
select st_collect(polygon) from final_plot;
```

Copy the result cell from the above query and paste it into geojson.io. You should get something similar to this (your image may have more/less points):



Positive : You may feel that these last few queries were a bit long and repetitive, but remember that the intention of this guide was to walk you through the progression of building these longer, more complicated queries by illustrating to you what happens at each step through the progression. By understanding how functions can be combined, it helps you to understand how you can do more advanced things with Snowflake geospatial features!

Conclusion

In this guide, you acquired geospatial data from the Snowflake Marketplace, explored how the `GEOGRAPHY` data type and its associated formats work, created data files with geospatial data in it, loaded those files into new tables with `GEOGRAPHY` typed columns, and queried geospatial data using parser, constructor, transformation and calculation functions on single tables and multiple tables with joins. You then saw how newly constructed geospatial objects could be visualized in tools like geojson.io or WKT Playground.

You are now ready to explore the larger world of Snowflake [geospatial support] and [geospatial functions].

What we've covered

- How to acquire a shared database from the Snowflake Marketplace
- The `GEOGRAPHY` data type, its formats `GeoJSON`, `WKT`, `EWKT`, `WKB`, and `EWKB`, and how to switch between them.
- How to unload and load data files with geospatial data.
- How to use parsers like `ST_X` and `ST_Y`.
- How to use constructors like `TO_GEOGRAPHY`, `ST_MAKEPOINT`, `ST_MAKELINE`, and `ST_MAKEPOLYGON`.
- How to use a transformation like `ST_COLLECT`.
- How to perform measurement calculations like `ST_DISTANCE`, `ST_LENGTH`, and `ST_PERIMETER`.
- How to perform relational calculations like `ST_DWITHIN` and `ST_WITHIN`.

Understanding Your Snowflake Utilization, Part 1: Warehouse Profiling

This series will take a deeper dive into the Information Schema (Snowflake's data dictionary) and show you some practical ways to use this data to better understand your usage of Snowflake.

In this lab, we will discuss how they can implement profiling for your Snowflake account on your own.

The answer is to utilize the Information Schema. Aside from providing a set of detailed views into Snowflake's metadata store, the Information Schema goes a step further and provides several powerful table functions that can be called directly in SQL. These functions can be used to return historical information about executed queries, data stored in databases and stages, and virtual warehouse (i.e. compute) usage.

In addition to these functions, I also recommend leveraging the recently implemented TABLE_STORAGE_METRICS view (also in the Information Schema) to dive even deeper into your analysis.

In this lab, I will show you how to leverage these easy-to-use function to gather detailed information about the usage of your virtual warehouses. So let's get started.

Warehouse Profiling

To profile your current warehouse usage, use the [WAREHOUSE_LOAD_HISTORY] and [WAREHOUSE_METERING_HISTORY] functions. A good way to think about the relationship between these two functions is that the first one shows how much work was done over a period of time (**load**) and the second one shows the cost for doing the work (**metering**).

The syntax for calling these functions is simple, and can be executed in the **Worksheet** in the Snowflake web interface. For example:

```
use warehouse mywarehouse;

select * from
table(information_schema.warehouse_load_history(date_range_start=>dateadd('hour',-1,curr



select * from
table(information_schema.warehouse_metering_history(dateadd('hour',-1,current_date()),cu
```

The above queries show warehouse load and credits used for the past hour for all your warehouses. Be sure to check out the *Usage Notes* section (in the documentation) for each function to understand all the requirements and rules. For example, the WAREHOUSE_LOAD_HISTORY function returns results in different intervals based on the timeframe you specify:

- 5-second intervals when the timeframe is less than 7 hours.
- 5-minute intervals when the timeframe is greater than 7 hours.

Here's an example of the output from the WAREHOUSE_LOAD_HISTORY query against SNOWHOUSE, a warehouse that we use internally:

row#	START_TIME	END_TIME	WAREHOUSE_NAME	AVG_RUNNING	AVG_QUEUED_LOAD	AVG_QUEUED_PROVISIONING	AVG_BLOCKED
1	2017-03-20 06:33:20.000 +0000	2017-03-20 06:33:25.000 +0000	SNOWHOUSE	0.02	0.00	0.00	0.00
2	2017-03-20 06:33:25.000 +0000	2017-03-20 06:33:30.000 +0000	SNOWHOUSE	1.12	0.00	0.00	0.00
3	2017-03-20 06:33:30.000 +0000	2017-03-20 06:33:35.000 +0000	SNOWHOUSE	2.02	0.00	0.00	0.00
4	2017-03-20 06:33:35.000 +0000	2017-03-20 06:33:40.000 +0000	SNOWHOUSE	4.29	0.00	0.00	0.00
5	2017-03-20 06:33:40.000 +0000	2017-03-20 06:33:45.000 +0000	SNOWHOUSE	4.54	0.00	0.00	0.00
6	2017-03-20 06:33:45.000 +0000	2017-03-20 06:33:50.000 +0000	SNOWHOUSE	4.97	0.00	0.00	0.00

Per our documentation:

- AVG_RUNNING -- Average number of queries executed.
- AVG_QUEUE_LOAD -- Average number of queries queued because the warehouse was overloaded.
- AVG_QUEUE_PROVISION -- Average number of queries queued because the warehouse was being provisioned.
- AVG_BLOCKED -- Average number of queries blocked by a transaction lock.

And here's an example of the output from the WAREHOUSE_METERING_HISTORY query against SNOWHOUSE:

ro...	START_TIME	END_TIME	WAREHOUSE_NAME	CREDITS_USED
1	Thu, 16 Mar 2017 10:00:00 -0700	Thu, 16 Mar 2017 11:00:00 -0700	ACCT	19.00
2	Fri, 10 Mar 2017 00:00:00 -0800	Fri, 10 Mar 2017 01:00:00 -0800	TESTMCWH	1.00
3	Fri, 10 Mar 2017 01:00:00 -0800	Fri, 10 Mar 2017 02:00:00 -0800	TESTMCWH	1.00
4	Fri, 10 Mar 2017 02:00:00 -0800	Fri, 10 Mar 2017 03:00:00 -0800	TESTMCWH	1.00
5	Fri, 10 Mar 2017 03:00:00 -0800	Fri, 10 Mar 2017 04:00:00 -0800	TESTMCWH	1.00
6	Fri, 10 Mar 2017 04:00:00 -0800	Fri, 10 Mar 2017 05:00:00 -0800	TESTMCWH	1.00
7	Fri, 10 Mar 2017 05:00:00 -0800	Fri, 10 Mar 2017 06:00:00 -0800	TESTMCWH	1.00
8	Fri, 10 Mar 2017 06:00:00 -0800	Fri, 10 Mar 2017 07:00:00 -0800	TESTMCWH	1.00
9	Fri, 10 Mar 2017 07:00:00 -0800	Fri, 10 Mar 2017 08:00:00 -0800	TESTMCWH	1.00
10	Fri, 10 Mar 2017 08:00:00 -0800	Fri, 10 Mar 2017 09:00:00 -0800	TESTMCWH	1.00

Now that we know the amount of work that was performed during the time period (via WAREHOUSE_LOAD_HISTORY) and the cost per time period (via WAREHOUSE_METERING_HISTORY), we can perform a simple efficiency ratio calculation for a particular warehouse. This example returns this information for a warehouse named XSMALL:

```
with cte as (
  select date_trunc('hour', start_time) as start_time, end_time, warehouse_name,
credits_used
  from
table(information_schema.warehouse_metering_history(dateadd('days',-1,current_date()),cu

  where warehouse_name = 'XSMALL')
select date_trunc('hour', a.start_time) as start_time, avg(AVG_RUNNING),
avg(credits_used), avg(AVG_RUNNING) / avg(credits_used) * 100
from
table(information_schema.warehouse_load_history(dateadd('days',-1,current_date()),curren
 a
join cte b on a.start_time = date_trunc('hour', a.start_time)
where a.warehouse_name = 'XSMALL'
```

```
group by 1
order by 1;
```

In the above query, we are treating the average of AVG_RUNNING as work and the average of CREDITS_USED as cost and we apply a simple efficiency ratio on both of these values. Feel free to experiment any way you like.

row#	START_TIME	AVG(AVG_RUNNING)	AVG(CREDITS_USED)	AVG(AVG_RUNNING) / AVG(CREDITS_USED) * 100
1	Mon, 20 Mar 2017 00:00:00 -0700	0.05000	1.00000	5.00000000
2	Mon, 20 Mar 2017 01:00:00 -0700	0.05000	1.00000	5.00000000
3	Mon, 20 Mar 2017 02:00:00 -0700	0.05000	1.00000	5.00000000
4	Mon, 20 Mar 2017 03:00:00 -0700	0.05000	1.00000	5.00000000
5	Mon, 20 Mar 2017 04:00:00 -0700	0.06000	1.00000	6.00000000
6	Mon, 20 Mar 2017 05:00:00 -0700	0.06000	1.00000	6.00000000
7	Mon, 20 Mar 2017 06:00:00 -0700	0.04000	1.00000	4.00000000
8	Mon, 20 Mar 2017 07:00:00 -0700	0.05000	1.00000	5.00000000
9	Mon, 20 Mar 2017 08:00:00 -0700	0.01000	1.00000	1.00000000
10	Mon, 20 Mar 2017 09:00:00 -0700	0.00000	1.00000	0.00000000
11	Mon, 20 Mar 2017 10:00:00 -0700	0.04000	1.00000	4.00000000
12	Mon, 20 Mar 2017 11:00:00 -0700	0.05000	1.00000	5.00000000

Next, let's talk about the specific use of WAREHOUSE_LOAD_HISTORY in our example above:

```
select date_trunc('hour', start_time), hour(start_time), avg(avg_running)
from
table(information_schema.warehouse_load_history(date_range_start=>dateadd('day',-1,curre

group by date_trunc('hour', start_time), hour(start_time)
order by date_trunc('hour', start_time) asc;
```

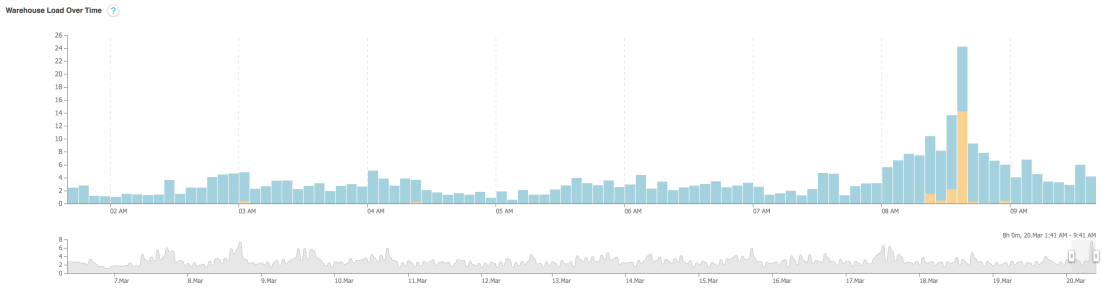
Here is the output:

row#	DATE_TRUNC('HOUR', START_TIME)	HOUR(START_TIME)	AVG(AVG_RUNNING)
1	2017-03-19 16:00:00.000 +0000	16	2.40800
2	2017-03-19 17:00:00.000 +0000	17	1.58916
3	2017-03-19 18:00:00.000 +0000	18	2.99333
4	2017-03-19 19:00:00.000 +0000	19	2.08916
5	2017-03-19 20:00:00.000 +0000	20	2.17500
6	2017-03-19 21:00:00.000 +0000	21	2.88750
7	2017-03-19 22:00:00.000 +0000	22	1.68500
8	2017-03-19 23:00:00.000 +0000	23	3.55333
9	2017-03-20 00:00:00.000 +0000	0	3.80916
10	2017-03-20 01:00:00.000 +0000	1	3.39333
11	2017-03-20 02:00:00.000 +0000	2	2.82583
12	2017-03-20 03:00:00.000 +0000	3	1.98083
13	2017-03-20 04:00:00.000 +0000	4	2.43000
14	2017-03-20 05:00:00.000 +0000	5	2.81833
15	2017-03-20 06:00:00.000 +0000	6	1.68916
16	2017-03-20 07:00:00.000 +0000	7	3.51833

In this case, I'm indeed asking for an average of an average. I'm grouping the values by hours so I can get a general overview of my warehouse workload. I can see my warehouse is working almost a full day. However, if I see some time gaps in this output, then I might do some additional investigation around those times and see if the warehouse should be doing work.

Another thing you can see in the output from this function is whether these time gaps repeat over a few days. If they do, then I would recommend that you script the warehouse to sleep when not in use (i.e. to save money), or enable AUTO_SUSPEND and AUTO_RESUME for that warehouse.

The Snowflake web interface also has a nice visual representation of this function (under the **Warehouse** tab):



Whether you use the visual chart or the manual query, for the four available metrics, pay particular attention to AVG_RUNNING. This should give you an idea how each warehouse performs. If you have split your workload across several different warehouses, it should tell you how well your queries are distributed.

AVG_QUEUE_LOAD and AVG_BLOCKED are also interesting and should provide you with good insight about how well your warehouses are sized. Keep in mind that queuing is not necessarily a bad thing and you shouldn't expect zero queuing. The idea is to accept a certain amount of queuing per time period based on your usage requirements.

Using these metrics, you can determine what to do:

- Increasing the warehouse size will provide more throughput in processing the queries and thereby can help reduce the queuing time.
- Increasing the cluster count (if using a multi-cluster warehouse) will allow more concurrency, which should also help reduce queuing and blocking.

Finding an Underutilized Warehouse

Is there a warehouse that's underutilized? For example, any similar sized warehouses being shared across several users could potentially be consolidated to a single warehouse. You can surface this information by comparing your AVG_RUNNING and AVG_QUEUE_LOAD scores across your warehouses:

- If you see a warehouse with a very low number of queries running, you may want to turn that warehouse off and redirect the queries to another less used warehouse.
- If a warehouse is running queries and queuing, perhaps it's time to review your workflow to increase your warehouse sizes.
- If you have built your own client application to interface with Snowflake, reviewing your client scripts / application code should also reveal any biases towards one warehouse over another.

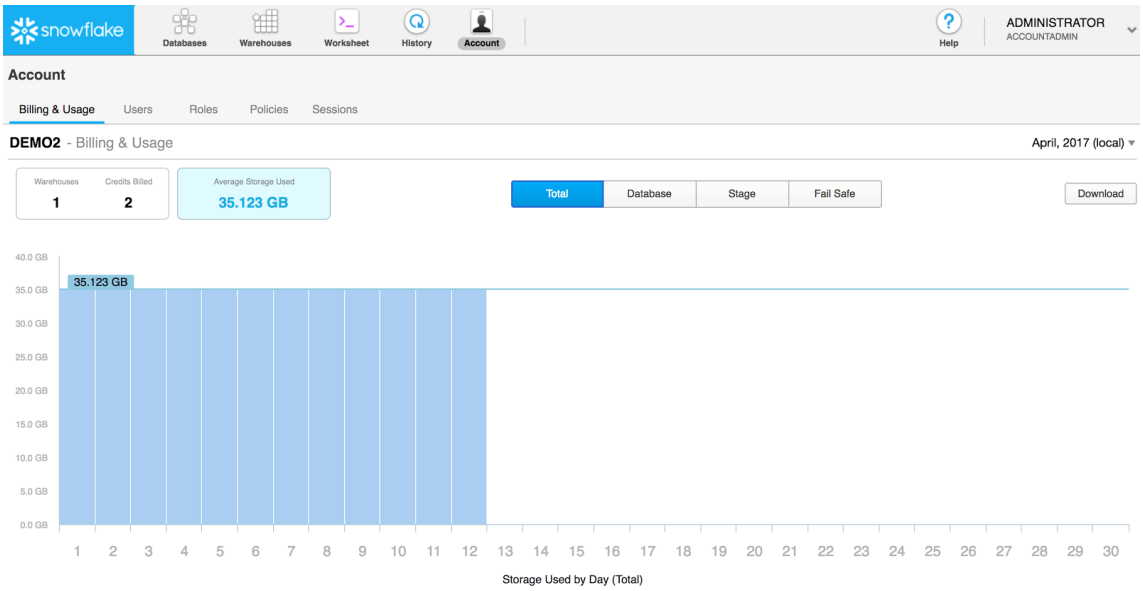
Understanding Your Snowflake Utilization, Part 2: Storage Profiling

In this lab, I provide a deep-dive into understanding how you are utilizing data storage in Snowflake at the database, stage, and table level. To do this, I will show you examples of two functions and a view provided in the Information Schema for monitoring storage usage. I will also show you a handy page in the UI that provides an account-level view of your storage. Keep in mind that you need ACCOUNTADMIN access to perform any of the tasks described in this lab.

Let's get started.

Summary Storage Profiling in the UI

Before diving into our detailed analysis of data storage, let's take a quick look at the summary, account-level storage view provided by Snowflake. As a user with the ACCOUNTADMIN role, you can navigate to the **Account** page in the Snowflake UI to get a visual overview of the data storage for your account.



This page provides a view, by month, of the average and daily storage usage across your entire account. You can use the filters on the page to filter by database, Snowflake stage, and data maintained in Fail-safe (for disaster recovery).

Detailed Storage Profiling Using the Information Schema

The Snowflake Information Schema provides two functions and one view for monitoring detailed data storage at the database, stage, and table level:

- DATABASE_STORAGE_USAGE_HISTORY (function)
- STAGE_STORAGE_USAGE_HISTORY (function)
- TABLE_STORAGE_METRICS (view)

The DATABASE_STORAGE_USAGE_HISTORY table function shows your database status and usage for all databases in your account or a specified database. Here's an example of the usage over the last 10 days for a database named `sales`:

```
use warehouse mywarehouse;

select * from
table(sales.information_schema.database_storage_usage_history(dateadd('days',-10,current
 'SALES'));
```

row#	USAGE_DATE	DATABASE_NAME	AVERAGE_DATABASE_BYTES	AVERAGE_FAILSAFE_BYTES
1	2017-03-31	SALES	5856246781	0
2	2017-04-01	SALES	5856246782	0
3	2017-04-02	SALES	5856246781	0
4	2017-04-03	SALES	5856246782	0
5	2017-04-04	SALES	5856246782	0
6	2017-04-05	SALES	5856246781	0
7	2017-04-06	SALES	5856246782	0
8	2017-04-07	SALES	5856246781	0
9	2017-04-08	SALES	5856246782	0
10	2017-04-09	SALES	5856246781	0
11	2017-04-10	SALES	5856246782	0

Note that the above screenshot only displays some of the output columns. Also, per the Snowflake documentation:

> *If a database has been dropped and its data retention period has passed (i.e. database cannot be recovered using Time Travel), then the database name is reported as DROPPED_id.*

At its core, the most useful insight from this function is the average growth in your database. Keep in mind, the output includes both AVERAGE_DATABASE_BYTES and AVERAGE_FAILSAFE_BYTES. Leveraging these data points to derive a percentage of `Fail-safe` over actual database size should give you an idea of how much you should be investing towards your Fail-safe storage. If certain data is not mission critical and doesn't require Fail-safe, try setting these tables to `transient`. More granular information about Fail-safe data is provided in TABLE_STORAGE_METRICS, which we will look at more closely later in this lab.

Next, let's look at STAGE_STORAGE_USAGE_HSTORY. This function shows you how much storage is being used for staged files across ***all*** your Snowflake staging locations, including named, internal *stages*. Note that this function does not allow querying storage on individual stages.

Here's an example of staged file usage for the last 10 days (using the same database, `sales`, from the previous example):

```
select * from
table(sales.information_schema.stage_storage_usage_history(dateadd('days',-10,current_da
```

row#	USAGE_DATE	AVERAGE_STAGE_BYTES
1	2017-03-31	448284316
2	2017-04-01	448284317
3	2017-04-02	448284316
4	2017-04-03	448284316
5	2017-04-04	448284317
6	2017-04-05	448284316
7	2017-04-06	448284317
8	2017-04-07	448284317
9	2017-04-08	448284316
10	2017-04-09	448284317
11	2017-04-10	448284318

Note that the above screenshot only displays some of the output columns.

Also note that you can only query up to 6 months worth of data using this function. Some of our users like to use Snowflake stages to store their raw data. For example, one user leverages table staging locations for their raw data storage just in case they need to access the data in the future. There's nothing wrong with this approach, and since Snowflake compresses your staged data files, it certainly makes sense; however, only the last 6 months of staged data storage is available.

Finally, the TABLE_STORAGE_METRICS view shows your table-level storage at runtime. This is a snapshot of your table storage which includes your active and Fail-safe storage. Additionally, you can derive cloned storage as well utilizing the CLONE_GROUP_ID column. As of today, this is the most granular level of storage detail available to users.

Here's a general use example (using the `sales` database):

```
select *
from sales.information_schema.table_storage_metrics
where table_catalog = 'SALES';
```

row#	TABLE_CAT...	TABLE_SC...	TABLE_NA...	ID	CLONE_GR...	IS_TRANSI...	ACTIVE_RO..	ACTIVE_BY...	TIME_TRAV...	FAILSAFE_...	OWNED_A...	TABLE_CR...	TABLE_DR...	TABLE_ENT..	CATALOG_...	CATALOG_...
1	SALES	PUBLIC	NATION	69646	69646	NO	0	0	0	0	0	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
2	SALES	PUBLIC	CUSTOMER	69648	69648	NO	15050000	1089571840	0	0	1089571840	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
3	SALES	PUBLIC	SUPPLIER	69650	69650	NO	0	0	0	0	0	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
4	SALES	PUBLIC	PRODUCT	69652	69652	NO	1	3584	0	0	3584	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
5	SALES	PUBLIC	PRODUCTH...	70669	70669	NO	1	1024	0	0	1024	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
6	SALES	PUBLIC	ORDERS	70671	70671	NO	150000000	4766642688	0	0	4766642688	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
7	SALES	PUBLIC	LINEITEM	70673	70673	NO	5302	26624	0	0	26624	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
8	SALES	PUBLIC	_after_reset	70675	70675	NO	1	1024	0	0	1024	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
9	SALES	PUBLIC	REGION	71688	71688	NO	0	0	0	0	0	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL
10	SALES	PUBLIC	PRODUCTS...	71690	71690	NO	0	0	0	0	0	2016-10-06 1...	NULL	NULL	2016-10-06 1...	NULL

Note that the above screenshot only shows a portion of the output columns.

In Snowflake, cloning data has no additional costs (until the data is modified or deleted) and it's done very quickly. All users benefit from "zero-copy cloning", but some are curious to know exactly what percentage of their table storage actually came from cloned data. To determine this, we'll leverage the CLONE_GROUP_ID column in TABLE_STORAGE_METRICS.

For example (using a database named `concurrency_wh`):

```
with storage_sum as (
  select clone_group_id,
         sum(owned_active_and_time_travel_bytes) as owned_bytes,
```

```
        sum(active_bytes) + sum(time_travel_bytes) as referred_bytes
  from concurrency_wh.information_schema.table_storage_metrics
  where active_bytes > 0
  group by 1)
select * , referred_bytes / owned_bytes as ratio
from storage_sum
where referred_bytes > 0 and ratio > 1
order by owned_bytes desc;
```

row#	CLONE_GROUP_ID	OWNED_BYTES	REFERRED_BYTES	RATIO
1	12292	41606396613632	49438168488960	1.188234803
2	1146387	6983569222656	10792538451456	1.545418697
3	186519	4700058698240	9124382190592	1.941333668
4	186542	1377472353280	2699929145344	1.960060497
5	184490	1077942566912	2155885121536	1.999999989
6	149508	465707046400	820531732480	1.761905341
7	186536	213844349440	328041306112	1.534019052
8	184454	107014242304	205778106368	1.922903923
9	746268	51754352128	121072992768	2.339378
10	15366	48444371456	82108175872	1.694896092
11	184452	31855170560	57569825792	1.807236464
12	349800	30342430720	91027292160	3
13	75778	20327801856	36357701120	1.788570224
14	163232	15405162496	15405174784	1.000000798
15	164389	14270820352	21313835008	1.493525564
16	349844	11844087808	35532263424	3

The ratio in the above query gives you an idea of how much of the original data is being "referred to" by the clone. In general, when you make a clone of a table, the CLONE_GROUP_ID for the original table is assigned to the new, cloned table. As you perform DML on the new table, your REFERRED_BYTES value gets updated. If you join the CLONE_GROUP_ID back into the original view, you get the output of the original table along with the cloned table. A ratio of 1 in the above example means the table data is not cloned.

If you need to find out the exact table name from the above query, then simply join the CTE back to the TABLE_STORAGE_METRICS view and ask for the TABLE_NAME column.

For example (using the same database, `concurrency_wh`, from the previous example):

```
with storage_sum as (
  select clone_group_id,
         sum(owned_active_and_time_travel_bytes) as owned_bytes,
         sum(active_bytes) + sum(time_travel_bytes) as referred_bytes
  from concurrency_wh.information_schema.table_storage_metrics
  where active_bytes > 0
  group by 1)
select b.table_name, a.* , referred_bytes / owned_bytes as ratio
from storage_sum a
join concurrency_wh.information_schema.table_storage_metrics b
on a.clone_group_id = b.clone_group_id
where referred_bytes > 0 and ratio > 1
order by owned_bytes desc;
```

row#	TABLE_NAME	CLONE_GROUP_ID	OWNED_BYTES	REFERRED_BYTES	RATIO
1	1cf01dc61ac4925e6326943f6c2faf73cc85b3f9	12292	43295438332928	51127210208256	1.180891387
2	1cf01dc61ac4925e6326943f6c2faf73cc85b3f9	12292	43295438332928	51127210208256	1.180891387
3	1cf01dc61ac4925e6326943f6c2faf73cc85b3f9	12292	43295438332928	51127210208256	1.180891387
4	1cf01dc61ac4925e6326943f6c2faf73cc85b3f9	12292	43295438332928	51127210208256	1.180891387
5	4d4971b5c2ca8eee185377de5378e7a8d4c44f05	186519	4756115311104	9180438803456	1.930238904
6	4d4971b5c2ca8eee185377de5378e7a8d4c44f05	186519	4756115311104	9180438803456	1.930238904
7	9398b7b93fba9ae62c3ae0b16a75a2b1b67dac09	1146387	4202746140672	8011715369472	1.906304854
8	5d4cb56f00f9ed3d6d30b223193eb39586b02b5f	1146387	4202746140672	8011715369472	1.906304854
9	45d9ebb3a323f9d945a1cfd737972e8faa02e63c	1146387	4202746140672	8011715369472	1.906304854
10	81f29e546c5bfbdfa1b95ed4c529fc6dc0024f4b	1146387	4202746140672	8011715369472	1.906304854
11	dd8d495f2ba9c65728c7904f85193e88dec5b225	186542	1389598218752	2712055010816	1.951682849
12	dd8d495f2ba9c65728c7904f85193e88dec5b225	186542	1389598218752	2712055010816	1.951682849
13	dd8d495f2ba9c65728c7904f85193e88dec5b225	186542	1389598218752	2712055010816	1.951682849
14	dd8d495f2ba9c65728c7904f85193e88dec5b225	186542	1389598218752	2712055010816	1.951682849
15	512d1121d7f781cb718336db4cb270d4f98a45bf	184490	1077942566912	2155885121536	1.999999989
16	765ec9e6af322a586f906c985902fb2d7558db4a	184490	1077942566912	2155885121536	1.999999989
17	712461a72b4df482a42b89dd765fb5a9fd6d02f9	149508	468484634112	823309320192	1.757388098
18	712461a72b4df482a42b89dd765fb5a9fd6d02f9	149508	468484634112	823309320192	1.757388098
19	a144cc44ed2bb4e14c6736caf5fb195f3b82ce39	186536	217674690560	331871647232	1.524622116

Conclusion

By utilizing the UI and the Information Schema functions and views described in this lab, you can profile your data storage to help you keep your storage costs under control and understand how your business is growing over time. It's a good idea to take regular snapshots of your storage so that you can analyze your growth month-over-month. This will help you both formulate usage insight and take actions.

I hope this lab has given you some good ideas for how to manage your Snowflake instance. Look for Part 3 of this series in coming weeks where I will show you how to analyze your query performance. As already shown in Parts 1 and 2, there are a lot of options to play with in Snowflake and they're all intended to give you the flexibility and control you need to best use Snowflake. Please share your thoughts with us!

Understanding Your Snowflake Utilization: Part 3 -- Query Profiling

This lab about query profiling is the third in a three-part series to help you utilize the functionality and data in Snowflake's Information Schema to better understand and effectively Snowflake.

In this lab, I will deep-dive into understanding query profiling. To do this, I will show you examples using the QUERY_HISTORY family of functions. I will also show you a handy page in the UI that provides a graphical view of each query. Keep in mind that you'll need warehouse MONITOR privileges to perform the tasks described in this lab. Typically, the SYSADMIN role has the necessary warehouse MONITOR privileges across your entire account; however, other lower-level roles may also have the necessary privileges.

Ready to get started? Here we go!

Query History Profiling

Query profiling is perhaps one of the more popular topics. Many people are interested in improving their query performance. Although every development team should strive to periodically refactor their code, many find it challenging to determine where to start. Going through this analysis should help with identifying a good starting point.

Let's look at some syntax, per our documentation for [QUERY_HISTORY]:

```
select *
from table(information_schema.query_history(dateadd('hours',-1,
current_timestamp()),current_timestamp()))
order by start_time;
```

This query provides a view into all of the queries run by the current user in the past hour:

row#	QUERY_ID	QUERY_TE...	DATABASE...	SCHEMA_N...	QUERY_TY...	SESSION_ID	USER_NAME	ROLE_NAME	WAREHOU...	WAREHOU...	WAREHOU...	QU
1	29ea94fb-303...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
2	eeb9e84a-83...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
3	ab162ebf-8d1...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
4	b3c188ff-1cc...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
5	49077d6c-58...	INSERT INTO...	SALES	PUBLIC	INSERT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
6	9765e6aa-8c...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
7	8e62d4c4-8c...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
8	826c47b9-d7...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
9	37c75df7-d23...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594752...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
10	e2aaaf91-bdd...	INSERT INTO...	SALES	PUBLIC	INSERT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
11	91faa8b9-611...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
12	beabf130-208...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	

We can also leverage the QUERY_HISTORY companion functions to narrow down your focus:

- QUERY_HISTORY_BY_SESSION
- QUERY_HISTORY_BY_USER
- QUERY_HISTORY_BY_WAREHOUSE

These are particularly useful if you have identified specific workflow issues you need to address.

Profiling Tip #1: Using HASH()

Now for a particularly useful tip: utilizing HASH on the QUERY_TEXT column can help you consolidate and group on similar queries (the HASH function will return the same result if any queries are exactly the same). As a general rule, identifying query groups and finding the max and min query runtime should help you sort through specific workflows. In the example below, I'm doing an analysis on average compile and execution time. Additionally, I'm collecting a count of the queries with the same syntax:

```
select hash(query_text), query_text, count(*), avg(compilation_time),
avg(execution_time)
from
table(information_schema.query_history(dateadd('hours',-1,current_timestamp()),current_t

group by hash(query_text), query_text
order by count(*) desc;
```

Output:

row#	HASH(QUERY_TEXT)	QUERY_TEXT	COUNT(*)	AVG(COMPILATION_TIME)	AVG(EXECUTION_TIME)
1	-102792116783286838	SELECT reg_key, looker, created_at, expires_at FR...	36	41.083	3.638
2	-5565463363353937792	SELECT 1 FROM looker_scratch.connection_reg_r...	12	20.000	5.083
3	-8443351143317024671	select hash(query_text), query_text, avg(compilatio...	4	52.000	1940.500
4	3249302775510350827	select * from table(information_schema.query_histo...	2	65.000	1105.000
5	1924843303589736206	select hash(query_text), count(*), avg(compilation_...	1	46.000	566.000
6	-52326745867448573322	select count(*) from json_table;	1	103.000	2.000
7	-1732845353771103631	select * from table(information_schema.warehouse...	1	59.000	299.000
8	-5897400655839230779	select hash(query_text), count(*), avg(compilation_...	1	63.000	720.000
9	2021467874986086700	desc view vw_json;	1	18.000	10.000
10	-79601723214758340004	select json_data:cust_values , json_data:fname::str...	1	31.000	517.000

Using the HASH function further allows a user to easily query a particular instance of this query from the QUERY_HISTORY function. In the example above, I could check for specific queries where the HASH of the query text converted to the value `-102792116783286838`. For example:

```
select *
from table(information_schema.query_history())
where hash(query_text) = -102792116783286838
order by start_time;
```

Output:

row#	QUERY_ID	QUERY_TE...	DATABASE...	SCHEMA_N...	QUERY_TY...	SESSION_ID	USER_NAME	ROLE_NAME	WAREHOU...	WAREHOU...	WAREHOU...	QUI
1	b411f5e2-008...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
2	984f5f0e-150...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
3	f2f41772-a46...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
4	7498c667-68...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
5	3e83e056-7a...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
6	b20e5e05-1b...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
7	59942a92-78...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
8	1b9d54e0-c9...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
9	bada7117-bd...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
10	6c60abe8-ae...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594712...	MARIE	ANALYST_US	NULL	NULL	STANDARD	

The above result shows you all of the times you have issued this particular query (going back 7 days). Pay specific attention to the following columns:

- COMPILATION_TIME

- EXECUTION_TIME
- QUEUED (times)

If a query is spending more time compiling (COMPILATION_TIME) than executing (EXECUTION_TIME), perhaps it is time to review the complexity of the query. Snowflake's query compiler will optimize your query and identify all of the resources required to perform the query in the most efficient manner. If a query is overly complex, the compiler needs to spend more time sorting through the query logic. Take a look at your query and see if there are many nested subqueries or unnecessary joins. Additionally, if there are more columns being selected than required, then perhaps be more specific in your SELECT statement by specifying certain columns.

QUEUED time is interesting because it could be an indicator about your warehouse size and the amount of workload you've placed on the warehouse. Snowflake is able to run concurrent queries and it does a very good job in doing so. However, there will be times when a particularly large query will require more resources and, thus, cause other queries to queue as they wait for compute resources to be freed up. If you see a lot of queries spending a long time in queue, you could either:

- Dedicate a warehouse to these large complex running queries, or
- Utilize Snowflake's multi-clustering warehouse feature to allow more parallel execution of the queries.

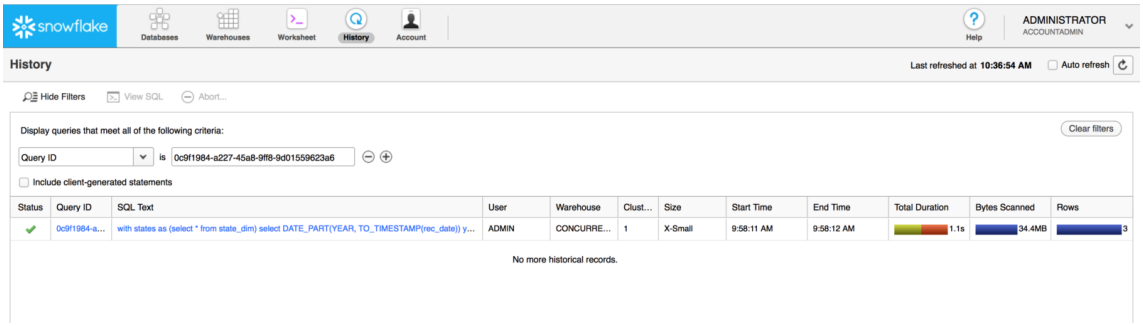
In the recent updates to our QUERY_HISTORY_* Information Schema functions, we have added more metadata references to the results and now you should have a range of metadata at your disposal:

- SESSION_ID
- USER_NAME , ROLE_NAME
- DATABASE_NAME , SCHEMA_NAME
- WAREHOUSE_NAME , WAREHOUSE_SIZE , WAREHOUSE_TYPE

These columns will help you identify the origin of the queries and help you fine tune your workflow. A simple example would be to find the warehouse with the longest-running queries. Or, find the user who typically issues these queries.

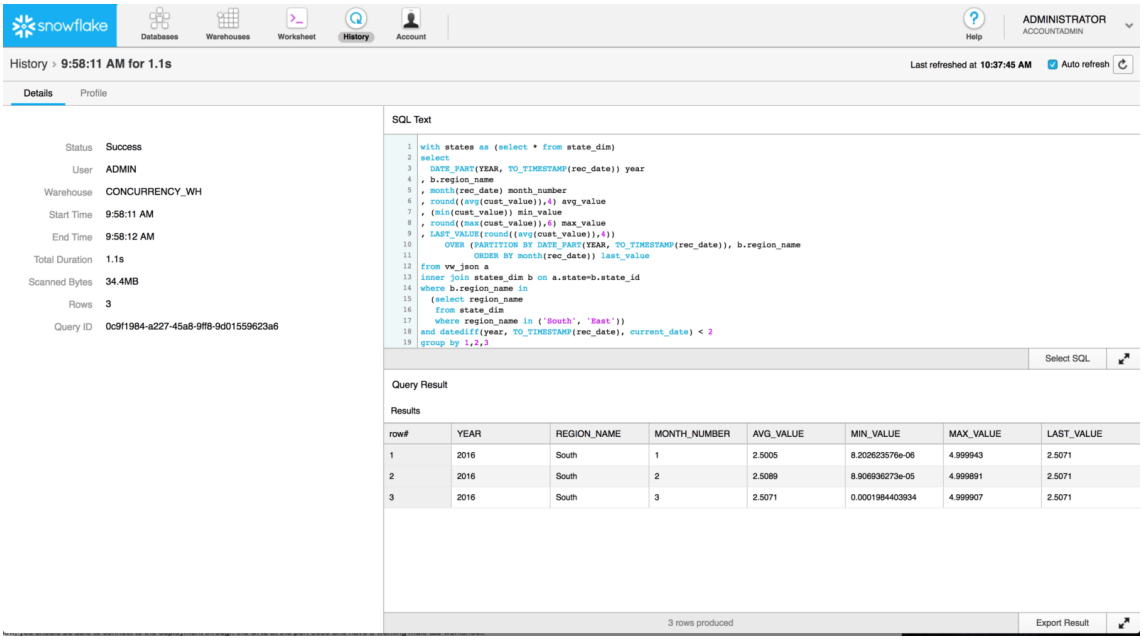
Profiling Tip #2: Using the UI

Once you have identified a particular query you would like to review, you can copy its QUERY_ID value and use this value to view its query plan in Snowflake's Query Profile. To do this, click on the **History** icon, add a QUERY ID filter, and paste the QUERY_ID in question. For example:



Hint: If you don't see a result, make sure you are using a role with the necessary warehouse MONITOR privilege (e.g. SYSADMIN or ACCOUNTADMIN) and you've selected the correct QUERY ID.

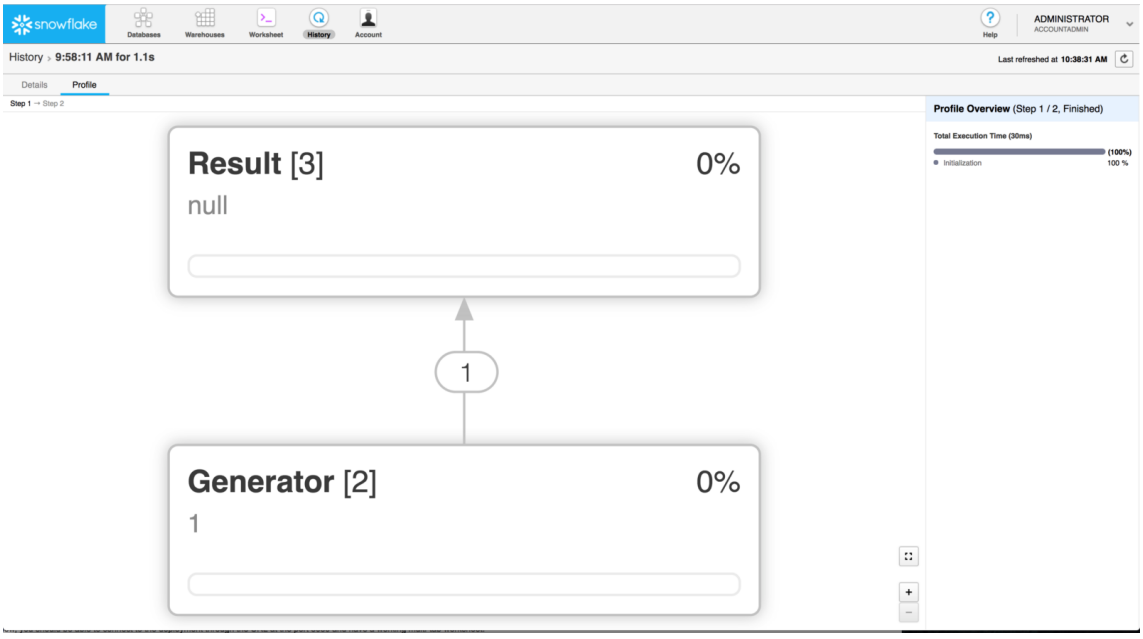
Once the search is complete, you should be able to click on the link provided under the **Query ID** column to go to the query's detail page:



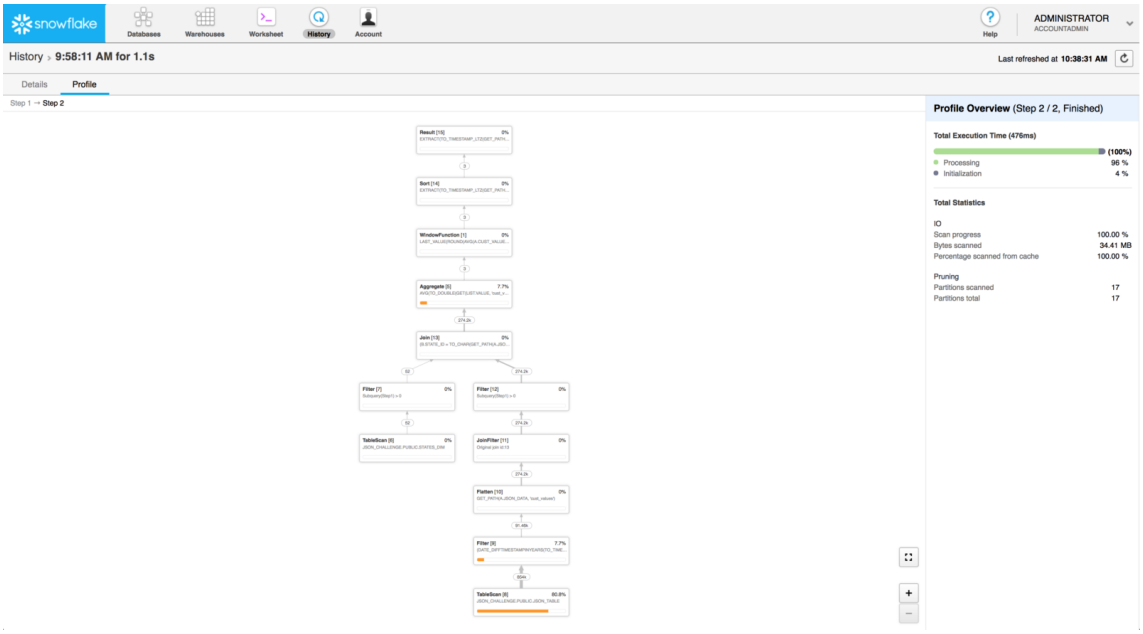
Now click on the **Profile** tab.

You should see a visualization of the Query Profile. In my example, Snowflake shows that this particular query executed in two steps:

Step 1:



Step 2:



Pay particular attention to the ***orange*** bar in this view. It indicates the percentage of overall query time spent on this particular process. In this instance, we can see our query spent most of the time reading data from the table. If we run this query often enough, we should see this time decrease because we'll be reading the data from cache instead of disk.

Conclusion

By utilizing the UI and the Information Schema functions and views described in this lab, you can use query profiling to help you understand your current workflow and identify queries that can be better optimized. This will help save you money in the long run and also improve your user experience. Snowflake will continue to invest in tools like these to help our users better understand and use our platform.

Zero to Snowflake: Automated Clustering in Snowflake

One of Snowflake's key selling points is automated clustering. However, it's not immediately clear what this actually means. It's easy to make the assumption that you will never need to think about how your data is clustered, and this is generally the case for tables under 1TB. But if you have tables larger than this (or if you're interested in general), definitely keep reading as this may help your query performance!

Snowflake is already designed to efficiently query data and return results quickly. For larger datasets, providing a little bit of guidance to Snowflake on how to store the data can significantly reduce query times. Snowflake cannot determine this itself (how could it know which columns you are most interested in without you telling it?). However, once you have told Snowflake which fields are important, it can automatically keep your data stored in a structure that supports rapid queries using your key fields. The combination of key fields the user provides is known as the Clustering Key, but we'll get to that.

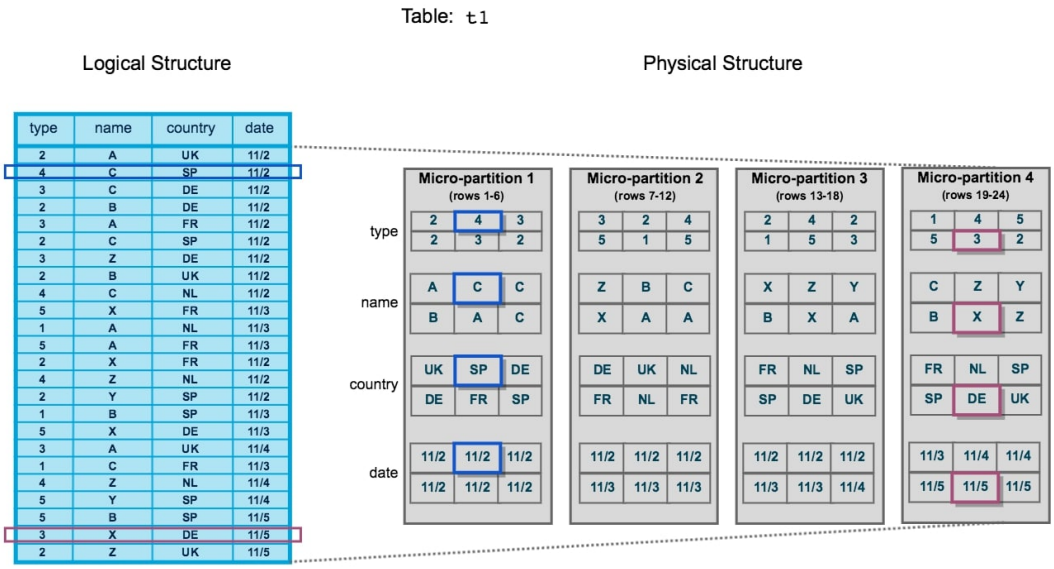
Data Storage Architecture and Micro-Partitions

Before we talk about clustering, let's talk about how data is stored and queried in Snowflake. To the user, Snowflake appears to store data in tables. To view data in a table, a user would simply use a query and view the results in the results pane. This is how the front-end works, but it is not how data is actually being managed and stored by Snowflake. Instead, Snowflake stores all data in encrypted files which are referred to as micro-partitions.

Each micro-partition will store a subset of the data, along with some accompanying metadata. Each micro-partition for a table will be similar in size, and from the name, you may have deduced that the micro-partition is small. Specifically, each file in Snowflake is stored in a compressed state, though if a micro-partition were to be decompressed, it would be between 50MB and 500MB in size.

The data inside each micro-partition usually spans multiple rows and columns, but this is not always the case, and it is unusual for a micro-partition to include every column in a table. Instead, micro-partitions typically span a handful of columns and rows and separate the columns within the file.

This is best explained with an example, which is supported by the following image:



Initially, this can be a confusing diagram to take in. Let's break it down, starting with the left-hand side. This is how a table would appear in Snowflake's user interface or as the result of a query. This is referred to as the logical structure. We have 24 rows of data spanning four columns: type, name, country and date. Two specific rows of data have been highlighted: row 2 and row 23.

On the right-hand side is a representation of how this data would be stored in the backend of Snowflake's architecture. This is referred to as the physical structure. Here, we can see four separate micro-partitions, each of which contains six rows of data. The first micro-partition contains data for rows 1 through 6. The data is stored by column instead of by row, enabling Snowflake to retrieve desired columns of data without breaking apart each row.

An alternative way to think about these micro-partitions is by thinking in terms of semi-structured data arrays. We could represent the physical structure by using four separate JSON files. For example, the first micro-partition could be represented as the following JSON array:

```
{
  "columns" : ["type", "name", "country", "date"],
  "rows" : [1, 2, 3, 4, 5, 6],
  "data" : [
    {
      "name": "type",
      "distinct values" : 3,
      "minimum value" : 2,
      "maximum value" : 4,
      "values" : [2, 4, 3, 2, 3, 2]
    },
    {
      "name": "name",
      "distinct values" : 3,
      "minimum value" : "A",
      "maximum value" : "C",
      "values" : ["A", "C", "C", "B", "A", "C"]
    },
    {
      "name": "country",
      "distinct values" : 4,
      "minimum value" : "DE",
      "maximum value" : "UK",
      "values" : ["UK", "SP", "DE", "DE", "FR", "SP"]
    },
    {
      "name": "date",
      "distinct values" : 1,
      "minimum value" : "11/2",
      "maximum value" : "11/2",
      "values" : ["11/2", "11/2", "11/2", "11/2", "11/2", "11/2"]
    }
  ]
}
```

This is a huge over-simplification of the storage method; however, this JSON representation demonstrates how some metadata is stored, as well as the values themselves.

Returning to the diagram above, rows 2 and 23 are highlighted on the left-hand side in blue and magenta, respectively. These align with the highlighted entries on the right-hand side, so we can see how these are stored in

micro-partitions 1 and 4 respectively.

Query Pruning

Remember how micro-partitions store metadata in addition to the data itself? This includes important pieces of information such as which fields are within the file, the number of distinct values for each field, the range of values for each field and a few other useful pieces of information to improve performance.

This metadata is a key part of the Snowflake architecture as it allows queries to determine whether or not the data inside a micro-partition should be queried. This way, when a query is executed, it does not need to scan the entire dataset but instead only queries the micro-partitions that hold relevant data. This process is known as query pruning, as the data is pruned before the query itself is executed.

Returning to our example above, imagine a query being executed on that data to return the **[type]** and **[country]** for records where the **[name]** is **Y**:

```
SELECT type, country
FROM MY_TABLE
WHERE name = "Y"
;
```

When this query executes in Snowflake, the micro-partitions are quickly scanned to determine which contain **Y** as a potential entry for the **[name]** field. The only micro-partitions that match this criterion are micro-partitions 3 and 4. Thus, the query pruning has reduced our total dataset to just these two partitions. In a similar way, only the **[type]** and **[country]** fields are required in the query output. Any micro-partitions that do not contain data for these columns would also be pruned. When the micro-partitions themselves are queried, only the required columns are queried, and Snowflake intelligently identifies which entries match which rows and can be aligned with the original **WHERE** clause.

Clustering

Clustering is a common word to encounter when looking behind the scenes of a data warehouse. The idea behind clustering is to organise the storage of data to better suit expected queries. The key objectives here are to improve query performance whilst reducing the system resources required to execute queries.

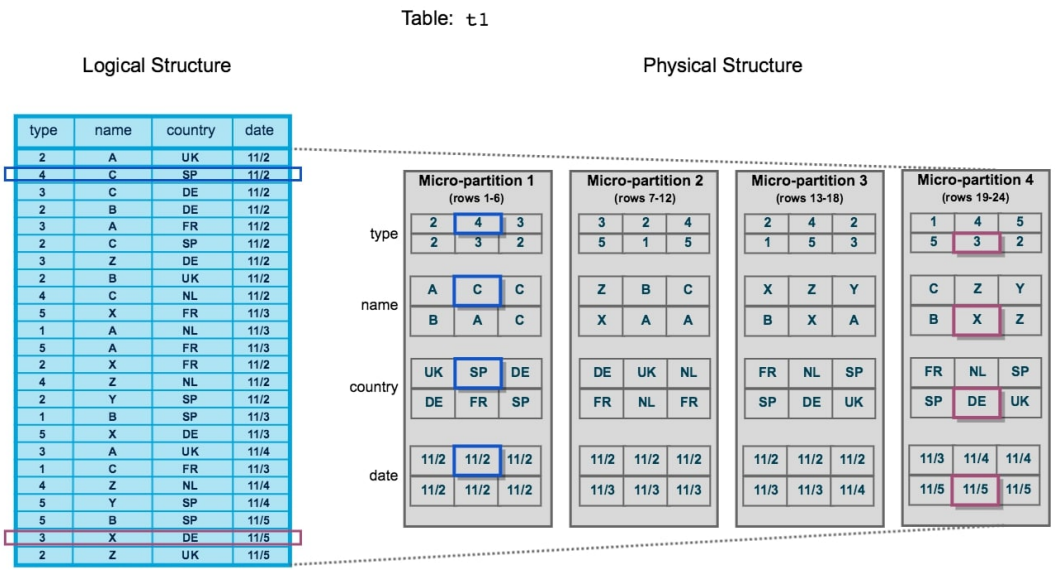
For example, imagine a table with over a billion records of data across many columns. One of these columns is **[category]**, which could be one of 10 possible values. Every day, millions of new records are added, again spanning these 10 categories.

In our example, whoever is querying this data is only ever interested in querying one or two categories at a time. With the standard method of storing data, a query with a **WHERE** clause that targets two categories would still query a large volume of data to return the right output. Whenever new data is added, this adds to the volume of data being queried, and there is no clean way to reduce the dataset before querying.

In Snowflake terminology, query pruning is less effective as far more micro-partitions contain multiple categories, and the desired categories in the query could easily be found in the majority of micro-partitions.

This is where clustering is effective. By restructuring how the data is stored, it is possible to improve query pruning and reduce the volume of data queried. Continuing our example, if our data was stored in a structure that was ordered by the **[category]** field, queries would not need to query the entire dataset. Instead, the query could skip the data up until a required **[category]** value appears, query the necessary records then skip the next set of records until it reaches another desired **[category]** value.

Let's return to our original example demonstrated by this image:



Again, we will consider the following query:

```
SELECT type, country
FROM MY_TABLE
WHERE name = "Y"
;
```

With how the data is currently stored, this query must investigate micro-partitions 3 and 4 since both contain the value **Y** in the **[name]** field. If the intended consumer of the data often filtered the data by the **[name]** field, the current method of storing the data would be inefficient. Instead, we can reorganise the data into the following logical structure:

Logical Structure

type	name	country	date
2	A	UK	11/2
3	A	FR	11/2
1	A	NL	11/3
5	A	FR	11/3
3	A	UK	11/4
2	B	DE	11/2
2	B	UK	11/2
1	B	SP	11/3
5	B	SP	11/5
4	C	SP	11/2
3	C	DE	11/2
2	C	SP	11/2
4	C	NL	11/2
1	C	FR	11/3
5	X	FR	11/3
2	X	FR	11/2
5	X	DE	11/3
3	X	DE	11/5
2	Y	SP	11/2
5	Y	SP	11/4
3	Z	DE	11/2
4	Z	NL	11/2
4	Z	NL	11/4
2	Z	UK	11/5

Physical Structure

	Micro-partition 1 (rows 1-6)			*Micro-partition 2 (rows 7-12)*			*Micro-partition 3 (rows 13-18)*			*Micro-partition 4 (rows 19-24)*		
type	2	3	1	2	1	5	4	1	5	2	5	3
	5	3	2	4	3	2	2	5	3	4	4	2
name	A	A	A	B	B	B	C	C	X	Y	Y	Z
	A	A	B	C	C	C	X	X	X	Z	Z	Z
country	UK	FR	NL	UK	SP	SP	NL	FR	FR	SP	SP	DE
	FR	UK	DE	SP	DE	SP	FR	DE	DE	NL	NL	UK
date	11/2	11/2	11/3	11/2	11/3	11/5	11/2	11/3	11/3	11/2	11/4	11/2
	11/3	11/4	11/2	11/2	11/2	11/2	11/2	11/3	11/5	11/2	11/4	11/5

The data is now stored and ordered based on the value of the **[name]** field. We can see that micro-partition 4 is now the only micro-partition that contains **[name]** values of **Y**. When we execute our query now, the query pruning will reduce our target data down to just micro-partition 4, which means our query has less data to interpret and thus will perform more efficiently.

Whilst our example has dealt with a small table of only 24 rows and 4 columns, the principle behind this approach is the same as table sizes increase. Indeed, as tables increase in size, the need to cluster the data increases with it. Similarly, smaller datasets benefit from clustering far less than larger datasets, and clustering on these smaller datasets is often fruitless. Whilst being useful for demonstrating the logic behind clustering, the table we have used in our example is far too small to benefit from clustering in any recognisable way; we would expect this query to complete in under a second whether the data was clustered or not.

A General Rule to Follow

A general rule to follow when considering clustering is to consider if the table is more than 1TB in size. If so, clustering is recommended. If not, depending on how large the table is and how well queries are performing, you may consider clustering but may find it more costly than it is worth. Personally, I often test clustering by cloning the table and testing different clustering approaches on said clone. If the query performance is noticeably improved and worth the cost, I will then consider applying the clustering approach to the original table and drop the clone. A few times here I have mentioned cost, which we will revisit this further down in this lab.

Clustering Keys

In the above example, we clustered our dataset based on the **[Name]** field, as this was a key field in our data and was used in our queries. As we have clustered based on this field, this field is referred to as the **clustering key**. In order to perform the clustering operation on an existing table using our key, the following SQL statement could be used:

```
ALTER TABLE MY_TABLE
CLUSTER BY (name)
;
```

In this situation, we are altering an existing table. This does not always need to be the case, however, as we can define clustering keys at the time of creation for a table as well:

```
CREATE TABLE MY_TABLE (
    type number
  , name string
  , country string
  , date date
)
CLUSTER BY (name)
;
```

These are very simple examples of creating clustering keys that achieve our current task. However, there are more capabilities here than meet the eye. A common question to ask at this stage is: *What if there are multiple fields that often affect our queries instead of one?*

Whilst we have only used a single field in our example of a clustering key, this is not the only option. If desired, multiple fields can be used to define a clustering key. Depending on who you talk to, some people refer to this as a **composite clustering key**, although it is common to stick with the original terminology of a clustering key since the requirement to use multiple fields is so common itself. In this situation, the order of the fields within the clustering

key determines the precedence. For example, if we wish to cluster by both **[name]** and **[date]**, the following query could be used:

```
ALTER TABLE MY_TABLE
CLUSTER BY (name, date)
;
```

As **[name]** comes before **[date]** in the clustering key, the data is first clustered by **[name]** and then the separate clusters are clustered by **[date]**. So we can include multiple fields in a clustering key, but why stop there? Another question to ask is: *What if our queries often apply a function to a specific field?*

Great news: expressions on fields can also be included in a cluster key. For example, the following query could be used to cluster our table based on the month and year of the **[date]** field:

```
ALTER TABLE MY_TABLE
CLUSTER BY (name, YEAR(date), MONTH(date))
;
```

So far, to be honest, nothing we have discussed in regard to clustering keys is particularly impressive for anybody accustomed to these tools. Most data warehouses have clustering capabilities and can define clustering keys based on multiple fields and expressions on those fields. Snowflake, however, has a large advantage over many of its competitors due to its impressive capabilities with semi-structured data. To demonstrate this effectively, we will need a new example column in our table. For simplicity, the following example only shows the first four rows of our table:

Updated Table with JSON Column

type	name	country	date	json_col
2	A	UK	11/2	`{ "type" : 2, "name": "A" }`
4	C	SP	11/2	`{ "type" : 4, "name": "C" }`
3	C	DE	11/2	`{ "type" : 3, "name": "C" }`
2	B	DE	11/2	`{ "type" : 2, "name": "B" }`

Our new field is called **[json_col]** and contains a simple JSON structure with the **[type]** and **[name]** for our data. In Snowflake, it is possible to directly query these entries:

```
SELECT
    json_col:type::number as type
  , json_col:name::string as name
FROM MY_TABLE
WHERE json_col:type::number = 2
;
```

In the exact same way, we can include these expressions in clustering keys:

```
ALTER TABLE MY_TABLE
CLUSTER BY (json_col:name::string as name, country, YEAR(date), MONTH(date),
json_col:type::number as type)
;
```

This last example demonstrates how any of the following can be included in a clustering key:

1. Base columns
2. Expressions on base columns
3. Expressions on paths in VARIANT columns

Keep in mind when defining clustering keys that it is certainly possible to go too heavy on the clustering. When too many elements are included in a clustering key, clustering the data can become too costly. Also, you may find the data is already clustered as well as it can be by the first field(s) in the clustering key. In this case, adding further fields to the clustering key will have no effect on query speeds but will still be costly to the system as it attempts to implement said further clustering.

A good principle to follow when choosing which fields should be included in a cluster key is to first consider which fields most commonly contribute to the filtering/WHERE clause in a given query. Secondly, you can consider which fields are used when joining to other tables.

Clustering Information

To assist with determining how well-clustered a table is, Snowflake has provided a system function called **SYSTEM$CLUSTERING_INFORMATION**. This function takes in a table name and a list of columns as inputs and outputs an overview of how well the table is clustered based on the list of columns. In effect, the list of columns is the clustering key for which the clustering information is being provided. You can thus use this function to determine how the data is clustered across several possible sets of columns, as well as potentially use this information to guide your final choice of clustering key for the table.

If no list of columns is provided, the function will instead return the clustering information for the table based on its current clustering key. If no current clustering key is defined on the table, the function will error. The function is executed using a **SELECT** statement, and it is important to note that both inputs are strings:

```
SELECT SYSTEM$CLUSTERING_INFORMATION('TABLE_NAME', '(COLUMN_1, COLUMN_2, ...,
COLUMN_N)')
```

This function returns a single JSON object containing multiple fields that provide details on various aspects of the table's clustering.

cluster_by_keys

Snowflake's description of this field:

> *[Columns in table used to return clustering information; can be any columns in the table.]{style="color: #818181;"}*

This is the list of columns that has been provided as an input to the function or the existing clustering key of the table if the input was left empty.

notes

Description of this field:

> *[This column can contain suggestions to make clustering more efficient. For example, this field might contain a warning if the cardinality of the clustering column is extremely high. This column can be empty.]{style="color: #818181;"}[For more information about how to cluster efficiently, see]{style="color: #818181;"}[Strategies for Selecting Clustering Keys].*

Snowflake have done well by including this field when it is relevant. When this field appears, it often provides guidance on whether or not clustering by the input clustering key is a good idea.

A standard and helpful message to see from these notes is that the cardinality of a particular field in the data is high and may result in expensive re-clustering. When seeing this message, it is recommended to consider alternative fields to use in the clustering key which may have a lower cardinality.

total_partition_count

Snowflake's description of this field:

> *[Total number of micro-partitions that comprise the table.]{style="color: #818181;"}*

This field simply provides the total number of micro-partitions used to store the data for the table.

total_constant_partition_count

Snowflake's description of this field:

> *[Total number of micro-partitions for which the value of the specified columns have reached a constant state (i.e. the micro-partitions will not benefit significantly from re-clustering). The number of constant micro-partitions in a table has an impact on pruning for queries. The higher the number, the more micro-partitions can be pruned from queries executed on the table, which has a corresponding impact on performance.]{style="color: #818181;"}*

This field is extremely useful when reviewing the clustering of a table. As this number increases, we can expect query pruning to improve and queries to execute more efficiently. Whilst not provided as a field by this function, I often find it useful to consider the **total_constant_partition_count** as a percentage of the **total_partition_count**. This provides a useful metric for what percentage of the data is clustered efficiently.

It is worth keeping in mind that this field should not be the only thing considered when reviewing clustering since it can lead to false positive scenarios. For example, our table could contain a boolean field which only ever has two values. We may find that the **total_constant_partition_count** is the full **total_partition_count**, which would suggest good clustering. However, any queries that filter on more than our single boolean field may not benefit from the clustering as much when pruning.

average_overlaps

Snowflake's description of this field:

> *[Average number of overlapping micro-partitions for each micro-partition in the table. A high number indicates the table is not well-clustered.]{style="color: #818181;"}*

In this situation, an overlap is when the same value for a field appears in more than one micro-partition. Let's return to our previous clustered example where we clustered based on the **[name]** field, specifically targeting this field:

After this clustering, we can see that each possible **[name]** value may still appear in multiple micro-partitions. Specifically, values **B** and **C** each appear in two micro-partitions:

Name by Micro-Partition

	Micro-partition 1 (rows 1-6)			*Micro-partition 2 (rows 7-12)*			*Micro-partition 3 (rows 13-18)*			*Micro-partition 4 (rows 19-24)*		
name	A	A	A	B	B	B	C	C	X	Y	Y	Z
	A	A	B	C	C	C	X	X	X	Z	Z	Z

value	micro-partitions list	micro-partitions count
A	1	1
B	1, 2	2
C	2, 3	2
X	3	1
Y	4	1
Z	4	1

From a micro-partition perspective, we have four micro-partitions with the following overlaps:

micro-partition	overlapping micro-partitions list	number of overlapping micro-partitions
1	2	1
2	1, 3	2
3	2	1
4		0
	Average	1

Here, we can see the average number of overlapping micro-partitions is 1 and thus the **average_overlaps** value from the **SYSTEM$CLUSTERING_INFORMATION** function will be 1.

average_depth

Snowflake's description of this field:

> *[Average overlap depth of each micro-partition in the table. A high number indicates the table is not well-clustered.]{style="color: #818181;"}*

*This value is also returned by **SYSTEM$CLUSTERING_DEPTH**.*

The overlap depth of a micro-partition is the average number of micro-partitions that an individual value may appear in when an overlap occurs. Let's return to our previous example again where we have clustered by **[name]**:

Name by Micro-Partition

	Micro-partition 1 (rows 1-6)			*Micro-partition 2 (rows 7-12)*			*Micro-partition 3 (rows 13-18)*			*Micro-partition 4 (rows 19-24)*		
name	A	A	A	B	B	B	C	C	X	Y	Y	Z
	A	A	B	C	C	C	X	X	X	Z	Z	Z

The following table again demonstrates the overlapping micro-partitions, along with the minimum and maximum value in each partition. We can see that the values **B** and **C** both span multiple micro-partitions. We could represent this information differently:

micro-partition	overlapping micro-partitions list	number of overlapping micro-partitions	minimum [name] value	maximum [name] value
1	2	1	A	B
2	1, 3	2	B	C
3	2	1	C	X
4		0	Y	Z

We can see from this diagram that we have three overlapping micro-partitions, but each value only overlaps with a single other micro-partition each time. Specifically, values **B** and **C** overlap. We could consider this in terms of values:

value	micro-partitions list	overlap?	depth (number of overlapping micro-partitions)
A	1	No	
B	1, 2	Yes	2
C	2, 3	Yes	2
X	3	No	
Y	4	No	
Z	4	No	
		Average	2

This tells us that whilst we have three overlapping micro-partitions (1, 2 and 3), the overlap depth is **2**. Note here how values that did not overlap at all are not counted in the depth. The purpose of the depth is to understand more about overlapping data, and including non-overlapping values would skew this result.

The depth is still considered in terms of micro-partitions instead of values, so we aggregate this back to the micro-partition level:

micro-partition	overlapping micro-partitions list	number of overlapping micro-partitions	average overlap depth
1	2	1	2
2	1, 3	2	2
3	2	1	2
4		0	1

partition_depth_histogram

Snowflake's description of this field:

> *[A histogram depicting the distribution of overlap depth for each micro-partition in the table. The histogram contains buckets with widths of]{style="color: #818181;"}[0 to 16 with increments of 1.]{style="color: #818181;"} [For buckets larger than 16, increments of twice the width of the previous bucket (e.g. 32, 64, 128, . . .)] {style="color: #818181;"}*

This value takes the overlap depth for each micro-partition and plots it in a histogram. Let's return to our previous example:

micro-partition	overlapping micro-partitions list	number of overlapping micro-partitions	average overlap depth
1	2	1	2
2	1, 3	2	2
3	2	1	2
4		0	1

Here, our histogram is quite small, but it works for our simple demonstration purposes:

average overlap depth	number of micro-partitions
1	1
2	3

Example Output

Of course, real-world scenarios can be much more complicated than this. Snowflake provides its own example of a demonstration output of **system$clustering_information** that I have included below:

```
{
  "cluster_by_keys" : "(COL1, COL3)",
  "total_partition_count" : 1156,
  "total_constant_partition_count" : 0,
  "average_overlaps" : 117.5484,
  "average_depth" : 64.0701,
  "partition_depth_histogram" : {
     "00000" : 0,
     "00001" : 0,
     "00002" : 3,
     "00003" : 3,
     "00004" : 4,
     "00005" : 6,
     "00006" : 3,
     "00007" : 5,
     "00008" : 10,
     "00009" : 5,
     "00010" : 7,
     "00011" : 6,
     "00012" : 8,
     "00013" : 8,
     "00014" : 9,
     "00015" : 8,
     "00016" : 6,
     "00032" : 98,
     "00064" : 269,
     "00128" : 698
    }
}
```

Learning how to read and understand these diagrams takes time as there is a lot of information here. A question to ask yourself: Based on the information provided in the output above, do you think the table is well clustered? If so, why? If not, why not?

The Cost of Re-clustering

In Snowflake, (re)clustering is performed in the services layer of the tool. This means that a virtual warehouse is not required, and Snowflake has its own way of keeping track of the credit cost. You can see this yourself in the **Billing & Usage** section of the **Account** area in your Snowflake environment, under the warehouse **AUTOMATIC_CLUSTERING**.

Automated Re-clustering Through Snowflake

This has been a pretty heavy-going lab covering a lot of content, and we have covered WHY you may want to cluster a table and HOW Snowflake achieves this clustering. But we haven't touched the best part yet--the part that separates Snowflake from other similar tools: Snowflake performs clustering automatically.

In most systems, processes must be put in place to re-cluster a table after data manipulation has occurred. Maybe one of the fields that contributes to the clustering key has had its valued changed, or maybe records have been added to or removed from the table. In most systems, somebody must either schedule clustering to occur (which may then run when not required) or manually apply re-clustering as and when it is deemed to be required.

Snowflake, on the other hand, uses an automated approach that is controlled by a single flag for each table. This flag is enabled or disabled using the following commands:

```
ALTER TABLE TABLE_NAME RESUME RECLUSTER;
ALTER TABLE TABLE_NAME SUSPEND RECLUSTER;
```

This is an extremely powerful tool as it takes the entire clustering process and removes all the irritating admin. The main thing to keep in mind though is that this can run away with you in terms of credit consumption if you are not paying attention to the number of objects. For example, if you create a clone of a table, that clone will default to have the same clustering information. Snowflake ensures clones disable automatic clustering by default, but it's recommended to verify that the clone is clustering the way you want before enabling automated clustering again.

We can review the clustering for each table with either the **SHOW TABLES** command or through the tables view of the information schema. Specifically, the **AUTO_CLUSTERING_ON** field can tell you if clustering is enabled or disabled, and the **CLUSTER_BY (SHOW TABLES)** or **CLUSTERING_KEY** (information schema) field can be used to determine the current clustering key.

Summary

This lab is an attempt to unlock some of the mystery behind clustering in Snowflake and explain some of the different ways this can be evaluated. There is a lot of content here, and I would recommend playing around with the functionality yourself to reinforce your understanding. Remember, you can make use of zero-copy cloning to give yourself a few test tables to play with and try out different clustering methods. As always, I'd love to hear your thoughts in the comments section down below. I hope you enjoy the rest of your day!