

Getting Started with Snowflake - Zero to Snowflake

Overview

Welcome to Snowflake! This entry-level guide designed for database and data warehouse administrators and architects will help you navigate the Snowflake interface and introduce you to some of our core capabilities. [Sign up for a free 30-day trial of Snowflake](#) and follow along with this lab exercise. Once we cover the basics, you'll be ready to start processing your own data and diving into Snowflake's more advanced features like a pro.

Prerequisites:

- Use of the [Snowflake free 30-day trial environment](#)
- Basic knowledge of SQL, database concepts, and objects
- Familiarity with CSV comma-delimited files and JSON semi-structured data

What You'll Learn:

- How to create stages, databases, tables, views, and virtual warehouses.
- How to load structured and semi-structured data.
- How to perform analytical queries on data in Snowflake, including joins between tables.
- How to clone objects.
- How to undo user errors using Time Travel.
- How to create roles and users, and grant them privileges.
- How to securely and easily share data with other accounts.
- How to consume datasets in the Snowflake Data Marketplace.

Prepare Your Lab Environment

If you haven't already, register for a [Snowflake free 30-day trial](#). The rest of the sections in this lab assume you are using a new Snowflake account created by registering for a trial.

The Snowflake edition (Standard, Enterprise, Business Critical, etc.) and cloud provider (AWS, Azure, GCP), and Region (US East, EU, etc.) you use for this lab do not matter. However, we suggest you select the region that is physically closest to you and Enterprise, our most popular offering, as your Snowflake edition.

After registering, you will receive an email with an activation link and URL for accessing your Snowflake account.

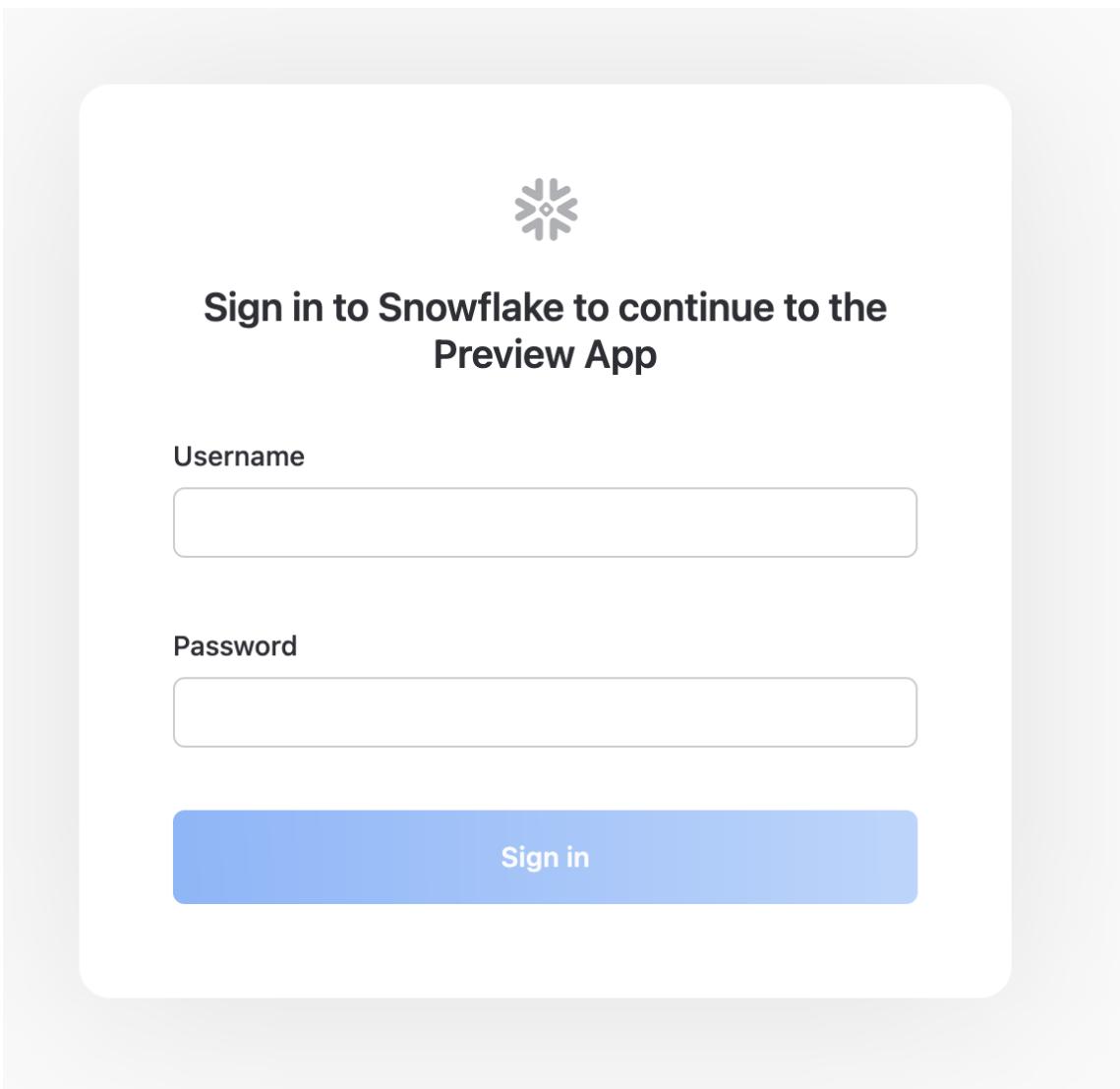
The Snowflake User Interface & Lab Story

Negative : **About the screenshots, sample code, and environment** Screenshots in this lab depict examples and results that may vary slightly from what you see when you complete the exercises.

Logging into the Snowflake User Interface (UI)

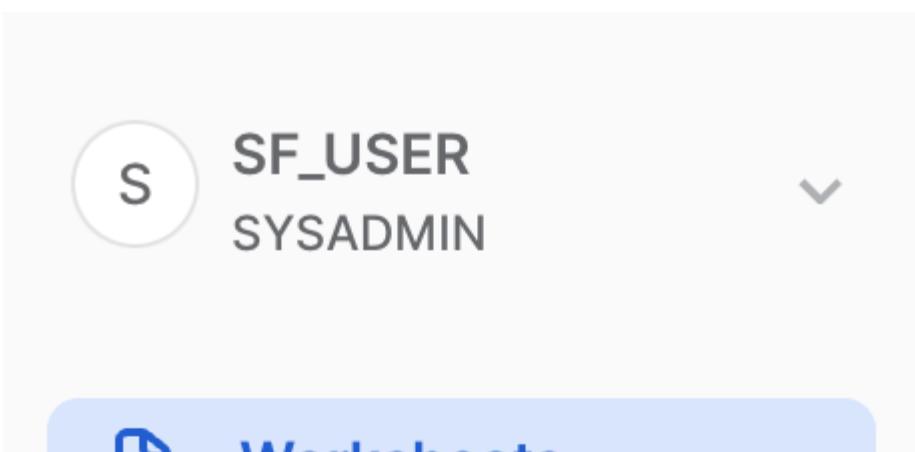
Open a browser window and enter the URL of your Snowflake 30-day trial environment that was sent with your registration email.

You should see the following login dialog. Enter the username and password that you specified during the registration:



Navigating the Snowflake UI

Let's get you acquainted with Snowflake! This section covers the basic components of the user interface. We will move from top to bottom on the left-hand side margin.





worksheets

Dashboards

Data

Marketplace

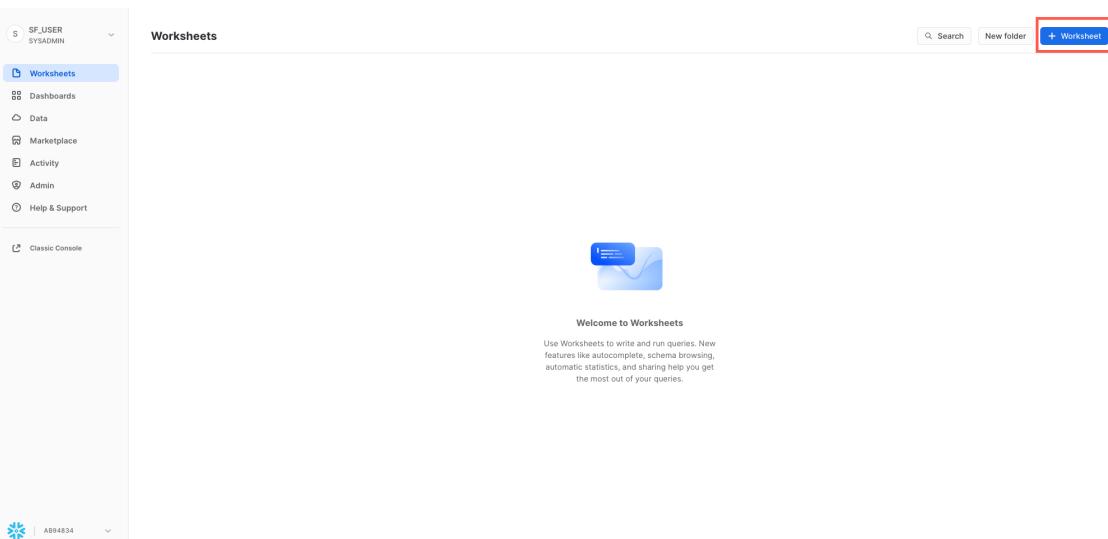
Activity

Admin

Help & Support



Classic Console



The **Worksheets** tab provides an interface for submitting SQL queries, performing DDL and DML operations, and viewing results as your queries or operations complete. A new worksheet is created by clicking **+ Worksheet** on the top right.

The top left corner contains the following:

- **Home** icon: Use this to get back to the main console/close the worksheet.
- **Worksheet_name** drop-down: The default name is the timestamp when the worksheet was created. Click the timestamp to edit the worksheet name. The drop-down also displays additional actions you can perform for the worksheet.
- **Manage filters** button: Custom filters are special keywords that resolve as a subquery or list of values.

The top right corner contains the following:

- **+ button**: This creates a new worksheet.

- **Context** box: This lets Snowflake know which role and warehouse to use during this session. It can be changed via the UI or SQL commands.
- **Share** button: Open the sharing menu to share to other users or copy the link to the worksheet.
- **Play/Run** button: Run the SQL statement where the cursor currently is or multiple selected statements.

The middle pane contains the following:

- Drop-down at the top for setting the database/schema/object context for the worksheet.
- General working area where you enter and execute queries and other SQL statements.

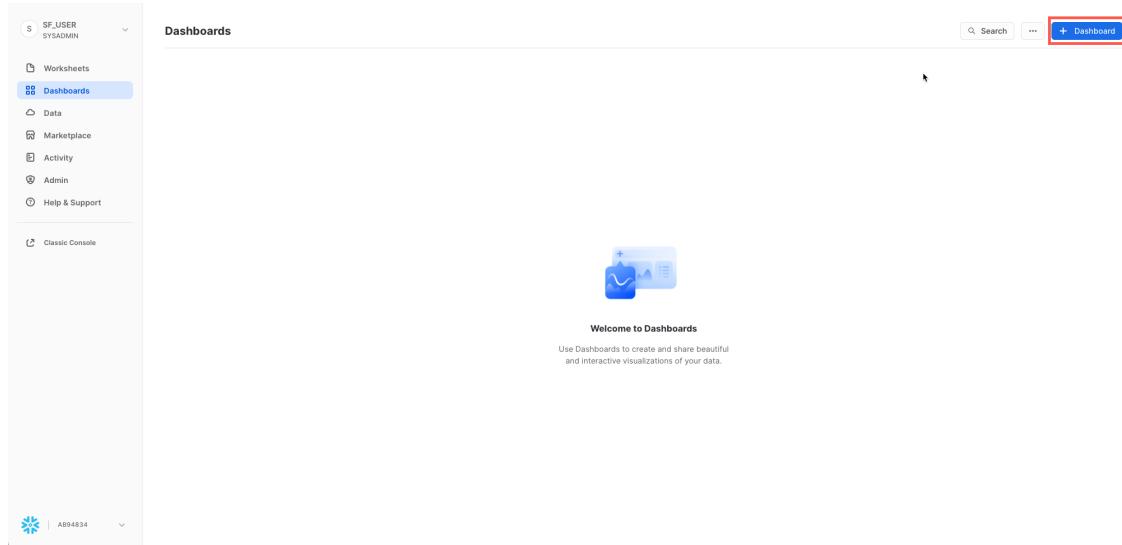
The middle-left panel contains the database objects browser which enables you to explore all databases, schemas, tables, and views accessible by the role currently in use for the worksheet.

The bottom pane displays the results of queries and other operations. Also includes 4 options (**Object**, **Query**, **Result**, **Chart**) that open/close their respective panels on the UI. **Chart** opens a visualization panel for the returned results. More on this later.

The various panes on this page can be resized by adjusting their sliders. If you need more room in the worksheet, collapse the database objects browser in the left panel. Many of the screenshots in this guide keep this panel closed.

Negative : **Worksheets vs the UI** Most of the exercises in this lab are executed using pre-written SQL within this worksheet to save time. These tasks can also be done via the UI, but would require navigating back-and-forth between multiple UI tabs.

Dashboards



The **Dashboards** tab allows you to create flexible displays of one or more charts (in the form of tiles, which can be rearranged). Tiles and widgets are produced by executing SQL queries that return results in a worksheet. Dashboards work at a variety of sizes with minimal configuration.

Databases

The screenshot shows the Snowflake UI with the left sidebar expanded. Under the 'Data' section, the 'Databases' tab is selected. The main area displays a table titled 'Databases' with one row:

NAME	SOURCE	OWNER	CREATED
SNOWFLAKE_SAMPLE_DATA	Share	ACCOUNTADMIN	1 hour ago

At the top right of the main area, there is a blue button labeled '+ Database'. The left sidebar also includes sections for 'Private Sharing', 'Provider Studio', 'Marketplace', 'Activity', 'Admin', and 'Help & Support'.

Under **Data**, the **Databases** tab shows information about the databases you have created or have permission to access. You can create, clone, drop, or transfer ownership of databases, as well as load data in the UI. Notice that a database already exists in your environment. However, we will not be using it in this lab.

Private Shared Data

The screenshot shows the Snowflake UI with the left sidebar expanded. Under the 'Data' section, the 'Private Sharing' tab is selected. The main area displays a table titled 'Shared With Me' with one row:

NAME	SOURCE	OWNER	CREATED
SNOWFLAKE_SAMPLE_DATA	Shared 1 hour ago		

At the top right of the main area, there is a blue button labeled 'Share Data'. The left sidebar also includes sections for 'Private Sharing', 'Provider Studio', 'Marketplace', 'Activity', 'Admin', and 'Help & Support'.

Also under **Data**, the **Private Shared Data** tab is where data sharing can be configured to easily and securely share Snowflake tables among separate Snowflake accounts or external users, without having to create a copy of the data. We will cover data sharing in Section 10.

Marketplace

The screenshot shows the Snowflake Marketplace page. At the top, there's a search bar and navigation links for 'Categories', 'Business Needs', 'Providers', and 'My Requests'. Below this, a filter bar includes 'Ready to Query', 'Free', 'Weather', 'Financial', '360-Degree Customer View', 'Demand Forecasting', 'Japan', and 'Last month'. The main content is divided into two sections: 'Most Popular' and 'Most Recent'. Each section contains four data set cards. The 'Most Popular' section includes 'Starschema COVID-19 Epidemiological Data' (Personalized), 'Worldwide Address Data' (Personalized), 'Exchange Data International SAMPLE: EDI Foreign Exchange Rates' (Personalized), and 'Knomena Economy Data Atlas' (Personalized). The 'Most Recent' section includes 'AHEAD Chicago Divvy Bike Station Status' (Personalized), 'Alesco Data Alesco Email Database (sample)' (Personalized), 'Arcadia Arcadia Research Data' (Personalized), and 'Hyland Software, Inc. OnBase Log Bundle' (Personalized). A 'More >' link is visible at the bottom right of each section.

The **Marketplace** tab is where any Snowflake customer can browse and consume data sets made available by providers. There are two types of shared data: Public and Personalized. Public data is free data sets available for querying instantaneously. Personalized data requires reaching out to the provider of data for approval of sharing data.

Query History

The screenshot shows the 'Query History' page under the 'Activity' tab. The left sidebar shows standard navigation options like Worksheets, Dashboards, Data, Marketplace, Activity, and Help & Support. The 'Query History' tab is selected. The main area displays a search bar and a message stating 'No Queries'. Below it, a note says 'There are no queries matching your filters.' At the bottom right, there are buttons for 'Status All', 'User SF_USER', 'Filters', and 'Columns'.

Under **Activity** there are two tabs **Query History** and **Copy History**:

- **QuerHistory** is where previous queries are shown, along with filters that can be used to hone results (user, warehouse, status, query tag, etc.). View the details of all queries executed in the last 14 days from your Snowflake account. Click a query ID to drill into it for more information.
- **Copy History** shows the status of copy commands run to ingest data into Snowflake.

Warehouses

The screenshot shows the Snowflake Admin interface. On the left, there's a sidebar with a user dropdown (SF_USER SYSADMIN), navigation links (Worksheets, Dashboards, Data, Marketplace, Activity, Admin, Usage, Warehouses, Resource Monitors, Users & Roles, Security, Billing, Partner Connect, Help & Support), and a classic console link. At the bottom of the sidebar is a session ID (AB94834). The main area is titled 'Warehouses' and shows a table with one row for 'COMPUTE_WH'. The table columns are NAME, STATUS, SIZE, CLUSTERS, RUNNING, QUEUED, OWNER, and CREATED. A search bar and filter buttons are at the top right of the table.

Under **Admin**, the **Warehouses** tab is where you set up and manage compute resources known as virtual warehouses to load or query data in Snowflake. If you don't see a warehouse called COMPUTE_WH in your environment, you can create it as shown below:

The screenshot shows the 'New Warehouse' dialog box. It has a title 'New Warehouse' and a note 'Creating as ACCOUNTADMIN'. The 'Name' field contains 'COMPUTE_WH' and the 'Size' dropdown is set to 'Small 2 credits/hour'. There's an optional 'Comment' field and a 'Query Acceleration' toggle switch. Below these are sections for 'Multi-cluster Warehouse' and 'Advanced Warehouse Options' (Auto Resume, Auto Suspend, Suspend After (min) set to 10). At the bottom are 'Cancel' and 'Create Warehouse' buttons.

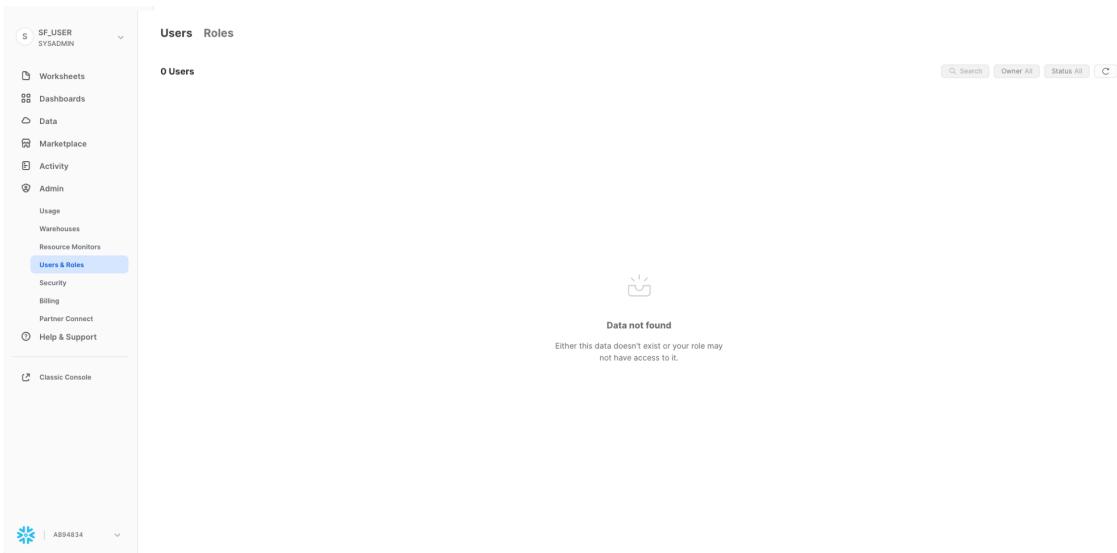
Resource Monitors

Under **Admin**, the **Resource Monitors** tab shows all the resource monitors that have been created to control the number of credits that virtual warehouses consume. For each resource monitor, it shows the credit quota, type of monitoring, schedule, and actions performed when the virtual warehouse reaches its credit limit.

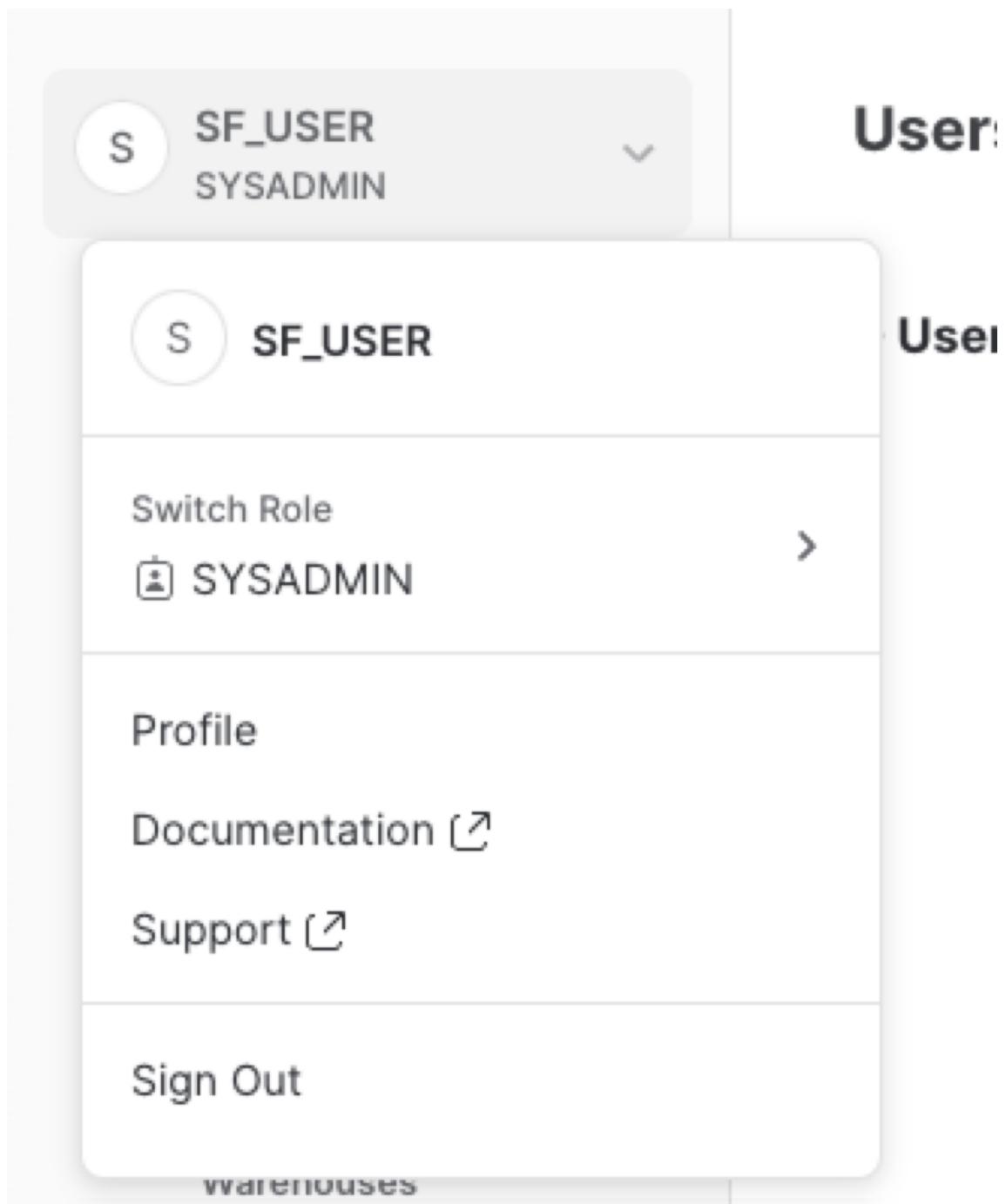
Roles

Under **Admin**, the **Roles** sub-tab of the **Users and Roles** tab shows a list of the roles and their hierarchies. Roles can be created, reorganized, and granted to users in this tab. The roles can also be displayed in tabular/list format by selecting the **Table** sub-tab.

Users



Also under **Admin** tab, the **Users** sub-tab of the **Users and Roles** tab shows a list of users in the account, default roles, and owner of the users. For a new account, no records are shown because no additional roles have been created. Permissions granted through your current role determine the information shown for this tab. To see all the information available on the tab, switch your role to ACCOUNTADMIN.



Clicking on your username in the top right of the UI allows you to change your password, roles, and preferences. Snowflake has several system defined roles. You are currently in the default role of `SYSADMIN` and will stay in this role for the majority of the lab.

Negative : **SYSADMIN** The `SYSADMIN` (aka System Administrator) role has privileges to create warehouses, databases, and other objects in an account.

The Lab Story

This lab is based on the analytics team at Citi Bike, a real, citywide bike sharing system in New York City, USA. The team wants to run analytics on data from their internal transactional systems to better understand their riders and how to best serve them.

We will first load structured `.csv` data from rider transactions into Snowflake. Later we will work with open-source, semi-structured JSON weather data to determine if there is any correlation between the number of bike rides and the weather.

Preparing to Load Data

Let's start by preparing to load the structured Citi Bike rider transaction data into Snowflake.

This section walks you through the steps to:

- Create a database and table.
- Create an external stage.
- Create a file format for the data.

Negative : **Getting Data into Snowflake** There are many ways to get data into Snowflake from many locations including the COPY command, Snowpipe auto-ingestion, external connectors, or third-party ETL/ELT solutions.

For the purposes of this lab, we use the COPY command and AWS S3 storage to load data manually. In a real-world scenario, you would more likely use an automated process or ETL solution.

The data we will be using is bike share data provided by Citi Bike NYC. The data has been exported and pre-staged for you in an Amazon AWS S3 bucket in the US-EAST region. The data consists of information about trip times, locations, user type, gender, age, etc. On AWS S3, the data represents 61.5M rows, 377 objects, and 1.9GB compressed.

Below is a snippet from one of the Citi Bike CSV data files:

```
"tripduration","starttime","stoptime","start station id","start station name","start station latitude","start station longitude","end station id","end station name","end station latitude","end station longitude","bikeid","name_localizedValue0","usertype","birth year","gender",196,"2018-01-01 00:01:51","2018-01-01 00:05:07",315,"South St & Gouverneur Ln",40.70355377,-74.00670227,259,"South St & Whitehall St",40.70122128,-74.01234218,18534,"Annual Membership","Subscriber",1997,1,207,"2018-01-01 00:02:44","2018-01-01 00:06:11",3224,"W 13 St & Hudson St",40.73997354103409,-74.00513872504234,470,"W 20 St & 8 Ave",40.74345335,-74.00004031,19651,"Annual Membership","Subscriber",1978,1,613,"2018-01-01 00:03:15","2018-01-01 00:13:28",386,"Centre St & Worth St",40.71494807,-74.00234482,2008,"Little West St & 1 Pl",40.70569254,-74.01677685,21678,"Annual Membership","Subscriber",1982,1
```

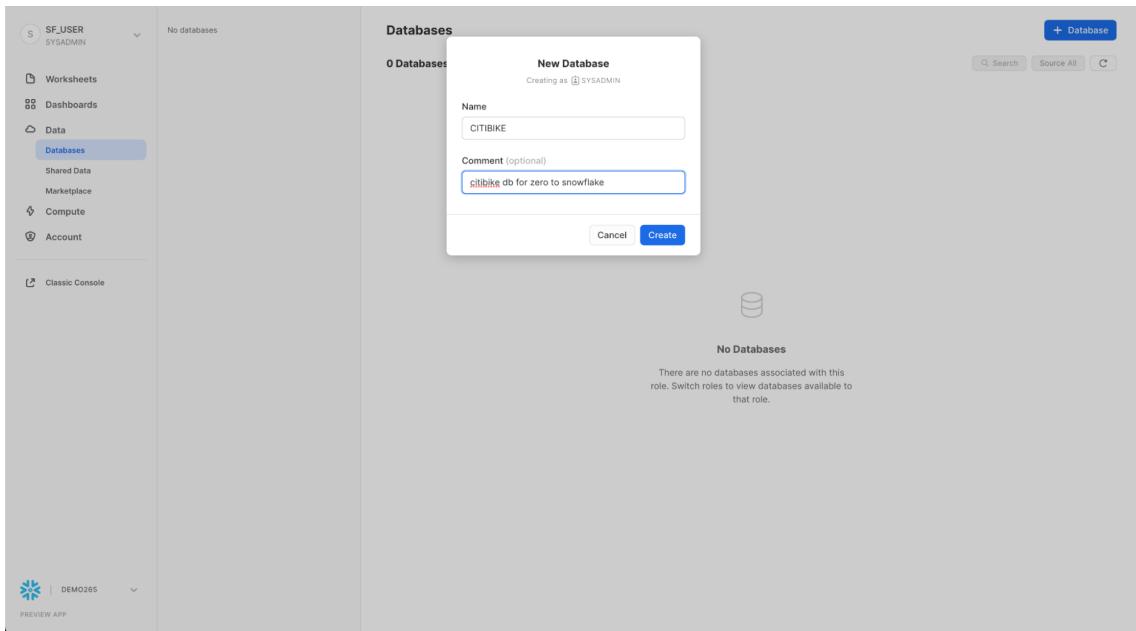
It is in comma-delimited format with a single header line and double quotes enclosing all string values, including the field headings in the header line. This will come into play later in this section as we configure the Snowflake table to store this data.

Create a Database and Table

First, let's create a database called `CITIBIKE` to use for loading the structured data.

Ensure you are using the sysadmin role by selecting **Switch Role > SYSADMIN**.

Navigate to the **Databases** tab. Click **Create**, name the database `CITIBIKE`, then click **CREATE**.



Now navigate to the **Worksheets** tab. You should see the worksheet we created in step 3.

We need to set the context appropriately within the worksheet. In the upper right corner of the worksheet, click the box next to the + to show the context menu. Here we control the elements you can see and run from each worksheet. We are using the UI here to set the context. Later in the lab, we will accomplish the same thing via SQL commands within the worksheet.

Select the following context settings:

Role: `SYSADMIN` Warehouse: `COMPUTE_WH`

The screenshot shows the Snowflake UI interface. In the top right corner, there are dropdown menus for 'Roles' (selected: SYSADMIN) and 'Warehouses' (selected: COMPUTE_WH). The main workspace shows a single query result:

```
1 select col from table where created = :daterange
```

Below the query results, there is a 'Query Details' panel showing:

- Query duration: 790ms
- Rows: 1
- COLUMN1: 0% filled, 100% blank

Next, in the drop-down for the database, select the following context settings:

Database: CITIBIKE Schema = PUBLIC

The screenshot shows the Snowflake UI interface. In the top right corner, there are dropdown menus for 'Databases' (selected: CITIBIKE) and 'Warehouses' (selected: COMPUTE_WH). The main workspace shows a single query result:

```
1 select col from table where created = :daterange
```

Below the query results, there is a 'Query Details' panel showing:

- Query duration: 56ms
- Rows: 1
- COLUMN1: 0% filled, 100% null

Negative : Data Definition Language (DDL) operations are free! All the DDL operations we have done so far do not require compute resources, so we can create all our objects for free.

To make working in the worksheet easier, let's rename it. In the top left corner, click the worksheet name, which is the timestamp when the worksheet was created, and change it to `CITIBIKE_ZERO_TO_SNOWFLAKE`.

Next we create a table called `TRIPS` to use for loading the comma-delimited data. Instead of using the UI, we use the worksheet to run the DDL that creates the table. Copy the following SQL text into your worksheet:

```

create or replace table trips
(tripduration integer,
starttime timestamp,
stoptime timestamp,
start_station_id integer,
start_station_name string,
start_station_latitude float,
start_station_longitude float,
end_station_id integer,
end_station_name string,
end_station_latitude float,
end_station_longitude float,
bikeid integer,
membership_type string,
usertype string,
birth_year integer,
gender integer);

```

Negative : Many Options to Run Commands. SQL commands can be executed through the UI, via the **Worksheets** tab, using our SnowSQL command line tool, with a SQL editor of your choice via ODBC/JDBC, or through our other connectors (Python, Spark, etc.). As mentioned earlier, to save time, we are performing most of the operations in this lab via pre-written SQL executed in the worksheet as opposed to using the UI.

Run the query by placing your cursor anywhere in the SQL text and clicking the blue **Play/Run** button in the top right of the worksheet. Or use the keyboard shortcut [Ctrl]/[Cmd]+[Enter].

Verify that your TRIPS table has been created. At the bottom of the worksheet you should see a Results section displaying a "Table TRIPS successfully created" message.

The screenshot shows a Snowflake Worksheet interface. The top navigation bar includes a 'CITIBIKE_ZERO_TO_SNOWFLAKE' database dropdown, a '+' icon for creating new worksheets, a user dropdown for 'SYSADMIN - COMPUTE_WH', a 'Share' button, and a timestamp 'Updated 1 minute ago'. The main area is divided into two sections: 'Pinned (0)' on the left and 'CITIBIKE.PUBLIC +' on the right. In the 'CITIBIKE.PUBLIC +' section, a code editor contains the SQL command to create the 'trips' table. The code is as follows:

```

create or replace table trips
(tripduration integer,
starttime timestamp,
stoptime timestamp,
start_station_id integer,
start_station_name string,
start_station_latitude float,
start_station_longitude float,
end_station_id integer,
end_station_name string,
end_station_latitude float,
end_station_longitude float,
bikeid integer,
membership_type string,
usertype string,
birth_year integer,
gender integer);

```

The code editor has line numbers from 1 to 18. Below the code editor is a results panel titled 'status' which displays the message 'Table TRIPS successfully created.' To the right of the results panel is a 'Query Details' sidebar showing a green progress bar for 'Query duration' at 277ms, 1 row processed, and a status of '100% filled'.

Navigate to the **Databases** tab by clicking the **HOME** icon in the upper left corner of the worksheet. Then click **Data** > **Databases**. In the list of databases, click **CITIBIKE** > **PUBLIC** > **TABLES** to see your newly created **TRIPS** table. If you don't see any databases on the left, expand your browser because they may be hidden.

The screenshot shows the Snowflake interface. On the left, the sidebar is visible with the user 'SF_USER' and the 'Databases' tab selected. In the main area, under the 'CITIBIKE' database and 'PUBLIC' schema, a new table named 'TRIPS' is being created. The table has one row and 0 bytes.

Click `TRIPS` and the **Columns** tab to see the table structure you just created.

The screenshot shows the 'Columns' tab for the 'TRIPS' table. It lists 16 columns:

NAME	TYPE	NULLABLE	DEFAULT
BIKEID	NUMBER(38,0)	Yes	NULL
BIRTH_YEAR	NUMBER(38,0)	Yes	NULL
END_STATION_ID	NUMBER(38,0)	Yes	NULL
END_STATION_LATITUDE	FLOAT	Yes	NULL
END_STATION_LONGITUDE	FLOAT	Yes	NULL
END_STATION_NAME	VARCHAR(16777216)	Yes	NULL
GENDER	NUMBER(38,0)	Yes	NULL
MEMBERSHIP_TYPE	VARCHAR(16777216)	Yes	NULL
STARTTIME	TIMESTAMP_NTZ(9)	Yes	NULL
START_STATION_ID	NUMBER(38,0)	Yes	NULL
START_STATION_LATITUDE	FLOAT	Yes	NULL
START_STATION_LONGITUDE	FLOAT	Yes	NULL
START_STATION_NAME	VARCHAR(16777216)	Yes	NULL
STOPTIME	TIMESTAMP_NTZ(9)	Yes	NULL
TRIPDURATION	NUMBER(38,0)	Yes	NULL
USERTYPE	VARCHAR(16777216)	Yes	NULL

Create an External Stage

We are working with structured, comma-delimited data that has already been staged in a public, external S3 bucket. Before we can use this data, we first need to create a Stage that specifies the location of our external bucket.

Positive : For this lab we are using an AWS-East bucket. To prevent data egress/transfer costs in the future, you should select a staging location from the same cloud provider and region as your Snowflake account.

From the **Databases** tab, click the `CITIBIKE` database and `PUBLIC` schema. In the **Stages** tab, click the **Create** button, then **Stage > Amazon S3**.

The screenshot shows the Snowflake interface. On the left, the sidebar has 'SF_USER' selected under 'Databases'. The main area shows the 'CITIBIKE / PUBLIC' schema with a 'Tables' section containing 'TRIPS'. A context menu is open over the 'Stages' section, listing options like 'Table', 'View', 'Stage', 'Pipe', 'Stream', 'Task', 'Function', and 'Procedure'. Below this, another context menu is open for 'External Stage', listing 'Amazon S3', 'Microsoft Azure', and 'Google Cloud Platform'. At the bottom, a message says 'No stages' and 'This schema contains no stages.'

In the "Create Securable Object" dialog that opens, replace the following values in the SQL statement:

```
stage_name : citibike_trips

url : s3://snowflake-workshop-lab/citibike-trips-csv/
```

Note: Make sure to include the final forward slash (/) at the end of the URL or you will encounter errors later when loading data from the bucket. Also ensure you have removed 'credentials = (...) statejment which is not required. The create stage command should resemble that show above exactly.

Positive : The S3 bucket for this lab is public so you can leave the credentials options in the statement empty. In a real-world scenario, the bucket used for an external stage would likely require key information.

The screenshot shows the 'Create Securable Object' dialog. The left sidebar shows pinned objects and a search bar. The main area contains the following SQL code:

```
create stage citibike_trips
url = 's3://snowflake-workshop-lab/citibike-trips/';
--credentials = (aws_secret_key = '<key>' aws_key_id = '<id>');
```

At the top right, there are buttons for 'Cancel', 'Create Stage', and 'Open in Worksheets'. At the bottom, there are tabs for 'Objects' and 'Query'.

Now let's take a look at the contents of the `citibike_trips` stage. Navigate to the **Worksheets** tab and execute the following SQL statement:

```
list @citibike_trips;
```

In the results in the bottom pane, you should see the list of files in the stage:

The screenshot shows the Snowflake UI with the following details:

- Top Bar:** CITIBIKE_ZERO_TO_SNOWFLAKE •, +, SYSADMIN, COMPUTE_WH, Share, Updated 7 seconds ago.
- Left Sidebar:** Pinned (0), No pinned objects, Q, Search (...), CITIBIKE.
- Middle Panel:** A code editor window titled "List @citibike_trips:" containing the following SQL code:

```
1 create or replace table trips
2   (tripduration integer,
3    starttime timestamp,
4    stoptime timestamp,
5    start_station_id integer,
6    start_station_name string,
7    start_station_latitude float,
8    start_station_longitude float,
9    end_station_id integer,
10   end_station_name string,
11   end_station_latitude float,
12   end_station_longitude float,
13   bikeid integer,
14   membership_type string,
15   usertype string,
16   birth_year integer,
17   gender integer);
18
19
20 | List @citibike_trips:
```
- Bottom Panel:** An object list view showing 12 files in the stage. The columns are name, size, md5, and last_modified. The data is as follows:

name	size	md5	last_modified
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/03/data_01a19496-c	219,733	5c4e4f21988b692a8cea379cdd094aca	Wed, 12 Jan 2022 13:10:38 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/04/data_01a19496-c	240,490	920744132d8da7cd8abbfd11465d8	Wed, 12 Jan 2022 13:10:37 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/05/data_01a19496-c	227,228	d982d1a2e3692abea51917f134e3925	Wed, 12 Jan 2022 13:10:36 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/06/data_01a19496-c	235,713	1ccc34af2eeb9c3eab0771f3ae193167	Wed, 12 Jan 2022 13:10:34 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/07/data_01a19496-c	224,639	82fe1f21949abeb6bd2c2e4579cfec3d3	Wed, 12 Jan 2022 13:10:39 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/08/data_01a19496-c	238,398	2d6cc208b20245127bc65beb1938	Wed, 12 Jan 2022 13:10:35 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/09/data_01a19496-c	227,747	4a6b6bcc6d3b3b379e23da4a466fc1a8c	Wed, 12 Jan 2022 13:10:40 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/10/data_01a19496-c	249,432	fe216ba3f84d7dfb52919986194c6	Wed, 12 Jan 2022 13:10:33 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/11/data_01a19496-c	285,569	c057bbb681d8420e42d86a24777d8	Wed, 12 Jan 2022 13:10:40 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/12/data_01a19496-c	261,702	91cb6989f12e2c83ff70e75aa2a071c	Wed, 12 Jan 2022 13:10:32 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/13/data_01a19496-c	269,385	1b6d78c7f48d8995ac76e85278ef1d2a2	Wed, 12 Jan 2022 13:10:40 GMT
s3://snowflake-workshop-lab/citibike-trips-parquet/2013/06/14/data_01a19496-c	266,097	d85727a36ea0df3e7f5ebe1b725863be	Wed, 12 Jan 2022 13:10:38 GMT

Query Details: Duration 3.6s, Rows 3.5K, 100% filled. Size 123, 4,898 to 9,720,801. MD5 100% filled.

Create a File Format

Before we can load the data into Snowflake, we have to create a file format that matches the data structure.

In the worksheet, run the following command to create the file format:

```
--create file format

create or replace file format csv type='csv'
compression = 'auto' field_delimiter = ',' record_delimiter = '\n'
skip_header = 0 field_optionally_enclosed_by = '\042' trim_space = false
error_on_column_count_mismatch = false escape = 'none' escape_unenclosed_field =
'\134'

date_format = 'auto' timestamp_format = 'auto' null_if = ('') comment = 'file format
for ingesting data for zero to snowflake';
```

The screenshot shows the Snowflake UI interface. At the top, there's a navigation bar with 'CITIBIKE_ZERO_TO_SNOWFLAKE', 'SYSADMIN', 'COMPUTE_WH', 'Share', and a 'Updated 4 seconds ago' timestamp. Below the navigation is a sidebar with sections for 'Pinned', 'No pinned objects', 'Search', and 'CITIBIKE'. The main area contains a code editor with the following SQL script:

```

5   starttime timestamp,
6   stoptime timestamp,
7   start.station_id integer,
8   start.station_name string,
9   start.station.latitude float,
10  start.station.longitude float,
11  end.station_id integer,
12  end.station.name string,
13  end.station.latitude float,
14  end.station.longitude float,
15  bikeid integer,
16  membership_type string,
17  usertype string,
18  birth_year integer,
19  gender integer
20 );
21 --created external stage. List files.
22 list @citibike_trips;
23
24 --create file format
25 CREATE FILE FORMAT "CITIBIKE"."PUBLIC" CSV TYPE = 'CSV' COMPRESSION = 'AUTO' FIELD_DELIMITER = ',' RECORD_DELIMITER = '\n' SKIP_HEADER = 0
FIELD_OPTIONALLY_ENCLOSED_BY = '\"' TRIM_SPACE = FALSE ERROR_ON_COLUMN_COUNT_MISMATCH = TRUE ESCAPE = 'NONE' ESCAPE_UNENCLOSED_FIELD = '\\"' DATE_FORMAT = 'AUTO'
TIMESTAMP_FORMAT = 'AUTO' NULL_IF = ('') COMMENT = 'creation of file format for zero to snowflake';

```

Below the code editor is a results table with one row:

	status
1	File format CSV successfully created.

On the right side of the results table, there's a 'Query Details' panel showing 'Query duration: 123ms', 'Rows: 1', and 'status: 100% filled'.

Verify that the file format has been created with the correct settings by executing the following command:

```
--verify file format is created

show file formats in database citibike;
```

The file format created should be listed in the result:

The screenshot shows the Snowflake UI interface. At the top, there's a navigation bar with 'CITIBIKE_ZERO_TO_SNOWFLAKE', 'SYSADMIN', 'COMPUTE_WH', 'Share', and a 'Updated 2 seconds ago' timestamp. Below the navigation is a sidebar with sections for 'Pinned', 'No pinned objects', 'Search', and 'CITIBIKE'. The main area contains a code editor with the following SQL script:

```

8   start.station.name string,
9   start.station.latitude float,
10  start.station.longitude float,
11  end.station_id integer,
12  end.station.name string,
13  end.station.latitude float,
14  end.station.longitude float,
15  bikeid integer,
16  membership_type string,
17  usertype string,
18  birth_year integer,
19  gender integer
20 );
21 --created external stage. List files.
22 list @citibike_trips;
23
24 --create file format
25 CREATE FILE FORMAT "CITIBIKE"."PUBLIC" CSV TYPE = 'CSV' COMPRESSION = 'AUTO' FIELD_DELIMITER = ',' RECORD_DELIMITER = '\n' SKIP_HEADER = 0
FIELD_OPTIONALLY_ENCLOSED_BY = '\"' TRIM_SPACE = FALSE ERROR_ON_COLUMN_COUNT_MISMATCH = TRUE ESCAPE = 'NONE' ESCAPE_UNENCLOSED_FIELD = '\\"' DATE_FORMAT = 'AUTO'
TIMESTAMP_FORMAT = 'AUTO' NULL_IF = ('') COMMENT = 'creation of file format for zero to snowflake';
26
27 --verify file format is created
28 show file formats in database citibike;

```

Below the code editor is a results table with one row:

	created_on	name	database_name	schema_name	type	owner	comment	for
1	2022-01-20 11:12:27.666 -0800	CSV	CITIBIKE	PUBLIC	CSV	SYSADMIN	creation of file format for zero to snowflake	(T)

On the right side of the results table, there's a 'Query Details' panel showing 'Query duration: 97ms', 'Rows: 1', and three detailed status rows for 'created_on', 'name', and 'database_name', all showing '100% filled'.

Loading Data

In this section, we will use a virtual warehouse and the COPY command to initiate bulk loading of structured data into the Snowflake table we created in the last section.

Resize and Use a Warehouse for Data Loading

Compute resources are needed for loading data. Snowflake's compute nodes are called virtual warehouses and they can be dynamically sized up or out according to workload, whether you are loading data, running a query, or performing a DML operation. Each workload can have its own warehouse so there is no resource contention.

Navigate to the **Warehouses** tab (under **Admin**). This is where you can view all of your existing warehouses, as well as analyze their usage trends.

Note the **+ Warehouse** option in the upper right corner of the top. This is where you can quickly add a new warehouse. However, we want to use the existing warehouse COMPUTE_WH included in the 30-day trial environment.

Click the row of the `COMPUTE_WH` warehouse. Then click the ... (dot dot dot) in the upper right corner text above it to see the actions you can perform on the warehouse. We will use this warehouse to load the data from AWS S3.

The screenshot shows the Snowflake Admin interface. On the left, a sidebar navigation bar includes options like Worksheets, Dashboards, Data, Marketplace, Activity, Admin, Usage, Warehouses (which is selected), Resource Monitors, Users & Roles, Security, Billing, Contacts, Accounts, Partner Connect, Help & Support, and Classic Console. The main content area displays the details for the `COMPUTE_WH` warehouse. At the top, there's a breadcrumb trail: `COMPUTE_WH`, `Warehouse`, `SYSADMIN`, and `1 month ago`. Below this is a chart titled "Warehouse Activity" showing data load activity from Sep 6 to Sep 20. To the right of the chart is a context menu with options: Edit, Suspend, Drop, Transfer Ownership, and a "..." button. The "Edit" option is highlighted with a red box. Below the chart is a "Details" section with various configuration parameters. At the bottom of the main content area is a "Privileges" section.

Click **Edit** to walk through the options of this warehouse and learn some of Snowflake's unique functionality.

Positive : If this account isn't using Snowflake Enterprise Edition (or higher), you will not see the **Mode** or **Clusters** options shown in the screenshot below. The multi-cluster warehouses feature is not used in this lab, but we will discuss it as a key capability of Snowflake.

Edit Warehouse

COMPUTE_WH as SYSADMIN

Name	Size ?
COMPUTE_WH	X-Large 16 credits/hour
Comment (optional)	
<hr/>	
Multi-cluster Warehouse	
Scale compute resources as query needs change	
Mode	Maximized
Clusters	1
<hr/>	
Advanced Warehouse Options ▾	
Auto Resume	On
Auto Suspend	On
Suspend After (min)	10
<hr/>	
<button>Cancel</button>	<button>Save Warehouse</button>

- The **Size** drop-down is where the capacity of the warehouse is selected. For larger data loading operations or more compute-intensive queries, a larger warehouse is recommended. The sizes translate to the underlying compute resources provisioned from the cloud provider (AWS, Azure, or GCP) where your Snowflake account is hosted. It also determines the number of credits consumed by the warehouse for each full hour it runs. The larger the size, the more compute resources from the cloud provider are allocated to

the warehouse and the more credits it consumes. For example, the `4X-Large` setting consumes 128 credits for each full hour. This sizing can be changed up or down at any time with a simple click.

- If you are using Snowflake Enterprise Edition (or higher) and the **Multi-cluster Warehouse** option is enabled, you will see additional options. This is where you can set up a warehouse to use multiple clusters of compute resources, up to 10 clusters. For example, if a `4X-Large` multi-cluster warehouse is assigned a maximum cluster size of 10, it can scale out to 10 times the compute resources powering that warehouse...and it can do this in seconds! However, note that this will increase the number of credits consumed by the warehouse to 1280 if all 10 clusters run for a full hour (128 credits/hour x 10 clusters). Multi-cluster is ideal for concurrency scenarios, such as many business analysts simultaneously running different queries using the same warehouse. In this use case, the various queries are allocated across multiple clusters to ensure they run quickly.
- Under **Advanced Warehouse Options**, the options allow you to automatically suspend the warehouse when not in use so no credits are needlessly consumed. There is also an option to automatically resume a suspended warehouse so when a new workload is sent to it, it automatically starts back up. This functionality enables Snowflake's efficient "pay only for what you use" billing model which allows you to scale your resources when necessary and automatically scale down or turn off when not needed, nearly eliminating idle resources.

Negative : **Snowflake Compute vs Other Data Warehouses** Many of the virtual warehouse and compute capabilities we just covered, such as the ability to create, scale up, scale out, and auto-suspend/resume virtual warehouses are easy to use in Snowflake and can be done in seconds. For on-premise data warehouses, these capabilities are much more difficult, if not impossible, as they require significant physical hardware, over-provisioning of hardware for workload spikes, and significant configuration work, as well as additional challenges. Even other cloud-based data warehouses cannot scale up and out like Snowflake without significantly more configuration work and time.

Warning - Watch Your Spend! During or after this lab, you should be careful about performing the following actions without good reason or you may burn through your \$400 of free credits more quickly than desired:

- Do not disable auto-suspend. If auto-suspend is disabled, your warehouses continues to run and consume credits even when not in use.
- Do not use a warehouse size that is excessive given the workload. The larger the warehouse, the more credits are consumed.

We are going to use this virtual warehouse to load the structured data in the CSV files (stored in the AWS S3 bucket) into Snowflake. However, we are first going to change the size of the warehouse to increase the compute resources it uses. After the load, note the time taken and then, in a later step in this section, we will re-do the same load operation with an even larger warehouse, observing its faster load time.

Change the **Size** of this data warehouse from `X-Small` to `Small`. then click the **Save Warehouse** button:

Edit Warehouse

COMPUTE_WH as SYSADMIN

Name: COMPUTE_WH

Size: Small 2 credits/hour

Comment (optional):

Multi-cluster Warehouse: Scale compute resources as query needs change

Mode: Maximized

Clusters: 1

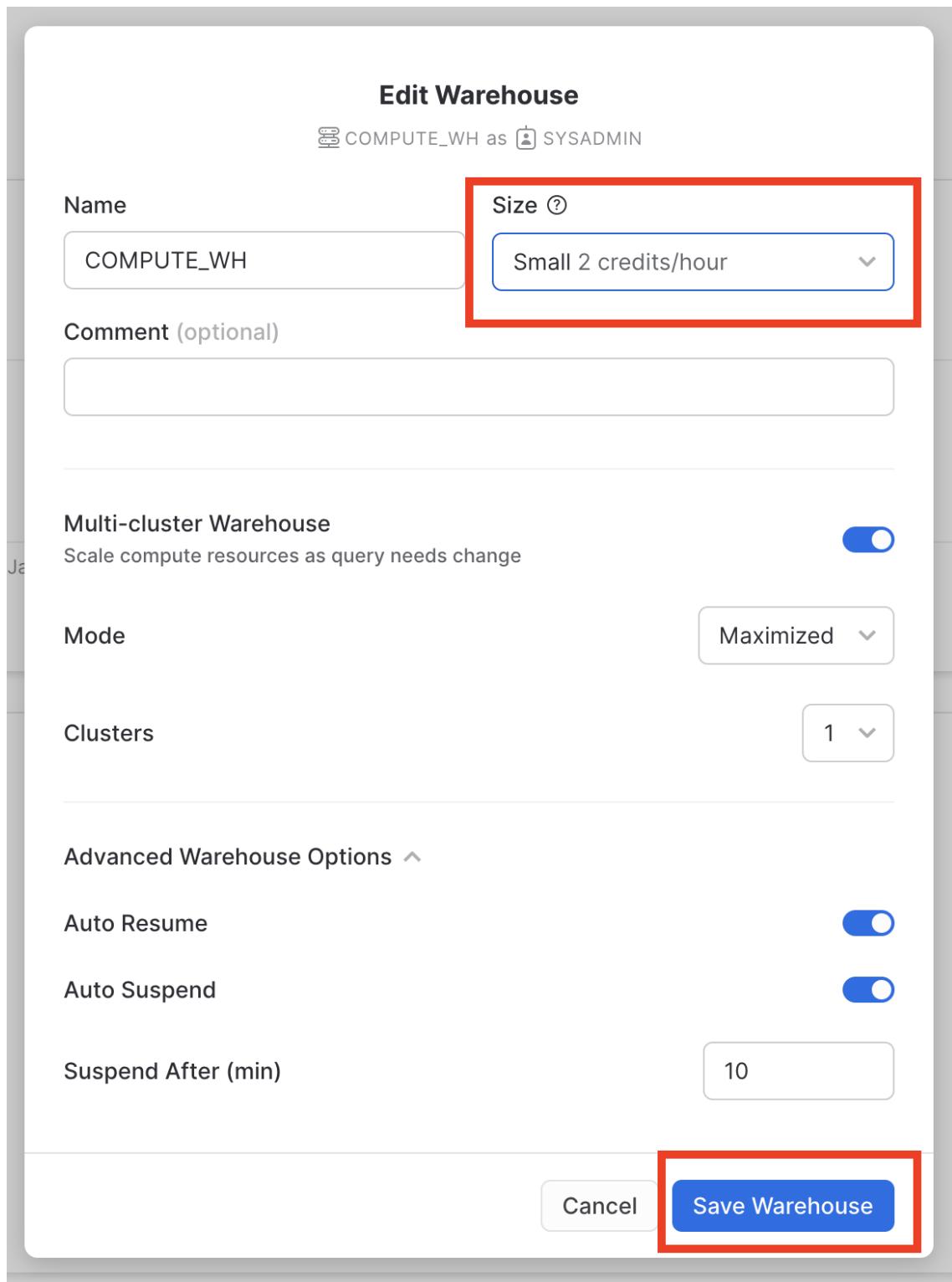
Advanced Warehouse Options ▾

Auto Resume:

Auto Suspend:

Suspend After (min): 10

Cancel **Save Warehouse**



Load the Data

Now we can run a COPY command to load the data into the `TRIPS` table we created earlier.

Navigate back to the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet in the **Worksheets** tab. Make sure the worksheet context is correctly set:

Role: `SYSADMIN` Warehouse: `COMPUTE_WH` Database: `CITIBIKE` Schema = `PUBLIC`

Execute the following statements in the worksheet to load the staged data into the table. This may take up to 30 seconds.

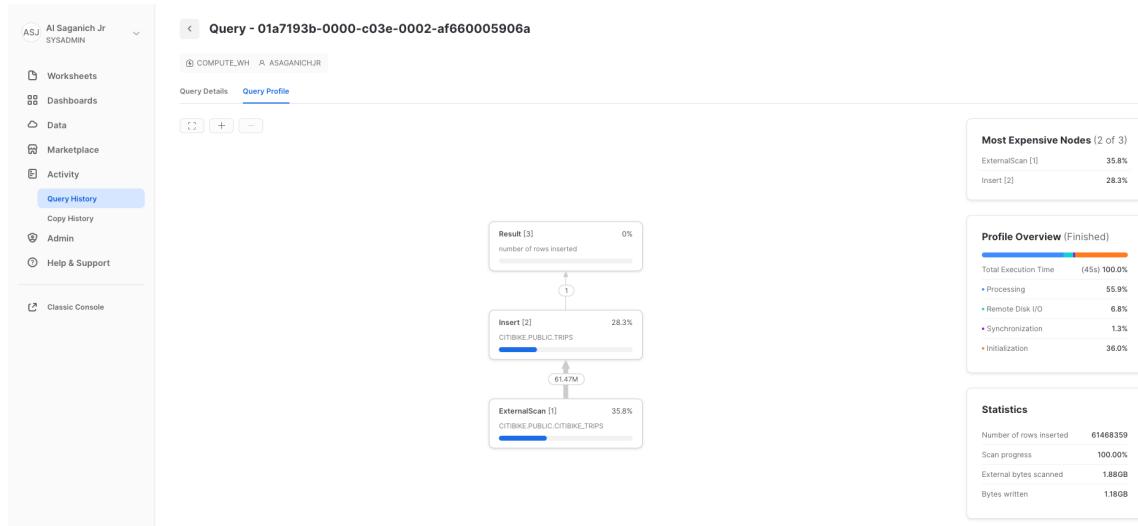
```
copy into trips from @citibike_trips file_format=csv PATTERN = '.*csv.*' ;
```

In the result pane, you should see the status of each file that was loaded. Once the load is done, in the **Query Details** pane on the bottom right, you can scroll through the various statuses, error statistics, and visualizations for the last statement executed:

file	status	rows_parsed	rows_loaded	error_limit	errors_seen	first_error
1 s3://snowflake-workshop-lab/citibike-trips/trips_2013_1_1_0.csv.gz	LOADED	112,123	112,123	1	0	
2 s3://snowflake-workshop-lab/citibike-trips/trips_2013_2_7_0.csv.gz	LOADED	93,143	93,143	1	0	
3 s3://snowflake-workshop-lab/citibike-trips/trips_2013_5_7_0.csv.gz	LOADED	111,042	111,042	1	0	
4 s3://snowflake-workshop-lab/citibike-trips/trips_2014_0_7_0.csv.gz	LOADED	112,334	112,334	1	0	
5 s3://snowflake-workshop-lab/citibike-trips/trips_2014_3_0_0.csv.gz	LOADED	126,789	126,789	1	0	
6 s3://snowflake-workshop-lab/citibike-trips/trips_2014_4_7_0.csv.gz	LOADED	109,835	109,835	1	0	
7 s3://snowflake-workshop-lab/citibike-trips/trips_2014_6_7_0.csv.gz	LOADED	106,608	106,608	1	0	
8 s3://snowflake-workshop-lab/citibike-trips/trips_2015_0_2_0.csv.gz	LOADED	125,082	125,082	1	0	
9 s3://snowflake-workshop-lab/citibike-trips/trips_2015_1_2_0.csv.gz	LOADED	154,940	154,940	1	0	

Next, navigate to the **Query History** tab by clicking the **Home** icon and then **Activity > Query History**. Select the query at the top of the list, which should be the COPY INTO statement that was last executed. Select the **Query**

Profile tab and note the steps taken by the query to execute, query details, most expensive nodes, and additional statistics.



Now let's reload the `TRIPS` table with a larger warehouse to see the impact the additional compute resources have on the loading time.

Go back to the worksheet and use the `TRUNCATE TABLE` command to clear the table of all data and metadata:

```
truncate table trips;
```

Verify that the table is empty by running the following command:

```
--verify table is clear
select * from trips limit 10;
```

The result should show "Query produced no results".

Change the warehouse size to `large` using the following `ALTER WAREHOUSE`:

```
--change warehouse size from small to large (4x)
alter warehouse compute_wh set warehouse_size='large';
```

Verify the change using the following `SHOW WAREHOUSES`:

```
--load data with large warehouse
show warehouses;
```

CITIBIKE_ZERO_TO_SNOWFLAKE

```

copy into trips from @citibike_trips file_format=csv;
--clear table and metadata on table trips for next test
truncate table trips;
--verify table is clear
select * from trips limit 10;
--change warehouse size from small to large (4x)
alter warehouse compute_wh set warehouse_size='large';
--verify large warehouse
show warehouses;

```

Query Details

	Value
Query duration	92ms
Rows	1

The size can also be changed using the UI by clicking on the worksheet context box, then the **Configure** (3-line) icon on the right side of the context box, and changing `Small` to `Large` in the **Size** drop-down:

The screenshot shows two adjacent tabs in the Snowflake interface. The left tab, titled 'Roles', lists several user roles: SYSADMIN (selected), ANALYST_CITIBIKE, DBA_CITIBIKE, DEV_CITIBIKE, and PUBLIC. The right tab, titled 'Warehouses', shows a single warehouse named 'COMPUTE_WH'.

This screenshot displays the configuration details for the 'COMPUTE_WH' warehouse. The 'Size' dropdown menu is open, showing various cluster sizes: X-Small, Small, Medium, Large (selected), X-Large, 2X-Large, 3X-Large, 4X-Large, and 5X-Large. The 'Large' option is currently selected.

Execute the same COPY INTO statement as before to load the same data again:

```
copy into trips from @citibike_trips  
file_format=CSV;
```

The screenshot shows the Snowflake UI interface. At the top, there's a navigation bar with icons for Home, Activity, and Query History. Below it is a search bar and a sidebar with sections for Pinned objects, No pinned objects, and a Search bar. The main area contains a code editor with a query script. The script includes comments like `--VERIFY 'CUBE' IS CLEAR;` and `--change warehouse size from small to large (4x);` followed by an `ALTER WAREHOUSE` command. A specific line, `copy into trips from @citibike_trips file_format=csv;`, is highlighted in blue. To the right of the code editor is a progress bar indicating the query is running. Below the code editor is a results table with columns: file, status, rows_parsed, rows_loaded, error_limit, errors_seen, first_error, and Query Details. The table lists 18 rows, each corresponding to a CSV file from S3. The status column shows all rows as LOADED. The Query Details section on the right provides a summary with a progress bar for duration (9.8s), a bar chart for rows (376), and histograms for status, rows_parsed (123), rows_loaded (123), and error_limit (123). The status histogram shows 100% filled.

file	status	rows_parsed	rows_loaded	error_limit	errors_seen	first_error	Query Details
1 s3://snowflake-workshop-lab/citibike-trips/trips_2013_1_0.csv.gz	LOADED	117,943	117,943	1	0		Query duration: 9.8s
2 s3://snowflake-workshop-lab/citibike-trips/trips_2013_1_3_0.csv.gz	LOADED	89,057	89,057	1	0		Rows: 376
3 s3://snowflake-workshop-lab/citibike-trips/trips_2014_1_5_0.csv.gz	LOADED	135,516	135,516	1	0		100% filled
4 s3://snowflake-workshop-lab/citibike-trips/trips_2014_1_3_0.csv.gz	LOADED	158,108	158,108	1	0		status: LOADED
5 s3://snowflake-workshop-lab/citibike-trips/trips_2015_1_1_0.csv.gz	LOADED	168,725	168,725	1	0		rows_parsed: 123
6 s3://snowflake-workshop-lab/citibike-trips/trips_2015_4_1_0.csv.gz	LOADED	141,423	141,423	1	0		rows_loaded: 123
7 s3://snowflake-workshop-lab/citibike-trips/trips_2015_7_5_0.csv.gz	LOADED	156,016	156,016	1	0		error_limit: 123
8 s3://snowflake-workshop-lab/citibike-trips/trips_2016_6_5_0.csv.gz	LOADED	189,709	189,709	1	0		100% filled
9 s3://snowflake-workshop-lab/citibike-trips/trips_2017_6_3_0.csv.gz	LOADED	266,912	266,912	1	0		
10 s3://snowflake-workshop-lab/citibike-trips/trips_2018_7_1_0.csv.gz	LOADED	111,386	111,386	1	0		
11 s3://snowflake-workshop-lab/citibike-trips/trips_2018_7_1_0.csv.gz	LOADED	124,464	124,464	1	0		
12 s3://snowflake-workshop-lab/citibike-trips/trips_2013_4_4_0.csv.gz	LOADED	104,107	104,107	1	0		
13 s3://snowflake-workshop-lab/citibike-trips/trips_2016_7_4_0.csv.gz	LOADED	121,945	121,945	1	0		
14 s3://snowflake-workshop-lab/citibike-trips/trips_2013_4_0_0.csv.gz	LOADED	85,271	85,271	1	0		
15 s3://snowflake-workshop-lab/citibike-trips/trips_2014_2_3_0.csv.gz	LOADED	102,555	102,555	1	0		
16 s3://snowflake-workshop-lab/citibike-trips/trips_2015_3_2_0.csv.gz	LOADED	125,585	125,585	1	0		
17 s3://snowflake-workshop-lab/citibike-trips/trips_2014_4_5_0.csv.gz	LOADED	149,304	149,304	1	0		
18 s3://snowflake-workshop-lab/citibike-trips/trips_2018_2_2_0.csv.gz	LOADED	218,112	218,112	1	0		

Once the load is done, navigate back to the **Queries** page (**Home** icon > **Activity** > **Query History**). Compare the times of the two COPY INTO commands. The load using the `Large` warehouse was significantly faster.

Create a New Warehouse for Data Analytics

Going back to the lab story, let's assume the Citi Bike team wants to eliminate resource contention between their data loading/ETL workloads and the analytical end users using BI tools to query Snowflake. As mentioned earlier, Snowflake can easily do this by assigning different, appropriately-sized warehouses to various workloads. Since Citi Bike already has a warehouse for data loading, let's create a new warehouse for the end users running analytics. We will use this warehouse to perform analytics in the next section.

Navigate to the **Admin** > **Warehouses** tab, click **+ Warehouse**, and name the new warehouse `` and set the size to `Large`.

If you are using Snowflake Enterprise Edition (or higher) and **Multi-cluster Warehouses** is enabled, you will see additional settings:

- Make sure **Max Clusters** is set to `1`.
- Leave all the other settings at their defaults.

The screenshot shows the Snowflake Web UI interface. On the left, there's a sidebar with navigation links like Worksheets, Dashboards, Data, Compute, History, Warehouses (which is selected), Resource Monitors, and Account. The main area is titled 'Warehouses' and shows a table with one row: '1 Warehouse'. The row contains 'COMPUTE_WH' under 'NAME' and 'Started' under 'STATUS'. A modal window titled 'New Warehouse' is open, prompting for a name ('ANALYTICS_WH'), size ('Large 8 credits/hour'), and a comment ('analytics warehouse for zero to snowflake'). It also includes settings for 'Multi-cluster Warehouse' (auto-scale is turned on), 'Mode' (Auto-scale), 'Min Clusters' (1), 'Max Clusters' (1), and 'Scaling Policy' (Standard). Under 'Advanced Warehouse Options', 'Auto Resume' and 'Auto Suspend' are turned on, with a 'Suspend After (min)' set to 10. At the bottom of the modal are 'Cancel' and 'Create Warehouse' buttons.

Click the **Create Warehouse** button to create the warehouse.

Working with Queries, the Results Cache, & Cloning

In the previous exercises, we loaded data into two tables using Snowflake's COPY bulk loader command and the `COMPUTE_WH` virtual warehouse. Now we are going to take on the role of the analytics users at Citi Bike who need to query data in those tables using the worksheet and the second warehouse `ANALYTICS_WH`.

Negative : Real World Roles and Querying Within a real company, analytics users would likely have a different role than SYSADMIN. To keep the lab simple, we are going to stay with the SYSADMIN role for this section. Additionally, querying would typically be done with a business intelligence product like Tableau, Looker, PowerBI, etc. For more advanced analytics, data science tools like Datarobot, Dataiku, AWS Sagemaker or many others can query Snowflake. Any technology that leverages JDBC/ODBC, Spark, Python, or any of the other supported programmatic interfaces can run analytics on the data in Snowflake. To keep this lab simple, all queries are being executed via the Snowflake worksheet.

Execute Some Queries

Go to the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet and change the warehouse to use the new warehouse you created in the last section. Your worksheet context should be the following:

Role: `SYSADMIN` Warehouse: `ANALYTICS_WH` (L) Database: `CITIBIKE` Schema = `PUBLIC`

The screenshot shows the Snowflake Worksheet interface. At the top, it displays the context: Role: `SYSADMIN`, Warehouse: `ANALYTICS_WH` (L), Database: `CITIBIKE`, Schema = `PUBLIC`. Below this, a context menu is open with two dropdowns. The 'Role' dropdown is set to `SYSADMIN` and has other options like `ANALYST_CITIBIKE`, `DBA_CITIBIKE`, `DEV_CITIBIKE`, and `PUBLIC`. The 'Warehouse' dropdown is set to `ANALYTICS_WH` and has other options like `COMPUTE_WH`. In the code editor area, there is a snippet of SQL:

```

CITIBIKE.PUBLIC <
1 create or replace table trips (
2     tripduration integer,
3     starttime timestamp,
4     stoptime timestamp,
5     start_station_id integer,
6     start_station_name string,
7     start_station_latitude float,
8     start_station_longitude float,
9     end_station_id integer,
10    end_station_name string
11);
12
13
14
15
16
17

```

Run the following query to see a sample of the `trips` data:

```
select * from trips limit 20;
```

CITIBIKE_ZERO_TO_SNOWFLAKE + SYSDADMIN · ANALYTICS_WH Share Updated 43 seconds ago

```
40
41 --clear table and metadata on table trips for next test
42 truncate table trips;
43
44 --verify table is clear
45 select * from trips limit 10;
46
47 --change warehouse size from small to large (4x)
48 alter warehouse compute_wh set warehouse_size='large';
49
50 --verify Large warehouse
51 show warehouses;
52
53 --load data in with large warehouse
54 copy into trips from @citibike_trips file_format=csv;
55
56 --preview trips data
57 select * from trips limit 20;
58
59
60
```

Objects Query Results Chart

TRIPDURATION	STARTTIME	STOPTIME	START_STATION_ID	START_STATION_NAME	START_STATION_LATIT
1	258 2018-04-03 18:31:11.000	2018-04-03 18:35:30.000	3,684	North Moore St & Greenwich St	40.720195
2	659 2018-04-03 18:31:12.000	2018-04-03 18:42:12.000	127	Barrow St & Hudson St	40.731724
3	1,629 2018-04-03 18:31:13.000	2018-04-03 18:58:22.000	3,002	South End Ave & Liberty St	40.7111!
4	286 2018-04-03 18:31:16.000	2018-04-03 18:36:05.000	465	Broadway & W 41 St	40.75513!
5	1,097 2018-04-03 18:31:19.000	2018-04-03 18:49:36.000	305	E 58 St & 3 Ave	40.76095
6	346 2018-04-03 18:31:19.000	2018-04-03 18:37:06.000	526	E 33 St & 5 Ave	40.74765!
7	313 2018-04-03 18:31:22.000	2018-04-03 18:36:35.000	519	Pershing Square North	40.7511
8	421 2018-04-03 18:31:24.000	2018-04-03 18:38:26.000	3,258	W 27 St & 10 Ave	40.750181!
9	1,185 2018-04-03 18:31:25.000	2018-04-03 18:51:10.000	3,263	Cooper Square & Astor Pl	40.729514!
10	272 2018-04-03 18:31:26.000	2018-04-03 18:35:59.000	3,140	1 Ave & E 79 St	40.77140+
11	509 2018-04-03 18:31:27.000	2018-04-03 18:39:56.000	528	2 Ave & E 31 St	40.74290!
12	454 2018-04-03 18:31:29.000	2018-04-03 18:39:04.000	248	Laight St & Hudson St	40.72185!
13	2,268 2018-04-03 18:31:34.000	2018-04-03 19:09:20.000	334	W 20 St & 7 Ave	40.74238!
14	358 2018-04-03 18:31:34.000	2018-04-03 18:37:32.000	418	Front St & Gold St	40.70

Query Details ...
Query duration 1.2s
Rows 20
TRIPDURATION 123
228 2,502
STARTTIME ○
100% filled
STOPTIME ○
100% filled
START_STATION_ID 123

Now, let's look at some basic hourly statistics on Citi Bike usage. Run the query below in the worksheet. For each hour, it shows the number of trips, average trip duration, and average trip distance.

```
select date_trunc('hour', starttime) as "date",
count(*) as "num trips",
avg(tripduration)/60 as "avg duration (mins)",
avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude,
end_station_longitude)) as "avg distance (km)"
from trips
group by 1 order by 1;
```

```

51 show warehouses;
52
53 --load data in with Large warehouse
54 copy into trips from @citibike_trips file_format=csv;
55
56 --preview trips data
57 select * from trips limit 20;
58
59 --hourly statistics on CITIBIKE usage
60 select date_trunc('hour', starttime) as "date",
61 count(*) as "num trips",
62 avg(tripduration)/60 as "avg duration (mins)",
63 avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude, end_station_longitude)) as "avg distance (km)"
64
65
66
67
68
69

```

	date	num trips	avg duration (mins)	avg distance (km)
1	2013-06-01 00:00:00	152	56.058442983333	2.127971476
2	2013-06-01 01:00:00	102	26.5251634	2.067906273
3	2013-06-01 02:00:00	67	36.1199005	2.31784827
4	2013-06-01 03:00:00	41	44.48536585	2.349126632
5	2013-06-01 04:00:00	16	23.278125	1.840026007
6	2013-06-01 05:00:00	13	34.584615383333	3.337844489
7	2013-06-01 06:00:00	40	22.3975	2.832927896
8	2013-06-01 07:00:00	93	66.397849466667	2.691774983
9	2013-06-01 08:00:00	177	18.530225983333	2.244399992
10	2013-06-01 09:00:00	280	40.2610119	2.292639556
11	2013-06-01 10:00:00	375	28.673466666667	2.22576826
12	2013-06-01 11:00:00	473	35.63520085	2.26058226
13	2013-06-01 12:00:00	604	33.8763521	2.177910979
14	2013-06-01 13:00:00	638	48.623093	2.270395192
15	2013-06-01 14:00:00	688	34.7400436	2.271981468

Query Details:

- Query duration: 1.3s
- Rows: 44.3K
- Date histogram: 2013-06-01 to 2018-08-30
- num trips histogram: 1 to 8,810
- avg duration (mins) histogram: 2.211111166667 to 5,898.822772866667
- avg distance (km) histogram: 123

Use the Result Cache

Snowflake has a result cache that holds the results of every query executed in the past 24 hours. These are available across warehouses, so query results returned to one user are available to any other user on the system who executes the same query, provided the underlying data has not changed. Not only do these repeated queries return extremely fast, but they also use no compute credits.

Let's see the result cache in action by running the exact same query again.

```

select date_trunc('hour', starttime) as "date",
count(*) as "num trips",
avg(tripduration)/60 as "avg duration (mins)",
avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude,
end_station_longitude)) as "avg distance (km)"
from trips
group by 1 order by 1;

```

In the **Query Details** pane on the right, note that the second query runs significantly faster because the results have been cached.

CITIBIKE_ZERO_TO_SNOWFLAKE

Pinned (0)
No pinned objects
Q Search CITIBIKE

```

61 count(*) as "num trips",
62 avg(tripduration)/60 as "avg duration (mins)",
63 avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude, end_station_longitude)) as "avg distance (km)"
64 from trips
65 group by 1 order by 1;
66
67 --run same query again to see performance improvement due to cache
68 select date_trunc('hour', starttime) as "date",
69 count(*) as "num trips",
70 avg(tripduration)/60 as "avg duration (mins)",
71 avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude, end_station_longitude)) as "avg distance (km)"
72 from trips
73 group by 1 order by 1;
74
75
76
77
78
79
80
81
82
83
84

```

Objects Query Results Chart

	date	num trips	...	avg duration (mins)	avg distance (km)
1	2013-06-01 00:00:00	152		56.058442983333	2.127971476
2	2013-06-01 01:00:00	102		26.5251634	2.067906273
3	2013-06-01 02:00:00	67		36.1199005	2.31794827
4	2013-06-01 03:00:00	41		44.48536585	2.349126632
5	2013-06-01 04:00:00	16		23.278125	1.840026007
6	2013-06-01 05:00:00	13		34.584615383333	3.337844489
7	2013-06-01 06:00:00	40		22.3975	2.832927896
8	2013-06-01 07:00:00	93		66.397849466667	2.691774983
9	2013-06-01 08:00:00	177		18.530225983333	2.244399992
10	2013-06-01 09:00:00	280		40.2610119	2.292639556
11	2013-06-01 10:00:00	375		28.673466666667	2.22576826
12	2013-06-01 11:00:00	473		35.63520085	2.26058226
13	2013-06-01 12:00:00	604		33.8763521	2.177910979
14	2013-06-01 13:00:00	638		48.623093	2.270395192
15	2013-06-01 14:00:00	688		34.7400436	2.271981468
...

Query Details

- Query duration: 55ms
- Rows: 44.3K
- date: Bar chart showing distribution from 2013-06-01 to 2018-08-30.
- num trips: Histogram showing distribution from 1 to 8,810.
- avg duration (mins): Histogram showing distribution from 2.11111166667 to 5,898.8227727286667.
- avg distance (km): Histogram showing distribution from 123 to 123.

Execute Another Query

Next, let's run the following query to see which months are the busiest:

```

select
monthname(starttime) as "month",
count(*) as "num trips"
from trips
group by 1 order by 2 desc;

```

CITIBIKE_ZERO_TO_SNOWFLAKE

Pinned (0)
No pinned objects
Q Search CITIBIKE

```

61 select date_trunc('hour', starttime) as "date",
62 count(*) as "num trips",
63 avg(tripduration)/60 as "avg duration (mins)",
64 avg(haversine(start_station_latitude, start_station_longitude, end_station_latitude, end_station_longitude)) as "avg distance (km)"
65 from trips
66 group by 1 order by 1;
67
68 --find busiest months
69 select
70 monthname(starttime) as "month",
71 count(*) as "num trips"
72 from trips
73 group by 1 order by 2 desc;
74
75
76
77
78
79
80
81
82
83
84

```

Objects Query Results Chart

month	num trips
Jun	7,800,765
Sep	6,804,899
Oct	6,550,164
Aug	6,516,652
May	6,388,309
Jul	6,013,644
Apr	4,959,244
Nov	4,719,798
Mar	3,405,178
Dec	3,349,319
Feb	2,817,291
Jan	2,541,096
...	...

Query Details

- Query duration: 1.2s
- Rows: 12
- month: Aa
- 100% filled
- num trips: Histogram showing distribution from 2,541,096 to 7,800,765.

Clone a Table

Snowflake allows you to create clones, also known as "zero-copy clones" of tables, schemas, and databases in seconds. When a clone is created, Snowflake takes a snapshot of data present in the source object and makes it available to the cloned object. The cloned object is writable and independent of the clone source. Therefore, changes made to either the source object or the clone object are not included in the other.

A popular use case for zero-copy cloning is to clone a production environment for use by Development & Testing teams to test and experiment without adversely impacting the production environment and eliminating the need to set up and manage two separate environments.

Negative : Zero-Copy Cloning A massive benefit of zero-copy cloning is that the underlying data is not copied. Only the metadata and pointers to the underlying data change. Hence, clones are "zero-copy" and storage requirements are not doubled when the data is cloned. Most data warehouses cannot do this, but for Snowflake it is easy!

Run the following command in the worksheet to create a development (dev) table clone of the `trips` table:

```
create table trips_dev clone trips;
```

Click the three dots (...) in the left pane and select **Refresh**. Expand the object tree under the `CITIBIKE` database and verify that you see a new table named `trips_dev`. Your Development team now can do whatever they want with this table, including updating or deleting it, without impacting the `trips` table or any other object.

The screenshot shows the Snowflake UI interface. At the top, there's a navigation bar with a house icon and the text "CITIBIKE_ZERO_TO_SNOWFLAKE". Below the navigation bar is a sidebar with sections for "Pinned (0)" and "No pinned objects". A search bar and a three-dot menu are also present. The main content area is a tree view under the "CITIBIKE" schema. The "Tables" node is expanded, showing "TRIPS" and "TRIPS_DEV". The "TRIPS_DEV" table is selected and highlighted with a light blue background. A tooltip is displayed over the "TRIPS_DEV" table, providing its properties:

Property	Value
Number of rows	61.5M
Size	1.9GB
Cluster Key	—
Owner	SYSADMIN
Created	1 minute ago

```

67
68     -- same query again to see
69     select date_trunc('hour', starttime)
70     count(*) as "num trips",
71     avg(tripduration)/60 as "avg"
72     avg(haversine(start_station_id,
73     end_station_id)) as "avg distance"
74
75     --find busiest months
76     select
77     monthname(starttime) as "month"
78     count(*) as "num trips"
79     from trips
80     group by 1 order by 2 desc;
81
82     --create dev table
  
```

Working with Semi-Structured Data, Views, & Joins

Positive : This section requires loading additional data and, therefore, provides a review of data loading while also introducing loading semi-structured data.

Going back to the lab's example, the Citi Bike analytics team wants to determine how weather impacts ride counts. To do this, in this section, we will:

- Load weather data in semi-structured JSON format held in a public S3 bucket.
- Create a view and query the JSON data using SQL dot notation.
- Run a query that joins the JSON data to the previously loaded `TRIPS` data.
- Analyze the weather and ride count data to determine their relationship.

The JSON data consists of weather information provided by *MeteoStat* detailing the historical conditions of New York City from 2016-07-05 to 2019-06-25. It is also staged on AWS S3 where the data consists of 75k rows, 36 objects, and 1.1MB compressed. If viewed in a text editor, the raw JSON in the GZ files looks like:

```
[{"temp":6.1,"dwpt":-3.3,"rhum":51.0,"prcp":null,"snow":null,"wdir":200.0,"wspd":27.7,"wpgt":null,"pres":1014.4,"tsun":null,"coco":null,"obsTime":"2017-01-01T00:00:00.000Z","stationID":"72502","name":"Newark Airport","country":"US","region":"NJ","wmo":"72502","icao":"KEWR","latitude":40.6833,"longitude":-74.0,"elevation":5.0,"timezone":"America/New_York","hourly_start":"1973-01-01T00:00:00.000Z","hourly_end":"2022-07-24T00:00:00.000Z","daily_start":"1893-01-01T00:00:00.000Z","daily_end":"2022-07-16T00:00:00.000Z","monthly_start":"1893-01T00:00:00.000Z","monthly_end":"2022-01-01T00:00:00.000Z","distance":7580.9151211035,"weatherCondition":null}, {"temp":6.1,"dwpt":-4.4,"rhum":47.0,"prcp":0.0,"snow":null,"wdir":210.0,"wspd":16.6,"wpgt":null,"pres":1013.9,"tsun":null,"coco":null,"obsTime":"2017-01-01T01:00:00.000Z","stationID":"72502","name":"Newark Airport","country":"US","region":"NJ","wmo":"72502","icao":"KEWR","latitude":40.6833,"longitude":-74.0,"elevation":5.0,"timezone":"America/New_York","hourly_start":"1973-01-01T00:00:00.000Z","hourly_end":"2022-07-24T00:00:00.000Z","daily_start":"1893-01-01T00:00:00.000Z","daily_end":"2022-01-01T00:00:00.000Z","monthly_start":"1893-01T00:00:00.000Z","monthly_end":"2022-01-01T00:00:00.000Z","distance":7580.9151211035,"weatherCondition":null}, {"temp":5.6,"dwpt":-4.3,"rhum":49.0,"prcp":0.0,"snow":null,"wdir":200.0,"wspd":18.4,"wpgt":null,"pres":1012.8,"tsun":null,"coco":null,"obsTime":"2017-01-01T03:00:00.000Z","stationID":"72502","name":"Newark Airport","country":"US","region":"NJ","wmo":"72502","icao":"KEWR","latitude":40.6833,"longitude":-74.0,"elevation":5.0,"timezone":"America/New_York","hourly_start":"1973-01-01T00:00:00.000Z","hourly_end":"2022-07-24T00:00:00.000Z","daily_start":"1893-01-01T00:00:00.000Z","daily_end":"2022-07-16T00:00:00.000Z","monthly_start":"1893-01T00:00:00.000Z","monthly_end":"2022-01-01T00:00:00.000Z","distance":7580.9151211035,"weatherCondition":null}, {"temp":6.1,"dwpt":-4.4,"rhum":47.0,"prcp":0.0,"snow":null,"wdir":230.0,"wspd":18.4,"wpgt":null,"pres":1012.8,"tsun":null,"coco":null,"obsTime":"2017-01-01T03:00:00.000Z","stationID":"72502","name":"Newark Airport","country":"US","region":"NJ","wmo":"72502","icao":"KEWR","latitude":40.6833,"longitude":-74.0,"elevation":5.0,"timezone":"America/New_York","hourly_start":"1973-01-01T00:00:00.000Z","hourly_end":"2022-07-24T00:00:00.000Z","daily_start":"1893-01-01T00:00:00.000Z","daily_end":"2022-07-16T00:00:00.000Z","monthly_start":"1893-01T00:00:00.000Z","monthly_end":"2022-01-01T00:00:00.000Z","distance":7580.9151211035,"weatherCondition":null} ]
```

Negative : SEMI-STRUCTURED DATA Snowflake can easily load and query semi-structured data such as JSON, Parquet, or Avro without transformation. This is a key Snowflake feature because an increasing amount of business-relevant data being generated today is semi-structured, and many traditional data warehouses cannot easily load and query such data. Snowflake makes it easy!

Create a New Database and Table for the Data

First, in the worksheet, let's create a database named `WEATHER` to use for storing the semi-structured JSON data.

```
create database weather;
```

Execute the following USE commands to set the worksheet context appropriately:

```
use role sysadmin;

use warehouse compute_wh;

use database weather;

use schema public;
```

Positive : Executing Multiple Commands Remember that you need to execute each command individually. However, you can execute them in sequence together by selecting all of the commands and then clicking the **Play/Run** button (or using the keyboard shortcut).

Next, let's create a table named `JSON_WEATHER_DATA` to use for loading the JSON data. In the worksheet, execute the following CREATE TABLE command:

```
create table json_weather_data (v variant);
```

Note that Snowflake has a special column data type called `VARIANT` that allows storing the entire JSON object as a single row and eventually query the object directly.

Negative : Semi-Structured Data Magic The VARIANT data type allows Snowflake to ingest semi-structured data without having to predefine the schema.

In the results pane at the bottom of the worksheet, verify that your table, `JSON_WEATHER_DATA`, was created:

```

77
78
79      select
80        monthname(starttime) as "month",
81        count(*) as "num trips"
82      from trips
83      group by 1 order by 2 desc;
84
85      create table trips_dev clone trips;
86
87
88      create database weather;
89
90      use role sysadmin;
91
92      use warehouse compute_wh;
93
94      use database weather;
95
96      use schema public;
97
98
99      create table json_weather_data (v variant);
100
101
102
103
104
105

```

Objects Query Results Chart

status

1 Table JSON_WEATHER_DATA successfully created.

Query Details

Query duration 188ms

Rows 1

status

100% filled

Create Another External Stage

In the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet, use the following command to create a stage that points to the bucket where the semi-structured JSON data is stored on AWS S3:

```
create stage nyc_weather
url = 's3://snowflake-workshop-lab/zero-weather-nyc';
```

Now let's take a look at the contents of the `nyc_weather` stage. Execute the following LIST command to display the list of files:

```
list @nyc_weather;
```

In the results pane, you should see a list of `.gz` files from S3:

```

WEATHER PUBLIC *
86   create table trips_dev clone trips;
87
88
89   create database weather;
90
91   use role sysadmin;
92
93   use warehouse compute_wh;
94
95   use database weather;
96
97   use schema public;
98
99   create table json_weather_data (v variant);
100
101
102   create stage nyc_weather
103     url = 's3://snowflake-workshop-lab/zero-weather-nyc';
104
105
106   | List @nyc_weather;|
107
108
109
110
111
112
113
114

```

name	size	md5	last_modified
1 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-10.json.gz	13,260	356d042ffac90934571e12fc7c348840	Mon, 25 Jul 2022 08:35
2 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-11.json.gz	12,930	dff30f516567061eb8d0fe9e953cd2f0	Mon, 25 Jul 2022 08:35
3 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-12.json.gz	13,574	4495224605a6a2bebe5405bf690438	Mon, 25 Jul 2022 08:35
4 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-7.json.gz	11,760	35b5738b75b05620b58c636bf31ef7	Mon, 25 Jul 2022 08:35
5 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-8.json.gz	12,898	36849474cd97946a2c4b69619b953ef	Mon, 25 Jul 2022 08:35
6 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-9.json.gz	12,656	50986ca04728278106982c5f354c10ed	Mon, 25 Jul 2022 08:35
7 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-1.json.gz	13,447	5d9664b36109e965ead983e17df827	Mon, 25 Jul 2022 08:35
8 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-10.json.gz	13,423	6eb5f864e4285bc93cb6431fc0d32d0	Mon, 25 Jul 2022 08:35
9 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-11.json.gz	13,055	4a24e38608b5dac9c8439d5767914090	Mon, 25 Jul 2022 08:35
10 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-12.json.gz	13,175	3ecb05ea4b17275e18d4ce398bc1a81	Mon, 25 Jul 2022 08:35
11 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-2.json.gz	12,432	70ba2fa598b8c1edd28700e02f0a887f	Mon, 25 Jul 2022 08:35
12 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-3.json.gz	14,065	cbb8cf0d224774e2e9017854ec67673ae	Mon, 25 Jul 2022 08:35
13 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-4.json.gz	13,276	7ca8af4ca3dfebc56d4691fd35c9f1	Mon, 25 Jul 2022 08:35
14 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-5.json.gz	13,461	976bf82ed554f9a9e33137cafc6600	Mon, 25 Jul 2022 08:35
15 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-6.json.gz	12,966	5cd814d9bf4e8e6ae9e9bbfbfb0c3f4	Mon, 25 Jul 2022 08:35
16 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-7.json.gz	13,187	834bc2452881ef07acf612d907355	Mon, 25 Jul 2022 08:35
17 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-8.json.gz	13,164	a5daad36c583664d55616704476c102e	Mon, 25 Jul 2022 08:35
18 e2://tennantdata/workshop-lab/zero-weather/nyc/hourlyData-2017-9.json.gz	12,360	10f31e40777a0077f6khakf358521ht	Mon, 25 Jul 2022 08:35

Query Details

- Query duration: 157ms
- Rows: 36
- name: Aa
100% filled
- size: 123
11,464 - 14,847
- md5: Aa
100% filled
- last_modified: Aa
Mon, 25 Jul 2022 08:35:12 GMT 2
Mon, 25 Jul 2022 08:35:25 GMT 2
Mon, 25 Jul 2022 08:35:08 GMT 1
+ 31 more

Load and Verify the Semi-structured Data

In this section, we will use a warehouse to load the data from the S3 bucket into the `JSON_WEATHER_DATA` table we created earlier.

In the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet, execute the COPY command below to load the data.

Note that you can specify a `FILE FORMAT` object inline in the command. In the previous section where we loaded structured data in CSV format, we had to define a file format to support the CSV structure. Because the JSON data here is well-formed, we are able to simply specify the JSON type and use all the default settings:

```

copy into json_weather_data
from @nyc_weather
file_format = (type = json strip_outer_array = true);

```

Verify that each file has a status of `LOADED`:

CITIBIKE_ZERO_TO_SNOWFLAKE +

SYSADMIN COMPUTE_WH Share Updated 20 seconds ago

Worksheets Databases

Pinned (0)

No pinned objects

Q All Objects ...

CITIBIKE

> INFORMATION_SCHEMA

> PUBLIC

Tables

- TRIPS
- TRIPS_DEV

> Views

> Stages

> Pipes

> Streams

> Tasks

> Functions

> Procedures

SNOWFLAKE_SAMPLE_DATA

```

WEATHER.PUBLIC +
86   create table trips_dev clone trips;
87
88
89   create database weather;
90
91   use role sysadmin;
92
93   use warehouse compute_wh;
94
95   use database weather;
96
97   use schema public;
98
99
100  create table json_weather_data (v variant);
101
102  create stage nyc_weather
103    url = 's3://snowflake-workshop-lab/zero-weather-nyc';
104
105  list @nyc.weather;
106
107
108
109
110  copy into json_weather_data
111    from @nyc.weather
112      file_format = (type = json strip_outer_array = true);
113
114

```

Objects Query Results Chart

file	status	rows_parsed	rows_loaded	error_limit	errors
1 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2019-3.json.gz	LOADED	744	744	1	
2 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-10.json.gz	LOADED	744	744	1	
3 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2019-2.json.gz	LOADED	672	672	1	
4 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-6.json.gz	LOADED	720	720	1	
5 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-9.json.gz	LOADED	720	720	1	
6 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-11.json.gz	LOADED	720	720	1	
7 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-7.json.gz	LOADED	648	648	1	
8 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-5.json.gz	LOADED	744	744	1	
9 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-4.json.gz	LOADED	720	720	1	
10 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2016-8.json.gz	LOADED	744	744	1	
11 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-3.json.gz	LOADED	744	744	1	
12 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-11.json.gz	LOADED	720	720	1	
13 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-2.json.gz	LOADED	672	672	1	
14 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-4.json.gz	LOADED	720	720	1	
15 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-10.json.gz	LOADED	744	744	1	
16 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-2.json.gz	LOADED	672	672	1	
17 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2017-5.json.gz	LOADED	744	744	1	
18 s3://snowflake-workshop-lab/zero-weather-nyc/hourlyData-2018-7.json.gz	LOADED	744	744	1	

Query Details

Query duration 4.2s

Rows 36

file 100% filled

status AA

status LOADED 36

rows_parsed 123

rows_parsed 577 744

rows_loaded 123

rows_loaded 577 744

error_limit 123

error_limit 100% filled

Now, let's take a look at the data that was loaded:

```
select * from json_weather_data limit 10;
```

Click any of the rows to display the formated JSON in the right panel:

```

    WEATHER PUBLIC +
    93  use warehouse compute_wh;
    94
    95  use database weather;
    96
    97  use schema public;
    98
    99
    100 create or replace table json_weather_data (v variant);
    101
    102      create or replace stage nyc_weather
    103          url = 's3://snowflake-workshop-lab/zero-weather-nyc';
    104
    105      list @nyc_weather;
    106
    107
    108
    109      copy into json_weather_data
    110          from @nyc_weather
    111              file_format = (type = json strip_outer_array = true);
    112
    113
    114      select * from json_weather_data limit 10;
    115
    116
    117
    118
    119
    120
    121

```

Row 1 / 10

V
{ "coco": 3, "country": "US", "dwpt": 12.8, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 3, "country": "US", "dwpt": 12.8, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 3, "country": "US", "dwpt": 11.8, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 3, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 3, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 4, "country": "US", "dwpt": 11.1, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 3, "country": "US", "dwpt": 11.8, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 4, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 4, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 4, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},
{ "coco": 4, "country": "US", "dwpt": 11.7, "elevation": 4, "icao": "KJFK", "latitude": 40.6333, "longitude": -73.7667, "name": "John F. Kennedy Airport", "obsTime": "2018-11-01T00:00:00Z", "pres": 1014.1, "region": "NY", "rhum": 90, "snow": null, "station": "74486", "temp": 14.4, "timezone": "America/New_York", "tsun": null, "wdir": 180, "weatherCondition": "Cloudy", "wmo": "74486", "wppt": null},

To close the display in the panel and display the query details again, click the **X** (Close) button that appears when you hover your mouse in the right corner of the panel.

Create a View and Query Semi-Structured Data

Next, let's look at how Snowflake allows us to create a view and also query the JSON data directly using SQL.

Negative : Views & Materialized Views A view allows the result of a query to be accessed as if it were a table. Views can help present data to end users in a cleaner manner, limit what end users can view in a source table, and write more modular SQL. Snowflake also supports materialized views in which the query results are stored as though the results are a table. This allows faster access, but requires storage space. Materialized views can be created and queried if you are using Snowflake Enterprise Edition (or higher).

Run the following command to create a columnar view of the semi-structured JSON weather data so it is easier for analysts to understand and query. The `72502` value for `station_id` corresponds to Newark Airport, the closest station that has weather conditions for the whole period.

```

// create a view that will put structure onto the semi-structured data
create or replace view json_weather_data_view as
select
    v:obsTime::timestamp as observation_time,
    v:station::string as station_id,
    v:name::string as city_name,
    v:country::string as country,
    v:latitude::float as city_lat,
    v:longitude::float as city_lon,

```

```

v:weatherCondition::string as weather_conditions,
v:coco::int as weather_conditions_code,
v:temp::float as temp,
v:prcp::float as rain,
v:tsun::float as tsun,
v:wdir::float as wind_dir,
v:wspd::float as wind_speed,
v:dwpt::float as dew_point,
v:rhum::float as relative_humidity,
v:pres::float as pressure
from
    json_weather_data
where
    station_id = '72502';

```

SQL dot notation `v:temp` is used in this command to pull out values at lower levels within the JSON object hierarchy. This allows us to treat each field as if it were a column in a relational table.

The new view should appear as `JSON_WEATHER_DATA` under `WEATHER > PUBLIC > Views` in the object browser on the left. You may need to expand or refresh the objects browser in order to see it.

WEATHER.PUBLIC > JSON_WEATHER_DATA_VIEW

Details **Definition**

Type: View
Number of columns: 16
Owner: SYSADMIN
Created: 19 minutes ago

	OBSERVATION_TIME	STATION_ID	CITY_NAME	COUNTRY	CITY_LAT	CITY_LON	WEATHER_CONDITIONS
1	2018-01-01 00:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
2	2018-01-01 01:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
3	2018-01-01 02:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
4	2018-01-01 03:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
5	2018-01-01 04:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
6	2018-01-01 05:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
7	2018-01-01 06:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
8	2018-01-01 07:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
9	2018-01-01 08:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
10	2018-01-01 09:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
11	2018-01-01 10:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
12	2018-01-01 11:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
13	2018-01-01 12:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
14	2018-01-01 13:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
15	2018-01-01 14:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
16	2018-01-01 15:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
17	2018-01-01 16:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
18	2018-01-01 17:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear

Query Details

- Query duration: 100ms
- Rows: 20

Observation Time

- 100% filled

Station ID

- 100% filled

City Name

- Newark Airport

Country

- US

City Lat

- 123

Verify the view with the following query:

```

select * from json_weather_data_view
where date_trunc('month',observation_time) = '2018-01-01'

```

```
limit 20;
```

Notice the results look just like a regular structured data source. Your result set may have different `observation_time` values:

The screenshot shows the Snowflake UI interface. At the top, there's a navigation bar with 'CITIBIKE_ZERO_TO_SNOWFLAKE' selected, 'SYSADMIN' as the role, and a 'Share' button. Below the navigation is a status message 'Updated 18 seconds ago'. The main area is divided into two sections: 'Worksheets' and 'Databases'. Under 'Worksheets', there are pinned objects (CITIBIKE, SNOWFLAKE_SAMPLE_DATA, WEATHER), all objects (including the JSON weather data view), and a 'Query' section containing the code for selecting data from the view where the month of observation is January 1, 2018, and limiting the results to 20 rows. The 'Results' tab is selected, displaying a table with columns: OBSERVATION_TIME, STATION_ID, CITY_NAME, COUNTRY, CITY_LAT, CITY_LON, and WEATHER_CONDITIONS. The data shows 19 rows of observations from Newark Airport (STATION_ID 72502) across various dates in January 2018, with most entries showing 'Clear' weather conditions. To the right of the table is a 'Query Details' sidebar showing metrics like Query duration (100ms), Rows (20), and various filters applied to the query.

	OBSERVATION_TIME	STATION_ID	CITY_NAME	COUNTRY	CITY_LAT	CITY_LON	WEATHER_CONDITIONS
1	2018-01-01 00:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
2	2018-01-01 01:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
3	2018-01-01 02:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
4	2018-01-01 03:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
5	2018-01-01 04:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
6	2018-01-01 05:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
7	2018-01-01 06:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
8	2018-01-01 07:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
9	2018-01-01 08:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
10	2018-01-01 09:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
11	2018-01-01 10:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
12	2018-01-01 11:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
13	2018-01-01 12:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
14	2018-01-01 13:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
15	2018-01-01 14:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
16	2018-01-01 15:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
17	2018-01-01 16:00:00.000	72502	Newark Airport	US	40.6833	-74	Clear
18	2018-01-01 17:00:00.000	72603	Newark Airport	US	40.6833	-74	Clear

Use a Join Operation to Correlate Against Data Sets

We will now join the JSON weather data to our `CITIBIKE.PUBLIC.TRIPS` data to answer our original question of how weather impacts the number of rides.

Run the query below to join `WEATHER` to `TRIPS` and count the number of trips associated with certain weather conditions:

Positive : Because we are still in the worksheet, the `WEATHER` database is still in use. You must, therefore, fully qualify the reference to the `TRIPS` table by providing its database and schema name.

```
select weather_conditions as conditions
, count(*) as num_trips
from citibike.public.trips
left outer join json_weather_data_view
on date_trunc('hour', observation_time) = date_trunc('hour', starttime)
where conditions is not null
group by 1 order by 2 desc;
```

```

134     v:wind::float as wind_dir,
135     v:spd::float as wind_speed,
136     v:dept::float as dew_point,
137     v:rhum::float as relative_humidity,
138     v:pres::float as pressure
139
140     from json_weather_data
141     where
142         station_id = '72502';
143
144
145
146     select * from json_weather_data_view
147     where date_trunc('month',observation_time) = '2018-01-01'
148
149     limit 20;
150
151
152     select weather_conditions as conditions
153     ,count(*) as num_trips
154     from citibike.public.trips
155     left outer join json_weather_data_view
156     on date_trunc('hour', observation_time) = date_trunc('hour', starttime)
157     where conditions is not null
158     group by 1 order by 2 desc;
159
160
161
162

```

CONDITIONS	NUM_TRIPS
1 Clear	27,651,743
2 Light Rain	1,577,552
3 Fair	717,754
4 Cloudy	676,990
5 Fog	606,701
6 Overcast	354,975
7 Rain	132,279
8 Light Snowfall	110,083
9 Thunderstorm	83,491
10 Heavy Rain	60,989
11 Heavy Thunderstorm	19,088
12 Sleet	12,398
13 Snowfall	5,123
14 Heavy Freezing Rain	1,068
15 Freezing Rain	1,003
16 Storm	653
17 Heavy Sleet	564
18 Heavy Snowfall	98

The initial goal was to determine if there was any correlation between the number of bike rides and the weather by analyzing both ridership and weather data. Per the results above we have a clear answer. As one would imagine, the number of trips is significantly higher when the weather is good!

Using Time Travel

Snowflake's powerful Time Travel feature enables accessing historical data, as well as the objects storing the data, at any point within a period of time. The default window is 24 hours and, if you are using Snowflake Enterprise Edition, can be increased up to 90 days. Most data warehouses cannot offer this functionality, but - you guessed it - Snowflake makes it easy!

Some useful applications include:

- Restoring data-related objects such as tables, schemas, and databases that may have been deleted.
- Duplicating and backing up data from key points in the past.
- Analyzing data usage and manipulation over specified periods of time.

Drop and Undrop a Table

First let's see how we can restore data objects that have been accidentally or intentionally deleted.

In the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet, run the following DROP command to remove the `JSON_WEATHER_DATA` table:

```
drop table json_weather_data;
```

Run a query on the table:

```
select * from json_weather_data limit 10;
```

In the results pane at the bottom, you should see an error because the underlying table has been dropped:

The screenshot shows the Snowflake UI interface. On the left is the object browser with a tree view of schemas and tables. In the center is a query editor window containing the following SQL code:

```
140  -- verify table is still present
141  select * from json_weather_data limit 10;
```

The line `select * from json_weather_data` is highlighted with a blue selection bar. At the bottom of the query editor, there is an error message in a red box:

Object 'JSON_WEATHER_DATA' does not exist or not authorized.

To the right of the query editor is a results pane showing the following details:

Query Details
Query duration: 32ms
Rows: 0

Now, restore the table:

```
undrop table json_weather_data;
```

The json_weather_data table should be restored. Verify by running the following query:

```
--verify table is undropped

select * from json_weather_data_view limit 10;
```

```

152
153
154 ///////////////////////////////////////////////////////////////////
155 //MODULE 8/
156 ///////////////////////////////////////////////////////////////////
157 --drop table json_weather_data;
158 drop table json_weather_data;
159 --verify table is dropped
160 select * from json_weather_data limit 10;
161
162
163 --undrop the table
164 undrop table json_weather_data;
165
166 --verify table is undropped
167 select * from json_weather_data_view limit 10;
168
169
170

```

	OBSERVATION_TIME	CITY_ID	CITY_NAME	COUNTRY	CITY_LAT	CITY_LON	CLOUDS	TEMP_AVG	TEMP
1	2016-10-23 07:55:33.000	5,128,638	New York	US	43.000351	-75.499901	88	4.723	4
2	2016-10-20 05:51:43.000	5,128,638	New York	US	43.000351	-75.499901	20	8.623	8
3	2016-10-22 06:41:04.000	5,128,638	New York	US	43.000351	-75.499901	92	7.309	7
4	2016-09-13 11:25:35.000	5,128,638	New York	US	43.000351	-75.499901	1	12.12	
5	2016-09-08 23:03:53.000	5,128,638	New York	US	43.000351	-75.499901	40	25.3	2
6	2016-09-05 18:25:39.000	5,128,638	New York	US	43.000351	-75.499901	1	27.82	2
7	2016-09-15 03:04:10.000	5,128,638	New York	US	43.000351	-75.499901	1	13.22	1
8	2016-11-26 21:07:52.000	5,128,638	New York	US	43.000351	-75.499901	90	3.33	
9	2016-08-06 23:31:04.000	5,128,638	New York	US	43.000351	-75.499901	44	25.29	2
10	2016-08-10 01:24:18.000	5,128,638	New York	US	43.000351	-75.499901	0	24.449	24

Roll Back a Table

Let's roll back the `TRIPS` table in the `CITIBIKE` database to a previous state to fix an unintentional DML error that replaces all the station names in the table with the word "oops".

First, run the following SQL statements to switch your worksheet to the proper context:

```

use role sysadmin;

use warehouse compute_wh;

use database citibike;

use schema public;

```

Run the following command to replace all of the station names in the table with the word "oops":

```
update trips set start_station_name = 'oops';
```

Now, run a query that returns the top 20 stations by number of rides. Notice that the station names result contains only one row:

```

select
start_station_name as "station",
count(*) as "rides"
from trips
group by 1
order by 2 desc
limit 20;

```

```

167  select * from json_weather_data_view limit 10;
168
169  set context;
170  use role sysadmin;
171  use warehouse compute_wh;
172  use database citibike;
173  use schema public;
174
175  --make a mistake by changing the column name
176  update trips set start_station_name='oops';
177
178  --query for top 20 stations by total number of rides
179  select start_station_name as "station",
180        count(*) as "rides"
181  from trips
182  group by 1
183  order by 2 desc
184
185  limit 20;

```

station	rides
oops	61,468,359

Query Details

- Query duration: 315ms
- Rows: 1
- station: 100% filled
- rides: 100% filled

Normally we would need to scramble and hope we have a backup lying around.

In Snowflake, we can simply run a command to find the query ID of the last UPDATE command and store it in a variable named `$QUERY_ID`.

```

set query_id =
(select query_id from table(information_schema.query_history_by_session
(result_limit=>5))
where query_text like 'update%' order by start_time desc limit 1);

```

Use Time Travel to recreate the table with the correct station names:

```

create or replace table trips as
(select * from trips before (statement => $query_id));

```

Run the previous query again to verify that the station names have been restored:

```

select
start_station_name as "station",
count(*) as "rides"
from trips
group by 1
order by 2 desc
limit 20;

```

```

--find the query id that was the last update to the table
187 set query_id =
188 (select query_id from table(information_schema.query_history_by_session (result_limit>5))
189 where query.text Like '%update%' order by start_time Limit 1);
190
191 --recreate the table with proper station names
192 create or replace table trips as
193 (select * from trips before (statement => $query_id));
194
195 --select from the restored table to verify
196 select
197 start_station_name as "station",
198 count(*) as "rides"
199 from trips
200 group by 1
201 order by 2 desc
202 limit 20;
203
204

```

station	rides
1 Pershing Square North	491,951
2 E 17 St & Broadway	481,065
3 W 21 St & 6 Ave	458,626
4 8 Ave & W 31 St	438,001
5 West St & Chambers St	432,518
6 Broadway & E 22 St	421,812
7 Lafayette St & E 8 St	397,724
8 Broadway & E 14 St	394,995
9 8 Ave & W 33 St	379,843
10 W 41 St & 8 Ave	359,838
11 Cleveland Pl & Spring St	358,485
12 W 20 St & 11 Ave	352,099
13 Carmine St & 6 Ave	348,158
14 University Pl & E 14 St	332,803
15 Greenwich Ave & 8 Ave	329,347

Working with Roles, Account Admin, & Account Usage

In this section, we will explore aspects of Snowflake's access control security model, such as creating a role and granting it specific permissions. We will also explore other usage of the ACCOUNTADMIN (Account Administrator) role, which was briefly introduced earlier in the lab.

Continuing with the lab story, let's assume a junior DBA has joined Citi Bike and we want to create a new role for them with less privileges than the system-defined, default role of SYSADMIN.

Negative : Role-Based Access Control Snowflake offers very powerful and granular access control that dictates the objects and functionality a user can access, as well as the level of access they have.

Create a New Role and Add a User

In the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet, switch to the ACCOUNTADMIN role to create a new role. ACCOUNTADMIN encapsulates the SYSADMIN and SECURITYADMIN system-defined roles. It is the top-level role in the account and should be granted only to a limited number of users.

In the `CITIBIKE_ZERO_TO_SNOWFLAKE` worksheet, run the following command:

```
use role accountadmin;
```

Notice that, in the top right of the worksheet, the context has changed to ACCOUNTADMIN:

```

287   order by z desc
288   limit 20;
289
290  ///////////////////////////////////////////////////////////////////
291  //MODULE v1
292  ///////////////////////////////////////////////////////////////////
293  --assume account admin role
294  use role accountadmin;
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320

```

status

1 Statement executed successfully.

Query Details

- Query duration: 51ms
- Rows: 1

status

100% filled

Before a role can be used for access control, at least one user must be assigned to it. So let's create a new role named `JUNIOR_DBA` and assign it to your Snowflake user. To complete this task, you need to know your username, which is the name you used to log in to the UI.

Use the following commands to create the role and assign it to you. Before you run the GRANT ROLE command, replace `YOUR_USERNAME_GOES_HERE` with your username:

```

create role junior_dba;

grant role junior_dba to user YOUR_USERNAME_GOES_HERE;

```

Positive : If you try to perform this operation while in a role such as `SYSADMIN`, it would fail due to insufficient privileges. By default (and design), the `SYSADMIN` role cannot create new roles or users.

Change your worksheet context to the new `JUNIOR_DBA` role:

```
use role junior_dba;
```

In the top right of the worksheet, notice that the context has changed to reflect the `JUNIOR_DBA` role.

```

284
285
286 ///////////////////////////////////////////////////////////////////
287 //MODULE 9//
288 ///////////////////////////////////////////////////////////////////
289 --assume account admin role
290 use role accountadmin;
291
292 --create new role and grant role to current user
293 create role junior_dba;
294 grant role junior_dba to user SF_USER;
295
296 --execute junior_dba role
297 use role junior_dba;
298
299
300
301
302
303

```

The screenshot shows a Snowflake SQL editor interface. The top bar includes a database dropdown set to 'CITIBIKE_ZERO_TO_SNOWFLAKE', a role dropdown set to 'JUNIOR_DBA', and buttons for 'Share' and 'Run as...'. A status message at the top right says 'Updated 21 seconds ago' and 'Draft'. The main area has tabs for 'Objects', 'Query', and 'Results'. The 'Query' tab is active, displaying the provided SQL code. A blue bar highlights the line 'use role junior_dba;'. A tooltip 'JUNIOR_DBA required to view results' appears over the 'Results' tab. On the left, a sidebar shows 'No databases' and 'Objects unavailable. Select a different role.'

Also, the warehouse is not selected because the newly created role does not have usage privileges on any warehouse. Let's fix it by switching back to ADMIN role and grant usage privileges to `COMPUTE_WH` warehouse.

```

use role accountadmin;

grant usage on warehouse compute_wh to role junior_dba;
```

Switch back to the `JUNIOR_DBA` role. You should be able to use `COMPUTE_WH` now.

```

use role junior_dba;

use warehouse compute_wh;
```

Finally, you can notice that in the database object browser panel on the left, the `CITIBIKE` and `WEATHER` databases no longer appear. This is because the `JUNIOR_DBA` role does not have privileges to access them.

Switch back to the ACCOUNTADMIN role and grant the `JUNIOR_DBA` the USAGE privilege required to view and use the `CITIBIKE` and `WEATHER` databases:

```

use role accountadmin;

grant usage on database citibike to role junior_dba;

grant usage on database weather to role junior_dba;
```

Switch to the `JUNIOR_DBA` role:

```
use role junior_dba;
```

Notice that the `CITIBIKE` and `WEATHER` databases now appear in the database object browser panel on the left. If they don't appear, try clicking ... in the panel, then clicking **Refresh**.

CITIBIKE_ZERO_TO_SNOWFLAKE

Search

Objects unavailable. Select a different role.

CITIBIKE WEATHER

```
use role junior_dbadmin;
--change role to give permissions to junior_dbadmin role
use role accountadmin;
grant usage on database citibike to junior_dbadmin;
grant usage on database weather to junior_dbadmin;
switch back to junior_dbadmin
use role junior_dbadmin;
```

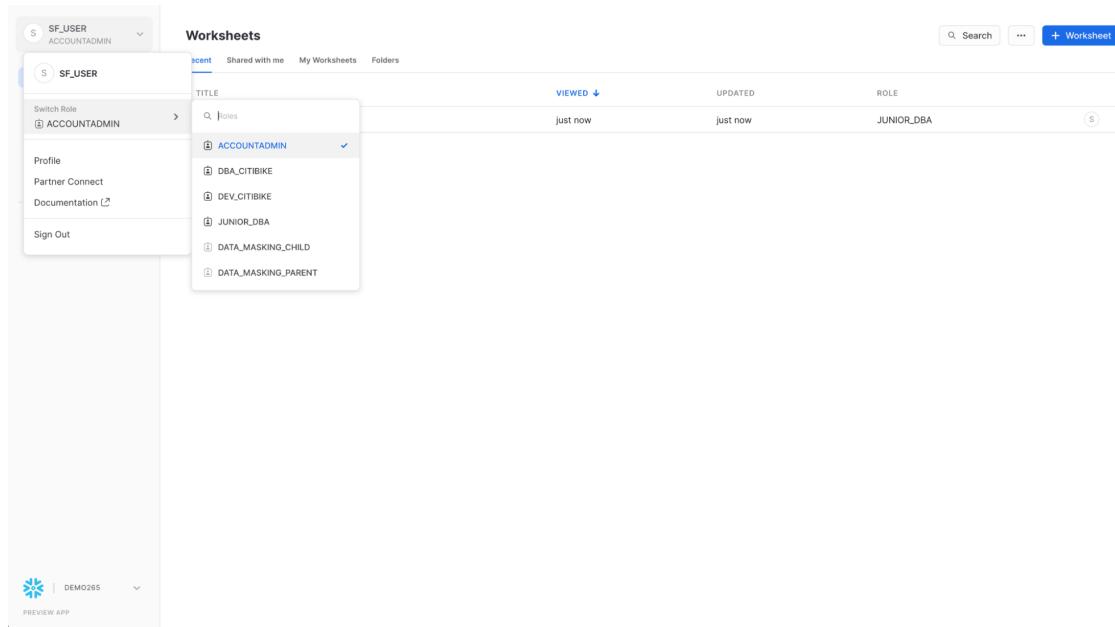
Objects Query Results Chart

JUNIOR_DBA required to view results

View the Account Administrator UI

Let's change our access control role back to `ACCOUNTADMIN` to see other areas of the UI accessible only to this role. However, to perform this task, use the UI instead of the worksheet.

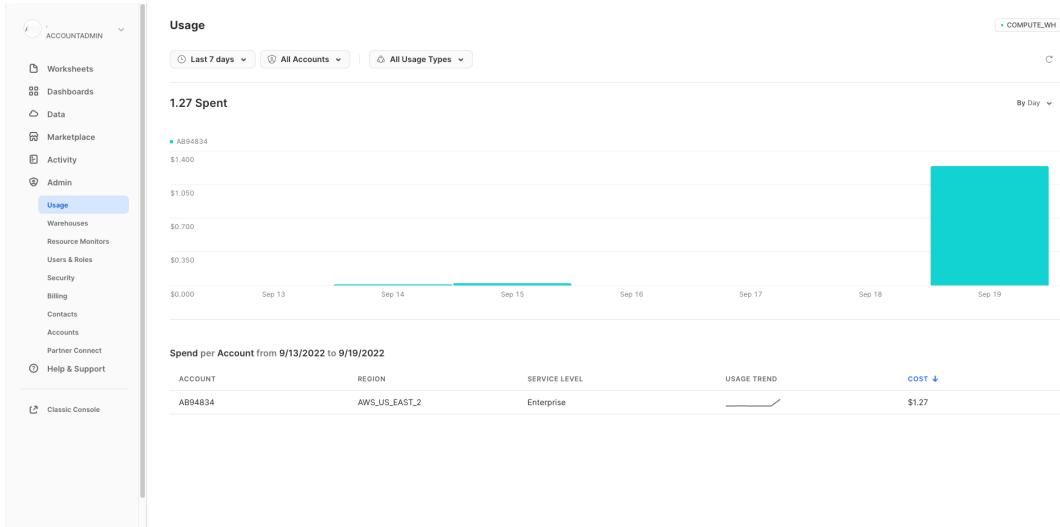
First, click the **Home** icon in the top left corner of the worksheet. Then, in the top left corner of the UI, click your name to display the user preferences menu. In the menu, go to **Switch Role** and select ACCOUNTADMIN.



Negative : Roles in User Preference vs Worksheet Why did we use the user preference menu to change the role instead of the worksheet? The UI session and each worksheet have their own separate roles. The UI session role controls the elements you can see and access in the UI, whereas the worksheet role controls only the objects and actions you can access within the role.

Notice that once you switch the UI session to the ACCOUNTADMIN role, new tabs are available under **Admin**.

Usage



The **Usage** tab shows the following, each with their own page:

- **Organization:** Credit usage across all the accounts in your organization.
- **Consumption:** Credits consumed by the virtual warehouses in the current account.
- **Storage:** Average amount of data stored in all databases, internal stages, and Snowflake Failsafe in the current account for the past month.
- **Transfers:** Average amount of data transferred out of the region (for the current account) into other regions for the past month.

The filters in the top right corner of each page can be used to break down the usage/consumption/etc. visualizations by different measures.

Security

The screenshot shows the Snowflake interface with the left sidebar expanded. The 'Security' tab is highlighted in blue. The main content area is titled 'Network Policies' and 'Sessions'. At the top right is a blue button labeled '+ Network Policy'. Below the title, there's a small icon of a document with a lock. A section titled 'No network policies' contains the text: 'Restrict or allow access to your Snowflake account based on IP address.' followed by a 'Learn More' link.

The **Security** tab contains network policies created for the Snowflake account. New network policies can be created by selecting "+ Network Policy" at the top right hand side of the page.

Billing

The screenshot shows the Snowflake interface with the left sidebar expanded. The 'Billing' tab is highlighted in blue. The main content area is titled 'Billing' and 'Payment Method'. It includes a note: 'Add a credit card to continue using Snowflake once your free trial ends.' and a blue button labeled '+ Credit Card'.

The **Billing** tab contains the payment method for the account:

- If you are a Snowflake contract customer, the tab shows the name associated with your contract information.
- If you are an on-demand Snowflake customer, the tab shows the credit card used to pay month-to-month, if one has been entered. If no credit card is on file, you can add one to continue using Snowflake when your trial ends.

For the next section, stay in the ACCOUNTADMIN role for the UI session.

Sharing Data Securely & the Data Marketplace

Snowflake enables data access between accounts through the secure data sharing features. Shares are created by data providers and imported by data consumers, either through their own Snowflake account or a provisioned Snowflake Reader account. The consumer can be an external entity or a different internal business unit that is required to have its own unique Snowflake account.

With secure data sharing:

- There is only one copy of the data that lives in the data provider's account.
- Shared data is always live, real-time, and immediately available to consumers.
- Providers can establish revocable, fine-grained access to shares.
- Data sharing is simple and safe, especially compared to older data sharing methods, which were often manual and insecure, such as transferring large `.csv` files across the internet.

Snowflake uses secure data sharing to provide account usage data and sample data sets to all Snowflake accounts. In this capacity, Snowflake acts as the data provider of the data and all other accounts.

Secure data sharing also powers the Snowflake Data Marketplace, which is available to all Snowflake customers and allows you to discover and access third-party datasets from numerous data providers and SaaS vendors. Again, in this data sharing model, the data doesn't leave the provider's account and you can use the datasets without any transformation.

View Existing Shares

In the home page, navigate to **Data > Databases**. In the list of databases, look at the **SOURCE** column. You should see two databases with `Local` in the column. These are the two databases we created previously in the lab. The other database, `SNOWFLAKE`, shows `Share` in the column, indicating it's shared from a provider.

The screenshot shows the Snowflake Data Marketplace interface. On the left, there is a sidebar with navigation links: Worksheets, Dashboards, Databases (which is selected and highlighted in blue), Shared Data, Marketplace, Compute, and Account. Below the sidebar is a link to the Classic Console. The main area displays a table titled "Databases" with the following data:

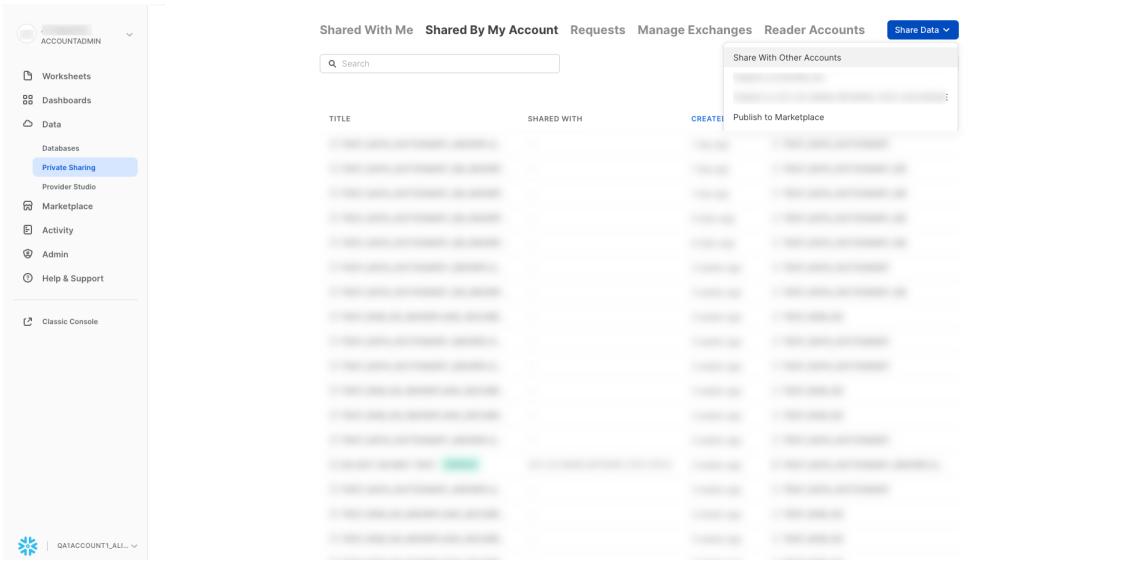
NAME	SOURCE	OWNER	CREATED	...
CITIBIKE	Local	SYSADMIN	13 hours ago	...
SNOWFLAKE	Share	—	9 months ago	...
WEATHER	Local	SYSADMIN	53 minutes ago	...

At the bottom of the interface, there is a footer bar with icons for a preview app and a DEMO265 link.

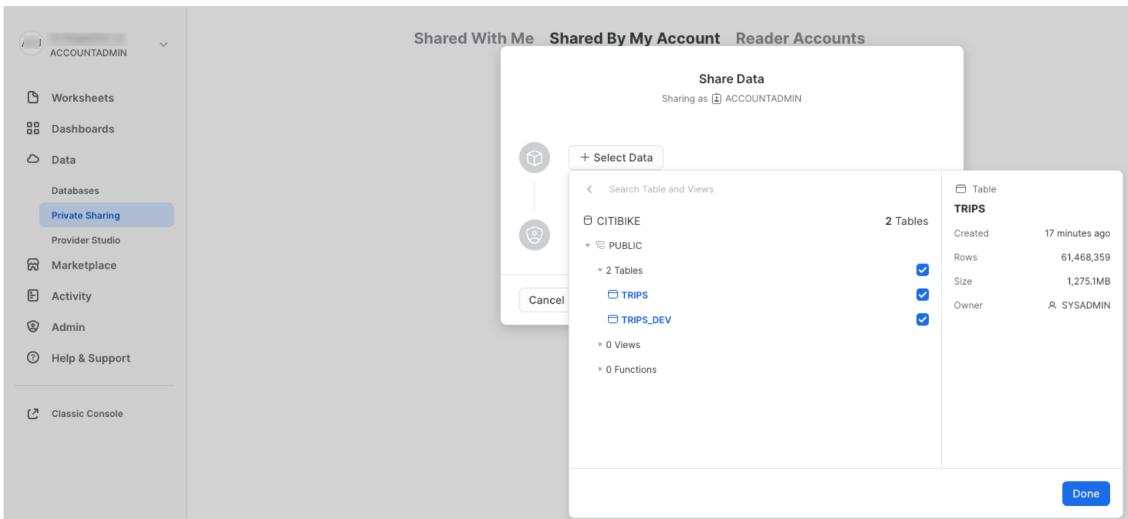
Create an Outbound Share

Let's go back to the Citi Bike story and assume we are the Account Administrator for Snowflake at Citi Bike. We have a trusted partner who wants to analyze the data in our `TRIPS` database on a near real-time basis. This partner also has their own Snowflake account in the same region as our account. So let's use secure data sharing to allow them to access this information.

Navigate to **Data > Shared Data**, then click **Shared by My Account** at the top of the tab. Click the **Share Data** button in the top right corner and select **Share with Other Accounts**:



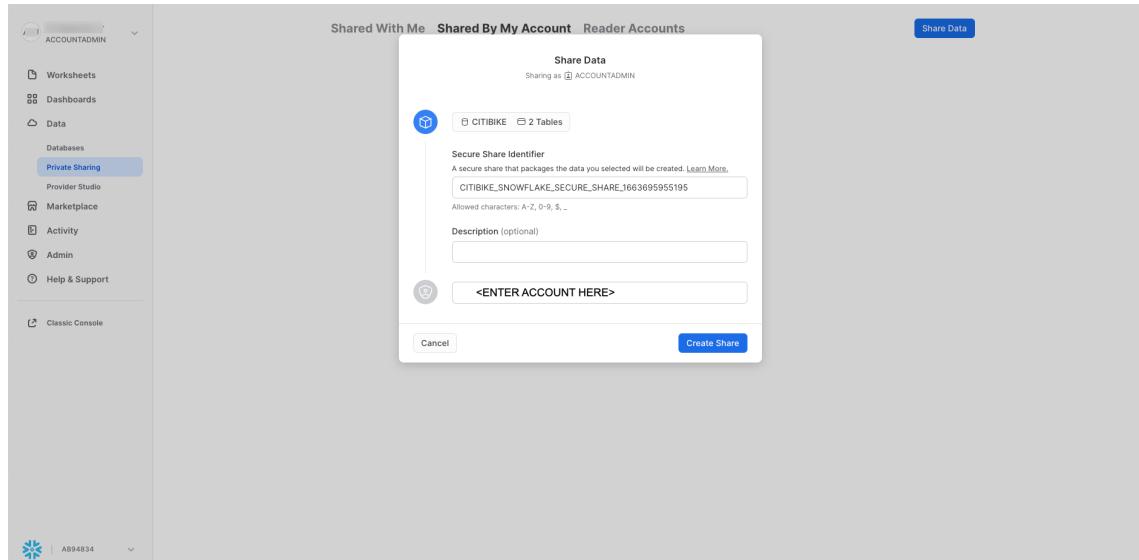
Click **+ Data** and navigate to the `CITIBIKE` database and `PUBLIC` schema. Select the 2 tables we created in the schema and click the **Done** button:



The default name of the share is a generic name with a random numeric value appended. Edit the default name to a more descriptive value that will help identify the share in the future (e.g. `ZERO_TO_SNOWFLAKE_SHARED_DATA`). You can also add a comment.

In a real-world scenario, the Citi Bike Account Administrator would next add one or more consumer accounts to the share, but we'll stop here for the purposes of this lab.

Click the **Create Share** button at the bottom of the dialog:



The dialog closes and the page shows the secure share you created:

A screenshot of the "ZERO_TO_SNOWFLAKE_DATA_SHARE" secure share details page. The page includes sections for "Shared With", "Description" (sharing data for zero to snowflake), and "Data" (listing two tables: TRIPS and TRIPS_DEV). The "Data" section has an "Edit" button.

You can add consumers, add/change the description, and edit the objects in the share at any time. In the page, click the < button next to the share name to return to the **Share with Other Accounts** page:

The screenshot shows the Snowflake Data Marketplace interface. On the left, there's a sidebar with navigation links: Worksheets, Dashboards, Data (with Shared Data selected), Databases, Marketplace, Compute, and Account. Below the sidebar, it says DEMO265 and PREVIEW APP. At the top, there are tabs for Shared With Me, Shared By My Account, Requests, Manage Exchanges, Reader Accounts, and a Share Data dropdown. A search bar is present, along with filters for Shared with All and All Types. The main area displays a table with columns: TITLE, SHARED WITH, CREATED (sorted by newest), and DATA. One entry is shown: ZERO_TO_SNOWFLAKE_SHARED_DATA, shared with CITIBIKE, created 1 minute ago.

We've demonstrated how it only takes seconds to give other accounts access to data in your Snowflake account in a secure manner with no copying or transferring of data required!

Snowflake provides several ways to securely share data without compromising confidentiality. In addition to tables, you can share secure views, secure UDFs (user-defined functions), and other secure objects. For more details about using these methods to share data while preventing access to sensitive information, see the [Snowflake documentation].

Snowflake Data Marketplace

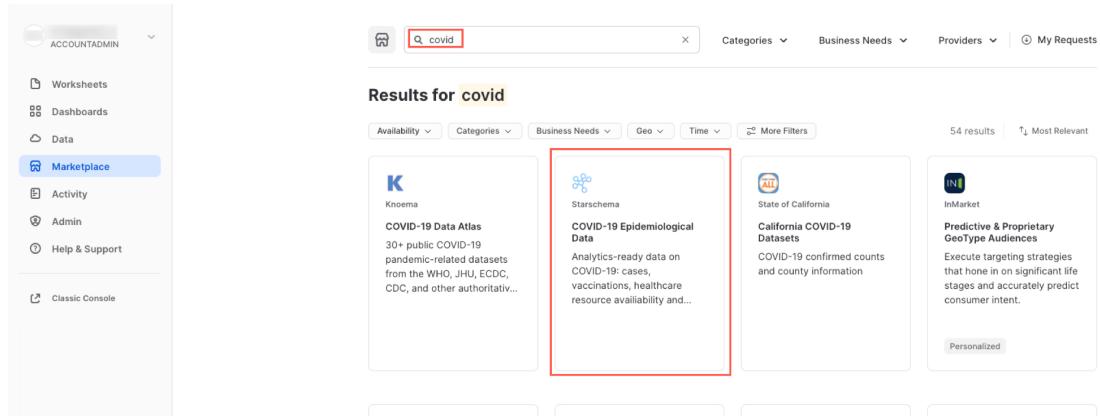
Make sure you're using the ACCOUNTADMIN role and, navigate to the **Marketplace**:

The screenshot shows the Snowflake Data Marketplace homepage. The left sidebar includes links for Worksheets, Dashboards, Data (with Marketplace selected), Activity, Admin, Help & Support, and a Classic Console link. The main content area features a search bar, categories (Business Needs, Providers), and a 'My Requests' button. A banner at the top reads "Together, we can build a sustainable future" with a call to join the Data Collaboration for the Climate initiative. Below the banner, there are sections for "Featured Providers" (Stripe, HubSpot, Braze, iPInfo) and "Most Popular" datasets (Starschema COVID-19 Epidemiological Data, Starschema Worldwide Address Data, Knoema Economy Data Atlas, EDI Exchange Data International). A "More" link is visible next to the popular datasets.

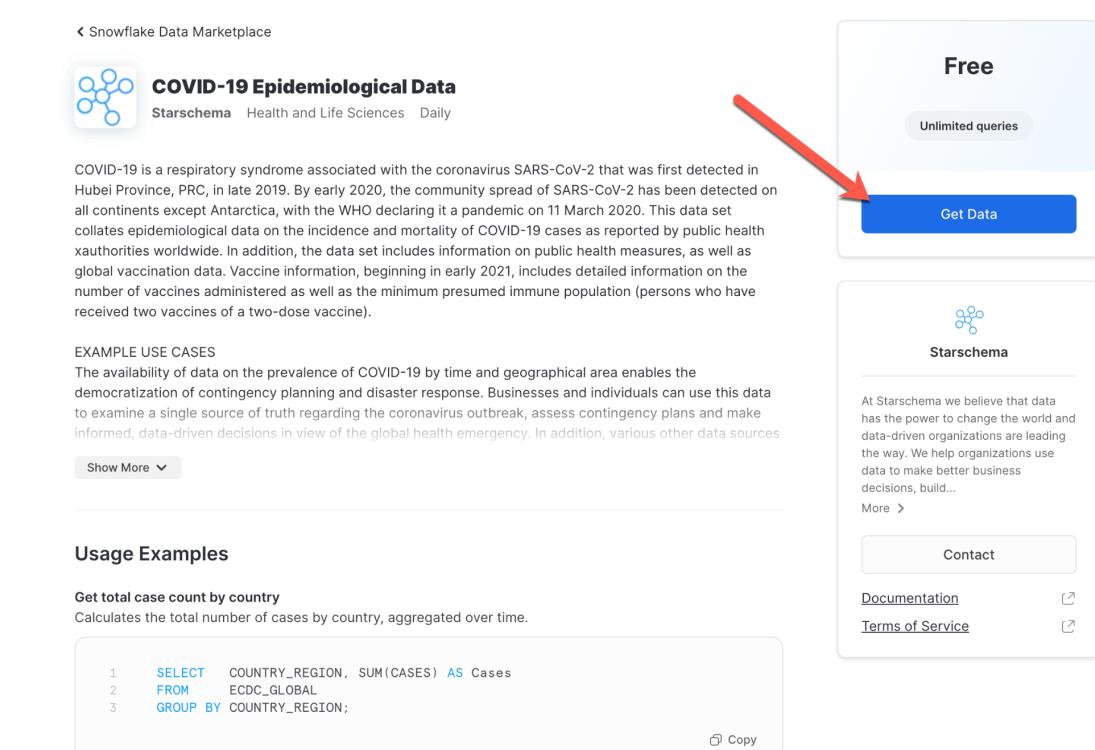
Find a listing

The search box at the top allows you to search for a listings. The drop-down lists to the right of the search box let you filter data listings by Provider, Business Needs, and Category.

Type **COVID** in the search box, scroll through the results, and select **COVID-19 Epidemiological Data** (provided by Starschema).



In the **COVID-19 Epidemiological Data** page, you can learn more about the dataset and see some usage example queries. When you're ready, click the **Get Data** button to make this information available within your Snowflake account:



COVID-19 Epidemiological Data
Starschema Health and Life Sciences Daily

COVID-19 is a respiratory syndrome associated with the coronavirus SARS-CoV-2 that was first detected in Hubei Province, PRC, in late 2019. By early 2020, the community spread of SARS-CoV-2 has been detected on all continents except Antarctica, with the WHO declaring it a pandemic on 11 March 2020. This data set collates epidemiological data on the incidence and mortality of COVID-19 cases as reported by public health authorities worldwide. In addition, the data set includes information on public health measures, as well as global vaccination data. Vaccine information, beginning in early 2021, includes detailed information on the number of vaccines administered as well as the minimum presumed immune population (persons who have received two vaccines of a two-dose vaccine).

EXAMPLE USE CASES
The availability of data on the prevalence of COVID-19 by time and geographical area enables the democratization of contingency planning and disaster response. Businesses and individuals can use this data to examine a single source of truth regarding the coronavirus outbreak, assess contingency plans and make informed, data-driven decisions in view of the global health emergency. In addition, various other data sources

Show More ▾

Usage Examples

Get total case count by country
Calculates the total number of cases by country, aggregated over time.

```
1  SELECT COUNTRY_REGION, SUM(CASES) AS Cases
2  FROM ECDC_GLOBAL
3  GROUP BY COUNTRY_REGION;
```

Copy

Free
Unlimited queries
Get Data

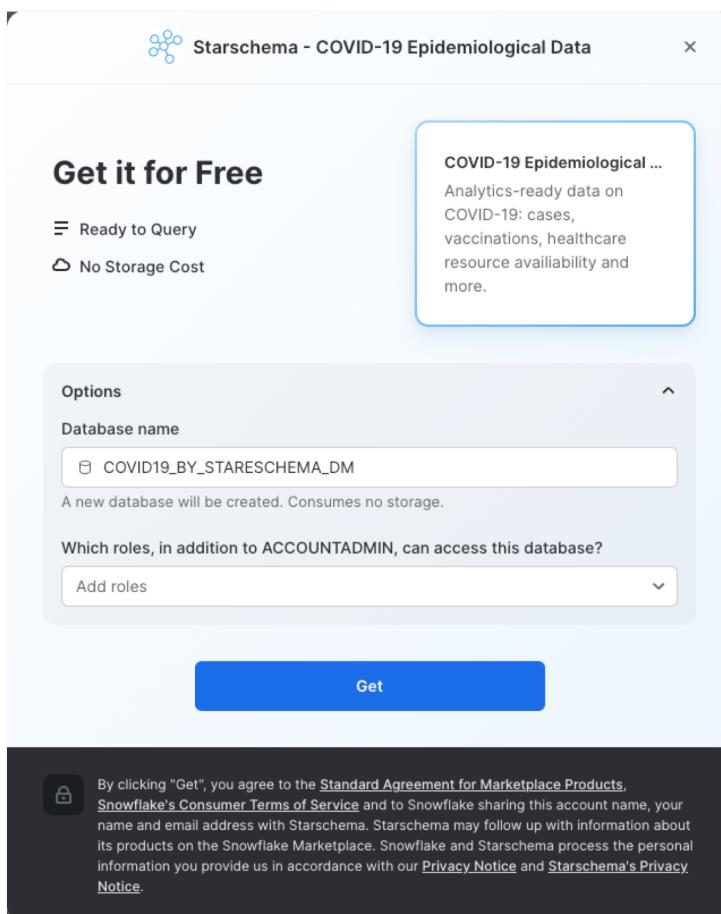
Starschema

At Starschema we believe that data has the power to change the world and data-driven organizations are leading the way. We help organizations use data to make better business decisions, build...

More ▾

Contact
Documentation Terms of Service

Review the information in the dialog and click **Get** again:



You can now click **Done** or choose to run the sample queries provided by Starschema:



Data is Ready to Query

COVID19_BY_STARESCHEMA_DM

Open ↗

Done

If you chose **Open**, a new worksheet opens in a new browser tab/window:

1. Select the query you want to run (or place your cursor in the query text).
2. Click the **Run/Play** button (or use the keyboard shortcut).
3. You can view the data results in the bottom pane.
4. When you are done running the sample queries, click the **Home** icon in the upper left corner.

Pinned (0)
No pinned objects

Databases

- APPLOG
- CITIBIKE
- CITIBIKE_RESET
- CITYBIKE2
- COVID19_BY_STARSCHEMA_DM
- ECONOMY_DATA_ATLAS
- JASON
- JASON_ANALYST_DB
- JASON_DATA
- JSON_DEMO
- KNOEMA_POVERTY_DATA_ATLAS
- MOVIELENS
- NETSPEND
- SAFEGRAPH_CENSUS_DATA2
- SALES
- SALES_AUSTRALIA
- SALES_COLOMBIA
- SALES_SWEDEN
- SECURE_VIEW_DEMO
- SNOWALERT
- SNOWCONVERT
- SNOWFLAKE
- SNOWFLAKE_SAMPLE_DATA
- STARBUCKS_PATTERNS_SAMPLE
- TIMF

```

1 // Get total case count by country
2 // Calculates the total number of cases by country, aggregated over time.
3 SELECT COUNTRY_REGION, SUM(CASES) AS Cases
4 FROM ECDC_GLOBAL
5 GROUP BY COUNTRY_REGION
6
7 // Change in mobility in over time
8 // Displays the change in visits to places like grocery stores and parks by
9 date, location and location type for a sub-region (Alexandria) of a state
10 (Virginia) of a country (United States).
11
12 SELECT DATE,
13 COUNTRY_REGION,
14 PROVINCE_STATE,
15 GROCERY_AND_PHARMACY_CHANGE_PERC,
16 PARKS_CHANGE_PERC,
17 RESIDENTIAL_CHANGE_PERC,
18 RETAIL_AND_RECREATION_CHANGE_PERC,
19 TRANSIT_STATIONS_CHANGE_PERC,
20 WORKPLACES_CHANGE_PERC
21
22 FROM GOOG_GLOBAL_MOBILITY_REPORT
23 WHERE COUNTRY_REGION = 'United States'
24 AND PROVINCE_STATE = 'Virginia'
25 AND SUB_REGION_2 = 'Alexandria';
    
```

Objects Query Results Chart

COUNTRY_REGION	CASES
Afghanistan	49,273
Albania	48,530
Algeria	92,102
Andorra	7,338
Angola	16,188
Anguilla	10
Antigua and Barbuda	148
Argentina	1,498,160
Armenia	148,682
Aruba	5,049

Query Details

- Query duration: 1.6s
- Rows: 214

COUNTRY_REGION

100% filled

CASES

Next:

- Click Data > Databases
- Click the COVID19_BY_STARSCHEMA_DM database.
- You can see details about the schemas, tables, and views that are available to query.

jason ACCOUNTADMIN

Worksheets

Dashboards

Data

Databases 1

Shared Data

Marketplace

Compute

Account

Organization

Classic Console

COVID19_BY_STARSCHEMA_DM 2

Database Database ACCOUNTADMIN 15 hours ago Snowflake Data Marketplace

Database Details Schemas 3

Source details

Provider Starschema

Title COVID-19 Epidemiological Data

Privileges

ACCOUNTADMIN (Current Role) OWNERSHIP

That's it! You have now successfully subscribed to the COVID-19 dataset from Starschema, which is updated daily with global COVID data. Notice we didn't have to create databases, tables, views, or an ETL process. We simply searched for and accessed shared data from the Snowflake Data Marketplace.

Resetting Your Snowflake Environment

If you would like to reset your environment by deleting all the objects created as part of this lab, run the SQL statements in a worksheet.

First, ensure you are using the ACCOUNTADMIN role in the worksheet:

```
use role accountadmin;
```

Then, run the following SQL commands to drop all the objects we created in the lab:

```
drop share if exists zero_to_snowflake_shared_data;
-- If necessary, replace "zero_to_snowflake-shared_data" with the name you used for
the share

drop database if exists citibike;

drop database if exists weather;

drop warehouse if exists analytics_wh;

drop role if exists junior_dba;
```

Conclusion & Next Steps

Congratulations on completing this introductory lab exercise! You've mastered the Snowflake basics and are ready to apply these fundamentals to your own data. Be sure to reference this guide if you ever need a refresher.

We encourage you to continue with your free trial by loading your own sample or production data and by using some of the more advanced capabilities of Snowflake not covered in this lab.

What we've covered:

- How to create stages, databases, tables, views, and virtual warehouses.
- How to load structured and semi-structured data.
- How to perform analytical queries on data in Snowflake, including joins between tables.
- How to clone objects.
- How to undo user errors using Time Travel.
- How to create roles and users, and grant them privileges.
- How to securely and easily share data with other accounts.
- How to consume datasets in the Snowflake Data Marketplace.

Getting Started with SnowSQL

Overview

SnowSQL is the software CLI tool used to interact with Snowflake. Using SnowSQL, you can control all aspects of your Snowflake Data Cloud, including uploading data, querying data, changing data, and deleting data. This guide will review SnowSQL and use it to create a database, load data, and learn helpful commands to manage your tables and data directly from your CLI.

What You'll Learn

- how to create a Snowflake account
- how to install SnowSQL locally
- how to set up a cloud database and table
- how to create a virtual warehouse
- how to migrate sample data to the cloud
- how to query cloud data
- how to insert additional data
- how to drop database objects and close SnowSQL connection

Be sure to check the needed computing requirements before beginning. Also, download the sample files to complete this tutorial and note the folder location for later use.

What You'll Need

- Local [Browser and OS Requirements]
- Download the [Sample Data Files]

What You'll Build

- A connection to cloud host and manage data with SnowSQL.

Set up SnowSQL

First, you'll get a Snowflake account and get comfortable navigating in the web console. After downloading the SnowSQL installer, you'll install and confirm your success.

Create a Snowflake Account

Snowflake lets you try out their services for free with a [trial account](#).

Access Snowflake's Web Console

`https://<account-name>.snowflakecomputing.com/console/login`

Log in to the web interface on your browser. The URL contains your [account name] and potentially the region.

Increase Your Account Permission

The Snowflake web interface has a lot to offer, but for now, we'll just switch the account role from the default `SYSADMIN` to `ACCOUNTADMIN`. This increase in permissions will allow the user account to create objects.



Download the SnowSQL Installer

SnowSQL can be downloaded and installed on Linux, Windows, or Mac. Download the SnowSQL Installer for Windows from here:

```
https://sfc-repo.snowflakecomputing.com/snowsql/bootstrap/1.2/windows_x86_64/snowsql-1.2.21-windows_x86_64.msi
```

Install SnowSQL Locally

- Double-click the installer file and walk through the wizard prompts.
- Confirm the install was successful by checking the version `$ snowsql -v`.
- Reboot your machine and check again if necessary.

After completing these steps, you'll be ready to use SnowSQL to make a database in the next section.

Create a Database

With your account active and SnowSQL installed, you'll use the terminal to sign in and create the needed objects for cloud storage.

Sign in From the Terminal

```
snowsql -a <account-name> -u <username>
```

The `-a` flag represents the Snowflake account, and the `-u` represents the username.

Create a Database and Schema

```
create or replace database sf_tuts;
```

The command [create or replace database] makes a new database and auto-creates the schema 'public.' It'll also make the new database active for your current session.

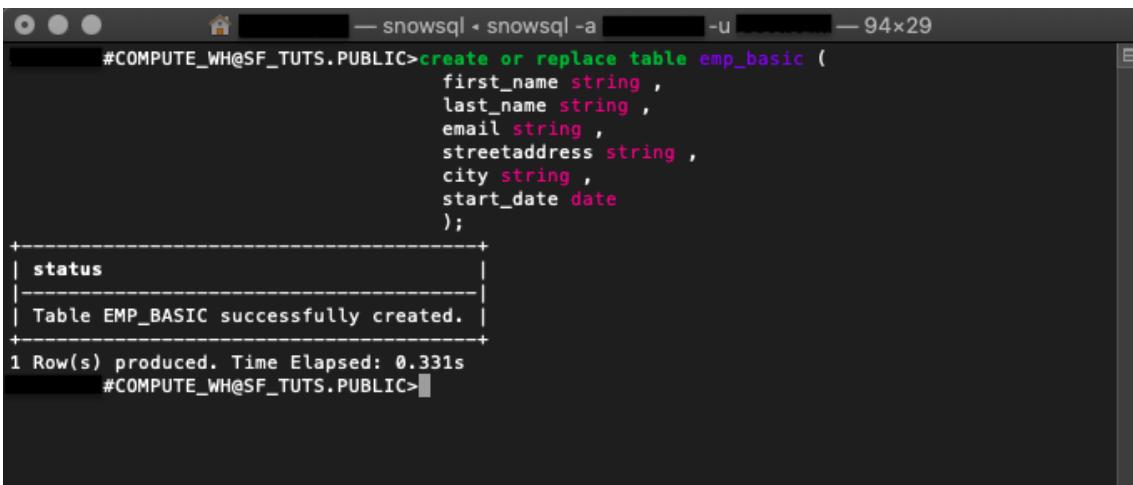
To check which database is in use for your current session, execute:

```
select current_database(),  
current_schema();
```

Generate a Table

```
create or replace table emp_basic (  
    first_name string,  
    last_name string,  
    email string,  
    streetaddress string,  
    city string,  
    start_date date  
) ;
```

Running [create or replace table] will build a new table based on the parameters specified. This example reflects the same columns in the sample CSV employee data files.

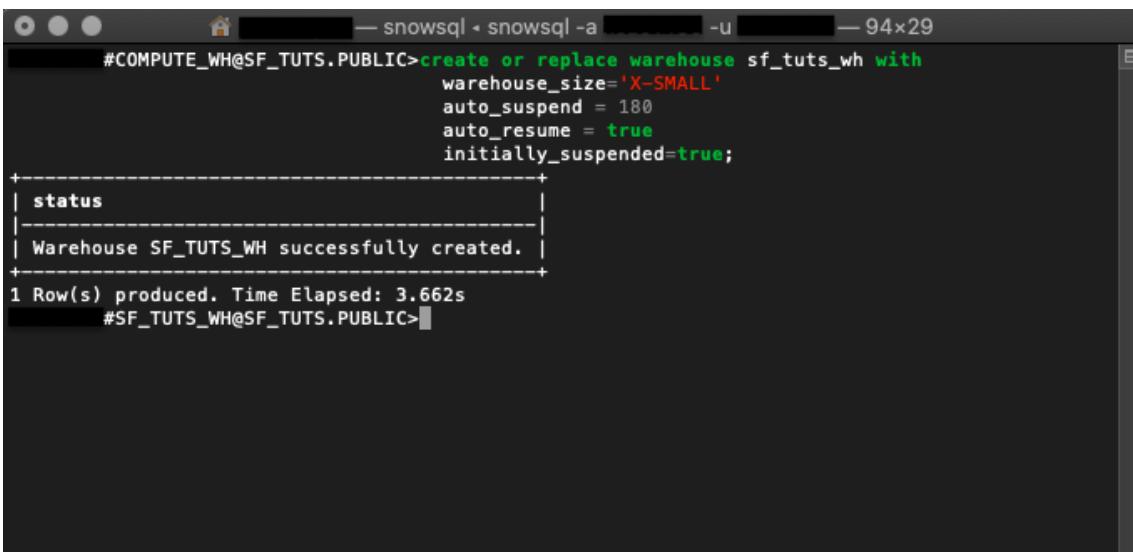


```
#COMPUTE_WH@SF_TUTS.PUBLIC>create or replace table emp_basic (
    first_name string ,
    last_name string ,
    email string ,
    streetaddress string ,
    city string ,
    start_date date
);
+-----+
| status |
+-----+
| Table EMP_BASIC successfully created. |
+-----+
1 Row(s) produced. Time Elapsed: 0.331s
#COMPUTE_WH@SF_TUTS.PUBLIC>
```

Make a Virtual Warehouse

```
create or replace warehouse sf_tuts_wh with
warehouse_size='X-SMALL'
auto_suspend = 180
auto_resume = true
initially_suspended=true;
```

After creation, this virtual warehouse will be active for your current session and begin running once the computing resources are needed.



```
#COMPUTE_WH@SF_TUTS.PUBLIC>create or replace warehouse sf_tuts_wh with
warehouse_size='X-SMALL'
auto_suspend = 180
auto_resume = true
initially_suspended=true;
+-----+
| status |
+-----+
| Warehouse SF_TUTS_WH successfully created. |
+-----+
1 Row(s) produced. Time Elapsed: 3.662s
#SF_TUTS_WH@SF_TUTS.PUBLIC>
```

With the database objects ready, you'll employ SnowSQL to move the sample data onto the `emp_basic` table.

Upload Data

In this section, you'll stage your sample CVS employee files and execute a SQL command to copy the data onto your table.

If you have not already done so, you can download the sample files here:

[Download Sample Data](#)

Stage Files With PUT

Linux

```
put file:///tmp/employees0*.csv @<database-name>.<schema-name>.%<table-name>;
```

Windows

```
put file://c:\temp\employees0*.csv @sf_tuts.public.%emp_basic;
```

- `file` specifies the *local* file path of the files to be staged. File paths are OS-specific.
- `@<database-name>.<schema-name>.%<table-name>` is the specific database, schema, and table the staged files are headed.
- The `@` sign before the database and schema name `@sf_tuts.public` indicates that the files are being uploaded to an internal stage, rather than an external stage. The `%` sign before the table name `%emp_basic` indicates that the internal stage being used is the stage for the table. For more details about stages, see [Staging Data Files from a Local File System](#).

```
put file:///tmp/employees0*.csv @sf_tuts.public.%emp_basic;
```

Here is a PUT call to stage the sample employee CSV files from a macOS `file:///tmp/` folder onto the `emp_basic` table within the `sf_tuts` database.

```

— snowsql - snowsql -a -u — 80x37
#SF_TUTS_WH@SF_TUTS.PUBLIC>put file:///tmp/employees0*.csv @sf_tuts.public.%emp_basic;
employees03.csv_c.gz(0.00MB): [#####] 100.00% Done (1.724s, 0.00MB/s).
employees01.csv_c.gz(0.00MB): [#####] 100.00% Done (1.203s, 0.00MB/s).
employees02.csv_c.gz(0.00MB): [#####] 100.00% Done (1.224s, 0.00MB/s).
employees04.csv_c.gz(0.00MB): [#####] 100.00% Done (0.790s, 0.00MB/s).
employees05.csv_c.gz(0.00MB): [#####] 100.00% Done (0.566s, 0.00MB/s).
+-----+
| source          | target          | source_size | target_size | source_compression | target_compression | status      | message   |
|-----+-----+-----+-----+-----+-----+-----+-----+
| employees01.csv | employees01.csv.gz | 370        | 288        | GZIP           | GZIP            | UPLOADED   |          |
| employees02.csv | employees02.csv.gz | 364        | 276        | GZIP           | GZIP            | UPLOADED   |          |
| employees03.csv | employees03.csv.gz | 407        | 298        | GZIP           | GZIP            | UPLOADED   |          |
| employees04.csv | employees04.csv.gz | 375        | 290        | GZIP           | GZIP            | UPLOADED   |          |
| employees05.csv | employees05.csv.gz | 404        | 303        | GZIP           | GZIP            | UPLOADED   |          |
+-----+
5 Row(s) produced. Time Elapsed: 5.661s
#SF_TUTS_WH@SF_TUTS.PUBLIC>

```

LIST Staged Files

```
list @<database-name>.<schema-name>.%<table-name>;
```

To check your staged files, run the `list` command.

```
list @sf_tuts.public.%emp_basic;
```

The example command above is to output the staged files for the `emp_basic` table.

```

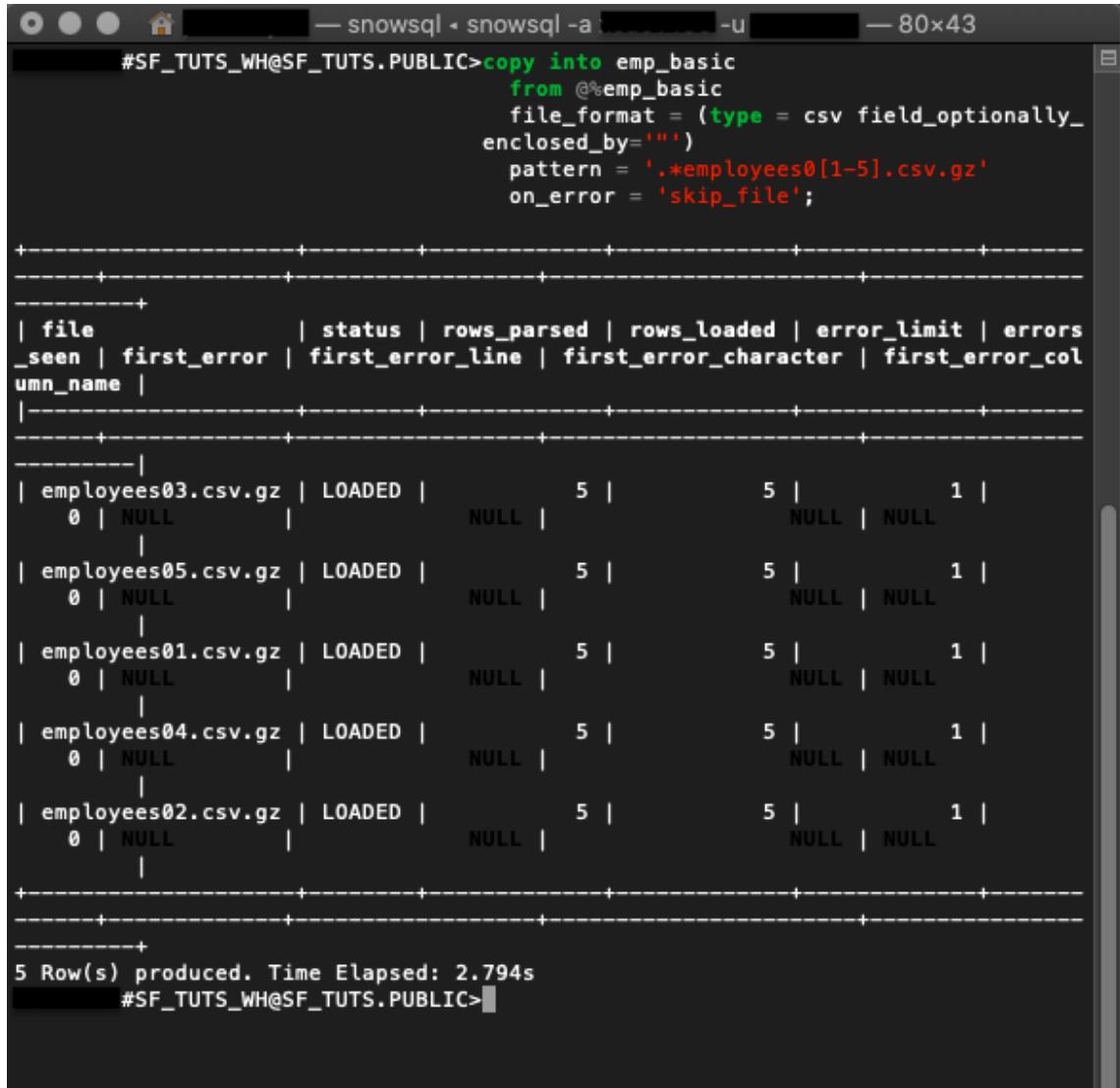
— snowsql - snowsql -a -u — 100x50
#SF_TUTS_WH@SF_TUTS.PUBLIC>list @sf_tuts.public.%emp_basic;
+-----+-----+-----+-----+
| name    | size  | md5          | last_modified |
|-----+-----+-----+-----+
| employees01.csv.gz | 304  | 63451cc8f4db701b3918d2475aa360a2 | GMT          |
| employees02.csv.gz | 288  | fc1e7fcfed992fd4f2d3da3464c7be0e4 | GMT          |
| employees03.csv.gz | 304  | 0f687fe91584c057a461163999a3baf3 | GMT          |
| employees04.csv.gz | 304  | adc308d8224f2d432cb57c91f6b36e5e | GMT          |
| employees05.csv.gz | 304  | d4c1882c6eafcdf3d70b9f6df18ac4b0 | GMT          |
+-----+-----+-----+-----+
5 Row(s) produced. Time Elapsed: 0.797s
#SF_TUTS_WH@SF_TUTS.PUBLIC>

```

[COPY INTO] Your Table

```
copy into emp_basic
  from @%emp_basic
  file_format = (type = csv field_optionally_enclosed_by='')
  pattern = '.*employees0[1-5].csv.gz'
  on_error = 'skip_file';
```

After getting the files staged, the data is copied into the `emp_basic` table. This DML command also auto-resumes the virtual warehouse made earlier.



The screenshot shows a terminal window with the title bar "snowsql - snowsql -a [REDACTED] -u [REDACTED] - 80x43". The command entered is:

```
#SF_TUTS_WH@SF_TUTS.PUBLIC>copy into emp_basic
  from @%emp_basic
  file_format = (type = csv field_optionally_
enclosed_by='')
  pattern = '.*employees0[1-5].csv.gz'
  on_error = 'skip_file';
```

The output shows the status of five files being loaded:

file	status	rows_parsed	rows_loaded	error_limit	errors
employees03.csv.gz	LOADED	5	5	1	1
employees05.csv.gz	LOADED	5	5	1	1
employees01.csv.gz	LOADED	5	5	1	1
employees04.csv.gz	LOADED	5	5	1	1
employees02.csv.gz	LOADED	5	5	1	1

At the bottom, it says "5 Row(s) produced. Time Elapsed: 2.794s".

The output indicates if the data was successfully copied and records any errors.

Query Data

With your data in the cloud, you need to know how to query it. We'll go over a few calls that will put your data on speed-dial.

- The `select` command followed by the wildcard `*` returns all rows and columns in `<table-name>`.

```
select * from emp_basic;
```

Here is an example command to `select` everything on the `emp_basic` table.

FIRST_NAME	LAST_NAME	EMAIL	STREETADDRESS	CITY	START_DATE
Althea	Featherstone	afeatherston@sf_tuts.com	8172 Browning Street, Apt B	Calatrava	2017-07-12
Hollis	Anneslie	hanneslie@sf_tuts.com	3249 Roth Park	Aleysk	2017-11-16
Betti	Cicco	bcicco@sf_tuts.com	121 Victoria Junction	Sinegor'ye	2017-06-22
Brendon	Durnall	bdurnalld@sf_tuts.com	26814 Weeping Birch Place	Sabadell	2017-11-14
Kylila	MacConnal	kmacconnale@sf_tuts.com	04 Valley Edge Court	Qingshu	2017-06-22
Arlene	Davidovits	adavidovitsk@sf_tuts.com	7571 New Castle Circle	Meniko	2017-05-03
Violette	Shermore	vshermore@sf_tuts.com	899 Merchant Center	Troitsk	2017-01-19
Ron	Mattys	rmattysm@sf_tuts.com	423 Lien Pass	Bayaguana	2017-11-15
Shurlocke	Oluwatoyin	soluwatoyinn@sf_tuts.com	40637 Portage Avenue	Seménovskoye	2017-09-12
Granger	Bassford	gbassford@sf_tuts.co.uk	6 American Ash Circle	Karditsa	2016-12-30
Lem	Boissier	lboissier@sf_tuts.com	3002 Ruskin Trail	Shikāpur	2017-08-25
Iain	Hanks	ihanks1@sf_tuts.com	2 Pankratz Hill	Monte-Carlo	2017-12-10
Avo	Laudham	alaudham2@sf_tuts.com	6948 Debs Park	Prażmów	2017-10-18
Emili	Cornner	ecornner3@sf_tuts.com	177 Magdelaine Avenue	Norrköping	2017-08-13
Harrietta	Goolding	hgoolding4@sf_tuts.com	458 Heath Trail	Osielsko	2017-11-27
Wallis	Sizey	wsizseyf@sf_tuts.com	36761 American Lane	Taibao	2016-12-30
Di	McGowan	dmcgowran@sf_tuts.com	1856 Maple Lane	Banjar Bengkelgede	2017-04-22
Carson	Bedder	cbedderh@sf_tuts.co.au	71 Clyde Gallagher Place	Leninskoye	2017-03-29
Dana	Avery	davoryi@sf_tuts.com	2 Holy Cross Pass	Wenlin	2017-05-11
Ronny	Talmadge	rtalmadgej@sf_tuts.co.uk	588 Chinook Street	Yawata	2017-06-02
Nyssa	Dorgan	ndorgan5@sf_tuts.com	7 Tomscot Way	Pampas Chico	2017-04-13
Catherin	Devereu	cdevereu6@sf_tuts.co.au	535 Basil Terrace	Magapit	2016-12-17
Grazia	Glaserman	gglaserman7@sf_tuts.com	162 Debra Lane	Shiquanhe	2017-06-06
Ivett	Casemore	icasemore8@sf_tuts.com	84 Holmberg Pass	Campina Grande	2017-03-29
Cesar	Hovie	chovie9@sf_tuts.com	5 7th Pass	Miami	2016-12-21

25 Row(s) produced. Time Elapsed: 2.603s
#SF_TUTS_WH@SF_TUTS.PUBLIC>

Sifting through everything on your table may not be the best use of your time. Getting specific results are simple, with a few functions and some query syntax.

- [WHERE] is an additional clause you can add to your select query.

```
select * from emp_basic where first_name = 'Ron';
```

This query returns a list of employees by the `first_name` of 'Ron' from the `emp_basic` table.

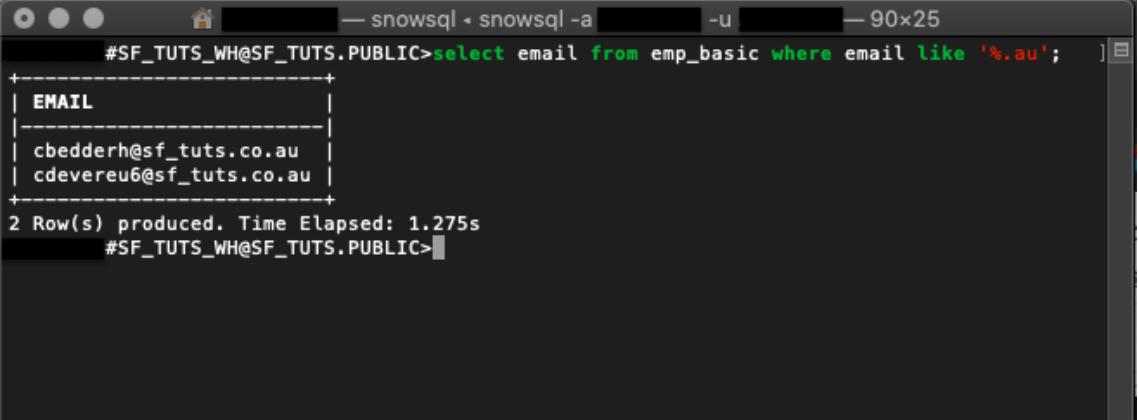
FIRST_NAME	LAST_NAME	EMAIL	STREETADDRESS	CITY	START_DATE
Ron	Mattys	rmattysm@sf_tuts.com	423 Lien Pass	Bayaguana	2017-11-15

1 Row(s) produced. Time Elapsed: 1.421s
#SF_TUTS_WH@SF_TUTS.PUBLIC>

- [LIKE] function supports wildcard `%` and `_`.

```
select email from emp_basic where email like '%.au';
```

The like function checks all emails in the `emp_basic` table for `.au` and returns a record.



A screenshot of a terminal window titled "snowsql - snowsql - 90x25". The command executed is `#SF_TUTS_WH@SF_TUTS.PUBLIC>select email from emp_basic where email like '%.au';`. The output shows two rows of data:

EMAIL
cbedderh@sf_tuts.co.au
cdevereu6@sf_tuts.co.au

Below the table, the message "2 Row(s) produced. Time Elapsed: 1.275s" is displayed. The prompt "#SF_TUTS_WH@SF_TUTS.PUBLIC>" is at the bottom.

Snowflake supports many [functions], [operators], and [commands]. However, if you need more specific tasks performed, consider setting up an [external function].

Manage and Delete Data

Often data isn't static. We'll review a few common ways to maintain your cloud database.

If HR updated the CSV file after hiring another employee, downloading, staging, and copying the whole CSV would be tedious. Instead, simply insert the new employee information into the targeted table.

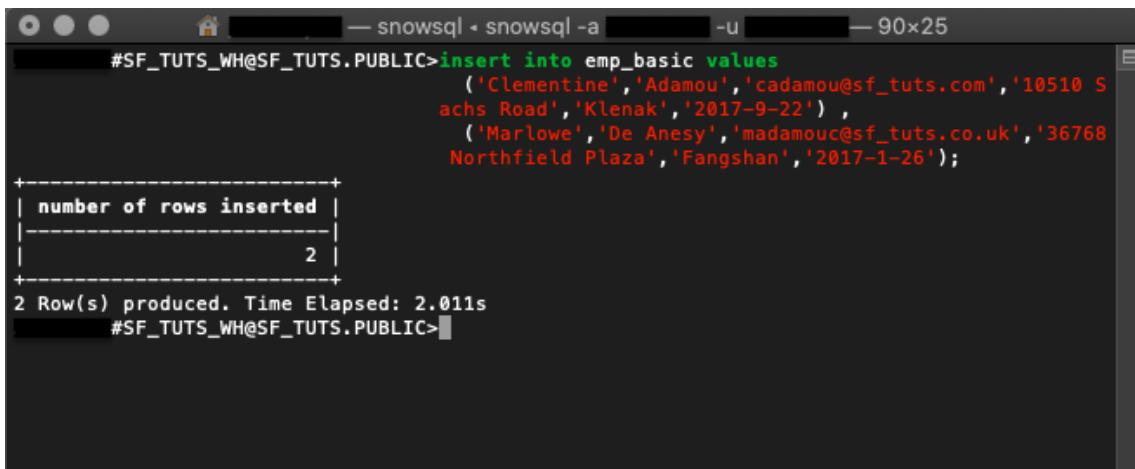
Insert Data

[INSERT] will update a table with additional values.

```
insert into emp_basic values
('Clementine','Adamou','cadamou@sf_tuts.com','10510 Sachs Road','Klenak','2017-9-22'),
('Marlowe','De Anesy','madamouc@sf_tuts.co.uk','36768 Northfield Plaza','Fangshan','2017-1-26');
```

Drop Objects

In the command displayed, `insert` is used to add two new employees to the `emp_basic` table.



A screenshot of a terminal window titled "snowsql - snowsql -a -u - 90x25". The command entered is:

```
#SF_TUTS_WH@SF_TUTS.PUBLIC>insert into emp_basic values
    ('Clementine','Adamou','cadamou@sf_tuts.com','10510 S
     aches Road','Klenak','2017-9-22') ,
    ('Marlowe','De Anesy','madamouc@sf_tuts.co.uk','36768
     Northfield Plaza','Fangshan','2017-1-26');
```

The output shows a table with one row:

number of rows inserted
2

2 Row(s) produced. Time Elapsed: 2.011s

#SF_TUTS_WH@SF_TUTS.PUBLIC>

- [DROP] objects no longer in use.

```
drop database if exists sf_tuts;
drop warehouse if exists sf_tuts_wh;
```

After practicing the basics covered in this tutorial, you'll no longer need the `sf-tuts` database and warehouse. To remove them altogether, use the `drop` command.

- Close Your Connection with `!exit` or `!disconnect`

For security reasons, it's best not to leave your terminal connection open unnecessarily. Once you're ready to close your SnowSQL connection, simply enter `!exit`.

Conclusion

Use SnowSQL for Your Application

You've created a Snowflake account, set up a cloud database with compute resources, and migrated data to the cloud with SnowSQL. Nice work! There are many advantages to using the cloud. Now that you know how easy getting started with Snowflake is, it's time to consider your next steps.

With your firm grasp of loading data with SnowSQL, start using it to run your application. Continue by [developing an application] with SnowSQL to learn how to connect your data to a Python application. If you already have application data, consider migrating it to the cloud with the same steps we used to complete the `emp_basic` table. Snowflake's tools and documentation are extensive and give you the power of cloud computing.

What we've covered

- SnowSQL setup
- Uploading data using SnowSQL
- Querying data using SnowSQL
- Managing and deleting data using SnowSQL

Extract Attributes from DICOM Files using a Java UDF

Overview

This lab is designed to help you understand the capabilities included in Snowflake's support for unstructured data and Snowpark. Although this guide is specific to processing DICOM files, you can apply this pattern of processing natively in Snowflake to many types of unstructured data. All source code for this guide can be found on [Github](#).

What You'll Need

- Snowflake account
- [SnowSQL] installed

What You'll Learn

- How to access DICOM files in cloud storage from Snowflake
- How to extract attributes from DICOM files natively using a Java User-Defined Function (UDF)

What You'll Build

- An external stage to access files in S3 from Snowflake
- A user-defined function using Snowflake's engine to process files

Prepare Your 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**.



Ready to start using Worksheets and Dashboards?

Snowflake's next-generation Worksheets and Dashboards are ready to be enabled for your account.

Whenever you're ready, click the button below to enable Worksheets and Dashboards for all users in your account. Worksheets will still be available in the Classic UI.

[Enable Worksheets and Dashboards](#)

Access the Data

Let's start by accessing DICOM files from 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.

For this lab, we will use an external stage, but processing and analysis workflows demonstrated in this lab can also be done using an internal stage.

Create a Database, Warehouse, and Stage

Let's create a database, warehouse, and stage that will be used for processing the files. 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 dicom;
create or replace warehouse quickstart;

use database dicom;
use schema public;
use warehouse quickstart;

create or replace stage dicom_external
url="s3://sfquickstarts/Extract DICOM Attributes/DICOM/"
directory = (enable = TRUE);
```

Verify if the DICOM files are accessible in your external stage by entering the following command on your Snowflake worksheet.

```
ls @dicom_external;
```

You should now see an identical list of files from the S3 bucket. Make sure you see 24 files.

	name	...	size
1	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0000 AGE_0060 CONTRAS		528,130
2	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0001 AGE_0069 CONTRAS		528,118
3	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0002 AGE_0074 CONTRAS		527,528
4	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0003 AGE_0075 CONTRAS		527,402
5	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0004 AGE_0056 CONTRAS		528,158
6	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0005 AGE_0048 CONTRAS		527,660
7	s3://sfquickstarts/Extract DICOM Attributes/DICOM/ID_0006 AGE_0075 CONTRAS		527,402

Extract Attributes from DICOM Files

In this section, we want to extract attributes from the DICOM files. The entities extracted are going to be fields like manufacturer, patient position, and study date. The goal is to have these fields to enrich the file-level metadata for analytics.

Creating a Java UDF in Snowflake

The Java code to parse DICOM files requires some dependencies. Instead of downloading those jar files and uploading to an internal stage, you can create an external stage and reference them when creating a UDF inline.

```
-- Create stage to store the jar file
create or replace stage jars_stage_internal;

-- Create external stage to import jars from S3
create or replace stage jars_stage
url = "s3://sfquickstarts/Common JARs/"
directory = (enable = true auto_refresh = false);

-- Create a java function to parse DICOM files
create or replace function read_dicom(file string)
returns String
language java
imports = ('@jars_stage/dcm4che-core-5.24.2.jar', '@jars_stage/log4j-1.2.17.jar',
           '@jars_stage/slf4j-api-1.7.30.jar', '@jars_stage/slf4j-log4j12-1.7.30.jar',
           '@jars_stage/gson-2.8.7.jar')
HANDLER = 'DicomParser.Parse'
```

```

as
$$
import org.dcm4che3.data.Attributes;
import org.dcm4che3.data.Tag;
import org.dcm4che3.io.DicomInputStream;
import org.xml.sax.SAXException;

import java.io.*;
import java.util.HashMap;
import java.util.Map;

import com.google.gson.Gson;

import com.snowflake.snowpark_java.types.SnowflakeFile;

public class DicomParser {
    public static String Parse(String file_url) throws IOException {
        SnowflakeFile file = SnowflakeFile.newInstance(file_url);
        String jsonStr = null;

        try {
            DicomInputStream dis = new DicomInputStream(file.getInputStream());
            DicomInputStream.IncludeBulkData includeBulkData =
DicomInputStream.IncludeBulkData.URI;
            dis.setIncludeBulkData(includeBulkData);
            Attributes attrs = dis.readDataset(-1, -1);

            Map<String, String> attributes = new HashMap<String, String>();
            attributes.put("PerformingPhysicianName",
attrs.getString(Tag.PerformingPhysicianName));
            attributes.put("PatientName", attrs.getString(Tag.PatientName));
            attributes.put("PatientBirthDate",
attrs.getString(Tag.PatientBirthDate));
            attributes.put("Manufacturer", attrs.getString(Tag.Manufacturer));
            attributes.put("PatientID", attrs.getString(Tag.PatientID));
            attributes.put("PatientSex", attrs.getString(Tag.PatientSex));
            attributes.put("PatientWeight", attrs.getString(Tag.PatientWeight));
            attributes.put("PatientPosition", attrs.getString(Tag.PatientPosition));
            attributes.put("StudyID", attrs.getString(Tag.StudyID));
            attributes.put("PhotometricInterpretation",
attrs.getString(Tag.PhotometricInterpretation));
            attributes.put("RequestedProcedureID",
attrs.getString(Tag.RequestedProcedureID));
            attributes.put("ProtocolName", attrs.getString(Tag.ProtocolName));
            attributes.put("ImagingFrequency",
attrs.getString(Tag.ImagingFrequency));
            attributes.put("StudyDate", attrs.getString(Tag.StudyDate));
            attributes.put("StudyTime", attrs.getString(Tag.StudyTime));
            attributes.put("ContentDate", attrs.getString(Tag.ContentDate));
            attributes.put("ContentTime", attrs.getString(Tag.ContentTime));
            attributes.put("InstanceCreationDate",
attrs.getString(Tag.InstanceCreationDate));
        }
    }
}

```

```

        attributes.put("SpecificCharacterSet",
attrs.getString(Tag.SpecificCharacterSet));
        attributes.put("StudyDescription",
attrs.getString(Tag.StudyDescription));
        attributes.put("ReferringPhysicianName",
attrs.getString(Tag.ReferringPhysicianName));
        attributes.put("ImageType", attrs.getString(Tag.ImageType));
        attributes.put("ImplementationVersionName",
attrs.getString(Tag.ImplementationVersionName));
        attributes.put("TransferSyntaxUID",
attrs.getString(Tag.TransferSyntaxUID));

        Gson gsonObj = new Gson();
        jsonStr = gsonObj.toJson(attributes);
    }
    catch (Exception exception) {
        System.out.println("Exception thrown :" + exception.toString());
        throw exception;
    }

    return jsonStr;
}
}
$$;

```

Invoking the Java UDF

The UDF can be invoked on any DICOM file with a simple SQL statement. First, make sure to refresh the directory table metadata for your external stage.

```

alter stage dicom_external refresh;

select read_dicom('@dicom_external/ID_0067_AGE_0060_CONTRAST_0_CT.dcm')
as dicom_attributes;

```

DICOM_ATTRIBUTES	
1	{"PatientPosition": "HFS", "SpecificCharacterSet": "ISO_IR 100", "ProtocolName": "1WBPETCT", "StudyDate": "19860422", "PatientName": "TCGA-17-Z058", "ImageType": "ORIGINAL", "StudyTime": "111534.486000", "PatientWeight": "70.824", "ContentTime": "111754.509051", "PhotometricInterpretation": "MONOCHROME2", "PatientSex": "M", "PatientID": "TCGA-17-Z058", "ContentDate": "19860422", "Manufacturer": "SIEMENS"}

Aa DICOM_ATTRIBUTES	
1	{"PatientPosition": "HFS", "SpecificCharacterSet": "ISO_IR 100", "ProtocolName": "1WBPETCT", "StudyDate": "19860422", "PatientName": "TCGA-17-Z058", "ImageType": "ORIGINAL", "StudyTime": "111534.486000", "PatientWeight": "70.824", "ContentTime": "111754.509051", "PhotometricInterpretation": "MONOCHROME2", "PatientSex": "M", "PatientID": "TCGA-17-Z058", "ContentDate": "19860422", "Manufacturer": "SIEMENS"}

The output is key-value pairs extracted from `ID_0067_AGE_0060_CONTRAST_0_CT.dcm`.

```
{
  "PatientPosition": "HFS",
  "SpecificCharacterSet": "ISO_IR 100",
  "ProtocolName": "1WBPETCT",
  "StudyDate": "19860422",
  "PatientName": "TCGA-17-Z058",
  "ImageType": "ORIGINAL",
  "StudyTime": "111534.486000",
  "PatientWeight": "70.824",
  "ContentTime": "111754.509051",
  "PhotometricInterpretation": "MONOCHROME2",
  "PatientSex": "M",
  "PatientID": "TCGA-17-Z058",
  "ContentDate": "19860422",
  "Manufacturer": "SIEMENS"
}
```

```

    "PatientWeight": "70.824",
    "ContentTime": "111754.509051",
    "PhotometricInterpretation": "MONOCHROME2",
    "PatientSex": "M",
    "PatientID": "TCGA-17-Z058",
    "ContentDate": "19860422",
    "Manufacturer": "SIEMENS"
}

```

UDFs are account-level objects. So if a developer familiar with Java creates a UDF, an analyst in the same account with proper permissions can invoke the UDF in their queries.

Extracting and Storing Attributes

We want to store the extracted attributes as columns in a table for analysts to be able to query, analyze, and retrieve files. This can be done easily with Snowflake's native support for semi-structured data.

```

create or replace table dicom_attributes as
select
    relative_path,
    file_url,
    parse_json(read_dicom('@dicom_external/' || relative_path)) as data,
    data:PatientName::string as PatientName,
    data:PatientID::string as PatientID,
    to_date(data:StudyDate::string, 'yyyyMMdd') as StudyDate,
    data:StudyTime::string as StudyTime,
    data:StudyDescription::string as StudyDescription,
    data:ImageType::string as ImageType,
    data:PhotometricInterpretation::string as PhotometricInterpretation,
    data:Manufacturer::string as Manufacturer,
    data:PatientPosition::string as PatientPosition,
    data:PatientSex::string as PatientSex,
    data:PerformingPhysicianName::string as PerformingPhysicianName,
    data:ImagingFrequency::string as ImagingFrequency,
    data:ProtocolName::string as ProtocolName
from directory(@dicom_external);

```

If you collapse and expand the `DICOM` database in the Objects pane on the left, you should now see a table named `DICOM_ATTRIBUTES`. Click on that table, and below you should see a preview of the fields you have created along with icons to indicate the data type. You can also see a preview of the view by clicking on the button that looks like a magnifier glass.

```

Pinned ⓘ
No pinned objects
...
as dicom_attributes;
create or replace table dicom_attributes as
select
relative_path,
file_name,
port.read('dicom/`dicom_external`/' || relative_path) as data,
data.PatientName:string as PatientName,
data.PatientID:string as PatientID,
to_date(data.StudyDate:'string','yyyymmdd') as StudyDate,
data.StudyDescription:string as StudyDescription,
data.ImageType:string as ImageType,
data.PhotometricInterpretation:string as PhotometricInterpretation,
data.Manufacturer:string as Manufacturer,
data.PatientPosition:string as PatientPosition,
data.PatientSex:string as PatientSex,
data.PerformingPhysicianName:string as PerformingPhysicianName,
...

```

status

1 Table DICOM_ATTRIBUTES successfully created.

Query Details

Query duration 8.1s
Rows 1

status 100% filled

One of the many new things you can do with Snowsight is quickly see summary statistics and distributions of field values in query results. For example, select the entire table `EXTRACTED_DICOM_ATTRIBUTES`. Then in the query results, click on the columns such as `MANUFACTURER`, `PATIENTPOSITION`, and `PATIENTSEX` to see the distribution of values in each column.

```
select * from dicom_attributes;
```

	STUDYDESCRIPTION	IMAGETYPE	PHOTOMETRICINTERPRETATION	MANUFACTURER	PATIENTPOSITION	PATIENTSEX	PERFORMING	Column
1	000 null	ORIGINAL	MONOCHROME2	SIEMENS	HFS	M	null	Aa PATIENTPOSITION
2	PET WB OR REG RESTAG HEAD+NECK	ORIGINAL	MONOCHROME2	SIEMENS	HFS	F	null	100% filled
3	000 null	ORIGINAL	MONOCHROME2	SIEMENS	HFS	F	null	4 distinct values
4	PET / CT TUMOR IMAGING	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	HFS	M	null	HFS
5	000 Outside Read or Comparison PET	ORIGINAL	MONOCHROME2	SIEMENS	HFS	M	null	FFS
6	000 null	ORIGINAL	MONOCHROME2	SIEMENS	HFS	M	null	HFSL
7	PET CT SKULL BASE TO MID-THIGH	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	HFS	F	null	FFP
8	CT CHEST WITHOUT CONTR	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	FFS	M	null	
9	CT CHEST WITHOUT CONTR	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	FFS	M	null	
10	CT LIMITED OR LOCALIZE	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	HFSL	M	null	
11	PET / CT TUMOR IMAGING	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	HFS	M	null	
12	PET CT SKULL BASE TO MID-THIGH	ORIGINAL	MONOCHROME2	GE MEDICAL SYSTEMS	HFS	M	null	
13	000 null	ORIGINAL	MONOCHROME2	SIEMENS	HFS	M	null	
14	000 null	ORIGINAL	MONOCHROME2	SIEMENS	HFS	M	null	

Conclusion

Congratulations! You used Snowflake to extract attributes from DICOM files.

What we've covered

- Accessing unstructured data with an **external stage**
- Processing unstructured data with a **Java UDF**

Analyze PDF Invoices using Java UDF and Snowsight

Overview

This lab is designed to help you understand the capabilities included in Snowflake's support for unstructured data and Snowpark. Although this guide is specific to processing PDF files, you can apply this pattern of processing natively in Snowflake to many types of unstructured data. All source code for this guide can be found on [Github](#).

What You'll Need

- Snowflake account
- SnowSQL installed

What You'll Learn

- How to access PDF invoices in cloud storage from Snowflake
- How to extract text from PDFs natively using a Java User-Defined Function (UDF)

What You'll Build

- An external stage to access files in S3 from Snowflake
- A user-defined function using Snowflake's engine to process files

Prepare Your 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**.



Ready to start using Worksheets and Dashboards?

Snowflake's next-generation Worksheets and Dashboards are ready to be enabled for your account.

Whenever you're ready, click the button below to enable Worksheets and Dashboards for all users in your account. Worksheets will still be available in the Classic UI.

[Enable Worksheets and Dashboards](#)

Access the Data

Let's start by loading the PDF invoices 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.

For this lab, we will use an external stage, but processing and analysis workflows demonstrated in this lab can also be done using an internal stage.

Create a Database, Warehouse, and Stage

Let's create a database, warehouse, and stage that will be used for loading and processing the PDFs. 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 pdf;
create or replace warehouse quickstart;

use database pdf;
use schema public;
use warehouse quickstart;

create or replace stage pdf_external
url="s3://sfquickstarts/Analyze PDF Invoices/Invoices/"
directory = (enable = TRUE);
```

Verify if the PDF files are accessible in your external stage by entering the following command on your Snowflake worksheet.

```
ls @pdf_external;
```

You should now see an identical list of files from the S3 bucket. Make sure you see 300 files.

The screenshot shows a Snowflake interface with a query results table. The table has two columns: 'name' and 'size'. The 'name' column lists file paths starting with 's3://sfquickstarts/Analyze PDF Invoices/Invoices/'. The 'size' column shows file sizes in bytes. The table contains 7 rows, with the first row being index 1 and the last row being index 7. The total number of files (300) is indicated at the top left of the table area.

	name	...	size
1	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice1.pdf		13,636
2	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice10.pdf		13,639
3	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice100.pdf		13,653
4	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice101.pdf		13,171
5	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice102.pdf		12,962
6	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice103.pdf		13,269
7	s3://sfquickstarts/Analyze PDF Invoices/Invoices/invoice104.pdf		12,683

Extract Text from PDFs

In this section, we want to extract attributes from the PDF invoices. The entities extracted are going to be fields like product names, unit cost, total cost, as well as business name. The goal is to have these fields to enrich the file-level metadata for analytics.

Creating a Java UDF in Snowflake

The Java code to parse PDFs requires some dependencies. Instead of downloading those jar files and uploading to an internal stage, you can create an external stage and reference them when creating a UDF inline.

```
-- Create stage to store the JAR file
create or replace stage jars_stage_internal;

-- Create external stage to import PDFBox from S3
create or replace stage jars_stage
url = "s3://sfquickstarts/Common JARs/"
directory = (enable = true auto_refresh = false);

-- Create a java function to parse PDF files
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;

import com.snowflake.snowpark_java.types.SnowflakeFile;

public class PdfParser {

    public static String ReadFile(String file_url) throws IOException {
SnowflakeFile file = SnowflakeFile.newInstance(file_url);
        try (PDDocument document = PDDocument.load(file.getInputStream())) {

            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;
    }
}
$$;

```

Invoking the Java UDF

The UDF can be invoked on any PDF file with a simple SQL statement. First, make sure to refresh the directory table metadata for your external stage.

```

alter stage pdf_external refresh;

select read_pdf('@pdf_external/invoice1.pdf')
as pdf_text;

```

The screenshot shows a Snowflake query interface with the following code:

```

68 | select read_pdf('@pdf_external/invoice1.pdf')
69 |   as pdf_text;

```

The results pane displays the extracted text from the PDF file "invoice1.pdf". The text is organized into sections: INVOICE, BILL TO, DATE, BALANCE DUE, ITEM LIST, and TOTAL.

INVOICE

1
Abbott Group
Bill To:
Aug 5, 2021
\$458.10
Date:
Balance Due:
Item Quantity Rate Amount
Flour - Corn, Fine 18 \$11.39 \$205.02
Hold Up Tool Storage Rack 14 \$9.54 \$133.56
Scallop - St. Jaques 9 \$13.28 \$119.52
\$458.10Total:

The output is text values extracted from `invoice1.pdf`.

```

INVOICE
# 1
Abbott Group
Bill To:
Aug 5, 2021
$458.10
Date:
Balance Due:
Item Quantity Rate Amount
Flour - Corn, Fine 18 $11.39 $205.02
Hold Up Tool Storage Rack 14 $9.54 $133.56
Scallop - St. Jaques 9 $13.28 $119.52
$458.10Total:

```

UDFs are account-level objects. So if a developer familiar with Java creates a UDF, an analyst in the same account with proper permissions can invoke the UDF in their queries.

Extracting and Storing Fields

We want to store the extracted text 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 need to create a table with the extracted text in its raw form. From this table, we can create views to parse the text into various fields for easier analysis.

```

create or replace table parsed_pdf as
select
    relative_path
    , file_url
    , read_pdf('@pdf_external/' || relative_path) as parsed_text
from directory(@pdf_external);

```

Using Snowflake's string functions, we can parse out specific values as fields like balance due, item name, item quantity, and more.

```

create or replace view v__parsed_pdf_fields as (
with items_to_array as (
    select
        *
        , split(
            substr(
                regexp_substr(parsed_text, 'Amount\n(.*)\n(.*)\n(.*)'))

```

```

        ), 8
    ), '\n'
)
as items
from parsed_pdf
)
, parsed_pdf_fields as (
    select
        substr(regexp_substr(parsed_text, '# [0-9]+'), 2)::int as invoice_number
        , to_number(substr(regexp_substr(parsed_text, '\$[^A-Z]+'), 2), 10, 2) as
balance_due
        , substr(
            regexp_substr(parsed_text, '[0-9]+\n[^n]+')
            , len(regexp_substr(parsed_text, '# [0-9]+'))
        ) as invoice_from
        , to_date(substr(regexp_substr(parsed_text, 'To:\n[^n]+'), 5), 'mon dd,
yyyy') as invoice_date
        , i.value::string as line_item
        , parsed_text
    from
        items_to_array
        , lateral flatten(items_to_array.items) i
)
select
    invoice_number
    , balance_due
    , invoice_from
    , invoice_date
    , rtrim(regexp_substr(line_item, ' ([0-9]+ \$)::string, '$)::integer as
item_quantity
    , to_number(ltrim(regexp_substr(line_item, '\$[^ ]+'))::string, '$'), 10, 2) as
item_unit_cost
    , regexp_replace(line_item, ' ([0-9]+ \$.*', '')::string as item_name
    , to_number(ltrim(regexp_substr(line_item, '\$[^ ]+', 1, 2))::string, '$'), 10, 2)
as item_total_cost
from parsed_pdf_fields
);

```

If you collapse and expand the `PDF` database in the Objects pane on the left, you should now see a view name `V__PARSED_PDF_FIELDS`. Click on that view, and below you should see a preview of the fields you have created along with icons to indicate the data type. You can also see a preview of the view by clicking on the button that looks like a magnifier glass.

Pinned PDF

No pinned objects

Q Search

DEMOS_DB

PDF

SNOWFLAKE_SAMPLE_DATA

UTL_DB

```

 90     -- ...
 91     from parsed_pdf
 92
 93     . parsed_pdf_fields as (
 94         select
 95             substr(regexp_substr(parsed_text, '# [0-9]+'), 2)::int as invoice_number
 96             , to_number(substr(regexp_substr(parsed_text, '\$[A-Z]+'), 2), 10, 2) as balance_due
 97             , substr(
 98                 regexp_substr(parsed_text, '[0-9]+\n[^\\n]+')
 99                 , len(regexp_substr(parsed_text, '# [0-9]+'))
100             ) as invoice_from
101             , to_date(substr(regexp_substr(parsed_text, 'To:\n[^\\n]+'), 5), 'mon dd, yyyy') as invoice_date
102             , i.value::string as line_item
103             , parsed_text
104         from
105             items_to_array
106             . lateral flatten(items_to_array.items) i
107     )
108     select
109         invoice_number
110         , balance_due
111         , invoice_from
112         , invoice_date
113         , trim(regexp_substr(line_item, ' ([0-9]+ \\\\$):string, '$)::integer as item_quantity
114             , to_number(trim(regexp_substr(line_item, '\$\w+'):string, '$'), 10, 2) as item_unit_cost
115             , regexp_replace(line_item, '([0-9]+) \\\\$.*', '$::string as item_name
116             , to_number(trim(regexp_substr(line_item, '\$\w+', 1, 2)::string, '$'), 10, 2) as item_total_cost
117     );

```

Objects Query Results Chart

status

1 View V_PARSEDPDFFIELDS successfully created.

Query Details

Query duration 224ms

Rows 1

status Aa

100% filled

Exploring Invoice Data

Now let's explore the data from the PDF invoices. What are the most purchased items based on quantity?

```

select
    sum(item_quantity)
    , item_name
from v__parsed_pdf_fields
group by item_name
order by sum(item_quantity) desc
limit 10;

```

```

118     --what are the 10 most purchased items?
119     select
120         sum(item_quantity)
121         , item_name
122     from v__parsed_pdf_fields
123     group by item_name
124     order by sum(item_quantity) desc
125     limit 10;

```

Objects Query Results Chart

	SUM(ITEM_QUANTITY)	ITEM_NAME
1	203	Apron
2	178	Lettuce - California Mix
3	169	Bar Special K
4	159	Oil - Margarine
5	158	Flower - Commercial Bronze
6	155	Apple - Fuji
7	154	Wine - Rhine Riesling Wolf Blass
8	152	Garam Marsala
9	151	Phyllo Dough
10	146	Salmon - Smoked, Sliced

What are the items on which the most money was spent?

```

select
    sum(item_total_cost)
    , item_name

```

```

from v__parsed_pdf_fields
group by item_name
order by sum(item_total_cost) desc
limit 10;

```

```

126
127    --on which 10 items was the most money spent?
128    select
129        sum(item_total_cost)
130        , item_name
131    from v__parsed_pdf_fields
132    group by item_name
133    order by sum(item_total_cost) desc
134    limit 10;

```

Objects Query Results Chart

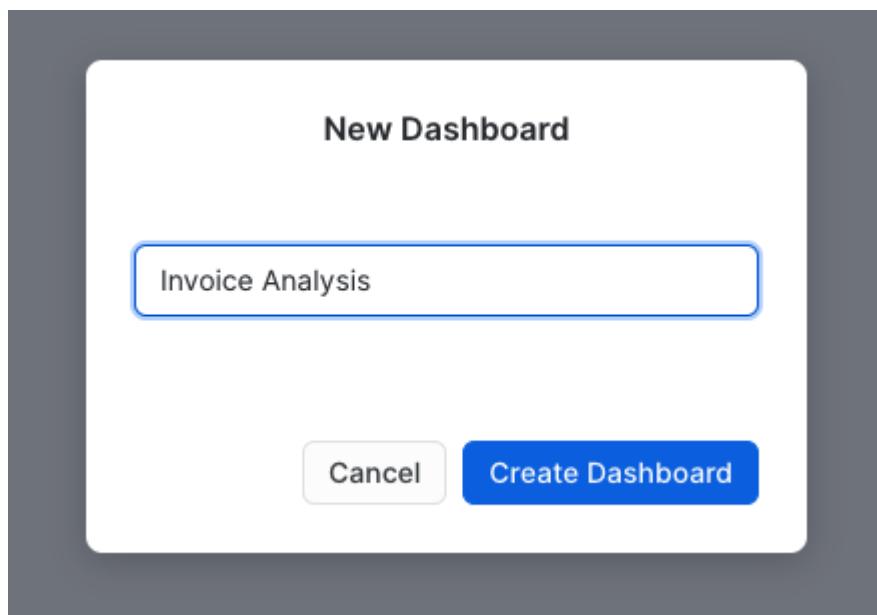
	SUM(ITEM_TOTAL_COST)	ITEM_NAME
1	3,214.68	Lettuce - California Mix
2	3,079.83	Oil - Margarine
3	2,644.85	Bar Special K
4	2,518.65	Pepper - Chili Powder
5	2,313.32	Phyllo Dough
6	2,173.98	Soup - Boston Clam Chowder
7	2,057.94	Sauce - Oyster
8	1,905.64	Cinnamon Rolls
9	1,865.92	Duck - Legs
10	1,825	Salmon - Smoked, Sliced

Analyze the Data with Snowsight

Now we can use Snowsight to visualize the data extracted from the PDF invoices.

Create a Dashboard

Let's use a dashboard as a collection of all of the visualizations we will create. Click on the Home button, then click on **Dashboards** in the pane on the left. Create a new dashboard by clicking the **+ Dashboard** button in the top-right. Name the dashboard `Invoice Analysis`, and click **Create Dashboard**.

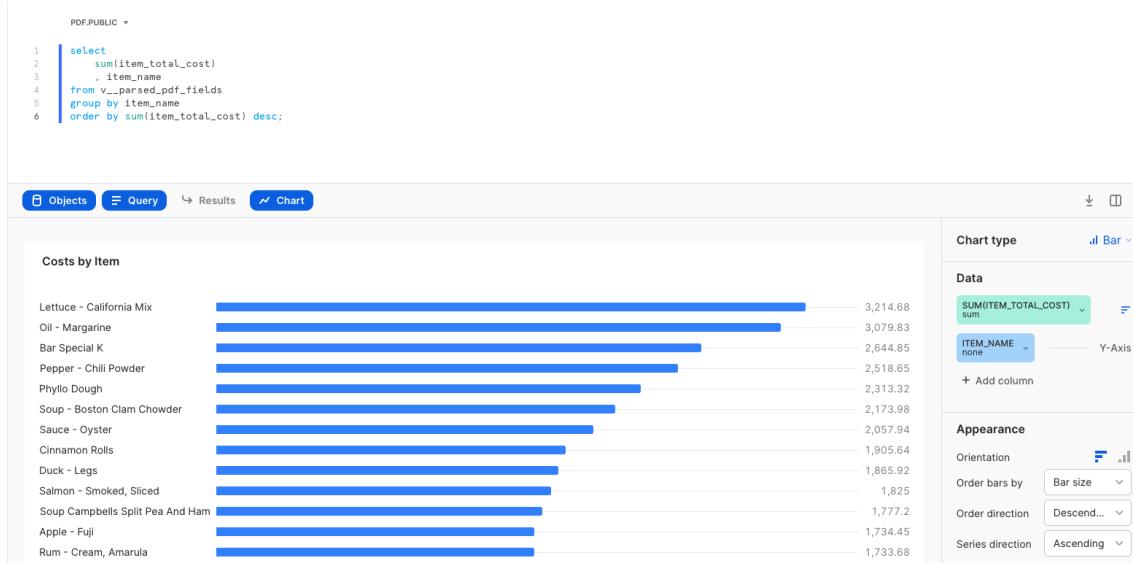


Now let's create the first tile on the Invoice Analysis dashboard by clicking the button **+ New Tile**. Give the tile a name by clicking on the timestamp at the top, and name it `Costs by Item`. In the canvas, copy/paste the SQL

below to query the data needed for the chart and run it.

```
select
    sum(item_total_cost)
    , item_name
from v__parsed_pdf_fields
group by item_name
order by sum(item_total_cost) desc;
```

Now click on **Chart**. Change the chart type from line to bar, X-Axis to `ITEM_NAME`, and orientation to horizontal. In this chart, you should see "Lettuce - California Mix" as the item on which the most money was spent, \$3,214.68.



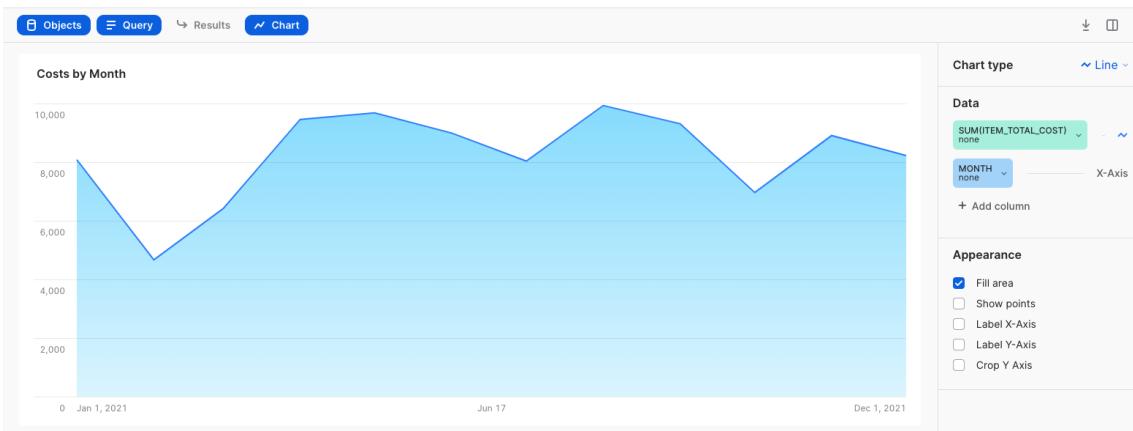
Now let's create another tile by clicking **Return to Invoice Analysis** in the top-left, then click on the + button, then **New Tile from Worksheet**. Name the tile `Costs by Month`. Now copy/paste and run this query.

```
select
    sum(item_total_cost)
    , date_trunc('month', invoice_date) as month
from v__parsed_pdf_fields
group by date_trunc('month', invoice_date);
```

Again, click **Chart**. You should see a line chart with `MONTH` on the x-axis and `SUM(ITEM_TOTAL_COST)` on the y-axis. Then click **Return to Invoice Analysis**. You should now see two tiles on your dashboard, which you can rearrange by clicking and dragging.

```
PDF.PUBLIC *
```

```
1 select
2     sum(item_total_cost)
3     , date_trunc('month', invoice_date) as month
4
5 from v__parsed.pdf_fields
group by date_trunc('month', invoice_date);
```



Conclusion

Congratulations! You used Snowflake to analyze PDF invoices.

What we've covered

- Accessing unstructured data with an **external stage**
- Processing unstructured data with a **Java UDF**
- Visualize data with **Snowsight**

Getting Started with Time Travel

Overview

Panic hits when you mistakenly delete data. Problems can come from a mistake that disrupts a process, or worse, the whole database was deleted. Thoughts of how recent was the last backup and how much time will be lost might have you wishing for a rewind button. Straightening out your database isn't a disaster to recover from with Snowflake's Time Travel. A few SQL commands allow you to go back in time and reclaim the past, saving you from the time and stress of a more extensive restore.

We'll get started in the Snowflake web console, configure data retention, and use Time Travel to retrieve historic data. Before querying for your previous database states, let's review the prerequisites for this guide.

What You'll Learn

- Snowflake account and user permissions
- Make database objects
- Set data retention timelines for Time Travel
- Query Time Travel data
- Clone past database states
- Remove database objects
- Next options for data protection

What You'll Need

- A [Snowflake](#) Account

What You'll Build

- Create database objects with Time Travel data retention

Get Started With the Essentials

First things first, let's get your Snowflake account and user permissions primed to use Time Travel features.

Create a Snowflake Account

Snowflake lets you try out their services for free with a [trial account](#). A **Standard** account allows for one day of Time Travel data retention, and an **Enterprise** account allows for 90 days of data retention. An **Enterprise** account is necessary to practice some commands in this tutorial.

Access Snowflake's Web Console

```
https://<account-name>.snowflakecomputing.com/console/login
```

Log in to the web interface on your browser. The URL contains your [account name] and potentially the region.

Increase Your Account Permission

Snowflake's web interface has a lot to offer, but for now, switch the account role from the default `SYSADMIN` to `ACCOUNTADMIN`. You'll need this increase in permissions later.



Now that you have the account and user permissions needed, let's create the required database objects to test drive Time Travel.

Generate Database Objects

Within the Snowflake web console, navigate to **Worksheets** and use a fresh worksheet to run the following commands.

Create Database

```
create or replace database timeTravel_db;
```

The screenshot shows a Snowflake worksheet interface. At the top, there is a toolbar with a 'Run' button, a 'Context' dropdown, and a three-dot menu. Below the toolbar, the query history shows a single query: 'create or replace database timeTravel_db;'. The results section is active, displaying the execution status: '145ms' and '1 rows'. A yellow box highlights the first row of the results table, which contains the message 'Database TIMETRAVEL_DB successfully created.'.

Use the above command to make a database called 'timeTravel_db'. The **Results** output will show a status message of Database TIMETRAVEL_DB successfully created.

Create Table

```
create or replace table timeTravel_table(ID int);
```

The screenshot shows a Snowflake worksheet interface. At the top, there is a toolbar with a 'Run' button, a 'Context' dropdown, and a three-dot menu. Below the toolbar, the query history shows a single query: 'create or replace table timeTravel_table(ID int);'. The results section is active, displaying the execution status: '240ms' and '1 rows'. A yellow box highlights the first row of the results table, which contains the message 'Table TIMETRAVEL_TABLE successfully created.'.

This command creates a table named 'timeTravel_table' on the timeTravel_db database. The **Results** output should show a status message of Table TIMETRAVEL_TABLE successfully created.

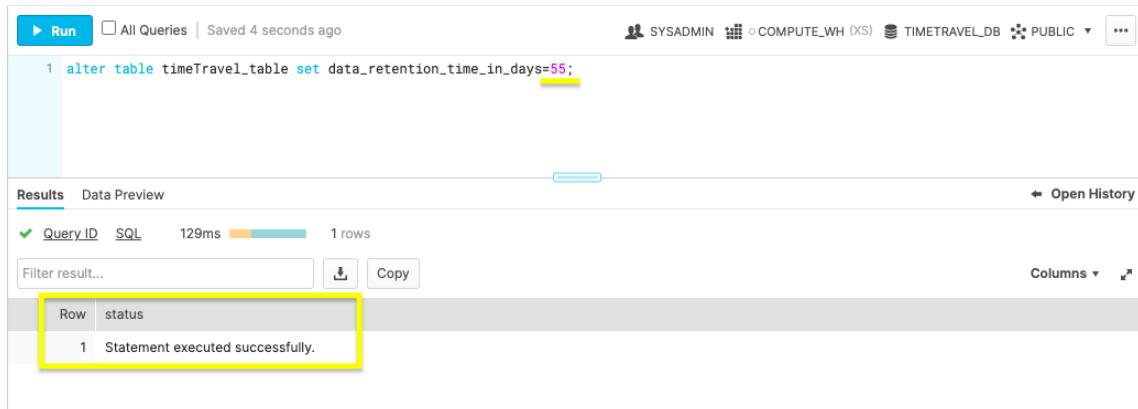
With the Snowflake account and database ready, let's get down to business by configuring Time Travel.

Prepare Your Database for Disaster

Be ready for anything by setting up data retention beforehand. The default setting is one day of data retention. However, if your one day mark passes and you need the previous database state back, you can't retroactively extend the data retention period. This section teaches you how to be prepared by preconfiguring Time Travel retention.

Alter Table

```
alter table timeTravel_table set data_retention_time_in_days=55;
```



The screenshot shows the Snowflake UI interface. At the top, there is a toolbar with a 'Run' button, a 'All Queries' dropdown, and a 'Saved 4 seconds ago' indicator. To the right are user roles (SYSADMIN, COMPUTE_WH (XS), TIMETRAVEL_DB, PUBLIC) and a three-dot menu. Below the toolbar, the query history shows a single entry: 'alter table timeTravel_table set data_retention_time_in_days=55;'. The results tab is selected, displaying a single row: 'Row' (1) and 'status' (Statement executed successfully). A yellow box highlights this row. The bottom of the interface includes a 'Filter result...' input, a 'Copy' button, and a 'Columns' dropdown.

The command above changes the table's data retention period to 55 days. If you opted for a **Standard** account, your data retention period is limited to the default of one day. An **Enterprise** account allows for 90 days of preservation in Time Travel.

Now you know how easy it is to [alter] your data retention, let's bend the rules of time by querying an old database state with Time Travel.

Query Your Time Travel Data

With your data retention period specified, let's turn back the clock with the `AT` and `BEFORE` [clauses].

At

```
select * from timeTravel_table at(timestamp => 'Fri, 23 Oct 2020 16:20:00 -0700'::timestamp);
```

Use `timestamp` to summon the database state **at** a specific date and time.

```
select * from timeTravel_table at(offset => -60*5);
```

Employ `offset` to call the database state **at** a time difference of the current time. Calculate the offset in seconds with math expressions. The example above states, `-60*5`, which translates to five minutes ago.

Before

```
select * from timeTravel_table before(statement => '<statement_id>');
```

If you're looking to restore a database state just **before** a transaction occurred, grab the transaction's statement id. Use the command above with your statement id to get the database state right before the transaction statement was executed.

By practicing these queries, you'll be confident in how to find a previous database state. After locating the desired database state, you'll need to get a copy by cloning in the next step.

Clone Past Database States

With the past at your fingertips, make a copy of the old database state you need with the `clone` keyword.

Clone Table

```
create table restoredTimeTravel_table clone timeTravel_table  
at(offset => -60*5);
```

The screenshot shows the Snowflake SQL interface. At the top, there is a toolbar with a 'Run' button, a 'All Queries' dropdown, and account information. Below the toolbar, the query history shows two rows:

```
1 create table restoredTimeTravel_table clone timeTravel_table  
2   at(offset => -60*5);
```

Below the query history, there are tabs for 'Results' and 'Data Preview'. The 'Results' tab is selected. It displays a table with one row:

Row	status
1	Table RESTOREDTIMETRAVEL_TABLE successfully created.

A yellow box highlights the status message 'Table RESTOREDTIMETRAVEL_TABLE successfully created.'

The command above creates a new table named `restoredTimeTravel_table` that is an exact copy of the table `timeTravel_table` from five minutes prior.

Cloning will allow you to maintain the current database while getting a copy of a past database state. After practicing the steps in this guide, remove the practice database objects in the next section.

Cleanup and Know Your Options

You've created a Snowflake account, made database objects, configured data retention, query old table states, and generate a copy of the old table state. Pat yourself on the back! Complete the steps to this tutorial by deleting the objects created.

Drop Table

```
drop table if exists timeTravel_table;
```

Run All Queries | Saved 33 seconds ago

ACCOUNTADMIN COMPUTE_WH (XS) TIMETRAVEL_DB PUBLIC ...

```
1 drop table if exists timeTravel_table;
2
```

Results Data Preview Open History

Query ID SQL 136ms 1 rows

Filter result... Copy Columns ▾

Row	status
1	TIMETRAVEL_TABLE successfully dropped.

By dropping the table before the database, the retention period previously specified on the object is honored. If a parent object(e.g., database) is removed without the child object(e.g., table) being dropped prior, the child's data retention period is null.

Drop Database

```
drop database if exists timeTravel_db;
```

Run All Queries | Saved 1 second ago

ACCOUNTADMIN COMPUTE_WH (XS) Select Database Select Schema ...

```
1 drop database if exists timeTravel_db;
```

Results Data Preview Open History

Query ID SQL 79ms 1 rows

Filter result... Copy Columns ▾

Row	status
1	TIMETRAVEL_DB successfully dropped.

With the database now removed, you've completed learning how to call, copy, and erase the past.

Building a Real-Time Data Vault in Snowflake

Overview

In this day and age, with the ever-increasing availability and volume of data from many types of sources such as IoT, mobile devices, and weblogs, there is a growing need, and yes, demand, to go from batch load processes to streaming or “real-time” (RT) loading of data. Businesses are changing at an alarming rate and are becoming more competitive all the time. Those that can harness the value of their data faster to drive better business outcomes will be the ones to prevail.

One of the benefits of using the Data Vault 2.0 architecture is that it was designed from inception not only to accept data loaded using traditional batch mode (which was the prevailing mode in the early 2000s when [Dan Linstedt](#) introduced Data Vault) but also to easily accept data loading in real or near-realtime (NRT). In the early 2000s, that was a nice-to-have aspect of the approach and meant the methodology was effectively future-proofed from that perspective. Still, few database systems had the capacity to support that kind of requirement. Today, RT or at least NRT loading is almost becoming a mandatory requirement for modern data platforms. Granted, not all loads or use cases need to be NRT, but most forward-thinking organizations need to onboard data for analytics in an NRT manner.

Those who have been using the Data Vault approach don’t need to change much other than figure out how to engineer their data pipeline to serve up data to the Data Vault in NRT. The data models don’t need to change; the reporting views don’t need to change; even the loading patterns don’t need to change. (NB: For those that aren’t using Data Vault already, if they have real-time loading requirements, this architecture and method might be worth considering.)

Data Vault on Snowflake

Luckily, streaming data is one of the [use-cases] that Snowflake was built to support, so we have many features to help us achieve this goal.

Prerequisites

- A Snowflake account. Existing or if you are not(yet) a Snowflake user, you can always get a [trial](#) account
- Familiarity with Snowflake and Snowflake objects
- Understanding of Data Vault concepts and modelling techniques

What You'll Learn

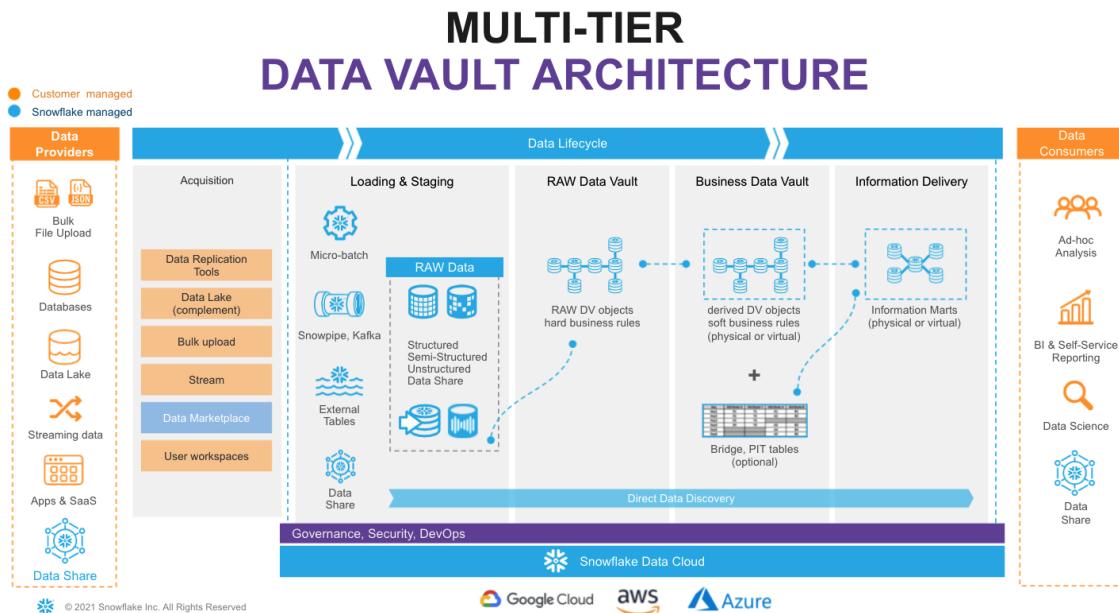
- how to use Data Vault on Snowflake
- how to build basic objects and write ELT code for it
- how to leverage [Snowpipe] and [Continuous Data Pipelines] to automate data processing
- how to apply data virtualization to accelerate data access

What You'll Build

- a Data Vault environment on Snowflake, based on sample dataset
- data pipelines, leveraging streams, tasks and Snowpipe

Reference Architecture

Let's start with the overall architecture to put everything in context.



On the very left of figure above we have a list of **data providers** that typically include a mix of existing operational databases, old data warehouses, files, lakes as well as 3rd party apps. There is now also the possibility to leverage Snowflake Data Sharing/Marketplace as a way to tap into new 3rd party data assets to augment your data set.

On the very right we have our ultimate **data consumers**: business users, data scientists, IT systems or even other companies you decided to exchange your data with.

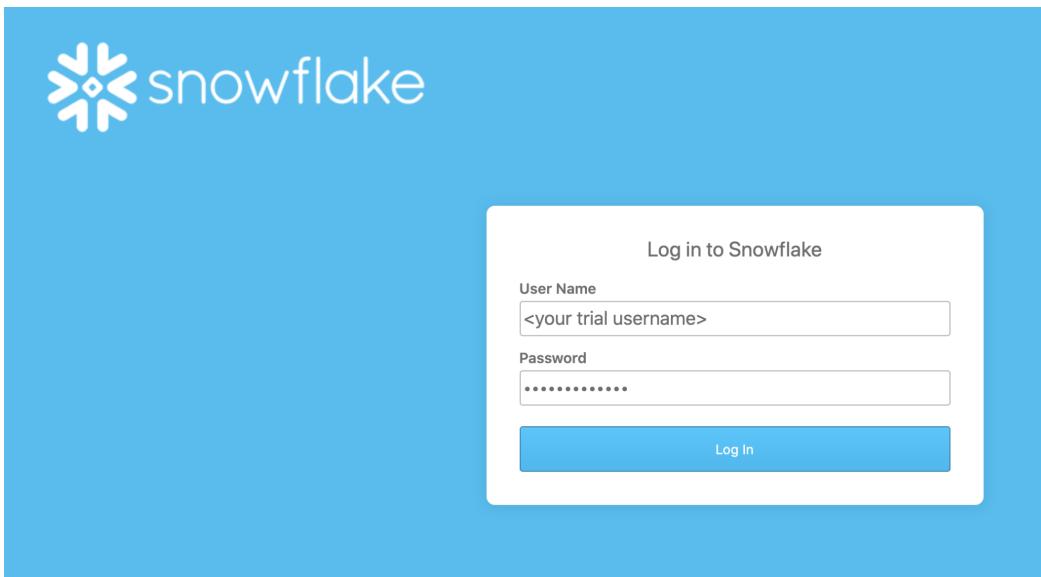
Architecturally, we will split the data lifecycle into following layers:

- **Data Acquisition:** extracting data from source systems and making it accessible for Snowflake.
- **Loading & Staging:** moving the source data into Snowflake. For this Snowflake has multiple options, including batch load, external tables and Snowpipe(our managed service for onboarding streaming data). Snowflake allows you to load and store structured and semi-structured in the original format whilst automatically optimizing the physical structure for efficient query access. The data is immutable and should be stored as it was received from source with no changes to the content. From a Data Vault perspective, functionally, this layer is also responsible for adding technical metadata (record source,, load date timestamp, etc.) as well as calculating business keys.
- **Raw Data Vault:** a data vault model with no soft business rules or transformations applied (only hard rules are allowed) loading all records received from source.
- **Business Data Vault:** data vault objects with soft business rules applied. The raw data vault data is getting augmented by the intelligence of the system. It is not a copy of the raw data vault, but rather a sparse addition with perhaps calculated satellites, mastered records,or maybe even commonly used aggregations. This could also optionally include PIT and Bridge tables helping to simplify access to bi-temporal view of the data. From a Snowflake perspective, raw and business data vaults could be separated by object naming convention or represented as different schemas or even different databases.
- **Information Delivery:** a layer of consumer-oriented models. This could be implemented as a set (or multiple sets) of views. It is common to see the use of dimensional models (star/snowflake) or denormalized flat tables (for example for data science or sharing) but it could be any other modeling style (e.g., unified star

schema, supernova, key-value, document object mode, etc.) that fits best for your data consumer. Snowflake's scalability will support the required speed of access at any point of this data lifecycle. You should consider Business Vault and Information Delivery objects materialization as optional.

Environment setup

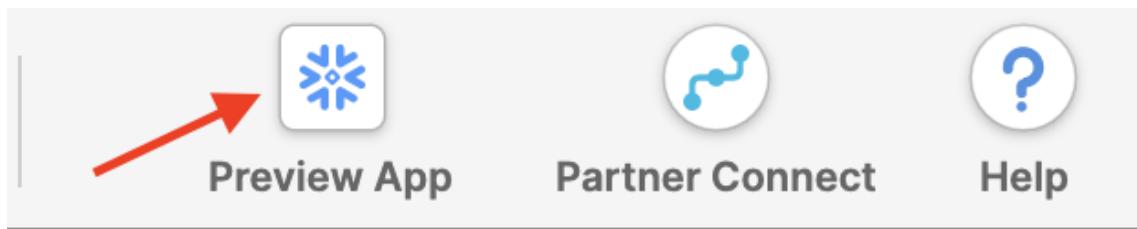
1. Login to your Snowflake trial account.



2. First page you are going to see would likely be [Snowflake Classic UI]:

A screenshot of the Snowflake Classic UI. The top navigation bar includes icons for Databases, Shares, Data Marketplace, Warehouses, Worksheets (which is selected), and History. On the far right, it shows the user "DANDERSEN" with a role of "SYSADMIN". Below the navigation is a toolbar with "New Worksheet", "Find database objects", "Run", "All Queries", "Saved 1 hour ago", and dropdown menus for "Select Role", "Select Warehouse", "Select Database", "Select Schema", and "Open History". The main workspace on the left shows a sidebar with "New Worksheet" and a search bar for "Find database objects" with "Starting with...". The main area is currently empty, showing the message "Query results will appear here.".

To keep things interesting, for the purpose of this lab let's use [Snowflake New Web Interface] also known as Snowsight. However, you absolutely can continue using Classic UI as all steps in this guide are expressed in SQL and will work regardless what interface is used. To switch into Snowsight, let's click the **Preview** button in the top-right corner:



Click Sign in to continue. You will need to use the same user and password that you used to login to your Snowflake account the first time.

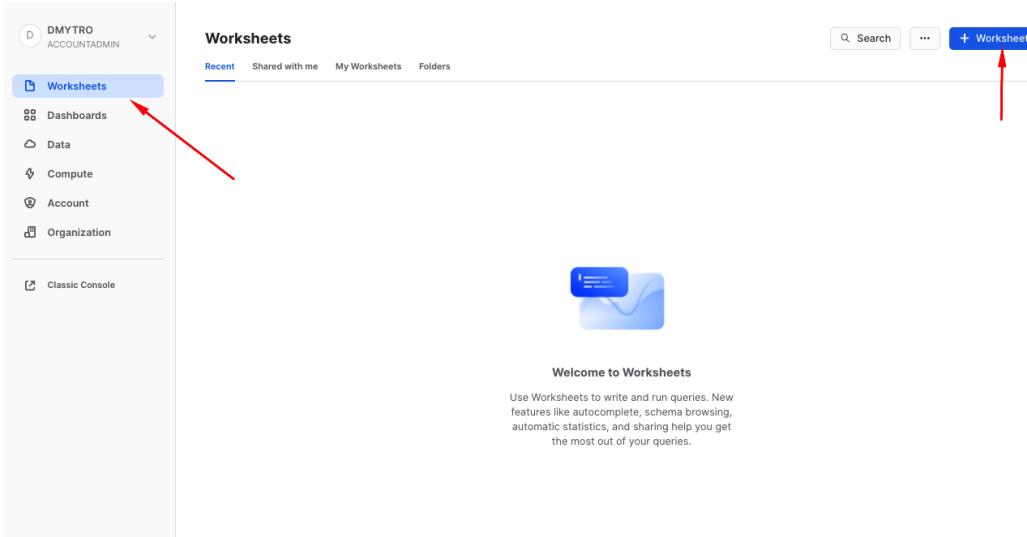


Snowflake

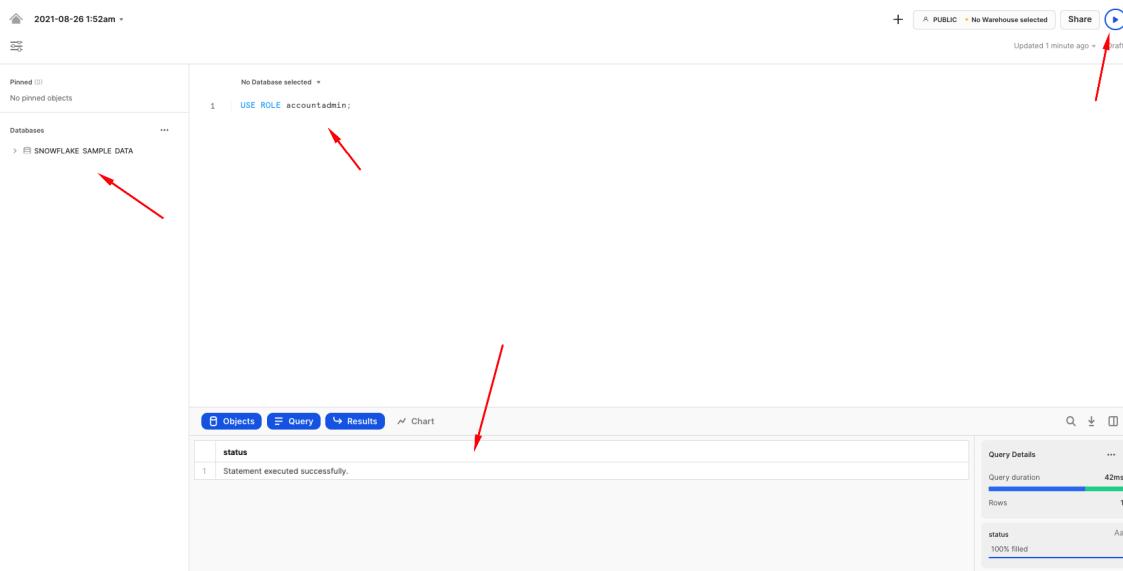
Sign in to continue

You're now in the new UI - Snowsight. It's pretty cool - with charting, dashboards, autocompletion and new capabilities that our engineering team will continue to add on weekly. Now, let's click on Worksheets...

3. Let's click on the worksheets -> + Worksheet



And without going into too much details, this is a fairly intuitive SQL workbench. It has a section for code we are going to be copy-pasting, object tree on the left, the 'run' button and of course the result panel at the bottom with the simple charting functionality.



4. We are going to start by setting up basics for the lab environment. Creating a clean database and logically dividing it into four different schemas, representing each functional area mentioned in the reference architecture.

To keep things simple, we are going to use the ACCOUNTADMIN role. We also going to create two [Snowflake virtual warehouses] to manage compute - one for generic use during the course of this lab and the other one (dv_rdv_wh) that is going to be used by our data pipelines. You might notice that the code for two more virtual warehouses (dv_bdv_wh, dv_id_wh) is commented - again, this is just to keep things simple for the guide but we wanted to illustrate the fact you can have as many of virtual warehouses of any size and configuration as you needed. For example having separate ones to deal with different layers in our Data Vault architecture.

```

-- setting up the environment

USE ROLE accountadmin;

CREATE OR REPLACE DATABASE dv_lab;

USE DATABASE dv_lab;

CREATE OR REPLACE WAREHOUSE dv_lab_wh WITH WAREHOUSE_SIZE = 'XSMALL' MIN_CLUSTER_COUNT = 1 MAX_CLUSTER_COUNT = 1 AUTO_SUSPEND = 60 COMMENT = 'Generic WH';
CREATE OR REPLACE WAREHOUSE dv_rdv_wh WITH WAREHOUSE_SIZE = 'XSMALL' MIN_CLUSTER_COUNT = 1 MAX_CLUSTER_COUNT = 1 AUTO_SUSPEND = 60 COMMENT = 'WH for Raw Data Vault object pipelines';
--CREATE OR REPLACE WAREHOUSE dv_bdv_wh WITH WAREHOUSE_SIZE = 'XSMALL'
MIN_CLUSTER_COUNT = 1 MAX_CLUSTER_COUNT = 1 AUTO_SUSPEND = 60 COMMENT = 'WH for Business Data Vault object pipelines';
--CREATE OR REPLACE WAREHOUSE dv_id_wh WITH WAREHOUSE_SIZE = 'XSMALL'
MIN_CLUSTER_COUNT = 1 MAX_CLUSTER_COUNT = 1 AUTO_SUSPEND = 60 COMMENT = 'WH for information delivery object pipelines';

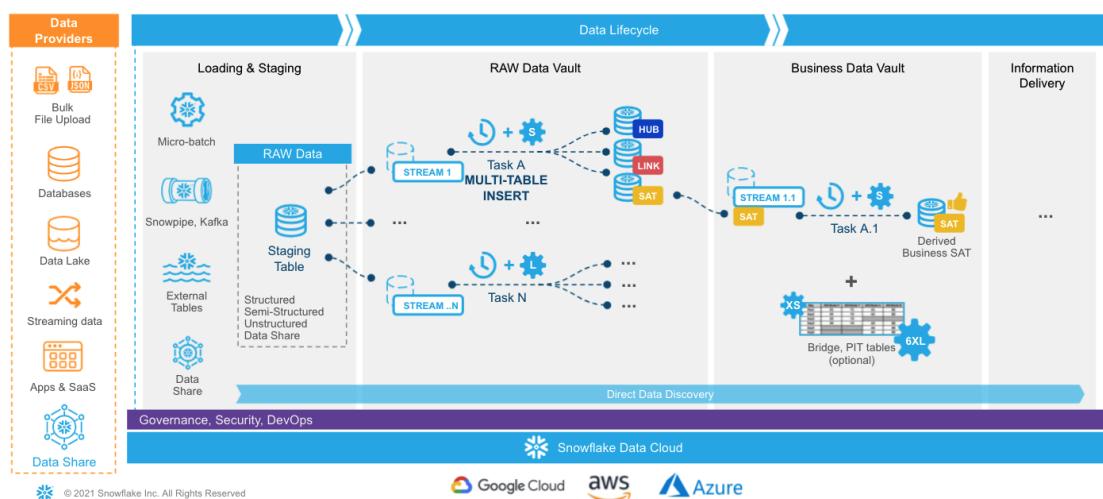
USE WAREHOUSE dv_lab_wh;

CREATE OR REPLACE SCHEMA 100_stg COMMENT = 'Schema for Staging Area objects';
CREATE OR REPLACE SCHEMA 110_rdv COMMENT = 'Schema for Raw Data Vault objects';
CREATE OR REPLACE SCHEMA 120_bdv COMMENT = 'Schema for Business Data Vault objects';
CREATE OR REPLACE SCHEMA 130_id COMMENT = 'Schema for Information Delivery objects';

```

Data Pipelines: Design

CONTINUOUS TRANSFORMATIONS



Snowflake supports multiple options for engineering data pipelines. In this post we are going to show one of the most efficient ways to implement incremental NRT integration leveraging Snowflake [Continuous Data Pipelines]. Let's take a look at the architecture diagram above to understand how it works.

Snowflake has a special [stream] object that tracks all data changes on a table (inserts, updates, and deletes). This process is 100% automatic and unlike traditional databases will never impact the speed of data loading. The change log from a stream is automatically 'consumed' once there is a successfully completed DML operation using the stream object as a source.

So, loading new data into a staging table, would immediately be reflected in a stream showing the ['delta'] that requires processing.

The second component we are going to use is [tasks]. It is a Snowflake managed data processing unit that will wake up on a defined interval (e.g., every 1-2 min), check if there is any data in the associated stream and if so, will run SQL to push it to the Raw Data Vault objects. Tasks could be arranged in a [tree-like dependency graph], executing child tasks the moment the predecessor finished its part.

Last but not least, following Data Vault 2.0 best practices for NRT data integration (to load data in parallel) we are going to use Snowflake's [multi-table insert (MTI)] inside tasks to populate multiple Raw Data Vault objects by a single DML command. (Alternatively you can create multiple streams & tasks from the same table in stage in order to populate each data vault object by its own asynchronous flow.)

Next step, you assign tasks to one or many virtual warehouses. This means you always have enough [compute power] (XS to 6XL) to deal with any size workload, whilst the [multi-cluster virtual warehouse] option will automatically scale-out and load balance all the tasks as you introduce more hubs, links and satellites to your vault.

Talking about tasks, Snowflake just introduced another fantastic capability - serverless tasks. This enables you to rely on compute resources managed by Snowflake instead of user-managed virtual warehouses. These compute resources are automatically resized and scaled up and down by Snowflake as required by each workload. This feature will be out of scope for this guide, but serverless compute model could reduce compute costs, in some cases significantly, allowing you to process more data, faster with even less management.

As your raw vault is updated, streams can then be used to propagate those changes to Business Vault objects (such as derived Sats, PITS, or Bridges, if needed) in the next layer. This setup can be repeated to move data through all the layers in small increments very quickly and efficiently. All the way until it is ready to be accessed by data consumers (if materialization of the data is required for performance).

Following this approach will result in a hands-off production data pipeline that feeds your Data Vault architecture.

Sample data & staging area

Every Snowflake account provides access to [sample data sets]. You can find corresponding schemas in SNOWFLAKE_SAMPLE_DATA database in your object explorer. For this guide we are going to use a subset of objects from [TPC-H] set, representing **customers** and their **orders**. We also going to take some reference data about **nations** and **regions**.

Dataset	Description	Source	Load Scenario	Mechanism
Nation	Static ref data	snowflake.sample_data.tpch_sf10.nation	one-off CTAS	SQL
Region	Static ref data	snowflake.sample_data.tpch_sf10.region	one-off CTAS	SQL
Customer	Customer data	snowflake.sample_data.tpch_sf10.customer	incremental JSON files	Snowpipe

Orders	Orders data	snowflake.sample_data.tpch_sf10.orders	incremental CSV files	Snowpipe
--------	-------------	--	-----------------------	----------

1. Let's start with the static reference data:

```
-- setting up staging area

-----



USE SCHEMA 100_stg;

CREATE OR REPLACE TABLE stg_nation
AS
SELECT src.*,
       CURRENT_TIMESTAMP()          ldts
      , 'Static Reference Data'    rscr
  FROM snowflake_sample_data.tpch_sf10.nation src;

CREATE OR REPLACE TABLE stg_region
AS
SELECT src.*,
       CURRENT_TIMESTAMP()          ldts
      , 'Static Reference Data'    rscr
  FROM snowflake_sample_data.tpch_sf10.region src;
```

2. Next, let's create staging tables for our data loading. This syntax should be very familiar with anyone working with databases before. It is ANSI SQL compliant DDL, with probably one key exception - for stg_customer we are going to load the full payload of JSON into raw_json column. For this, Snowflake has a special data type [VARIANT].

As we load data we also going to add some technical metadata, like load data timestamp, row number in a file.

```
CREATE OR REPLACE TABLE stg_customer
(
  raw_json          VARIANT
, filename         STRING  NOT NULL
, file_row_seq    NUMBER  NOT NULL
, ldts             STRING  NOT NULL
, rscr             STRING  NOT NULL
);

CREATE OR REPLACE TABLE stg_orders
(
  o_orderkey        NUMBER
, o_custkey         NUMBER
, o_orderstatus     STRING
, o_totalprice      NUMBER
, o_orderdate       DATE
, o_orderpriority   STRING
, o_clerk           STRING
, o_shipppriority   NUMBER
, o_comment          STRING
, filename          STRING  NOT NULL
);
```

```

, file_row_seq      NUMBER  NOT NULL
, ldts              STRING   NOT NULL
, rscr              STRING   NOT NULL
);

```

3. Tables we just created are going to be used by Snowpipe to drip-feed the data as it lands in the stage. In order to easily detect and incrementally process the new portion of data we are going to create [streams] on these staging tables:

```

CREATE OR REPLACE STREAM stg_customer_strm ON TABLE stg_customer;
CREATE OR REPLACE STREAM stg_orders_strm ON TABLE stg_orders;

```

4. Next we are going to produce some sample data. And for the sake of simplicity we are going to take a bit of a shortcut here. We are going to generate data by unloading subset of data from our TPCH sample dataset into files and then use Snowpipe to load it back into our Data Vault lab, simulating the streaming feed. Let's start by creating two stages for each data class type (orders, customers data). In real-life scenarios these could be internal or external stages as well as these feeds could be sourced via Kafka connector. The world is your oyster.

```

CREATE OR REPLACE STAGE customer_data FILE_FORMAT = (TYPE = JSON);
CREATE OR REPLACE STAGE orders_data FILE_FORMAT = (TYPE = CSV) ;

```

5. Generate and unload sample data. There are couple of things going on. First, we are using [object_construct] as a quick way to create a object/document from all columns and subset of rows for customer data and offload it into customer_data stage. Orders data would be extracted into compressed CSV files. There are many additional options in [COPY INTO stage] construct that would fit most requirements, but in this case we are using INCLUDE_QUERY_ID to make it easier to generate new incremental files as we are going to run these commands over and over again, without a need to deal with file overriding.

```

COPY INTO @customer_data
FROM
(SELECT object_construct(*)
  FROM snowflake_sample_data.tpch_sf10.customer limit 10
)
INCLUDE_QUERY_ID=TRUE;

COPY INTO @orders_data
FROM
(SELECT *
  FROM snowflake_sample_data.tpch_sf10.orders limit 1000
)
INCLUDE_QUERY_ID=TRUE;

```

You can now run the following to validate that the data is now stored in files:

```

list @customer_data;
SELECT METADATA$FILENAME,$1 FROM @customer_data;

```

+ ACCOUNTADMIN + DV_LAB_WH Share

Updated 1 minute ago ▾

```
DV_LAB.LOOG_STG *
1 COPY INTO @customer_data
2 FROM
3 (SELECT object_construct(*)
4   FROM snowflake_sample_data.tpch_sf10.customer LIMIT 10
5 )
6 ) INCLUDE_QUERY_ID=TRUE;
7
8 COPY INTO @orders_data
9 FROM
10 (SELECT *
11   FROM snowflake_sample_data.tpch_sf10.orders LIMIT 1000
12 )
13 ) INCLUDE_QUERY_ID=TRUE;
14
15 list @customer_data;
16 | SELECT METADATASFILENAME, $1 FROM @customer_data;
```

The screenshot shows the Snowflake interface with a table titled 'METADATASFILENAME'. A red arrow points from the table to a tooltip window. The tooltip contains the following information:

	\$1
{ "C_ACCTBAL": 3721.47, "C_ADDRESS": "k5jk4s9kvJ9PUzx3EFzrM4", "C_COMMENT": "ily regular acco	Cell in \$1
{ "C_ACCTBAL": 9818.32, "C_ADDRESS": "ppu10.nyp34A6iCkaJy2r,2CuY6s2wC", "C_COMMENT": "apad	Clear selection
{ "C_ACCTBAL": 3596, "C_ADDRESS": "eJIEpg74BRuCb264YATBCJ3Snr", "C_COMMENT": "ie furiously as	RECORD TEXT
{ "C_ACCTBAL": 1909, "C_ADDRESS": "2hM3ZXizf3J8iCeOK5Hh,Qd4RYD", "C_COMMENT": "sly final i	
{ "C_ACCTBAL": 6444.3, "C_ADDRESS": "FkkXanVhIzjnzvfu59RtpM", "C_COMMENT": "requests, furic	
{ "C_ACCTBAL": 1422.73, "C_ADDRESS": "dqN8zGhZ.uwKHWMEvgl5cdwX35fcdf398YCs", "C_COMMENT": "	
{ "C_ACCTBAL": 6753.74, "C_ADDRESS": "tn4flmwF35cQff5tBk6CbmYblhGQsviSMje2N", "C_COMMENT": "	
{ "C_ACCTBAL": 822.31, "C_ADDRESS": "ytRFpVp44+8RKdQtfHfU03PgoZfN", "C_COMMENT": "shras	
{ "C_ACCTBAL": 4356.55, "C_ADDRESS": "YFa7jn7n3TNeGeRqJVq8icPDbtISO", "C_COMMENT": "st reg	
{ "C_ACCTBAL": 2486.05, "C_ADDRESS": "PHR09AnJb,8MKhgM8ac", "C_COMMENT": "ts according to th	

6. Next, we are going to setup Snowpipe to load data from files in a stage into staging tables. In this guide, for better transparency we are going to trigger Snowpipe explicitly to scan for new files, but in real projects you will likely going to enable AUTO_INGEST, connecting it with your cloud storage events (like AWS SNS) and process new files automatically.

```
CREATE OR REPLACE PIPE stg_orders_pp
AS
COPY INTO stg_orders
FROM
(
SELECT $1,$2,$3,$4,$5,$6,$7,$8,$9
, metadata$filename
, metadata$file_row_number
, CURRENT_TIMESTAMP()
, 'Orders System'
FROM @orders_data
);

CREATE OR REPLACE PIPE stg_customer_pp
--AUTO_INGEST = TRUE
--aws sns topic = 'arn:aws:sns:mybucketdetails'
AS
COPY INTO stg_customer
FROM
```

```

(
SELECT $1
    , metadata$filename
    , metadata$file_row_number
    , CURRENT_TIMESTAMP()
    , 'Customers System'
FROM @customer_data
);

ALTER PIPE stg_customer_pp REFRESH;

ALTER PIPE stg_orders_pp     REFRESH;

```

Once this done, you should be able to see data appearing in the target tables and the stream on these tables. As you would notice, number of rows in a stream is exactly the same as in the base table. This is because we didn't process/consumed the delta of that stream yet. Stay tuned!

```

SELECT 'stg_customer', count(1) FROM stg_customer
UNION ALL
SELECT 'stg_orders', count(1) FROM stg_orders
UNION ALL
SELECT 'stg_orders_strm', count(1) FROM stg_orders_strm
UNION ALL
SELECT 'stg_customer_strm', count(1) FROM stg_customer_strm
;

```

The screenshot shows a Data Vault Lab interface with a query editor and a results table.

Query Editor:

```

11     , CURRENT_TIMESTAMP()
12     , 'Orders System'
13   FROM @orders_data
14 );
15
16 CREATE OR REPLACE PIPE stg_customer_pp
17   --AUTO_INGEST = TRUE
18   --aws_sns_topic = 'arn:aws:sns:mybucketdetails'
19 AS
20 COPY INTO stg_customer
21   FROM
22 (
23   SELECT $1
24     , metadata$filename
25     , metadata$file_row_number
26     , CURRENT_TIMESTAMP()
27     , 'Customers System'
28   FROM @customer_data
29 );
30
31 ALTER PIPE stg_customer_pp REFRESH;
32
33 ALTER PIPE stg_orders_pp     REFRESH;
34
35   SELECT 'stg_customer', count(1) FROM stg_customer
36 UNION ALL
37   SELECT 'stg_orders', count(1) FROM stg_orders
38 UNION ALL
39   SELECT 'stg_orders_strm', count(1) FROM stg_orders_strm
40 UNION ALL
41   SELECT 'stg_customer_strm', count(1) FROM stg_customer_strm
42 ;

```

Results Table:

'STG_CUSTOMER'	COUNT(1)
1 stg_customer	10
2 stg_orders	1,000
3 stg_orders_strm	1,000
4 stg_customer_strm	10

Query Details Panel:

- Query duration: 320ms
- Rows: 4
- STG_CUSTOMER: 100% filled
- COUNT(1): 123

7. Finally, now that we established the basics and new data is knocking at our door (stream), let's see how we can derive some of the business keys for the Data Vault entities we are going to model. In this example, we

will model it as a view on top of the stream that should allow us to perform data parsing (raw_json -> columns) and business_key, hash_diff derivation on the fly. Another thing to notice here is the use of SHA1_BINARY as hasing function. For this lab, we are going to use fairly common SHA1 and its BINARY version from Snowflake arsenal of functions that use less bytes to encode value than STRING.

```

CREATE OR REPLACE VIEW stg_customer_strm_outbound AS
SELECT src.*
    , raw_json:C_CUSTKEY::NUMBER          c_custkey
    , raw_json:C_NAME::STRING            c_name
    , raw_json:C_ADDRESS::STRING        c_address
    , raw_json:C_NATIONKEY::NUMBER      C_nationcode
    , raw_json:C_PHONE::STRING          c_phone
    , raw_json:C_ACCTBAL::NUMBER        c_acctbal
    , raw_json:C_MKTSEGMENT::STRING     c_mktsegment
    , raw_json:C_COMMENT::STRING        c_comment
-----
-- derived business key
-----
    , SHA1_BINARY(UPPER(TRIM(c_custkey)))  sha1_hub_customer
    , SHA1_BINARY(UPPER(ARRAY_TO_STRING(ARRAY_CONSTRUCT(
                    NVL(TRIM(c_name)           , '-1')
                    , NVL(TRIM(c_address)       , '-1')
                    , NVL(TRIM(c_nationcode)   , '-1')
                    , NVL(TRIM(c_phone)         , '-1')
                    , NVL(TRIM(c_acctbal)       , '-1')
                    , NVL(TRIM(c_mktsegment)   , '-1')
                    , NVL(TRIM(c_comment)       , '-1')
                ), '^'))) ) AS customer_hash_diff
FROM stg_customer_strm src
;

CREATE OR REPLACE VIEW stg_order_strm_outbound AS
SELECT src.*
-----
-- derived business key
-----
    , SHA1_BINARY(UPPER(TRIM(o_orderkey)))      sha1_hub_order
    , SHA1_BINARY(UPPER(TRIM(o_custkey)))        sha1_hub_customer
    , SHA1_BINARY(UPPER(ARRAY_TO_STRING(ARRAY_CONSTRUCT( NVL(TRIM(o_orderkey)
    , '-1')
                    , NVL(TRIM(o_custkey)
    , '-1'))
                ), '^'))) ) AS
sha1_lnk_customer_order
    , SHA1_BINARY(UPPER(ARRAY_TO_STRING(ARRAY_CONSTRUCT( NVL(TRIM(o_orderstatus)
    , '-1')
                    , NVL(TRIM(o_totalprice)
    , '-1'))
                    , NVL(TRIM(o_orderdate)
    , '-1'))
                    , NVL(TRIM(o_orderpriority)
    , '-1')))) ) AS

```

```

        , NVL(Trim(o_clerk)           ,
        , NVL(Trim(o_shipppriority)   ,
        , NVL(Trim(o_comment)         ,
        ) , '^')) ) AS order_hash_diff
  FROM stg_orders_strm src
;

```

Finally let's query these views to validate the results:

```

2021-08-26 1:52am · + ACCOUNTADMIN · DV_LAB_WH · Share ▶
Updated 1 minute ago ·

Pinned (0)
No pinned objects

Databases ...
  DBT_HOL_DEV
  DBT_HOL_DEV_TO_SWAP
  DBT_HOL_PROD
  DV_LAB
    INFORMATION_SCHEMA
    LOB_STG
  STG_CUSTOMER_STRM_OUTBOUND
    RAW_JSON
      FILENAME
        FILE_ROW_SEQ
        LOTS
        RSCR
        METADATASACTION
        METADATASISUPDATE
        METADATAROWID
        CUSTKEY
        CNAME
        CADDRESS
        CATIONCODE
        CPHON
        CACCTBAL
        CMKTSEGMENT
        CCOMMENT
        SHA1HUBCUSTOMER
        CUSTOMERHASHDIFF
  METADATAROWID
  CUSTKEY
  CNAME
  CADDRESS
  CATIONCODE
  CPHON
  CACCTBAL
  CMKTSEGMENT
  CCOMMENT
  SHA1HUBCUSTOMER
  CUSTOMERHASHDIFF

CREATE OR REPLACE VIEW stg_order_strm_outbound AS
SELECT src.*
  , NVL(Trim(c_address) , '-' )
  , NVL(Trim(c_nationcode) , '-' )
  , NVL(Trim(c_acctbal) , '-' )
  , NVL(Trim(c_mktsegment) , '-' )
  , NVL(Trim(c_comment) , '-' )
, '^')) ) AS customer_hash_diff
FROM stg_customer_strm src
;
CREATE OR REPLACE VIEW stg_order_strm_outbound AS
SELECT src.*
  , NVL(Trim(o_orderkey) , '-' )
  , NVL(Trim(o_custkey) , '-' )
  , NVL(Trim(ARRAY_TO_STRING(ARRAY_CONSTRUCT( NVL(Trim(o_orderkey) , '-' )
, NVL(Trim(o_custkey) , '-' )
, NVL(Trim(o_orderdate) , '-' )
, NVL(Trim(o_orderpriority) , '-' )
, NVL(Trim(o_clerk) , '-' )
, NVL(Trim(o_shipppriority) , '-' )
, NVL(Trim(o_comment) , '-' )
), '^')) ) AS order_hash_diff
FROM stg_orders_strm src
;
  | SELECT * FROM stg_customer_strm_outbound;

```

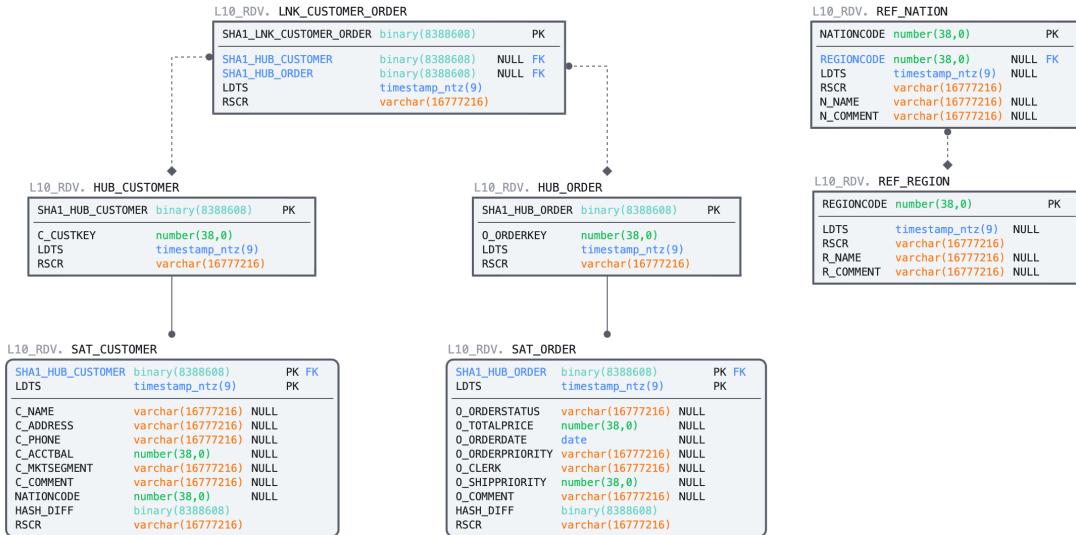
Objects	Query	Results
C_COMMENT		
1	lyl regular accounts, blithely final ideas sleep blithely	6d8cf58f41433a8182c20d8146fb7cf0b05585c5 ae1844477cbf14111212777b52f4ab07db0354
2	apades detect carefully final packages, final, pend	248fa151b26402600ced537e7e5453273e9e4ff 8d540249a8ad6dbad559e56203997d6073f8171
3	le furiously asymptotic, final, ironic foxes nag quickly silent packages. huffily	e4b080e0f46bc7baee172d62b5ad52d240299101 5c592d6da205ea8f0fe67b033a498cd4a5fd27
4	slify final accounts use blithely among the slyly bold pinto beans; always final pack	f7d077b8b87b879daeb0dd34fc01870553af83 d21fe2654c3a230a74ea3d41faa3x32addca25
5	requests, furiously stealthy deposits toward the notorious haggis blithely regular thr	f9d7d3fb8b4e37558ba3d05f0e418adec5222a78 d73f9706e19d33d4dad5a0023e9e3fb1c1214
6	test blithely regular theodolites	e530b631054cc6cf522d9ee0070721d0052ab541 651e3f5a4c55a0022330d0078ec2bad477be469
7	wick ideas are, quickly ironic requests are quick	d5e436453fb8d8df3f1628218d180a1a4dc2b666 7b7b483656a0ea41d1798acece5379d51b47847
8	s trash, furiously pending attainments	496c101d98c02e74d6811e5f80119939ef517d7 a78b726223b0a3271c0111dddb91814b4bc33
9	if regularly regular deposits wakely pending ideas, unusual, ironic packages	1812002c5ac0c5a5bd7473d5d8d0471cd9836 d40c17354bb6b0791u02290a1b4f5e34b4c408
10	ts according to the blithely regular theodolites, raise furiously pending attainments, are	c11f8fhr57a9laa8a47n4r8e6r8a394r87r7nfdrn R7dhf4h725K33d4rakehkhala8c3c7zefndaaa

Well done! We build our staging/inbound pipeline, ready to accomodate streaming data and derived business keys that we are going to use in our Raw Data Vault. Let's move on to the next step!

Build: Raw Data Vault

In this section, we will start building structures and pipelines for **Raw Data Vault** area.

Here is the ER model of the objects we are going to deploy using the script below:



1. We'll start by deploying DDL for the HUBs, LINKs and SATellite tables. As you can imagine, this guide has no chance to go in the detail on data vault modelling process. This is something we usually highly recommend to establish by working with experts & partners from Data Vault Alliance.

```

-- setting up RDV

-----



USE SCHEMA 110_rdv;

-- hubs

CREATE OR REPLACE TABLE hub_customer
(
  sha1_hub_customer      BINARY      NOT NULL
, c_custkey               NUMBER      NOT NULL
, ldts                     TIMESTAMP   NOT NULL
, rscr                     STRING      NOT NULL
, CONSTRAINT pk_hub_customer PRIMARY KEY(sha1_hub_customer)
);

CREATE OR REPLACE TABLE hub_order
(
  sha1_hub_order      BINARY      NOT NULL
, o_orderkey             NUMBER      NOT NULL
, ldts                     TIMESTAMP   NOT NULL
, rscr                     STRING      NOT NULL
, CONSTRAINT pk_hub_order PRIMARY KEY(sha1_hub_order)
);
  
```

```

);

-- sats

CREATE OR REPLACE TABLE sat_customer
(
    sha1_hub_customer      BINARY      NOT NULL
    , ldts                  TIMESTAMP NOT NULL
    , c_name                STRING
    , c_address              STRING
    , c_phone                STRING
    , c_acctbal              NUMBER
    , c_mktsegment            STRING
    , c_comment                STRING
    , nationcode              NUMBER
    , hash_diff                BINARY      NOT NULL
    , rscr                  STRING      NOT NULL
    , CONSTRAINT pk_sat_customer PRIMARY KEY(sha1_hub_customer, ldts)
    , CONSTRAINT fk_sat_customer FOREIGN KEY(sha1_hub_customer) REFERENCES
hub_customer
);

CREATE OR REPLACE TABLE sat_order
(
    sha1_hub_order      BINARY      NOT NULL
    , ldts                  TIMESTAMP NOT NULL
    , o_orderstatus            STRING
    , o_totalprice              NUMBER
    , o_orderdate                DATE
    , o_orderpriority            STRING
    , o_clerk                  STRING
    , o_shipppriority            NUMBER
    , o_comment                STRING
    , hash_diff                BINARY      NOT NULL
    , rscr                  STRING      NOT NULL
    , CONSTRAINT pk_sat_order PRIMARY KEY(sha1_hub_order, ldts)
    , CONSTRAINT fk_sat_order FOREIGN KEY(sha1_hub_order) REFERENCES hub_order
);
;

-- links

CREATE OR REPLACE TABLE lnk_customer_order
(
    sha1_lnk_customer_order BINARY      NOT NULL
    , sha1_hub_customer      BINARY
    , sha1_hub_order          BINARY
    , ldts                  TIMESTAMP NOT NULL
    , rscr                  STRING      NOT NULL
    , CONSTRAINT pk_lnk_customer_order PRIMARY KEY(sha1_lnk_customer_order)
    , CONSTRAINT fk1_lnk_customer_order FOREIGN KEY(sha1_hub_customer) REFERENCES
hub_customer
    , CONSTRAINT fk2_lnk_customer_order FOREIGN KEY(sha1_hub_order)      REFERENCES

```

```

hub_order
);

-- ref data

CREATE OR REPLACE TABLE ref_region
(
    regioncode          NUMBER
   , ldts               TIMESTAMP
   , rscr               STRING NOT NULL
   , r_name              STRING
   , r_comment            STRING
   , CONSTRAINT PK_REF_REGION PRIMARY KEY (REGIONCODE)
)
AS
SELECT r_regionkey
   , ldts
   , rscr
   , r_name
   , r_comment
FROM 100_stg.stg_region;

CREATE OR REPLACE TABLE ref_nation
(
    nationcode          NUMBER
   , regioncode          NUMBER
   , ldts               TIMESTAMP
   , rscr               STRING NOT NULL
   , n_name              STRING
   , n_comment            STRING
   , CONSTRAINT pk_ref_nation PRIMARY KEY (nationcode)
   , CONSTRAINT fk_ref_region FOREIGN KEY (regioncode) REFERENCES ref_region(regioncode)
)
AS
SELECT n_nationkey
   , n_regionkey
   , ldts
   , rscr
   , n_name
   , n_comment
FROM 100_stg.stg_nation;

```

2. Now we have source data waiting in our staging streams & views, we have target RDV tables. Let's connect the dots. We are going to create tasks, one per each stream so whenever there is new records coming in a stream, that delta will be incrementally propagated to all dependent RDV models in one go. To achieve that, we are going to use multi-table insert functionality as described in design section before. As you can see, tasks can be set up to run on a pre-defined frequency (every 1 minute in our example) and use dedicated virtual warehouse as a compute power (in our guide we are going to use same warehouse for all tasks, thou this could be as granular as needed). Also, before waking up a compute resource, tasks are going to check that there is data in a corresponding stream to process. Again, you are paying only for the compute when you actually use it.

```

CREATE OR REPLACE TASK customer_strm_tsk
WAREHOUSE = dv_rdv_wh
SCHEDULE = '1 minute'
WHEN
  SYSTEM$STREAM_HAS_DATA('L00_STG.STG_CUSTOMER_STRM')
AS
  INSERT ALL
  WHEN (SELECT COUNT(1) FROM hub_customer tgt WHERE tgt.sha1_hub_customer =
src_sha1_hub_customer) = 0
  THEN INTO hub_customer
  ( sha1_hub_customer
  , c_custkey
  , ldts
  , rscr
  )
  VALUES
  ( src_sha1_hub_customer
  , src_c_custkey
  , src_ldts
  , src_rscr
  )
  WHEN (SELECT COUNT(1) FROM sat_customer tgt WHERE tgt.sha1_hub_customer =
src_sha1_hub_customer AND tgt.hash_diff = src_customer_hash_diff) = 0
  THEN INTO sat_customer
  (
    sha1_hub_customer
  , ldts
  , c_name
  , c_address
  , c_phone
  , c_acctbal
  , c_mktsegment
  , c_comment
  , nationcode
  , hash_diff
  , rscr
  )
  VALUES
  (
    src_sha1_hub_customer
  , src_ldts
  , src_c_name
  , src_c_address
  , src_c_phone
  , src_c_acctbal
  , src_c_mktsegment
  , src_c_comment
  , src_nationcode
  , src_customer_hash_diff
  , src_rscr
  )
  SELECT sha1_hub_customer    src_sha1_hub_customer

```

```

        , c_custkey          src_c_custkey
        , c_name             src_c_name
        , c_address          src_c_address
        , c_nationcode       src_nationcode
        , c_phone            src_c_phone
        , c_acctbal          src_c_acctbal
        , c_mktsegment       src_c_mktsegment
        , c_comment           src_c_comment
        , customer_hash_diff src_customer_hash_diff
        , ldts               src_ldts
        , rscr               src_rscr
FROM 100_stg.stg_customer_strm_outbound src
;

CREATE OR REPLACE TASK order_strm_tsk
WAREHOUSE = dv_rdv_wh
SCHEDULE = '1 minute'
WHEN
  SYSTEM$STREAM_HAS_DATA('L00_STG.STG_ORDERS_STRM')
AS
INSERT ALL
WHEN (SELECT COUNT(1) FROM hub_order tgt WHERE tgt.sha1_hub_order =
src_sha1_hub_order) = 0
THEN INTO hub_order
(
  sha1_hub_order
  , o_orderkey
  , ldts
  , rscr
)
VALUES
(
  src_sha1_hub_order
  , src_o_orderkey
  , src_ldts
  , src_rscr
)
WHEN (SELECT COUNT(1) FROM sat_order tgt WHERE tgt.sha1_hub_order = src_sha1_hub_order
AND tgt.hash_diff = src_order_hash_diff) = 0
THEN INTO sat_order
(
  sha1_hub_order
  , ldts
  , o_orderstatus
  , o_totalprice
  , o_orderdate
  , o_orderpriority
  , o_clerk
  , o_shipppriority
  , o_comment
  , hash_diff
  , rscr
)

```

```

VALUES
(
    src_shal_hub_order
, src_ldts
, src_o_orderstatus
, src_o_totalprice
, src_o_orderdate
, src_o_orderpriority
, src_o_clerk
, src_o_shipppriority
, src_o_comment
, src_order_hash_diff
, src_rscr
)
WHEN (SELECT COUNT(1) FROM lnk_customer_order tgt WHERE tgt.shal_lnk_customer_order =
src_shal_lnk_customer_order) = 0
THEN INTO lnk_customer_order
(
    sha1_lnk_customer_order
, sha1_hub_customer
, sha1_hub_order
, ldts
, rscr
)
VALUES
(
    src_shal_lnk_customer_order
, src_shal_hub_customer
, src_shal_hub_order
, src_ldts
, src_rscr
)
SELECT sha1_hub_order      src_shal_hub_order
      , sha1_lnk_customer_order src_shal_lnk_customer_order
      , sha1_hub_customer      src_shal_hub_customer
      , o_orderkey            src_o_orderkey
      , o_orderstatus          src_o_orderstatus
      , o_totalprice           src_o_totalprice
      , o_orderdate             src_o_orderdate
      , o_orderpriority         src_o_orderpriority
      , o_clerk                src_o_clerk
      , o_shipppriority         src_o_shipppriority
      , o_comment               src_o_comment
      , order_hash_diff        src_order_hash_diff
      , ldts                   src_ldts
      , rscr                   src_rscr
FROM 100_stg.stg_order_strm_outbound src;

ALTER TASK customer_strm_tsk RESUME;
ALTER TASK order_strm_tsk     RESUME;

```

2. Once tasks are created and RESUMED (by default, they are initially suspended) let's have a look on the task execution history to see how the process will start.

```
SELECT *
FROM table(information_schema.task_history())
ORDER BY scheduled_time DESC;
```

Notice how after successfull execution, next two tasks run were automatically SKIPPED as there were nothing in the stream and there nothing to do.

	CONDITION_TEXT	STATE	ERROR_CODE	ERROR_MESSAGE
1	ha1_hub_d SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_ORDERS_STRM')	SCHEDULED		
2	gt.sha1_h SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_CUSTOMER_STRM')	SCHEDULED		
3	ha1_hub_d SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_ORDERS_STRM')	SKIPPED	0040003	Conditional expression for task evaluated to false.
4	gt.sha1_h SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_CUSTOMER_STRM')	SKIPPED	0040003	Conditional expression for task evaluated to false.
5	ha1_hub_d SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_ORDERS_STRM')	SUCCEEDED		
6	gt.sha1_h SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_CUSTOMER_STRM')	SUCCEEDED		
7	ha1_hub_d SYSTEM\$STREAM_HAS_DATA('L00_STG.STG_ORDERS_STRM')	FAILED	002003	SQL compilation error: Object 'LNK_CUSTOMER_ORDER' does not exist or no

3. We can also check content and stats of the objects involved. Please notice that views on streams in our staging area are no longer returning any rows. This is because that delta of changes was consumed by a successfully completed DML transaction (in our case, embedded in tasks). This way you don't need to spend any time implementing incremental detection/processing logic on the application side.

```
SELECT 'hub_customer', count(1) FROM hub_customer
UNION ALL
SELECT 'hub_order', count(1) FROM hub_order
UNION ALL
SELECT 'sat_customer', count(1) FROM sat_customer
UNION ALL
SELECT 'sat_order', count(1) FROM sat_order
UNION ALL
SELECT 'lnk_customer_order', count(1) FROM lnk_customer_order
UNION ALL
SELECT '100_stg.stg_customer_strm_outbound', count(1) FROM
100_stg.stg_customer_strm_outbound
UNION ALL
SELECT '100_stg.stg_order_strm_outbound', count(1) FROM
100_stg.stg_order_strm_outbound;
```

```
DV_LAB.L10.RDV *

1   SELECT 'hub_customer', count(1) FROM hub_customer
2   UNION ALL
3   SELECT 'hub_order', count(1) FROM hub_order
4   UNION ALL
5   SELECT 'sat_customer', count(1) FROM sat_customer
6   UNION ALL
7   SELECT 'sat_order', count(1) FROM sat_order
8   UNION ALL
9   SELECT 'lnk_customer_order', count(1) FROM lnk_customer_order
10  UNION ALL
11  SELECT 'l00_stg_stg_customer_strm_outbound', count(1) FROM l00_stg_stg_customer_strm_outbound
12  UNION ALL
13  SELECT 'l00_stg_stg_order_strm_outbound', count(1) FROM l00_stg_stg_order_strm_outbound;
```

Objects Query Results Chart

'HUB_CUSTOMER'	COUNT(1)
1 hub_customer	10
2 hub_order	1,000
3 sat_customer	10
4 sat_order	1,000
5 lnk_customer_order	1,000
6 l00_stg_stg_customer_strm_outbound	0
7 l00_stg_stg_order_strm_outbound	0

Query Details

- Query duration: 1.7s
- Rows: 7

'HUB_CUSTOMER' Aa

100% filled

COUNT(1)

Great. We now have data in our **Raw Data Vault** core structures. Let's move on and talk about the concept of virtualization for building your near-real time Data Vault solution.

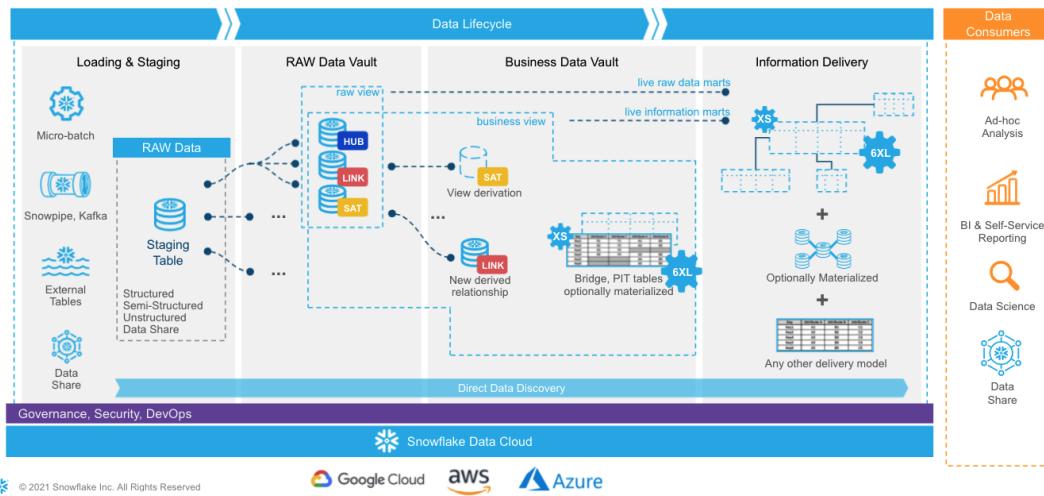
Views for Agile Reporting

One of the great benefits of having the compute power from Snowflake is that now it is totally possible to have most of your business vault and information marts in a Data Vault architecture be built exclusively from views. There are numerous customers using this approach in production today. There is no longer a need to have the argument that there are "too many joins" or that the response won't be fast enough. The elasticity of the Snowflake virtual warehouses combined with our dynamic optimization engine have solved that problem.

If you really want to deliver data to the business users and data scientists in NRT, in our opinion using views is the only option. Once you have the streaming loads built to feed your Data Vault, the fastest way to make that data visible downstream will be views. Using views allows you to deliver the data faster by eliminating any latency that would be incurred by having additional ELT processes between the Data Vault and the data consumers downstream.

All the business logic, alignment, and formatting of the data can be in the view code. That means fewer moving parts to debug, and reduces the storage needed as well.

DATA VAULT VIRTUALIZATION



Looking at the diagram above you will see an example of how virtualization could fit in the architecture. Here, solid lines are representing physical tables and dotted lines - views. You incrementally ingest data into **Raw Data Vault** and all downstream transformations are applied as views. From a data consumer perspective when working with a virtualized information mart, the query always shows everything known by your data vault, right up to the point the query was submitted.

With Snowflake you have the ability to provide as much compute as required, on-demand, without a risk of causing performance impact on any surrounding processes and pay only for what you use. This makes materialization of transformations in layers like **Business Data Vault** and **Information delivery** an option rather than a must-have. Instead of “optimizing upfront” you can now make this decision based on the usage pattern characteristics, such as frequency of use, type of queries, latency requirements, readiness of the requirements etc.

Many modern data engineering automation frameworks are already actively supporting virtualization of logic. Several tools offer a low-code or configuration-like ability to switch between materializing an object as a view or a physical table, automatically generating all required DDL & DML. This could be applied on specific objects, layers or/and be environment specific. So even if you start with a view, you can easily refactor to use a table if user requirements evolve.

As said before, virtualization is not only a way to improve time-to-value and provide near real time access to the data, given the scalability and workload isolation of Snowflake, virtualization also is a design technique that could make your Data Vault excel: minimizing cost-of-change, accelerating the time-to-delivery and becoming an extremely agile, future proof solution for ever growing business needs.

Build: Business Data Vault

As a quick example of using views for transformations we just discussed, here is how enrichment of customer descriptive data could happen in Business Data Vault, connecting data received from source with some reference data.

1. Let's create a view that will perform these additional derivations on the fly. Assuming non-functional capabilities are satisfying our requirements, deploying (and re-deploying a new version) transformations in this way is super easy.

```

USE SCHEMA 120_bdv;

CREATE OR REPLACE VIEW sat_customer_bv
AS
SELECT rsc.shal_hub_customer
, rsc.ldts
, rsc.c_name
, rsc.c_address
, rsc.c_phone
, rsc.c_acctbal
, rsc.c_mktsegment
, rsc.c_comment
, rsc.nationcode
, rsc.rscr
-- derived
, rrn.n_name          nation_name
, rrr.r_name          region_name
FROM 110_rdv.sat_customer      rsc
LEFT OUTER JOIN 110_rdv.ref_nation rrn
ON (rsc.nationcode = rrn.nationcode)
LEFT OUTER JOIN 110_rdv.ref_region rrr
ON (rrn.regioncode = rrr.regioncode)
;

```

2. Now, let's imagine we have a heavier transformation to perform that it would make more sense to materialize it as a table. It could be more data volume, could be more complex logic, PITs, bridges or even an object that will be used frequently and by many users. For this case, let's first build a new business satellite that for illustration purposes will be deriving additional classification/tiering for orders based on the conditional logic.

```

CREATE OR REPLACE TABLE sat_order_bv
(
    sha1_hub_order      BINARY      NOT NULL
, ldts                TIMESTAMP NOT NULL
, o_orderstatus       STRING
, o_totalprice        NUMBER
, o_orderdate         DATE
, o_orderpriority     STRING
, o_clerk             STRING
, o_shipppriority     NUMBER
, o_comment            STRING
, hash_diff           BINARY      NOT NULL
, rscr                STRING      NOT NULL
-- additional attributes
, order_priority_bucket STRING
, CONSTRAINT pk_sat_order PRIMARY KEY(sha1_hub_order, ldts)
, CONSTRAINT fk_sat_order FOREIGN KEY(sha1_hub_order) REFERENCES 110_rdv.hub_order
)
AS
SELECT sha1_hub_order
, ldts
, o_orderstatus

```

```

, o_totalprice
, o_orderdate
, o_orderpriority
, o_clerk
, o_shipppriority
, o_comment
, hash_diff
, rscr
-- derived additional attributes
, CASE WHEN o_orderpriority IN ('2-HIGH', '1-URGENT') AND
o_totalprice >= 200000 THEN 'Tier-1'
WHEN o_orderpriority IN ('3-MEDIUM', '2-HIGH', '1-URGENT') AND
o_totalprice BETWEEN 150000 AND 200000 THEN 'Tier-2'
ELSE 'Tier-3'
END order_priority_bucket
FROM l10_rdv.sat_order;

```

3. What we are going to do from processing/orchestration perspective is extending our order processing pipeline so that when the task populates a **l10_rdv.sat_order** this will generate a new stream of changes and these changes are going to be propagated by a dependent task to **l20_bdv.sat_order_bv**. This is super easy to do as tasks in Snowflake can be not only schedule-based but also start automatically once the parent task is completed.

```

CREATE OR REPLACE STREAM l10_rdv.sat_order_strm ON TABLE l10_rdv.sat_order;

ALTER TASK l10_rdv.order_strm_tsk SUSPEND;

CREATE OR REPLACE TASK l10_rdv.hub_order_strm_sat_order_bv_tsk
WAREHOUSE = dv_rdv_wh
AFTER l10_rdv.order_strm_tsk
AS
INSERT INTO l20_bdv.sat_order_bv
SELECT
    sha1_hub_order
, ldts
, o_orderstatus
, o_totalprice
, o_orderdate
, o_orderpriority
, o_clerk
, o_shipppriority
, o_comment
, hash_diff
, rscr
-- derived additional attributes
, CASE WHEN o_orderpriority IN ('2-HIGH', '1-URGENT') AND o_totalprice >=
200000 THEN 'Tier-1'
WHEN o_orderpriority IN ('3-MEDIUM', '2-HIGH', '1-URGENT') AND o_totalprice
BETWEEN 150000 AND 200000 THEN 'Tier-2'
ELSE 'Tier-3'
END order_priority_bucket
FROM sat_order_strm;

```

```
ALTER TASK l10_rdv.hub_order_strm_sat_order_bv_tsk RESUME;
ALTER TASK l10_rdv.order_strm_tsk RESUME;
```

4. Now, let's go back to our staging area to process another slice of data to test the task.

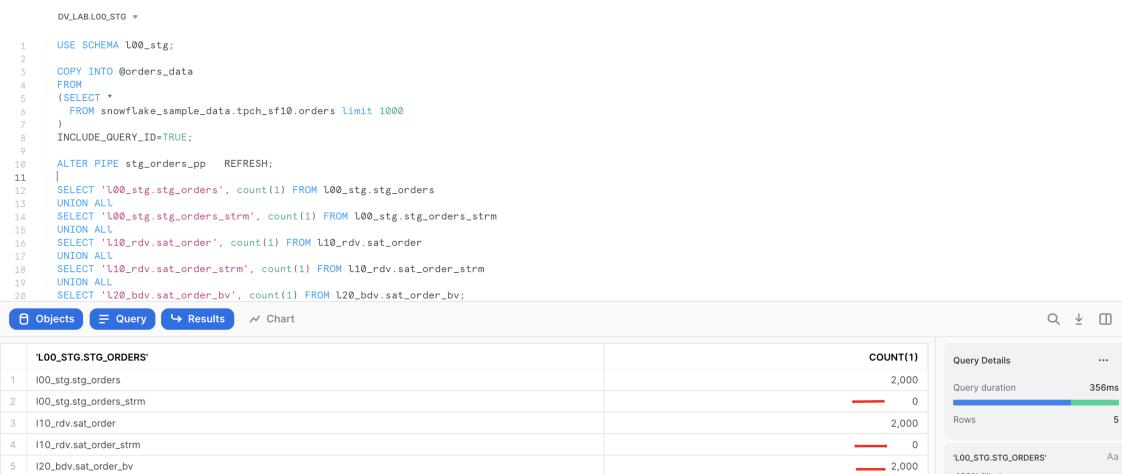
```
USE SCHEMA 100_stg;

COPY INTO @orders_data
FROM
(SELECT *
 FROM snowflake_sample_data.tpch_sf10.orders limit 1000
)
INCLUDE_QUERY_ID=TRUE;

ALTER PIPE stg_orders_pp REFRESH;
```

5. Data is not automatically flowing through all the layers via asynchronous tasks. With the results you can validate:

```
SELECT '100_stg.stg_orders', count(1) FROM 100_stg.stg_orders
UNION ALL
SELECT '100_stg.stg_orders_strm', count(1) FROM 100_stg.stg_orders_strm
UNION ALL
SELECT 'l10_rdv.sat_order', count(1) FROM l10_rdv.sat_order
UNION ALL
SELECT 'l10_rdv.sat_order_strm', count(1) FROM l10_rdv.sat_order_strm
UNION ALL
SELECT 'l20_bdv.sat_order_bv', count(1) FROM l20_bdv.sat_order_bv;
```



Great. Hope this example illustrated few ways of managing **Business Data Vault** objects in our pipeline. Let's finally move into the **Information Delivery** layer.

Build: Information Delivery

When it comes to Information Delivery area we are not changing the meaning of data, but we may change format to simplify users to access and work with the data products/output interfaces. Different consumers may have different needs and preferences, some would prefer star/snowflake dimensional schemas, some would adhere to use flattened objects or even transform data into JSON/parquet objects.

1. First things we would like to add to simplify working with satellites is creating views that shows latest version for each key.

```
-- RDV curr views
-----  
  
USE SCHEMA 110_rdv;  
  
CREATE VIEW sat_customer_curr_vw  
AS  
SELECT *  
FROM sat_customer  
QUALIFY LEAD(ldts) OVER (PARTITION BY sha1_hub_customer ORDER BY ldts) IS NULL;  
  
CREATE OR REPLACE VIEW sat_order_curr_vw  
AS  
SELECT *  
FROM sat_order  
QUALIFY LEAD(ldts) OVER (PARTITION BY sha1_hub_order ORDER BY ldts) IS NULL;  
  
-- BDV curr views
-----  
  
USE SCHEMA 120_bdv;  
  
CREATE VIEW sat_order_bv_curr_vw  
AS  
SELECT *  
FROM sat_order_bv  
QUALIFY LEAD(ldts) OVER (PARTITION BY sha1_hub_order ORDER BY ldts) IS NULL;  
  
CREATE VIEW sat_customer_bv_curr_vw  
AS  
SELECT *  
FROM sat_customer_bv  
QUALIFY LEAD(ldts) OVER (PARTITION BY sha1_hub_customer ORDER BY ldts) IS NULL;
```

2. Let's create a simple dimensional structure. Again, we will keep it virtual(as views) to start with, but you already know that depending on access characteristics required any of these could be selectively materialized.

```
USE SCHEMA 130_id;  
  
-- DIM TYPE 1
```

```

CREATE OR REPLACE VIEW dim1_customer
AS
SELECT hub.shal_hub_customer
      , sat.ldts
      , hub.c_custkey
      , sat.rscr
      , sat.*
FROM 110_rdv.hub_customer
      , 120_bdv.sat_customer_bv_curr_vw
WHERE hub.shal_hub_customer
      = sat.shal_hub_customer;

-- DIM TYPE 1
CREATE OR REPLACE VIEW dim1_order
AS
SELECT hub.shal_hub_order
      , sat.ldts
      , hub.o_orderkey
      , sat.rscr
      , sat.*
FROM 110_rdv.hub_order
      , 120_bdv.sat_order_bv_curr_vw
WHERE hub.shal_hub_order
      = sat.shal_hub_order;

-- FACT table

CREATE OR REPLACE VIEW fct_customer_order
AS
SELECT lnk.ldts
      , lnk.rscr
      , lnk.shal_hub_customer
      , lnk.shal_hub_order
      AS effective_dts
      AS record_source
      AS dim_customer_key
      AS dim_order_key
-- this is a factless fact, but here you can add any measures, calculated or derived
FROM 110_rdv.lnk_customer_order
      AS lnk;

```

3. All good so far? Now lets try to query **fct_customer_order** and at least in my case this view was not returning any rows. Why? If you remember, when we were unloading sample data, we took a subset of random orders and a subset of random customers. Which in my case didn't have any overlap, therefore doing the inner join with dim1_order was resulting in all rows being eliminated from the resultset. Thankfully we are using Data Vault and all need to do is go and load full customer dataset. Just think about it, there is no need to reprocess any links or fact tables simply because customer/reference feed was incomplete. I am sure for those of you who were using different architectures for data engineering and warehousing have painful experience when such situations occur. So, lets go and fix it:

```

USE SCHEMA 100_stg;

COPY INTO @customer_data
FROM
(SELECT object_construct(*)
FROM snowflake_sample_data.tpch_sf10.customer
-- removed LIMIT 10
)
INCLUDE_QUERY_ID=TRUE;

```

```
ALTER PIPE stg_customer_pp    REFRESH;
```

All you need to do now is just wait a few seconds whilst our continuous data pipeline will automatically propagate new customer data into Raw Data Vault. Quick check for the records count in customer dimension now shows that there are 1.5Mn records:

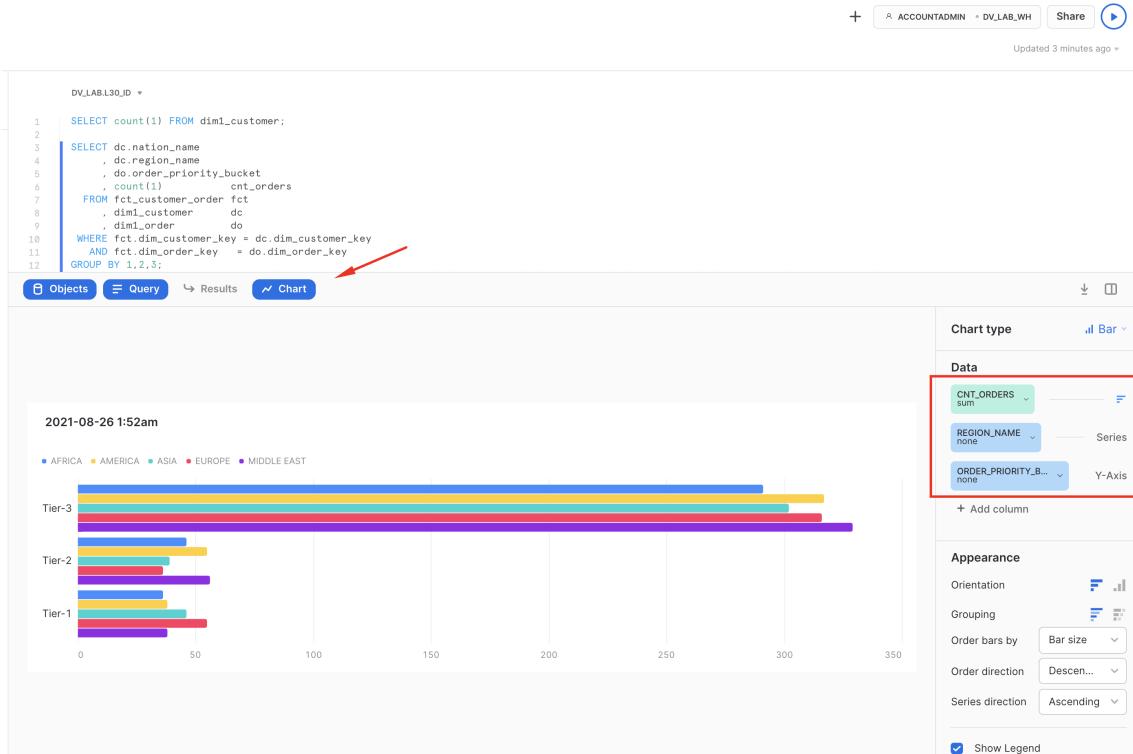
```
USE SCHEMA 130_id;

SELECT COUNT(1) FROM dim1_customer;

COUNT(1)
-----
1,500,000
```

4. Finally lets wear user's hat and run a query to break down orders by nation,region and order_priority_bucket (all attributes we derived in **Business Data Vault**). As we are using Snowsight, why not quickly creating a chart from this result set to better understand the data. For this simply click on the 'Chart' section on the bottom pane and put attributes/measures as it is shown on the screenshot below.

```
SELECT dc.nation_name
      , dc.region_name
      , do.order_priority_bucket
      , COUNT(1)                               cnt_orders
FROM fct_customer_order
      , dim1_customer
      , dim1_order
WHERE fct.dim_customer_key
      AND fct.dim_order_key
      = dc.dim_customer_key
      = do.dim_order_key
GROUP BY 1,2,3;
```



Voila! This concludes our journey for this guide. Hope you enjoyed it and lets summarise key points in the next section.

Conclusion

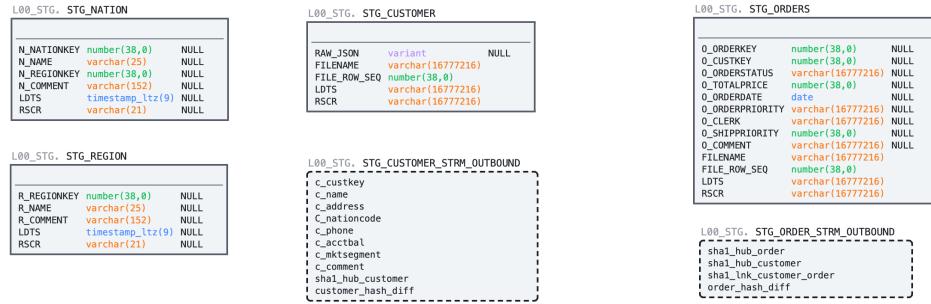
Simplicity of engineering, openness, scalable performance, enterprise-grade governance enabled by the core of the Snowflake platform are now allowing teams to focus on what matters most for the business and build truly agile, collaborative data environments. Teams can now connect data from all parts of the landscape, until there are no stones left unturned. They are even tapping into new datasets via live access to the Snowflake Data Marketplace. The Snowflake Data Cloud combined with a Data Vault 2.0 approach is allowing teams to democratize access to all their data assets at any scale. We can now easily derive more and more value through insights and intelligence, day after day, bringing businesses to the next level of being truly data-driven.

Delivering more usable data faster is no longer an option for today's business environment. Using the Snowflake platform, combined with the Data Vault 2.0 architecture it is now possible to build a world class analytics platform that delivers data for all users in near real-time.

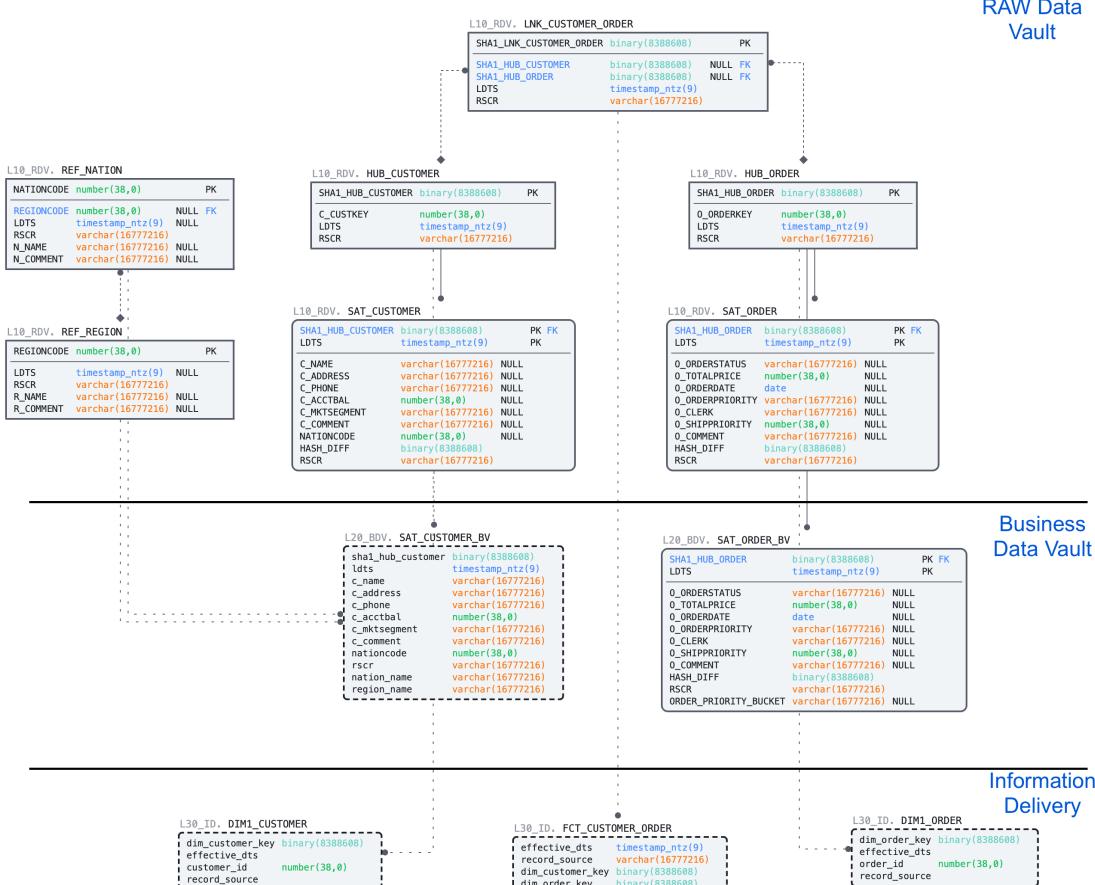
What we've covered

- unloading and loading back data using COPY and Snowpipe
- engineering data pipelines using virtualization, streams and tasks
- building multi-layer Data Vault environment on Snowflake:

Staging



RAW Data Vault



Call to action

- seeing is believing. Try it!
- we made examples limited in size, but feel free to scale the data volumes and virtual warehouse size to see scalability in action
- tap into numerous communities of practice for Data Engineering on Snowflake and Data Vault in particular
- talk to us about modernizing your data landscape! Whether it is Data Vault or not you have on your mind, we have the top expertise and product to meet your demand
- feedback is super welcome!
- Enjoy your journey!

Resource Optimization: Billing Metrics

Overview

This resource optimization guide represents one module of the four contained in the series. These guides are meant to help customers better monitor and manage their credit consumption. Helping our customers build confidence that their credits are being used efficiently is key to an ongoing successful partnership. In addition to this set of Snowflake Quickstarts for Resource Optimization, Snowflake also offers community support as well as Training and Professional Services offerings.

Billing Metrics

Billing queries are responsible for identifying total costs associated with the high level functions of the Snowflake Cloud Data Platform, which includes warehouse compute, snowpipe compute, and storage costs. If costs are noticeably higher in one category versus the others, you may want to evaluate what might be causing that.

These metrics also seek to identify those queries that are consuming the most amount of credits. From there, each of these queries can be analyzed for their importance (do they need to be run as frequently, if at all) and explore if additional controls need to be in place to prevent excessive consumption (i.e. resource monitors, statement timeouts, etc.).

What You'll Learn

- how to identify and analyze Snowflake consumption across all services
- how to analyze most resource-intensive queries
- how to analyze serverless consumption

What You'll Need

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

Related Materials

- Resource Optimization: Setup & Configuration
- Resource Optimization: Usage Monitoring
- 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.

Billing Metrics (T1)

TIER 1

Description:

Identify key metrics as it pertains to total compute costs from warehouses, serverless features, and total storage costs.

How to Interpret Results:

Where are we seeing most of our costs coming from (compute, serverless, storage)? Are seeing excessive costs in any of those categories that are above expectations?

Primary Schema:

Account_Usage

SQL

```
/* These queries can be used to measure where costs have been incurred by
the different cost vectors within a Snowflake account including:
1) Warehouse Costs
2) Serverless Costs
3) Storage Costs

To accurately report the dollar amounts, make changes to the variables
defined on lines 17 to 20 to properly reflect your credit price, the initial
capacity purchased, when your contract started and the term (default 12 months)

If unsure, ask your Sales Engineer or Account Executive
*/
USE DATABASE SNOWFLAKE;
USE SCHEMA ACCOUNT_USAGE;

SET CREDIT_PRICE = 4.00; --edit this number to reflect credit price
SET TERM_LENGTH = 12; --integer value in months
SET TERM_START_DATE = '2019-01-01';
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(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
)

--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')

UNION ALL

SELECT
    NULL as WAREHOUSE_GROUP_NAME
    ,NULL as WAREHOUSE_NAME
    ,NULL as GROUP_CONTACT
    ,NULL as GROUP_COST_CENTER
    ,NULL as GROUP_COMMENT
    ,DA.D_DATE::timestamp as START_TIME
    ,DA.D_DATE::timestamp as END_TIME

```

```

        , PU.CREDITS_PER_DAY AS CREDITS_USED
        , PU.CREDIT_PRICE
        , PU.DOLLARS_PER_DAY AS DOLLARS_USED
        , 'PROJECTED COMPUTE' AS MEASURE_TYPE
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

UNION ALL

SELECT
    NULL as WAREHOUSE_GROUP_NAME
    ,NULL as WAREHOUSE_NAME
    ,NULL as GROUP_CONTACT
    ,NULL as GROUP_COST_CENTER
    ,NULL as GROUP_COMMENT
    ,NULL as START_TIME
    ,NULL as END_TIME
    ,NULL AS CREDITS_USED
    ,PU.CREDIT_PRICE
    ,PU.TOTAL_CONTRACT_VALUE AS DOLLARS_USED
    , 'CONTRACT VALUES' AS MEASURE_TYPE
FROM PROJECTED_USAGE PU
;

```

Screenshot

	WAREHOUSE_GROUP_NAME	WAREHOUSE_NAME	GROUP_CONTACT	GROUP_COST_CENTER	GROUP_COMMENT	START_TIME	END_TIME	CREDITS_USED	\$CREDIT_PRICE	DOLLARS_USED	MEASURE_TYPE
1	WH Compute	COMPUTE_WH				2021-07-09 13:00:00.000-0700	2021-07-09 14:00:00.000-0700	0.399820556	4	1.598482224	ACTUAL_COMPUTE
2	WH Compute	RESET_WH				2021-07-09 12:00:00.000-0700	2021-07-09 13:00:00.000-0700	0.000081389	4	0.000245556	ACTUAL_COMPUTE
3	WH Compute	RESET_WH				2021-07-29 06:00:00.000-0700	2021-07-29 07:00:00.000-0700	2.752222222	4	11.008888888	ACTUAL_COMPUTE
4	Storage	Storage				2021-03-21 00:00:00.000-0700	2021-03-21 00:00:00.000-0700	4	0.625538534001	ACTUAL_COMPUTE	
5	Storage	Storage				2020-10-10 00:00:00.000-0700	2020-10-10 00:00:00.000-0700	4	0.625538534001	ACTUAL_COMPUTE	
6	Storage	Storage				2020-08-25 00:00:00.000-0700	2020-08-25 00:00:00.000-0700	4	0.625538534003	ACTUAL_COMPUTE	
7	Storage	Storage				2020-11-10 00:00:00.000-0800	2020-11-10 00:00:00.000-0800	4	0.646398819087	ACTUAL_COMPUTE	
8	Storage	Storage				2021-07-04 00:00:00.000-0700	2021-07-04 00:00:00.000-0700	4	0.625538534829	ACTUAL_COMPUTE	
9	Storage	Storage				2021-05-19 00:00:00.000-0700	2021-05-19 00:00:00.000-0700	4	0.625538534829	ACTUAL_COMPUTE	
10	Storage	Storage				2020-12-26 00:00:00.000-0800	2020-12-26 00:00:00.000-0800	4	0.8255385348	ACTUAL_COMPUTE	
11	Storage	Storage				2021-08-06 00:00:00.000-0700	2021-08-06 00:00:00.000-0700	4	0.625539930966	ACTUAL_COMPUTE	
12	Storage	Storage				2021-02-12 00:00:00.000-0800	2021-02-12 00:00:00.000-0800	4	0.69256052045	ACTUAL_COMPUTE	
13	Storage	Storage				2021-07-20 00:00:00.000-0700	2021-07-20 00:00:00.000-0700	4	0.625539522069	ACTUAL_COMPUTE	

Most Expensive Queries (T2)

TIER 2

Description:

This query orders the most expensive queries from the last 30 days. It takes into account the warehouse size, assuming that a 1 minute query on larger warehouse is more expensive than a 1 minute query on a smaller warehouse

How to Interpret Results:

This is an opportunity to evaluate expensive queries and take some action. The admin could:

- look at the query profile
- contact the user who executed the query
- take action to optimize these queries

Primary Schema:

Account_Usage

SQL

```
WITH WAREHOUSE_SIZE AS
(
    SELECT WAREHOUSE_SIZE, NODES
    FROM (
        SELECT 'XSMALL' AS WAREHOUSE_SIZE, 1 AS NODES
        UNION ALL
        SELECT 'SMALL' AS WAREHOUSE_SIZE, 2 AS NODES
        UNION ALL
        SELECT 'MEDIUM' AS WAREHOUSE_SIZE, 4 AS NODES
        UNION ALL
        SELECT 'LARGE' AS WAREHOUSE_SIZE, 8 AS NODES
        UNION ALL
        SELECT 'XLARGE' AS WAREHOUSE_SIZE, 16 AS NODES
        UNION ALL
        SELECT '2XLARGE' AS WAREHOUSE_SIZE, 32 AS NODES
        UNION ALL
        SELECT '3XLARGE' AS WAREHOUSE_SIZE, 64 AS NODES
        UNION ALL
        SELECT '4XLARGE' AS WAREHOUSE_SIZE, 128 AS NODES
    )
),
QUERY_HISTORY AS
(
    SELECT QH.QUERY_ID
        ,QH.QUERY_TEXT
        ,QH.USER_NAME
        ,QH.ROLE_NAME
        ,QH.EXECUTION_TIME
        ,QH.WAREHOUSE_SIZE
    FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY QH
    WHERE START_TIME > DATEADD(month, -2, CURRENT_TIMESTAMP())
)
SELECT QH.QUERY_ID
    ,'https://' || current_account() ||
'.snowflakecomputing.com/console#/monitoring/queries/detail?queryId=' || QH.QUERY_ID AS QU
    ,QH.QUERY_TEXT
    ,QH.USER_NAME
    ,QH.ROLE_NAME
    ,QH.EXECUTION_TIME AS EXECUTION_TIME_MILLISECONDS
    ,(QH.EXECUTION_TIME/(1000)) AS EXECUTION_TIME_SECONDS
    ,(QH.EXECUTION_TIME/(1000*60)) AS EXECUTION_TIME_MINUTES
    ,(QH.EXECUTION_TIME/(1000*60*60)) AS EXECUTION_TIME_HOURS
    ,WS.WAREHOUSE_SIZE
    ,WS.NODES
    ,(QH.EXECUTION_TIME/(1000*60*60))*WS.NODES AS RELATIVE_PERFORMANCE_COST
```

```

FROM QUERY_HISTORY QH
JOIN WAREHOUSE_SIZE WS ON WS.WAREHOUSE_SIZE = upper(QH.WAREHOUSE_SIZE)
ORDER BY RELATIVE_PERFORMANCE_COST DESC
LIMIT 200
;

```

Average Cost per Query by Warehouse (T2)

TIER 2

Description:

This summarize the query activity and credit consumption per warehouse over the last month. The query also includes the ratio of queries executed to credits consumed on the warehouse

How to Interpret Results:

Highlights any scenarios where warehouse consumption is significantly out of line with the number of queries executed. Maybe auto-suspend needs to be adjusted or warehouses need to be consolidated.

Primary Schema:

Account_Usage

SQL

```

set credit_price = 4; --edit this value to reflect your credit price

SELECT
    COALESCE(WC.WAREHOUSE_NAME,QC.WAREHOUSE_NAME) AS WAREHOUSE_NAME
    ,QC.QUERY_COUNT_LAST_MONTH
    ,WC.CREDITS_USED_LAST_MONTH
    ,WC.CREDIT_COST_LAST_MONTH
    ,CAST((WC.CREDIT_COST_LAST_MONTH / QC.QUERY_COUNT_LAST_MONTH) AS decimal(10,2) )
AS COST_PER_QUERY

FROM (
    SELECT
        WAREHOUSE_NAME
        ,COUNT(QUERY_ID) as QUERY_COUNT_LAST_MONTH
    FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
    WHERE TO_DATE(START_TIME) >= TO_DATE(DATEADD(month,-1,CURRENT_TIMESTAMP()))
    GROUP BY WAREHOUSE_NAME
) QC
JOIN (
    SELECT
        WAREHOUSE_NAME
        ,SUM(CREDITS_USED) as CREDITS_USED_LAST_MONTH
        ,SUM(CREDITS_USED)*($CREDIT_PRICE) as CREDIT_COST_LAST_MONTH
    FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_METERING_HISTORY
    WHERE TO_DATE(START_TIME) >= TO_DATE(DATEADD(month,-1,CURRENT_TIMESTAMP()))
    GROUP BY WAREHOUSE_NAME
) WC

```

```

    ON WC.WAREHOUSE_NAME = QC.WAREHOUSE_NAME

ORDER BY COST_PER_QUERY DESC
;

```

Screenshot

WAREHOUSE_NAME	QUERY_COUNT_LAST_MONTH	CREDITS_USED_LAST_MONTH	CREDIT_COST_LAST_MONTH	COST_PER_QUERY
DATALAKE_WH	53	8.392294444	33.569177776	0.63
COMPUTE_WH	234	10.023920556	40.095682224	0.17
RESET_WH	109	3.836485278	15.345941112	0.14
ANALYTICS_WH	189	2.526540556	10.106162224	0.05
LOAD_WH	219	2.752667501	11.010670004	0.05
TASK_WH	302	1.077893611	4.311574444	0.01
BI_MEDIUM_WH	2	0.000060000	0.000240000	0.00
BI_LARGE_WH	16734	0.481809443	1.927237772	0.00

AutoClustering Cost History (by Day by Object) (T3)

TIER 3

Description:

Full list of tables with auto-clustering and the volume of credits consumed via the service over the last 30 days, broken out by day.

How to Interpret Results:

Look for irregularities in the credit consumption or consistently high consumption

Primary Schema:

Account_Usage

SQL

```

SELECT TO_DATE(START_TIME) as DATE
,DATABASE_NAME
,SCHEMA_NAME
,TABLE_NAME
,SUM(CREDITS_USED) as CREDITS_USED

FROM "SNOWFLAKE"."ACCOUNT_USAGE"."AUTOMATIC_CLUSTERING_HISTORY"

WHERE START_TIME >= dateadd(month,-1,current_timestamp())
GROUP BY 1,2,3,4
ORDER BY 5 DESC
;

```

Screenshot

CLUSTERING_DATE	DATABASE_NAME	SCHEMA_NAME	TABLE_NAME	CREDITS_USED
2020-02-19	ASHISH_TEST_DB	PUBLIC	GENOTYPES_COPY	266.442864197
2020-04-28	PACIONE_TEST	PUBLIC	COLLAT_AMT_FACT_AC	118.290533383
2020-10-13	CARLINENG	TPCDS_UNCLUSTERED	AUTOCLUSTER_SOLD_DATE_ITEM	78.132147290
2020-10-13	CARLINENG	TPCDS_UNCLUSTERED	AUTOCLUSTER_TICKET_NUMBER	67.328286503
2020-10-13	CARLINENG	TPCDS_UNCLUSTERED	AUTOCLUSTER_SOLD_DATE	57.013410098
2020-01-08	GENOMES_STAGE	GENOME1K	GENOTYPES	29.837784249
2019-12-18	JCLARKE_DB	PUBLIC	SNOW121900_CBD	25.405727086
2020-06-18	SKIPSTANDBOX	PUBLIC	WEB_SALES	18.709104655
2019-12-18	JCLARKE_DB	PUBLIC	SNOW121900_CBTS	11.821605786
2020-06-18	SKIPSTANDBOX	PUBLIC	TEST_CLUSTER_IMPACT_ON_CARD_EST	9.956016226
2020-07-01	GEOSPATIAL	NASANEX	NEX_GDOP_ALL	8.324776052
2020-06-25	JCLARKE_DB	PUBLIC	C_TEST	4.867021840
2020-07-30	SKIPSTANDBOX	PUBLIC	SEARCH_OPTIMIZATION_ON_TABLE_ID-2778292	4.259085124
2020-04-28	NTIERNEY_TEST_DB	CLUSTER_TEST	V_ORDERS_CUSTOMER	3.167348136
2020-06-22	CARLOS_TEST_DB	PUBLIC	LINEITEM	2.060566449
2020-06-26	JCLARKE_DB	PUBLIC	C_TEST	1.202380758
2020-07-02	GEOSPATIAL	NASANEX	NEX_GDOP_ALL	0.698955203
2020-04-28	NTIERNEY_TEST_DB	CLUSTER_TEST	ORDERS	0.575332238
2020-06-23	CARLOS_TEST_DB	PUBLIC	LINEITEM	0.363553380
2020-09-23	CARLINENG	PUBLIC	ORDERS	0.212436370

Materialized Views Cost History (by Day by Object) (T3)

TIER 3

Description:

Full list of materialized views and the volume of credits consumed via the service over the last 30 days, broken out by day.

How to Interpret Results:

Look for irregularities in the credit consumption or consistently high consumption

Primary Schema:

Account_Usage

SQL

```

SELECT

TO_DATE(START_TIME) as DATE
,DATABASE_NAME
,SCHEMA_NAME
,TABLE_NAME
, SUM(CREDITS_USED) as CREDITS_USED

FROM "SNOWFLAKE"."ACCOUNT_USAGE"."MATERIALIZED_VIEW_REFRESH_HISTORY"

WHERE START_TIME >= dateadd(month,-1,current_timestamp())
GROUP BY 1,2,3,4
ORDER BY 5 DESC
;

```

Search Optimization Cost History (by Day by Object) (T3)

TIER 3

Description:

Full list of tables with search optimization and the volume of credits consumed via the service over the last 30 days, broken out by day.

How to Interpret Results:

Look for irregularities in the credit consumption or consistently high consumption

Primary Schema:

Account_Usage

SQL

```
TSELECT  
  
TO_DATE(START_TIME) as DATE  
, DATABASE_NAME  
, SCHEMA_NAME  
, TABLE_NAME  
, SUM(CREDITS_USED) as CREDITS_USED  
  
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."SEARCH_OPTIMIZATION_HISTORY"  
  
WHERE START_TIME >= dateadd(month, -1, current_timestamp())  
GROUP BY 1,2,3,4  
ORDER BY 5 DESC  
;
```

Snowpipe Cost History (by Day by Object) (T3)

TIER 3

Description:

Full list of pipes and the volume of credits consumed via the service over the last 30 days, broken out by day.

How to Interpret Results:

Look for irregularities in the credit consumption or consistently high consumption

Primary Schema:

Account_Usage

SQL

```
SELECT  
  
TO_DATE(START_TIME) as DATE  
, PIPE_NAME  
, SUM(CREDITS_USED) as CREDITS_USED  
  
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."PIPE_USAGE_HISTORY"  
  
WHERE START_TIME >= dateadd(month, -1, current_timestamp())  
GROUP BY 1,2
```

```
ORDER BY 3 DESC  
;
```

Replication Cost History (by Day by Object) (T3)

TIER 3

Description:

Full list of replicated databases and the volume of credits consumed via the replication service over the last 30 days, broken out by day.

How to Interpret Results:

Look for irregularities in the credit consumption or consistently high consumption

Primary Schema:

Account_Usage

SQL

```
SELECT  
TO_DATE(START_TIME) as DATE  
, DATABASE_NAME  
, SUM(CREDITS_USED) as CREDITS_USED  
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."REPLICATION_USAGE_HISTORY"  
WHERE START_TIME >= dateadd(month, -1, current_timestamp())  
GROUP BY 1,2  
ORDER BY 3 DESC  
;
```

Resource Optimization: Performance

Performance

The queries provided in this guide are intended to help you setup and run queries pertaining to identifying areas where poor performance might be causing excess consumption, driven by a variety of factors.

What You'll Learn

- how to identify areas in which performance can be improved
- how to analyze workloads with poor performance causing excess consumption
- how to identify warehouses that would benefit from scaling up or out

What You'll Need

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

Related Materials

- Resource Optimization: Setup & Configuration
- Resource Optimization: Usage Monitoring
- Resource Optimization: Billing Metrics

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.

Data Ingest with Snowpipe and "Copy" (T1)

TIER 1

Description:

This query returns an aggregated daily summary of all loads for each table in Snowflake showing average file size, total rows, total volume and the ingest method (copy or snowpipe)

How to Interpret Results:

With this high-level information you can determine if file sizes are too small or too big for optimal ingest. If you can map the volume to credit consumption you can determine which tables are consuming more credits per TB loaded.

Primary Schema:

Account_Usage

SQL

```
SELECT
    TO_DATE(LAST_LOAD_TIME) AS LOAD_DATE
    , STATUS
    , TABLE_CATALOG_NAME AS DATABASE_NAME
    , TABLE_SCHEMA_NAME AS SCHEMA_NAME
    , TABLE_NAME
    , CASE WHEN PIPE_NAME IS NULL THEN 'COPY' ELSE 'SNOWPIPE' END AS INGEST_METHOD
    , SUM(ROW_COUNT) AS ROW_COUNT
    , SUM(ROW_PARSED) AS ROWS_PARSED
    , AVG(FILE_SIZE) AS AVG_FILE_SIZE_BYTES
    , SUM(FILE_SIZE) AS TOTAL_FILE_SIZE_BYTES
    , SUM(FILE_SIZE)/POWER(1024,1) AS TOTAL_FILE_SIZE_KB
    , SUM(FILE_SIZE)/POWER(1024,2) AS TOTAL_FILE_SIZE_MB
    , SUM(FILE_SIZE)/POWER(1024,3) AS TOTAL_FILE_SIZE_GB
    , SUM(FILE_SIZE)/POWER(1024,4) AS TOTAL_FILE_SIZE_TB
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."COPY_HISTORY"
GROUP BY 1,2,3,4,5,6
ORDER BY 3,4,5,1,2
;
```

Screenshot

LOAD_DATE	STATUS	DATABASE_NAME	SCHEMA_NAME	TABLE_NAME	INGEST_METHOD	ROW_COUNT	ROWS_PARSED	AVG_FILE_SIZE_BYTES	TOTAL_FILE_SIZE_BYTES	TOTAL_FILE_SIZE_KB	TOTAL_FILE_SIZE_MB	TOTAL_FILE_SIZE_GB	TOTAL_FILE_SIZE_TB
2020-08-14	Loaded	DEMO_NW	TASKS	USERS	COPY	5	5	35000000	105	0.0125300425	0.001001358032	9.778857033e-08	9.549604369e-11
2020-09-20	Load failed	DEMO_NW	TASKS	USERS	COPY	0	3	35000000	70	0.0083950375	6.675722015e-05	6.510258922e-08	6.360462912e-11
2020-08-20	Loaded	DEMO_NW	TASKS	USERS	COPY	18	18	34272777	377	0.3681640625	0.0003959352173	3.510180506e-07	3.428790266e-10
2020-08-20	Partially loaded	DEMO_NW	TASKS	USERS	COPY	2	4	39000000	78	0.070171875	7.43665668e-05	7.243195802e-08	7.094058074e-11
2020-12-02	Loaded	DEV_RALVAREZ	COMMON	CTIIRIKE_TRIPS_RESULTS	COPY	45	45	95000000	460	0.46875	0.0024577839719	4.470348359e-07	4.365374568e-10
2020-12-03	Loaded	DEV_RALVAREZ	COMMON	CTIIRIKE_TRIPS_RESULTS	COPY	9	9	96000000	96	0.09375	9.155271438e-05	8.9406967156e-08	8.731149137e-11
2020-12-02	Loaded	DEV_RALVAREZ	COMMON	CTIIRIKE_TRIPS_TEMP2	COPY	18	18	64000000	584	0.375	0.005562103375	5.576278567e-07	5.492459056e-10
2020-12-03	Loaded	DEV_RALVAREZ	COMMON	CTIIRIKE_TRIPS_TEMP2	COPY	3	3	64000000	64	0.0625	0.0031515625e-05	5.960464479e-08	5.820768901e-11
2020-09-28	Loaded	DEV_RALVAREZ	COMMON	TWITTER	SNOWPIPE	4004	4004	541931869	18777707	18337604492788	17.807818887	0.0748810243	1.707235039e-05
2020-09-29	Loaded	DEV_RALVAREZ	COMMON	TWITTER	SNOWPIPE	12502	12502	10894850275	93300128	91113.407226562	88.779736745	0.08688251835	8.4855973e-05
2020-09-30	Loaded	DEV_RALVAREZ	COMMON	TWITTER	SNOWPIPE	7164	7164	6971419435	40991911	4003116205938	39.092932701	0.0387979209	3.728791537e-05
2020-10-01	Loaded	DEV_RALVAREZ	COMMON	TWITTER	SNOWPIPE	2845	2845	5436112100	1343268	13117205864602	12.8103510794	0.0125010142	1.21690854e-05
2020-01-13	Load failed	TEMP_DB	PUBLIC	APP VERSIONS	COPY	0	2160	184000000	555	5.300425	0.005964282277	5.144900619e-09	5.02010754e-09
2020-01-14	Load failed	TEMP_DB	PUBLIC	APP VERSIONS	COPY	1002	1002	760400000	2512	21.984375	0.02146017621	2.06619338e-05	2.04745473e-08
2020-01-10	Loaded	TEMP_DB	PUBLIC	ARRAYLOC	COPY	41	41	184000000	1840	1.796875	0.001754760742	1.7138633537e-06	1.673470251e-09

Scale Up vs. Out (Size vs. Multi-cluster) (T2)

TIER 2

Description:

Two separate queries that list out the warehouses and times that could benefit from either a MCW setting OR scaling up to a larger size

How to Interpret Results:

Use this list to determine reconfiguration of a warehouse and the times or users that are causing contention on the warehouse

Primary Schema:

Account_Usage

SQL

```
--LIST OF WAREHOUSES AND DAYS WHERE MCW COULD HAVE HELPED
SELECT TO_DATE(START_TIME) AS DATE
,WAREHOUSE_NAME
,SUM(AVG_RUNNING) AS SUM_RUNNING
,SUM(AVG_QUEUED_LOAD) AS SUM_QUEUED
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."WAREHOUSE_LOAD_HISTORY"
WHERE TO_DATE(START_TIME) >= DATEADD(month,-1,CURRENT_TIMESTAMP())
GROUP BY 1,2
HAVING SUM(AVG_QUEUED_LOAD) >0
;

--LIST OF WAREHOUSES AND QUERIES WHERE A LARGER WAREHOUSE WOULD HAVE HELPED WITH
REMOTE SPILLING
SELECT QUERY_ID
,USER_NAME
,WAREHOUSE_NAME
,WAREHOUSE_SIZE
,BYTES_SCANNED
,BYTES_SPILLED_TO_REMOTE_STORAGE
,BYTES_SPILLED_TO_REMOTE_STORAGE / BYTES_SCANNED AS SPILLING_READ_RATIO
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
WHERE BYTES_SPILLED_TO_REMOTE_STORAGE > BYTES_SCANNED * 5 -- Each byte read was
spilled 5x on average
ORDER BY SPILLING_READ_RATIO DESC
;
```

Screenshot

DATE	WAREHOUSE_NAME	SUM_RUNNING	SUM_QUEUED
2020-10-10	SGURSOY_MCW_CLUSTER	23.134146666	71.554796667
2020-10-06	DGARDNER_MCW_CLUSTER	30.532923333	108.237089999
2020-10-27	AP_WAREHOUSE	4.105586666	0.328790000
2020-10-06	VLAD	12.718840002	4.664140001

Warehouse Cache Usage (T3)

TIER 3

Description:

Aggregate across all queries broken out by warehouses showing the percentage of data scanned from the warehouse cache.

How to Interpret Results:

Look for warehouses that are used from querying/reporting and have a low percentage. This indicates that the warehouse is suspending too quickly

Primary Schema:

Account_Usage

SQL

```

SELECT WAREHOUSE_NAME
, COUNT(*) AS QUERY_COUNT
, SUM(BYTES_SCANNED) AS BYTES_SCANNED
, SUM(BYTES_SCANNED*PERCENTAGE_SCANNED_FROM_CACHE) AS BYTES_SCANNED_FROM_CACHE
, SUM(BYTES_SCANNED*PERCENTAGE_SCANNED_FROM_CACHE) / SUM(BYTES_SCANNED) AS
PERCENT_SCANNED_FROM_CACHE
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
WHERE START_TIME >= dateadd(month, -1, current_timestamp())
AND BYTES_SCANNED > 0
GROUP BY 1
ORDER BY 5
;

```

Screenshot

WAREHOUSE_NAME	QUERY_COUNT	BYTES_SCANNED	BYTES_SCANNED_FROM_CACHE	PERCENT_SCANNED_FROM_CACHE
ADMIN_WH	2	3656736768	0	0
MRAINEY_WH	8	4075008	0	0
DEMO_WH	1	237568	0	0
SKUMAR_WH	2	115639296	1536	1.328268204e-05
SCWH	3	4732139520	2174976	0.0004596178939
REF_ER_WH	25	676613120	1774080	0.00262200059
MPROCTOR_XS_WH	8	10470936656	83157504	0.007941744538
ADHOC	60	5188376544	79983072	0.01541581867
MHENSON_WH	221	353100925440	6399581184	0.01812394339

Heavy Scanners (T3)

TIER 3

Description:

Ordered list of users that run queries that scan a lot of data.

How to Interpret Results:

This is a potential opportunity to train the user or enable clustering.

Primary Schema:

Account_Usage

SQL

```

select
    User_name
    , warehouse_name
    , avg(case when partitions_total > 0 then partitions_scanned / partitions_total else 0
end) avg_pct_scanned
from snowflake.account_usage.query_history
where start_time::date > dateadd('days', -45, current_date)
group by 1, 2
order by 3 desc
;

```

Full Table Scans by User (T3)

TIER 3

Description:

These queries are the list of users that run the most queries with near full table scans and then the list of the queries themselves.

How to Interpret Results:

This is a potential opportunity to train the user or enable clustering.

Primary Schema:

Account_Usage

SQL

```

--who are the users with the most (near) full table scans
SELECT USER_NAME
,COUNT(*) as COUNT_OF_QUERIES
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
WHERE START_TIME >= dateadd(month,-1,current_timestamp())
AND PARTITIONS_SCANNED > (PARTITIONS_TOTAL*0.95)
AND QUERY_TYPE NOT LIKE 'CREATE%'
group by 1
order by 2 desc;

-- This gives all queries in the last month with nearly a full table scan :) > 95%,
-- ordered by the worst offending
SELECT *
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."QUERY_HISTORY"
WHERE START_TIME >= dateadd(month,-1,current_timestamp())
AND PARTITIONS_SCANNED > (PARTITIONS_TOTAL*0.95)
AND QUERY_TYPE NOT LIKE 'CREATE%'
ORDER BY PARTITIONS_SCANNED DESC
LIMIT 50 -- Configurable threshold that defines "TOP N=50"
;
```

Top 10 Spillers Remote (T3)

TIER 3

Description:

Identifies the top 10 worst offending queries in terms of bytes spilled to remote storage.

How to Interpret Results:

These queries should most likely be run on larger warehouses that have more local storage and memory.

Primary Schema:

Account_Usage

SQL

```
select query_id, substr(query_text, 1, 50) partial_query_text, user_name,
warehouse_name, warehouse_size,
BYTES_SPOILED_TO_REMOTE_STORAGE, start_time, end_time, total_elapsed_time/1000
total_elapsed_time
from snowflake.account_usage.query_history
where BYTES_SPOILED_TO_REMOTE_STORAGE > 0
and start_time::date > dateadd('days', -45, current_date)
order by BYTES_SPOILED_TO_REMOTE_STORAGE desc
limit 10
;
```

AutoClustering History & 7-Day Average (T3)

TIER 3**Description:**

Average daily credits consumed by Auto-Clustering grouped by week over the last year.

How to Interpret Results:

Look for anomalies in the daily average over the course of the year. Opportunity to investigate the spikes or changes in consumption.

Primary Schema:

Account_Usage

SQL

```
WITH CREDITS_BY_DAY AS (
SELECT TO_DATE(START_TIME) as DATE
, SUM(CREDITS_USED) as CREDITS_USED

FROM "SNOWFLAKE"."ACCOUNT_USAGE"."AUTOMATIC_CLUSTERING_HISTORY"

WHERE START_TIME >= dateadd(year, -1, current_timestamp())
GROUP BY 1
ORDER BY 2 DESC
)

SELECT DATE_TRUNC('week', DATE)
```

```
, AVG(CREDITS_USED) AS AVG_DAILY_CREDITS
FROM CREDITS_BY_DAY
GROUP BY 1
ORDER BY 1
;
```

Materialized Views History & 7-Day Average (T3)

TIER 3

Description:

Average daily credits consumed by Materialized Views grouped by week over the last year.

How to Interpret Results:

Look for anomalies in the daily average over the course of the year. Opportunity to investigate the spikes or changes in consumption.

Primary Schema:

Account_Usage

SQL

```
WITH CREDITS_BY_DAY AS (
SELECT TO_DATE(START_TIME) AS DATE
, SUM(CREDITS_USED) AS CREDITS_USED

FROM "SNOWFLAKE"."ACCOUNT_USAGE"."MATERIALIZED_VIEW_REFRESH_HISTORY"

WHERE START_TIME >= dateadd(year,-1,current_timestamp())
GROUP BY 1
ORDER BY 2 DESC
)

SELECT DATE_TRUNC('week',DATE)
, AVG(CREDITS_USED) AS AVG_DAILY_CREDITS
FROM CREDITS_BY_DAY
GROUP BY 1
ORDER BY 1
;
```

Search Optimization History & 7-Day Average (T3)

TIER 3

Description:

Average daily credits consumed by Search Optimization grouped by week over the last year.

How to Interpret Results:

Look for anomalies in the daily average over the course of the year. Opportunity to investigate the spikes or changes in consumption.

Primary Schema:

Account_Usage

SQL

```
WITH CREDITS_BY_DAY AS (
  SELECT TO_DATE(START_TIME) as DATE
  , SUM(CREDITS_USED) as CREDITS_USED

  FROM "SNOWFLAKE"."ACCOUNT_USAGE"."SEARCH_OPTIMIZATION_HISTORY"

  WHERE START_TIME >= dateadd(year,-1,current_timestamp())
  GROUP BY 1
  ORDER BY 2 DESC
)

SELECT DATE_TRUNC('week',DATE)
, AVG(CREDITS_USED) as AVG_DAILY_CREDITS
FROM CREDITS_BY_DAY
GROUP BY 1
ORDER BY 1
;
```

Snowpipe History & 7-Day Average (T3)

TIER 3**Description:**

Average daily credits consumed by Snowpipe grouped by week over the last year.

How to Interpret Results:

Look for anomalies in the daily average over the course of the year. Opportunity to investigate the spikes or changes in consumption.

Primary Schema:

Account_Usage

SQL

```
WITH CREDITS_BY_DAY AS (
  SELECT TO_DATE(START_TIME) as DATE
  , SUM(CREDITS_USED) as CREDITS_USED

  FROM "SNOWFLAKE"."ACCOUNT_USAGE"."PIPE_USAGE_HISTORY"

  WHERE START_TIME >= dateadd(year,-1,current_timestamp())
  GROUP BY 1
  ORDER BY 2 DESC
)
```

```
SELECT DATE_TRUNC('week', DATE)
, AVG(CREDITS_USED) as AVG_DAILY_CREDITS
FROM CREDITS_BY_DAY
GROUP BY 1
ORDER BY 1
;
```

Replication History & 7-Day Average (T3)

TIER 3

Description:

Average daily credits consumed by Replication grouped by week over the last year.

How to Interpret Results:

Look for anomalies in the daily average over the course of the year. Opportunity to investigate the spikes or changes in consumption.

Primary Schema:

Account_Usage

SQL

```
WITH CREDITS_BY_DAY AS (
SELECT TO_DATE(START_TIME) as DATE
, SUM(CREDITS_USED) as CREDITS_USED
FROM "SNOWFLAKE"."ACCOUNT_USAGE"."REPLICATION_USAGE_HISTORY"
WHERE START_TIME >= dateadd(year, -1, current_timestamp())
GROUP BY 1
ORDER BY 2 DESC
)

SELECT DATE_TRUNC('week', DATE)
, AVG(CREDITS_USED) as AVG_DAILY_CREDITS
FROM CREDITS_BY_DAY
GROUP BY 1
ORDER BY 1
;
```

Getting Started With Snowflake SQL API

Overview

Welcome! The Snowflake SQL API is a [REST API](#) that you can use to access and update data in a Snowflake database. You can use this API to execute [standard queries] and most [DDL] and [DML] statements.

This getting started guide will walk you through executing a SQL statement with the API and retrieving the results.

Prerequisites

- A familiarity with Snowflake
- A familiarity with SQL

What You'll Learn

- Perform simple queries
- Manage your deployment (e.g., provision users and roles, create tables, etc.)
- Submit one SQL statement for execution per API call.
- Check the status of the execution of a statement.
- Cancel the execution of a statement.

What You'll Need

- A Snowflake Account with an accessible warehouse, database, schema, and role
- SnowSQL 1.2.17 or higher
- Working [Key-Pair authentication](#)

What You'll Build

- An execution of a statement using the Snowflake SQL API

Introducing the API

Head to the SQL API by navigating to your version of the following URL, replacing `*account_locator*` with the account locator for your own Snowflake account:

```
https://*account_locator*.snowflakecomputing.com/api/v2
```

Negative : Note that the account locator might include additional segments for your region and cloud provider. See [Specifying Region Information in Your Account Hostname] for details.

Now let's break down the parts of the API before we begin using it. The API consists of the `/api/v2/statements/` resource and provides the following endpoints:

- `/api/v2/statements` : You'll use this endpoint to [submit a SQL statement for execution].
- `/api/v2/statements/*statementHandle*` : You'll use this endpoint to [check the status of the execution of a statement].
- `/api/v2/statements/*statementHandle*/cancel` : You'll use this endpoint to [cancel the execution of a statement].

In the steps to come, you shall use all these endpoints to familiarize yourself with the API.

Positive : You can use development tools and libraries for REST APIs (e.g., Postman) to send requests and handle responses.

Limitations of the SQL API

It's important to be aware of the [limitations that the SQL API] currently has. In particular noting that `GET` and `PUT` are not supported.

Assigning a Unique Request ID for Resubmitting Requests

In some cases, it might not be clear if Snowflake executed the SQL statement in an API request (e.g., due to a network error or a timeout). You might choose to resubmit the same request to Snowflake again in case Snowflake did not execute the statement.

If Snowflake already executed the statement in the initial request and you resubmit the request again, the statement is executed twice. For some types of requests, repeatedly executing the same statement can have unintended consequences (e.g., inserting duplicate data into a table).

To prevent Snowflake from executing the same statement twice when you resubmit a request, you can use a request ID to distinguish your request from other requests. Suppose you specify the same request ID in the initial request and in the resubmitted request. In that case, Snowflake does not execute the statement again if the statement has already executed successfully.

To specify a request ID, generate a [universally unique identifier \(UUID\)](#), and include this identifier in the `requestId` query parameter:

```
POST /api/v2/statements?requestId=<UUID> HTTP/1.1
```

If Snowflake fails to process a request, you can submit the same request again with the same request ID. Using the same request ID indicates to the server that you are submitting the same request again.

Now let's move on to additional information you need to include in requests: authentication parameters.

Authenticating to the Server

When you send a request, the request must include authentication information. There are two options for providing authentication: OAuth and JWT key pair authentication. You can use whichever one you have previously implemented or whichever one you are most comfortable with. This example will be detailing authentication with [JWT](#).

If you haven't done so already, make sure you have [key pair authentication] working with Snowflake already.

You can test to make sure you can successfully connect to Snowflake Key Pairs using the following command:

```
$ snowsql -a <account> -u <user> --private-key-path <path to private key>
```

After you've verified you can connect to Snowflake using key-pair authentication, you'll need to generate a JWT token. This JWT token is time limited token which has been signed with your key and Snowflake will know that you authorized this token to be used to authenticate as you for the SQL API.

```
$ snowsql -a <account> -u <user> --private-key-path <path to private key> --generate-jwt
<returns JWT token>
```

You'll need the JWT token generated to be used for using the SQL API. The following headers need be set in each API request that you send within your application code:

- `Authorization: Bearer *jwt_token*` where `*jwt_token*` is the generated JWT token from SnowSQL

- X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT

Altogether, your request query and header will take the following form:

```
POST /api/v2/statements?requestId=<UUID> HTTP/1.1
Authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT
```

Now that you have been introduced to authentication and unique request IDs, you can now move to actually making a request to execute a SQL statement.

Submitting a Request to Execute a SQL Statement

To submit a SQL statement for execution, send a [POST request to the /api/v2/statements/ endpoint]:

```
POST /api/v2/statements?requestId=<UUID> HTTP/1.1
Authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT

(request body)
```

In the request URL, you can also set query parameters to:

- Execute the statement asynchronously: `async=true`

For the [body of the request], set the following fields:

- Set the `statement` field to the SQL statement that you want to execute.
- To specify the warehouse, database, schema, and role to use, set the `warehouse`, `database`, `schema`, and `role` fields.

Negative : Note: the values in these fields are case-sensitive.

- To set a timeout for the statement execution, set the `timeout` field to the maximum number of seconds to wait. If the `timeout` field is not set, the timeout specified by the [STATEMENT_TIMEOUT_IN_SECONDS] parameter is used.

```
POST /api/v2/statements HTTP/1.1
Authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT

{
  "statement": "select * from T",
```

```
"timeout": 60,  
"database": "<your_database>",  
"schema": "<your_schema>",  
"warehouse": "<your_warehouse>",  
"role": "<your_role>"  
}
```

Let's go over some specific fields in this request:

- The `statement` field specifies the SQL statement to execute.
- The `timeout` field specifies that the server allows 60 seconds for the statement to be executed.

If the statement was executed successfully, Snowflake returns the HTTP response code 200 and the first results in a [ResultSet] object. We'll go over how to check the status and retrieve the results after we look at including bind variables.

Now we'll look at how you can include bind variables (`?` placeholders) in the statement and set the `bindings` field to an object that specifies the corresponding Snowflake data types and values for each variable.

Using Bind Variables in a Statement

If you want to use bind variables (`?` placeholders) in the statement, use the `bindings` field to specify the values that should be inserted.

Set this field to a JSON object that specifies the [Snowflake data type] and value for each bind variable.

```
...  
"statement": "select * from T where c1=?",  
...  
"bindings": {  
  "1": {  
    "type": "FIXED",  
    "value": "123"  
  }  
},  
...
```

Choose the binding type that corresponds to the type of the value that you are binding. For example, if the value is a string representing a date (e.g. `2021-04-15`) and you want to insert the value into a DATE column, use the `TEXT` binding type.

The following table specifies the values of the `type` field that you can use to bind to different [Snowflake data types] for this preview release.

- The first column on the left specifies the binding types that you can use.
- The rest of the columns specify the Snowflake data type of the column where you plan to insert the data.
- Each cell specifies the type of value that you can use with a binding type to insert data into a column of a particular Snowflake data type.

Negative : If the cell for a binding type and Snowflake data type is empty, you cannot use the specified binding type to insert data into a column of that Snowflake data type.

Binding types supported for different Snowflake data types [¶]										
Snowflake Data Types	INT / NUMBER	FLOAT	VARCHAR	BINARY	BOOLEAN	DATE	TIME	TIMESTAMP_TZ	TIMESTAMP_LTZ	TIMESTAMP_NTZ
Binding Types										
FIXED	integer	integer	integer		0 (false) / nonzero (true)					
REAL	integer	int or float	int or float		0/non-0					
TEXT	integer	int or float	any text	hexdec	"true" / "false"	see notes below				
BINARY										
BOOLEAN		true/false, 0/1		true/false						
DATE		epoch (ms)		epoch (ms)		epoch (ms)		epoch (ms)		
TIME		epoch (nano)		epoch (nano)						
TIMESTAMP_TZ		epoch (nano)		epoch (nano)		epoch (nano)		epoch (nano)		
TIMESTAMP_LTZ		epoch (nano)		epoch (nano)		epoch (nano)		epoch (nano)		
TIMESTAMP_NTZ		epoch (nano)		epoch (nano)		epoch (nano)		epoch (nano)		

For additional information on binding specific data types, see the documentation's section on [Using Bind Variables in a Statement]

If the value is in a format not supported by Snowflake, the API returns an error:

```
{
  code: "100037",
  message: "<bind type> value '<value>' is not recognized",
  sqlState: "22018",
  statementHandle: "<ID>"
}
```

Whether you use bind variables or not, you'll want to check the status of your statements. Let's look at that next.

Checking the Status of the Execution of the Statement

When you submit a SQL statement for execution, Snowflake returns a 202 response code if the execution of the statement has not yet been completed or if you submitted an asynchronous query.

In the body of the response, Snowflake includes a [QueryStatus] object. The `statementStatusUrl` field in this object specifies the URL to the [/api/v2/statements/ endpoint] that you can use to check the execution status:

```
{
  "code": "090001",
  "sqlState": "00000",
  "message": "successfully executed",
  "statementHandle": "e4ce975e-f7ff-4b5e-b15e-bf25f59371ae",
  "statementStatusUrl": "/api/v2/statements/e4ce975e-f7ff-4b5e-b15e-bf25f59371ae"
}
```

As illustrated by the URL above, in requests to check the status of a statement and cancel the execution of a statement, you specify the statement handle (a unique identifier for the statement) as a path parameter in order to identify the statement to use.

Note that the `QueryStatus` object also provides the statement handle as a separate value in the `statementHandle` field.

To check the status of the execution of the statement, send a GET request using this URL:

```
GET /api/v2/statements/{statementHandle}
```

For example, the following request checks the execution status of the statement with the handle e4ce975e-f7ff-4b5e-b15e-bf25f59371ae :

```
GET /api/v2/statements/e4ce975e-f7ff-4b5e-b15e-bf25f59371ae HTTP/1.1
Authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT
```

If the statement has finished executing successfully, Snowflake returns the HTTP response code 200 and the first results in a [ResultSet] object. However, if an error occurred when executing the statement, Snowflake returns the HTTP response code 422 with a [QueryFailureStatus] object.

Once the statement has executed successfully, you can then retrieve the results, detailed in the next step.

Cancelling the Execution of a SQL Statement

To cancel the execution of a statement, send a POST request to the [cancel endpoint].

```
POST /api/v2/statements/{statementHandle}/cancel
```

Retrieving the Results

If you [submit a SQL statement for execution] or [check the status of statement execution], Snowflake returns a [ResultSet] object in the body of the response if the statement was executed successfully.

The following is an example of a `ResultSet` object that is returned for a query, truncated for brevity.

```
{
  "code" : "090001",
  "statementStatusUrl" : "/api/v2/statements/01a288b9-0603-af68-0000-328502422e7e?
requestId=f8cccd534-7cd5-4c06-b673-f25361e96d7f",
  "requestId" : "f8cccd534-7cd5-4c06-b673-f25361e96d7f",
  "sqlState" : "00000",
  "statementHandle" : "01a288b9-0603-af68-0000-328502422e7e",
  "message" : "Statement executed successfully.",
  "createdOn" : 1645742998434,
  "resultSetMetaData" : {
    "rowType" : [ {
      "name" : "HIGH_NDV_COLUMN",
      "database" : "",
      "schema" : "",
      "table" : "",
      "type" : "fixed",
      "scale" : 0,
      "precision" : 19,
      "byteLength" : null,
      "nullable" : false,
      "isTemporary" : false
    } ]
  }
}
```

```

    "collation" : null,
    "length" : null
}, {
    "name" : "LOW_NDV_COLUMN",
    "database" : "",
    "schema" : "",
    "table" : "",
    "type" : "fixed",
    "scale" : 0,
    "precision" : 2,
    "byteLength" : null,
    "nullable" : false,
    "collation" : null,
    "length" : null
}, {
    "name" : "CONSTANT_COLUM",
    "database" : "",
    "schema" : "",
    "table" : "",
    "type" : "fixed",
    "scale" : 0,
    "precision" : 1,
    "byteLength" : null,
    "nullable" : false,
    "collation" : null,
    "length" : null
} ],
" numRows" : 100000000,
"format" : "jsonv2",
"partitionInfo" : [ {
    "rowCount" : 8192,
    "uncompressedSize" : 152879,
    "compressedSize" : 22412
}, {
    "rowCount" : 53248,
    "uncompressedSize" : 1048161,
    "compressedSize" : 151251
}, {
    "rowCount" : 86016,
    "uncompressedSize" : 1720329,
    "compressedSize" : 249447
}, {

```

Notice that there is an `ARRAY` of `partitionInfo` objects. These partitions objects give you some information, such as `rowCount` and `size`, about the partitions that are available for retrieval. The API returns the data data inline in JSON of the first partition, or partition `0` with the response and we'll cover how to retrieve subsequent partitions in a later section.

Getting Metadata About the Result

It also includes, in the response the `rowType` which gives additional metadata about the datatypes and names of data returned from the query. This metadata is only included in the initial response, and no metadata is returned when retrieving subsequent partitions.

In the `ResultSet` object returned in the response, the `resultSetMetaData` field contains a `[ResultSet_resultSetMetaData]` object that describes the result set (for example, the format of the results, etc.).

In this object, the `rowType` field contains an array of `[ResultSet_resultSetMetaData_rowType]` objects. Each object describes a column in the results. The `type` field specifies the Snowflake data type of the column.

```
{  
  "resultSetMetaData": {  
    "rowType": [  
      {  
        "name": "ROWNUM",  
        "type": "FIXED",  
        "length": 0,  
        "precision": 38,  
        "scale": 0,  
        "nullable": false  
      }, {  
        "name": "ACCOUNT_NAME",  
        "type": "TEXT",  
        "length": 1024,  
        "precision": 0,  
        "scale": 0,  
        "nullable": false  
      }, {  
        "name": "ADDRESS",  
        "type": "TEXT",  
        "length": 16777216,  
        "precision": 0,  
        "scale": 0,  
        "nullable": true  
      }, {  
        "name": "ZIP",  
        "type": "TEXT",  
        "length": 100,  
        "precision": 0,  
        "scale": 0,  
        "nullable": true  
      }, {  
        "name": "CREATED_ON",  
        "type": "TIMESTAMP_NTZ",  
        "length": 0,  
        "precision": 0,  
        "scale": 3,  
        "nullable": false  
      }  
    ]  
  },  
}
```

Retrieving Result Partitions

Partitions are [retrieved] using the `partition=n` query parameter at `/api/v2/statements/<handle>?partition=<partition_number>` endpoint. This can be used to iterate, or even retrieve in parallel, the results from the SQL API call.

```
GET /api/v2/statements/e4ce975e-f7ff-4b5e-b15e-bf25f59371ae?partition=1 HTTP/1.1
Authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT
```

The above query call, with `partition=1` returns just the data below. Notice that the `data` object does not contain any of the additional metadata about the result. Only the data, in gzipped format that needs to be uncompressed.

```
{"data": [
  ["32768", "3", "5"],
  ["32772", "4", "5"],
  ["32776", "3", "5"],
  ["32780", "3", "5"],
  ["32784", "2", "5"],
  ...
]}
```

Each array within the array contains the data for a row:

- The first element in each array is a JSON string containing a sequence ID that starts from 0.
- The rest of the elements in each array represent the data in a row.

The data in the result set is encoded in JSON v1.0, which means that all data is expressed as strings, regardless of the Snowflake data type of the column.

For example, the value `1.0` in a `NUMBER` column is returned as the string `" 1.0"`. As another example, timestamps are returned as the number of nanoseconds since the epoch. For example, the timestamp for Thursday, January 28, 2021 10:09:37.123456789 PM is returned as `"1611871777123456789"`.

You are responsible for converting the strings to the appropriate data types.

Helpful URLs to iterate and retrieve partitions

Since your results may contain significant number of partitions, the Snowflake SQL API provides a special header, `Link`, which assists in helping clients traverse and retrieve those results.

To get the next partition of results or other partitions, use the URLs provided in the [Link header] in the HTTP response. The `Link` header specifies the URLs for retrieving the first, next, previous, and last partitions of results:

```
HTTP/1.1 200 OK
Link: </api/v2/statements/01a288b9-0603-af68-0000-328502422e7e?requestId=918e2211-
d1d6-4c53-bec3-457d047651f7&partition=0>; rel="first"
,</api/v2/statements/01a288b9-0603-af68-0000-328502422e7e?requestId=51ad0c0a-e514-
4d8a-9cf2-cf537d439e39&partition=1>; rel="next"
,</api/v2/statements/01a288b9-0603-af68-0000-328502422e7e?requestId=c6a17bb3-7593-
489d-bbac-5e3b268bc6da&partition=47>; rel="last"
...
```

Getting Started with Snowpark Python

Overview

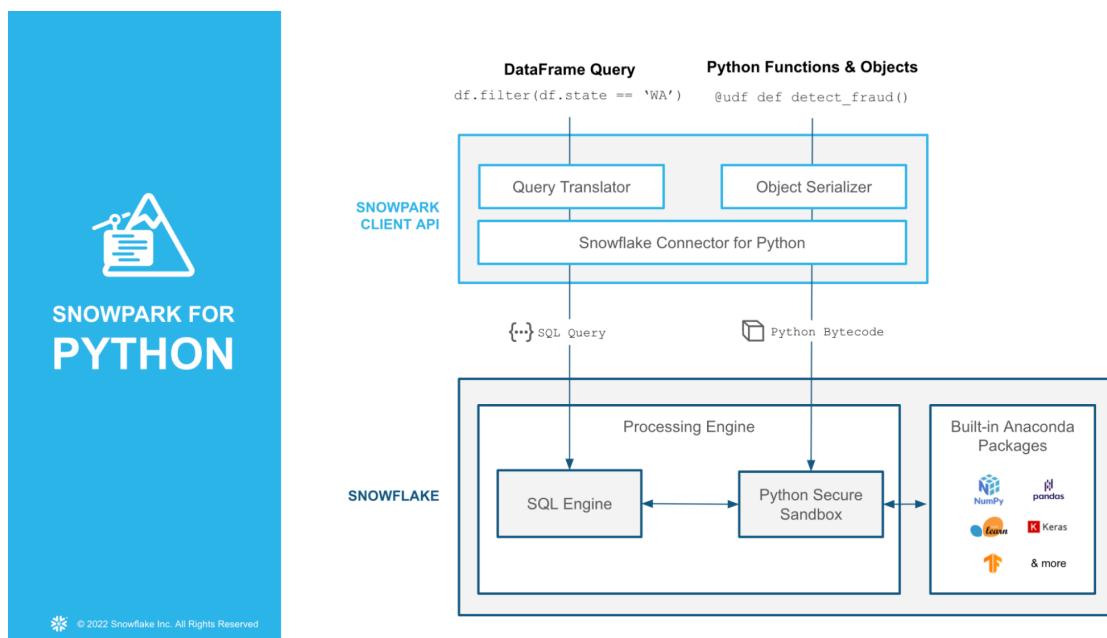
Python is the language of choice for Data Science and Machine Learning workloads. Snowflake has long supported Python via the Python Connector, allowing data scientists to interact with data stored in Snowflake from their preferred Python environment. This did, however, require data scientists to write verbose SQL queries. To provide a more friendly, expressive, and extensible interface to Snowflake, we built **Snowpark Python**, a native Python experience with a pandas and PySpark-like API for data manipulation. This includes a client-side API to allow users to write Python code in a Spark-like API without the need to write verbose SQL. Python UDF and Stored Procedure support also provides more general additional capabilities for compute pushdown.

Snowpark includes client-side APIs and server-side runtimes that extends Snowflake to popular programming languages including Scala, Java, and Python. Ultimately, this offering provides a richer set of tools for Snowflake users (e.g. Python's extensibility and expressiveness) while still leveraging all of Snowflake's core features, and the underlying power of SQL, and provides a clear path to production for machine learning products and workflows.

A key component of Snowpark for Python is that you can "Bring Your Own IDE"- anywhere that you can run a Python kernel, you can run client-side Snowpark Python. You can use it in your code development the exact same way as any other Python library or module. In this lab, we will be using Jupyter Notebooks, but you could easily replace Jupyter with any IDE of your choosing.

Throughout this lab, we will specifically explore the power of the Snowpark Python Dataframe API, as well as server-side Python runtime capabilities, and how Snowpark Python can enable and accelerate end-to-end Machine Learning workflows.

The source code for this lab is available on [GitHub](#).



What You'll Learn

- How to create a DataFrame that loads data from a stage
- How to perform data and feature engineering using Snowpark DataFrame API
- How to bring a trained ML model into Snowflake as a UDF to score new data

What You'll Need

- A Snowflake Account with [Anaconda Integration enabled by ORGADMIN]
- if you do not already have a Snowflake account, you can register for a [free trial account](#)
- A Snowflake login with the `ACCOUNTADMIN` role. If you have this role in your corporate environment, you may choose to use it directly. If not, you will either need to (1) register for a free trial account above, (2) use a different role that has the ability to create and use a database, schema and tables, and UDFs (edit the `config.py` file and Jupyter notebooks to use this alternative role), OR (3) use an existing database and schema in which you can create tables and UDFs (edit the `config.py` file and notebook to use that role, database, and schema). If you're confused, it is best to just sign up for a free Enterprise-level trial account via the link above.
- Python 3.8
- Jupyter Notebook
- Snowpark Python 0.7.0 API (included in the [Quickstart GitHub repo](#))

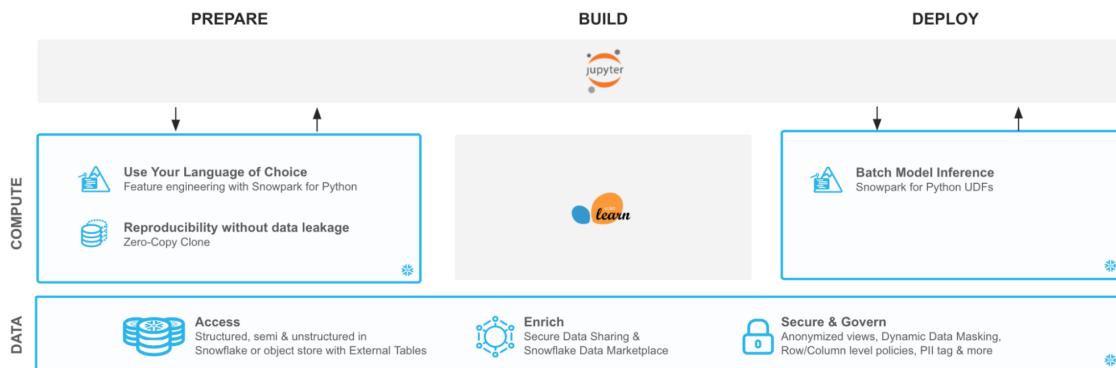
What You'll Build

You will build an end-to-end data science workflow leveraging Snowpark for Python to load, clean and prepare data and then deploy our trained model in Snowflake using Python UDF for inference.

Use-Case: Predicting Customer Churn

You are part of a team of data engineers and data scientists at a Telecom company that has been tasked to reduce customer churn using a machine learning based solution.

To build this, you have access to customer demographic and billing data. Using Snowpark, we will ingest, analyze and transform this data to train a model that will then be deployed inside Snowflake to score new data.



With Snowflake, it is easy to make all relevant data instantly accessible to your machine learning models whether it is for training or inference. For this guide, we are going to do all of our data and feature engineering with Snowpark for Python but users can choose to work with SQL or any of the other Snowpark supported languages including Java and Scala without the need for separate environments.

To streamline your path to production, we will learn how to bring trained models (whether trained inside Snowflake or in an external environment) to run directly inside Snowflake as a UDF bringing models where the data and data pipelines live.

Setup

Let's set up the Python environment necessary to run this lab:

First, clone the source code for this repo to your local environment (you can also download the code [here](#) and unzip the files into the location of your choosing in your local environment):

```
git clone git@github.com:fenago/getting-started-snowpark-python.git  
cd getting-started-snowpark-python/customer-churn-prediction
```

Snowpark Python via Anaconda

If you are using [Anaconda](#) on your local machine, create a conda env for this lab:

```
conda env create -f jupyter_env.yml  
conda activate getting_started_snowpark_python
```

Conda will automatically install `snowflake-snowpark-python==0.7.0` and all other dependencies for you.

Once Snowpark is installed, create a kernel for Jupyter:

```
python -m ipykernel install --user --name=getting_started_snowpark_python
```

Now, launch Jupyter Notebook on your local machine:

```
jupyter notebook
```

Open up the [config.py](#) file in Jupyter and modify with your account, username, and password information:

If your Snowflake URL is <https://mycompany-sandpit.snowflakecomputing.com/console/login#/>, your username is `THOMAS`, and the role you will use is `ACCOUNTADMIN`, then you would fill it in like so:

```
snowflake_conn_prop = {  
  
    "account": "mycompany-sandpit",  
  
    "user": "THOMAS",  
  
    "password": "YourPasswordHere",  
  
    "role": "ACCOUNTADMIN",  
  
    "database": "snowpark_quickstart",  
  
    "schema": "TELCO",  
  
    "warehouse": "sp_qs_wh",  
  
}
```

Now, you are ready to get started with the notebooks. For each notebook, make sure that you select the `getting_started_snowpark_python` kernel when running. You can do this by navigating to: `Kernel -> Change Kernel` and selecting `getting_started_snowpark_python` after launching each Notebook.

Load Data Using Snowpark Python Client API

Persona: DBA/Platform Administrator/Data Engineer

What You'll Do:

- Establish the Snowpark Python session
- Create the database, schema, and warehouses needed for the remainder of the lab
- Load raw parquet data into Snowflake using Snowpark Python

Open up the [01-Load-Data-with-Snowpark](#) Jupyter notebook and run each of the cells to explore loading and transforming data with Snowpark Python.

Analyze Data Using Snowpark Python Dataframe API

Persona: Data Scientist

What You'll Do:

- Understand and analyze the data set
- Perform data and feature discovery

Open up the [02-Data-Analysis](#) Jupyter notebook and run each of the cells to explore data analysis using Snowpark Python.

Deploy Trained Models using Snowpark Python UDFs

Persona: Data Scientist/ML Engineer

What You'll Do:

- Perform feature selection, model training, and model selection
- Deploy the model into Snowflake for inference using Snowpark Python UDFs

Open up the [03-Snowpark-UDF-Deployment](#) Jupyter notebook and run each of the cells to train a model and deploy it for in-Snowflake inference using Snowpark Python UDFs

Conclusion

Through this lab we were able to experience how Snowpark for Python enables you to use familiar syntax and constructs to process data where it lives with Snowflake's elastic, scalable and secure engine, accelerating the path to production for data pipelines and ML workflows. Here's what you were able to complete:

- Load data into a Snowpark DataFrame
- Clean and prepare data using the Snowpark DataFrame API
- Deploy a trained machine learning model in Snowflake as Python UDF

Getting Started with Snowpark and the Dataframe API

Overview



This project will demonstrate how to get started with Jupyter Notebooks on [Snowpark], a new product feature announced by Snowflake for [public preview] during the 2021 Snowflake Summit. With this tutorial you will learn how to tackle real world business problems as straightforward as ELT processing but also as diverse as math with rational numbers with unbounded precision, sentiment analysis and machine learning.

Snowpark not only works with Jupyter Notebooks but with a variety of IDEs. Instructions on how to set up your favorite development environment can be found in the Snowpark documentation under [Setting Up Your Development Environment for Snowpark].

Snowpark

Snowpark is a new developer framework of Snowflake. It brings deeply integrated, DataFrame-style programming to the languages developers like to use, and functions to help you expand more data use cases easily, all executed inside of Snowflake. Snowpark support starts with Scala API, Java UDFs, and External Functions.

With Snowpark, developers can program using a familiar construct like the DataFrame, and bring in complex transformation logic through UDFs, and then execute directly against Snowflake's processing engine, leveraging all of its performance and scalability characteristics in the Data Cloud.

Snowpark provides several benefits over how developers have designed and coded data driven solutions in the past:

1. Simplifies architecture and data pipelines by bringing different data users to the same data platform, and process against the same data without moving it around.

2. Accelerates data pipeline workloads by executing with performance, reliability, and scalability with Snowflake's elastic performance engine.
3. Eliminates maintenance and overhead with managed services and near-zero maintenance.
4. Creates a single governance framework and a single set of policies to maintain by using a single platform.
5. Provides a highly secure environment with administrators having full control over which libraries are allowed to execute inside the Java/Scala runtimes for Snowpark.

The following tutorial highlights these benefits and lets you experience Snowpark in your environment.

Prerequisites:

This [repo](#) is structured in multiple parts. Each part has a notebook with specific focus areas. All notebooks in this series require a Jupyter Notebook environment with a Scala kernel.

All notebooks will be fully self contained, meaning that all you need for processing and analyzing datasets is a Snowflake account. If you do not have a Snowflake account, you can sign up for a [free trial](#). It doesn't even require a credit card.

- Use of the [Snowflake free 30-day trial environment](#)
- All notebooks in this series require a Jupyter Notebook environment with a Scala kernel.
 - If you do not already have access to that type of environment, Follow the instructions below to either run Jupyter locally or in the AWS cloud.
- Instructions on how to set up your favorite development environment can be found in the Snowpark documentation under [Setting Up Your Development Environment for Snowpark].

What you will learn

- [Part 1](#)

The first notebook in this series provides a quick-start guide and an introduction to the Snowpark DataFrame API. The notebook explains the steps for setting up the environment (REPL), and how to resolve dependencies to Snowpark. After a simple "Hello World" example you will learn about the Snowflake DataFrame API, projections, filters, and joins.

- [Part 2](#)

The second notebook in the series builds on the quick-start of the first part. Using the TPCH dataset in the sample database, it shows how to use aggregations and pivot functions in the Snowpark DataFrame API. Then it introduces UDFs and how to build a stand-alone UDF: a UDF that only uses standard primitives. From there, we will learn how to use third party Scala libraries to perform much more complex tasks like math for numbers with unbounded (unlimited number of significant digits) precision and how to perform sentiment analysis on an arbitrary string.

- [Part 3](#)

The third notebook combines what you learned in part 1 and 2. It implements an end-to-end ML use case including data ingestion, ETL/ELT transformations, model training, model scoring, and result visualization.

Preparing your lab environment

Option 1: Running Jupyter locally

1. Clone the GitHub Lab [repo](#):

```
git clone https://github.com/fenago/snowpark_on_jupyter.git
```

2. Starting your Local Jupyter environment

```
`jupyter lab`
```

The output should be similar to the following

```
To access the server, open this file in a browser:  
file:///home/jovyan/.local/share/jupyter/runtime/jpserver-15-open.html  
Or copy and paste one of these URLs:  
http://162e383e431c:8888/lab?  
token=bdae06c9944057a86f9d8a823cebad4ce66799d855be5d  
http://127.0.0.1:8888/lab?  
token=bdae06c9944057a86f9d8a823cebad4ce66799d855be5d
```

8. Start a browser session (Chrome, ...). Paste the line with the local host address (127.0.0.1) printed in **your shell window** into the browser status bar and update the port (8888) to **your port** in case you have changed the port in the step above.

Positive : Once you have completed this step, you can move on to the Setup Credentials Section.

Setup Credentials

First, we have to set up the Jupyter environment for our notebook. The full instructions for setting up the environment are in the Snowpark documentation [Configure Jupyter].

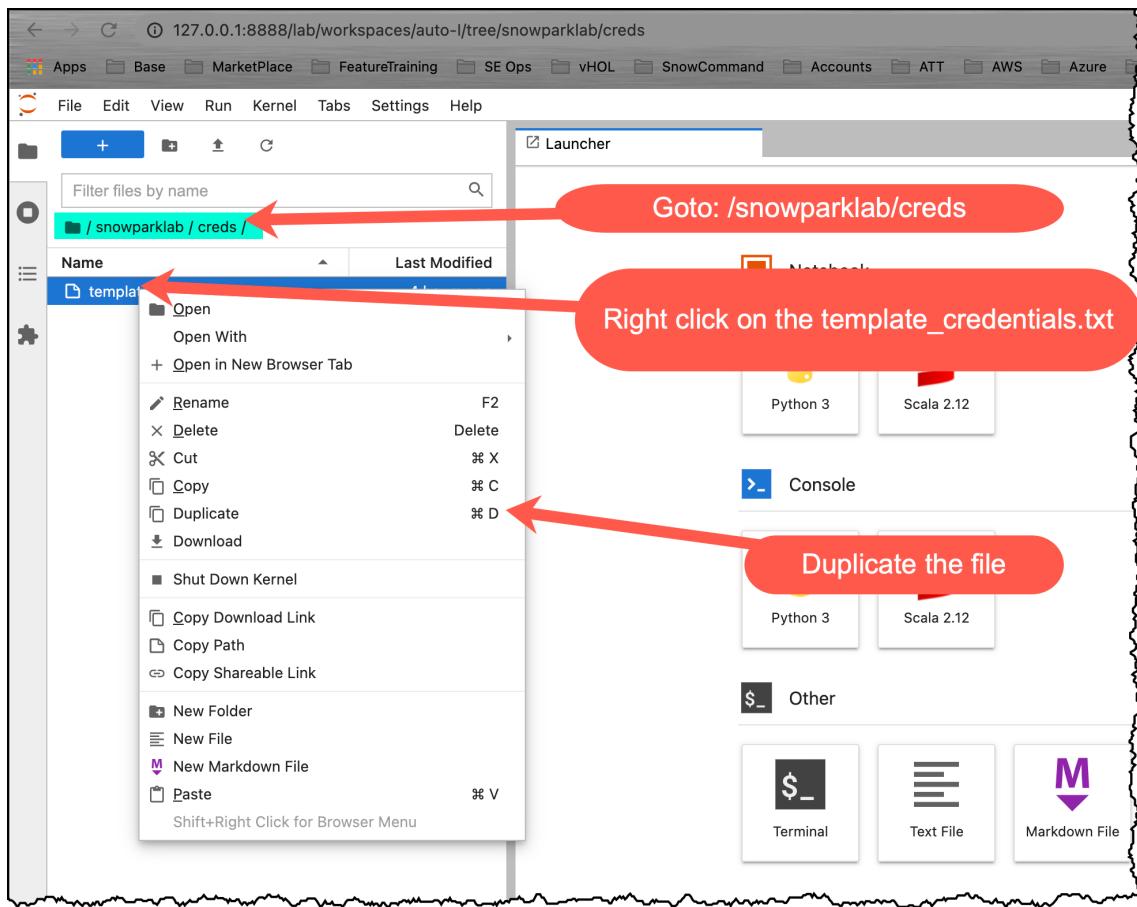
Setup your credentials file

To create a Snowflake session, we need to authenticate to the Snowflake instance. Though it might be tempting to just override the authentication variables with hard coded values in your Jupyter notebook code, it's not considered best practice to do so. If you share your version of the notebook, you might disclose your credentials by mistake to the recipient. Even worse, if you upload your notebook to a public code repository, you might advertise your credentials to the whole world. To prevent that, you should keep your credentials in an external file (like we are doing here).

Then, update your credentials in that file and they will be saved on your local machine. Even better would be to switch from user/password authentication to [private key authentication].

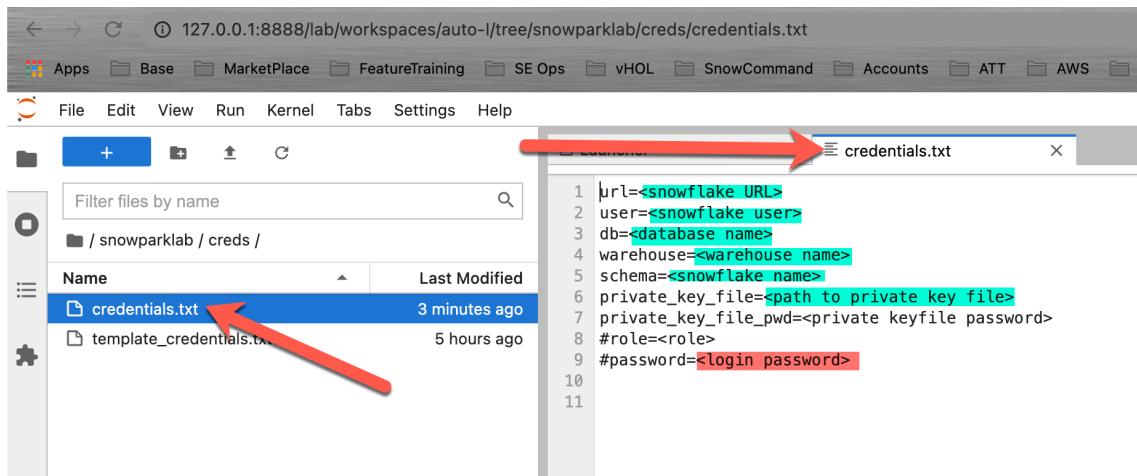
Positive : Put your key pair files into the same directory or update the location in your credentials file.

- Open your Jupyter environment in your web browser
- Navigate to the folder: /snowparklab/creds
- Duplicate the file **template_credentials.txt**



- Rename the duplicated file to **credentials.txt** (right click menu)
- Double click the **credential.txt** file to open and edit it in the Jupyter environment
- Update the file to your Snowflake environment connection parameters

Positive : You can comment out parameters by putting a # at the beginning of the line.

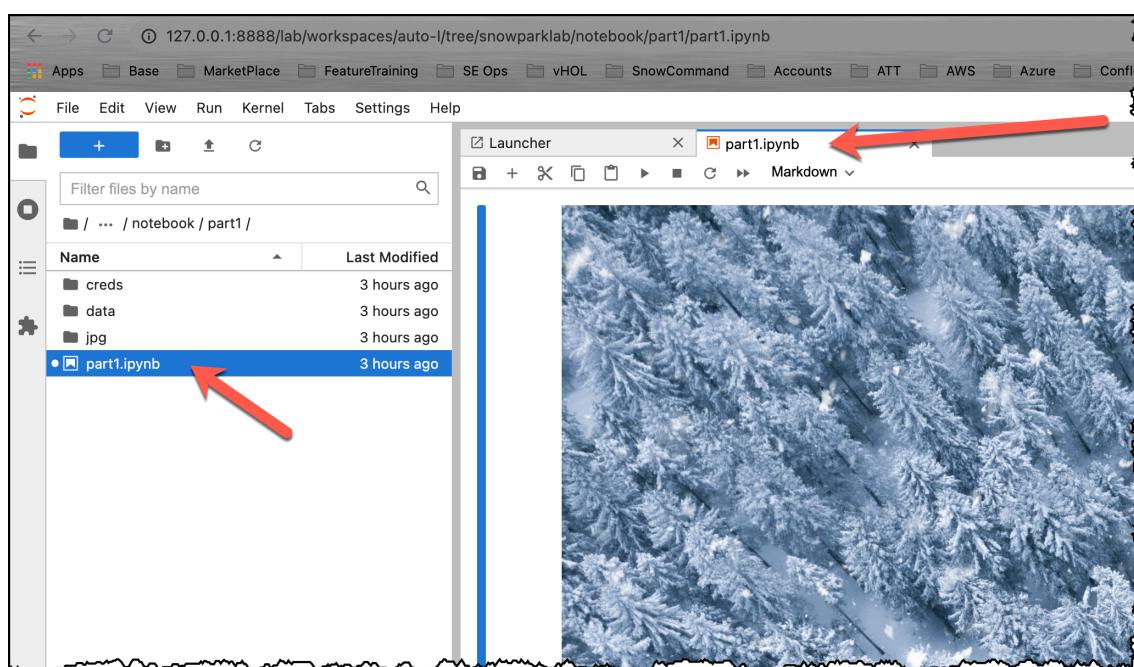
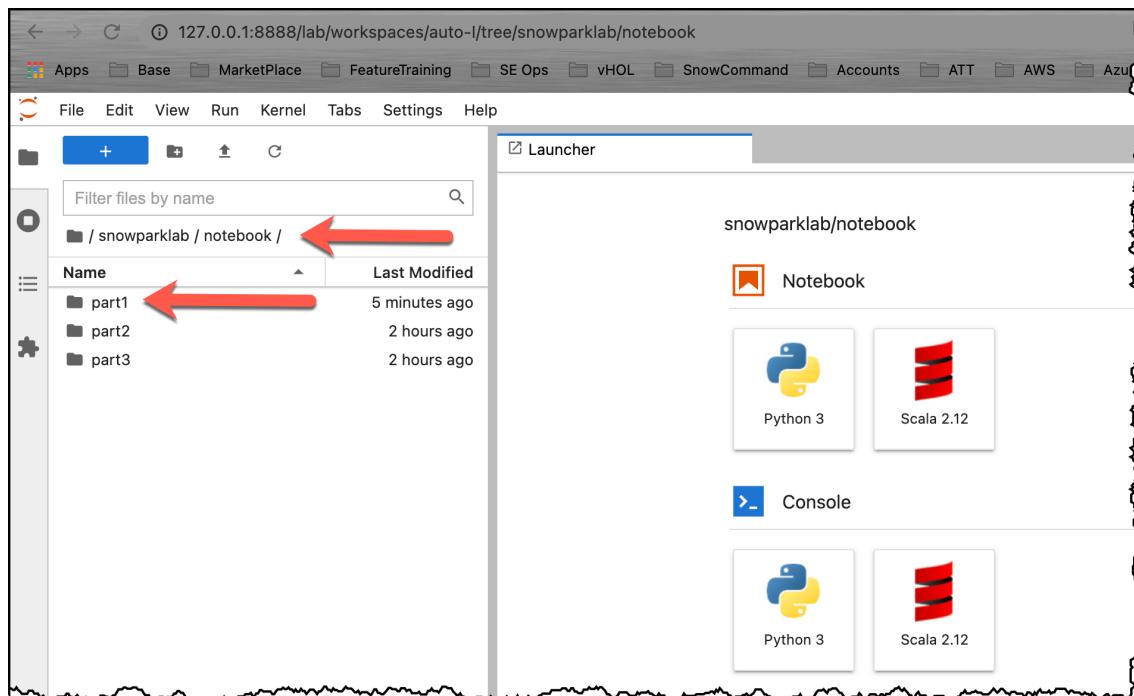


Part 1: Introduction to the Snowpark DataFrame API

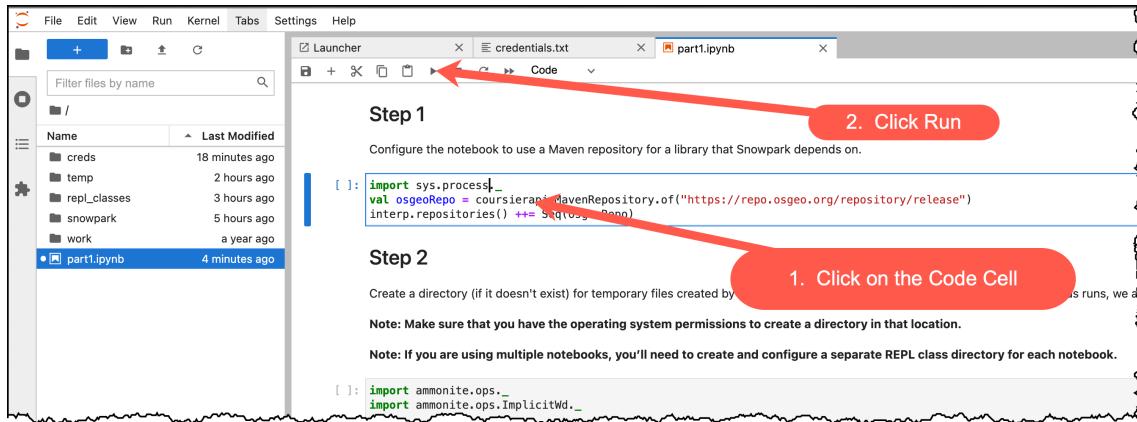
This is the first notebook of a series to show how to use Snowpark on Snowflake. This notebook provides a quick-start guide and an introduction to the Snowpark DataFrame API. The notebook explains the steps for setting up the environment (REPL), and how to resolve dependencies to Snowpark. After a simple "Hello World" example you will learn about the Snowflake DataFrame API, projections, filters, and joins.

Open the Part 1 notebook

Open your Jupyter environment. Navigate to the folder snowparklab/notebook/part1 and Double click on the part1.ipynb to open it



Now read through and run each step in the notebook.



Positive : **Return here once you have finished the first notebook.**

Part 2: Aggregations, Pivots, and UDF's

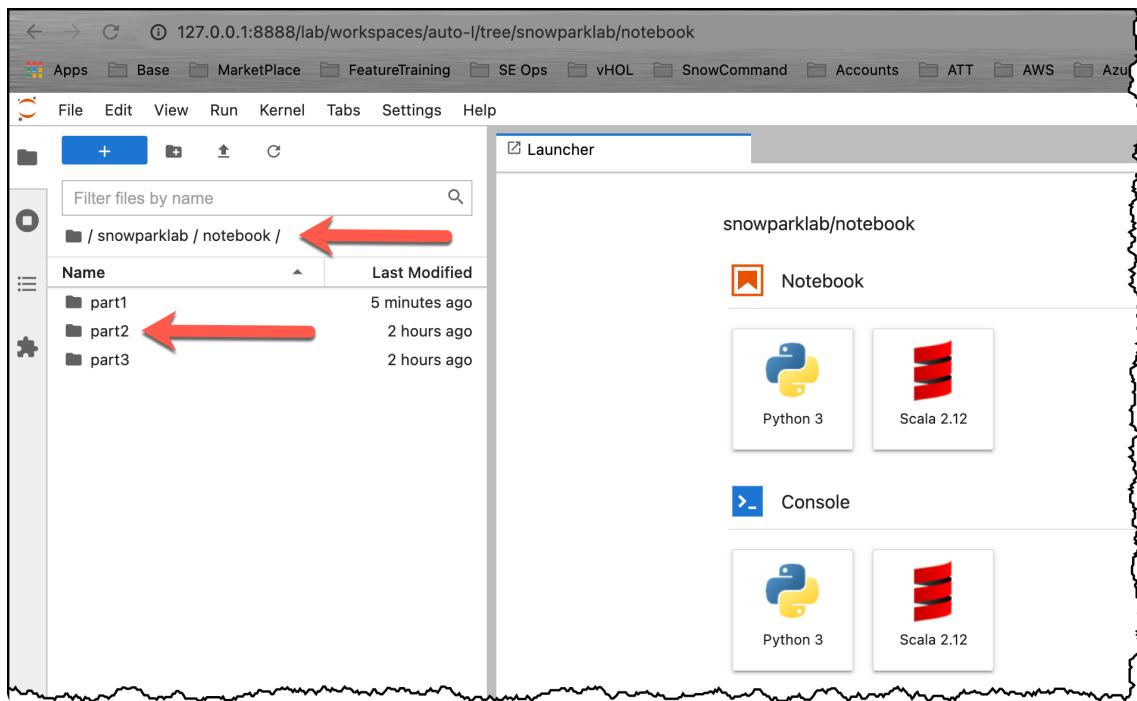
This is the second notebook in the series. It builds on the quick-start of the first part. Using the TPCH dataset in the sample database, we will learn how to use aggregations and pivot functions in the Snowpark DataFrame API. Then, it introduces user defined functions (UDFs) and how to build a stand-alone UDF: a UDF that only uses standard primitives. From there, we will learn how to use third party Scala libraries to perform much more complex tasks like math for numbers with unbounded (unlimited number of significant digits) precision and how to perform sentiment analysis on an arbitrary string.

In this session, the focus will be on:

- Advanced API features and visualization
- User-defined functions

Open the Part 2 notebook

Open your Jupyter environment. Navigate to the folder snowparklab/notebook/part2 and Double click on the part2.ipynb to open it



A screenshot of a Jupyter Notebook interface. The URL in the address bar is '127.0.0.1:8888/lab/tree/snowparklab/notebook/part2/part2.ipynb'. On the left, a file tree shows a folder structure: '... / notebook / part2 /'. A red arrow points to the 'part2.ipynb' file, which is highlighted with a blue selection bar. The right side of the interface shows a preview of the notebook content, featuring a snowy landscape image. Below the preview, there is descriptive text and a bulleted list of prerequisites.

This is the second notebook in the series. It builds on the quick-start of the first one, but only uses standard primitives. From there, we will learn how to use third party libraries.

In this session, the focus will be on:

- Advanced API features and visualization
- User-defined functions

Prerequisites

Now read through and run each step in the notebook.

Step 1

```
[ ]: import sys.process._  
val osgorepo = coursierapi.MavenRepository.of("https://repo.osgeo.org/repository/release")  
interp.repositories() += Seq(osgorepo)
```

Step 2

```
[ ]: import ammonite.ops._  
import ammonite.ons.TmpImplicitWd.
```

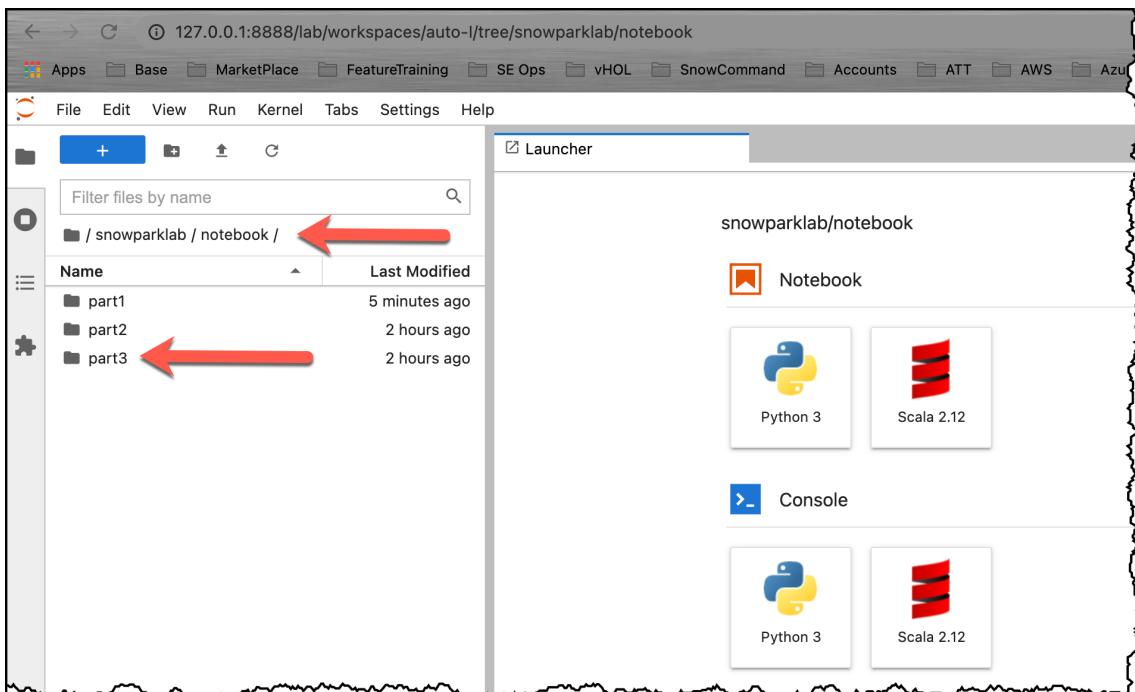
Positive : **Return here once you have finished the second notebook.**

Part 3: Data Ingestion, transformation, and model training

The third notebook builds on what you learned in part 1 and 2. It implements an end-to-end ML use-case including data ingestion, ETL/ELT transformations, model training, model scoring, and result visualization.

Open the Part 3 notebook

Open your Jupyter environment. Navigate to the folder snowparklab/notebook/part2 and Double click on the part2.ipynb to open it



The screenshot shows a Jupyter Notebook interface with the URL `127.0.0.1:8888/lab/tree/snowparklab/notebook/part3/part3.ipynb`. The left sidebar lists files: creds, data, jpg, and part3.ipynb. The right panel shows the notebook content with a title 'Prerequisites' and a section 'Quick Start'. A red arrow points to the tab bar at the top, which displays 'part3.ipynb'.

Now read through and run each step in the notebook.

The screenshot shows the same Jupyter Notebook interface. The 'Step 1' section is visible with its configuration text and code cell. A red circle labeled '2' is on the tab bar, and another red circle labeled '1' is on the first code cell, highlighting the line `val osgeoRepo = coursierapi.MavenRepository.of("https://repo.osgeo.org/repository/release")`.

Positive : Return here once you have finished the third notebook so you can read the conclusion & Next steps, and complete the guide.

Conclusion & Next Steps

Snowpark is a brand new developer experience that brings scalable data processing to the Data Cloud. In Part1 of this series, we learned how to set up a Jupyter Notebook and configure it to use Snowpark to connect to the Data Cloud. Next, we built a simple Hello World! program to test connectivity using embedded SQL. Then we enhanced that program by introducing the Snowpark Dataframe API. Lastly we explored the power of the Snowpark Dataframe API using filter, projection, and join transformations.

We encourage you to continue with your free trial by loading your own sample or production data and by using some of the more advanced capabilities of Snowflake not covered in this lab.

What we've covered:

- Quick Start: Set up the environment
- Hello World: First steps
- Snowflake DataFrame API: Query the Snowflake Sample Datasets via Snowflake DataFrames
- Aggregations, Pivots, and UDF's using the Snowpark API
- Data Ingestion, transformation, and model training

Lab 12: Defining Virtual Warehouses

One of the first things you'll do when setting up your Snowflake environment is to establish your virtual warehouses. What this boils down to, essentially, is how much compute you want to use and how segregated that compute will be.

What Is a Virtual Warehouse?

Warehouses are required for queries, as well as for operations like loading data into tables. They come in a variety of T-shirt sizes ranging from extra small to 4XL, with each representing an increase in the hardware resources available for you to use.

Warehouses can be started and stopped at any time. They can also be resized at any time, even while running, to accommodate the need for more or fewer compute resources, depending on the type of operations you are performing at that time.

If you are on the Enterprise or higher edition of Snowflake, the warehouses can also be configured to scale outwards by adding more servers to a cluster. Doing so provides a boost in parallel processing capabilities, which is useful when you have a surge in concurrent activity.

A warehouse can be created using the web interface by any Snowflake user with the appropriate rights. Simply select Warehouses from the menu ribbon and then click on the Create button. You'll see a dialog like this where you can define the name, size and other attributes about the warehouse:

Create Warehouse

The screenshot shows the 'Create Warehouse' dialog box. It contains fields for Name (my_first_warehouse), Size (Medium (4 credits / hour)), Maximum Clusters (2), Minimum Clusters (1), Scaling Policy (Standard), Auto Suspend (10 minutes), and Auto Resume (checked). There is also a Comment field and a 'Show SQL' link at the bottom. Buttons for 'Cancel' and 'Finish' are at the bottom right.

Name *	my_first_warehouse
Size	Medium (4 credits / hour)
Learn more about virtual warehouse sizes here	
Maximum Clusters	2
Multi-cluster warehouses improve the query throughput for high concurrency workloads.	
Minimum Clusters	1
The number of active clusters will vary between the specified minimum and maximum values, based on number of concurrent users/queries.	
Scaling Policy	Standard
The policy used to automatically start up and shut down clusters.	
Auto Suspend	10 minutes
The maximum idle time before the warehouse will be automatically suspended.	
<input checked="" type="checkbox"/> Auto Resume ?	
Comment	
Show SQL	Cancel Finish

You can also script a warehouse. You can use the **Show SQL** link on the above dialog to give you a headstart, as it will translate the information you enter and the selections you make into the SQL needed to create that warehouse:

SQL

```
1 CREATE WAREHOUSE my_first_warehouse WITH WAREHOUSE_SIZE = 'MEDIUM'  
WAREHOUSE_TYPE = 'STANDARD' AUTO_SUSPEND = 600 AUTO_RESUME = TRUE  
MIN_CLUSTER_COUNT = 1 MAX_CLUSTER_COUNT = 2 SCALING_POLICY = 'STANDARD';
```

Select SQL

Close

But how many warehouses do I need? And when do you know to scale up vs. scale out a warehouse?

How Many Warehouses Do I Need?

Well, the good news is that creating warehouses doesn't cost you anything, so you can have as many as you need. I tend to say there are four different types of warehouses, and it's worth considering whether your organisation will need some, or all, of these. You can optionally combine types here, but as you'll see later in this article, they don't necessarily share the same setup, and if you do combine, you don't fully optimise. Below are the types of warehouses available:

- **Staging:** Used by any process that brings information into Snowflake
- **Integration:** Used when you want to transform and integrate information within Snowflake
- **Consumption:** Used by any process that extracts or uses information hosted within Snowflake; for example, Tableau connecting and gathering information for use by a dashboard
- **Labs:** Optional compute reserved exclusively for high-intensity requests from Data Scientists and Data Citizens within your organisation

How many of each warehouse type you need is influenced by several contributing factors:

- **Isolation of Workload:** If you want to load a lot of information into Snowflake from various sources in parallel---as might be the case with a nightly load---or maybe you want to reserve or secure compute for specific users, it can be useful to define multiple warehouses of either a staging or consumption type.
- **Usage Chargeback:** If your data analytics delivery model relies on funding from different departments, then creating separate consumption and/or lab warehouses for each consumer group can help you to track and chargeback as applicable.

If you're wondering what makes these warehouses different, let me elaborate.

Staging

A staging warehouse switches on for just long enough to stage information from a source:

```
CREATE OR REPLACE WAREHOUSE "STG_XXXX"  
WITH WAREHOUSE_SIZE = 'X-SMALL'  
AUTO_SUSPEND = 60  
AUTO_RESUME = TRUE  
MIN_CLUSTER_COUNT = 1  
MAX_CLUSTER_COUNT = 1  
SCALING_POLICY = 'STANDARD'  
INITIALLY_SUSPENDED = TRUE  
COMMENT = 'For the staging of information from XXXX source';
```

- Start with the smallest warehouse and scale up as needed to load information in a timely manner.

- Experiment with scaling outward to improve performance if you are on Enterprise edition and already beyond an XS T-shirt size.
- You might set up access rights for this type of warehouse to be sized up or down by your ELT platform as it needs the extra compute.

Integration

An integration warehouse stays on for longer and generally has a lot to do when transforming and integrating information:

```
CREATE OR REPLACE WAREHOUSE "INT_General"
  WITH WAREHOUSE_SIZE = 'SMALL'
    AUTO_SUSPEND = 300
    AUTO_RESUME = TRUE
    MIN_CLUSTER_COUNT = 1
    MAX_CLUSTER_COUNT = 2
    SCALING_POLICY = 'STANDARD'
    INITIALLY_SUSPENDED = TRUE
    COMMENT = 'For all data integration within the EDW';
```

- Start with a small warehouse and scale up as needed to integrate information in a timely manner.
- Review your cluster counts as you increase warehouse size, but consciously increment the maximum number of servers in the cluster to reach optimal processing capabilities. Don't just set it to the maximum available for that T-shirt size.
- If you see the warehouse suspending routinely during the integration process, increase the auto-suspend timeout.

You could combine staging and integration warehouse types and have just one type here; a lot of people do. I always consider the separation when those influencing factors I mentioned above come into play or where I may need to separate resources for data security purposes.

Consumption

Consumption warehouses are split either by function or department---whichever is a natural and required level of separation within your business. They should stay on for longer and only scale when resources are in high demand:

```
CREATE OR REPLACE WAREHOUSE "OUT_XXXX"
  WITH WAREHOUSE_SIZE = 'X-SMALL'
    AUTO_SUSPEND = 600
    AUTO_RESUME = TRUE
    MIN_CLUSTER_COUNT = 1
    MAX_CLUSTER_COUNT = 1
    SCALING_POLICY = 'ECONOMY'
    INITIALLY_SUSPENDED = TRUE
    COMMENT = 'VW for all data analytics delivered by XXXX';
```

- Start with the smallest warehouse and scale up by benchmarking performance from the BI tool or the process that is extracting information.
- Increase the auto-suspend timeout if the BI tool (or process) makes infrequent requests. This will maximise cache usage and result in a faster return of results for subsequent requests for similar information.

Labs

Labs are your secret weapon when a group wants dedicated resources and may put extraordinary load on Snowflake (which it can take), but maybe you want to keep a watchful eye on their usage.

The definition of the warehouse really depends upon their needs, but I also establish a Resource Monitor to monitor their consumption for me:

```
CREATE RESOURCE MONITOR "RM_LAB_OPERATIONS"
WITH CREDIT_QUOTA = 30, frequency = 'MONTHLY'
TRIGGERS
ON 80 PERCENT DO NOTIFY
ON 120 PERCENT DO SUSPEND;

ALTER WAREHOUSE "LAB_OPERATIONS" SET RESOURCE_MONITOR = "RM_LAB_OPERATIONS";
```

You must be an account admin to establish these resource monitors, but doing so gives you an early indication of excessive usage. They can be set up to inform you and the users that they are approaching or exceeding the assigned quota. The monitor can be configured to solely notify, or it can take steps to restrict usage.

Tips for Naming Your Warehouses

When it comes to naming your warehouses, I always recommend these good habits:

- Distinguish which type of warehouse it is. I use the prefixes STG, INT, OUT and LAB.
- Keep your names short. When you select a warehouse in the context menu of a worksheet, you can only see the first 14 characters, so by keeping it short, you'll be able to see the full name.
- Don't include the size of the warehouse in the name. As we've already said, a warehouse can be scaled up or down at any time, and the interface tells you in other ways what size it is presently.
- Include a more verbose description of the warehouse purpose in the comments attribute. Hovering over a warehouse at different places in the web interface often pops up the comments in the accompanying tooltip, helping those with access to multiple warehouses decide on the most appropriate to use.

Lab 13: Table Types in Snowflake

Tables are database objects logically structured as a collection of rows and columns. All data in Snowflake is stored in database tables. Apart from standard database tables, Snowflake supports other table types that are especially useful for storing data that does not need to be maintained for extended periods of time.

When data is loaded into Snowflake, Snowflake reorganizes that data into its internal optimized, compressed, columnar format. Snowflake stores this optimized data in cloud storage.

Snowflake supports different types of tables

- Permanent Table
- Transient Table
- Temporary Table
- External Table

Permanent Table:

These are the standard, regular database tables. Permanent tables are the default table type in Snowflake and do not need any additional syntax while creating to make them permanent.

```
create database TableTypes;
create schema Demo_TableTypes;
-- Permanent Table
create table employee (empid number, name varchar(50));
```

The screenshot shows the Snowflake UI interface. At the top, there is a toolbar with a 'Run' button and a link to 'All Queries | Saved 31 seconds ago'. Below the toolbar, the SQL query is displayed in a code editor:

```
1 create database TableTypes;
2 |
3 create schema Demo_TableTypes;
4
5 -- Permanent Table
6
7 create table employee (empid number, name varchar(50));
```

Below the code editor, there are two tabs: 'Results' (which is selected) and 'Data Preview'. The 'Results' tab shows the execution status: a green checkmark, 'Query_ID', 'SQL', '171ms', and '1 rows'. There is also a progress bar indicating the execution time. A 'Filter result...' input field, a download icon, and a 'Copy' button are located below the status bar. The results table itself has a header row 'Row' and 'status', and one data row: '1 Table EMPLOYEE successfully created.'

The data stored in permanent tables consumes space and contributes to the storage charges that Snowflake bills your account. It also comes with additional features like Time-Travel and Fail-Safe which helps in data availability and recovery.

Transient Table:

Transient tables in Snowflake are similar to permanent tables except that they do not have a Fail-safe period and only have a very limited Time-Travel period. These are best suited in scenarios where the data in your table is not

critical and can be recovered from external means if required.

Transient tables, like permanent tables, contribute to your account's overall storage expenses. However, since Transient Tables do not use Fail-safe, there are no Fail-safe costs (i.e. the costs associated with maintaining the data required for Fail-safe disaster recovery).

To create a Transient table in Snowflake, You need to mention **transient** in the create table syntax.

```
create transient table student (regno number, studentname varchar(50));
```

The screenshot shows the Snowflake UI interface. In the code editor, lines 8 through 12 are visible, defining a transient table named 'student' with columns 'regno' and 'studentname'. Below the code, the 'Results' tab is selected, showing a success message: 'Table STUDENT successfully created.' The results table has two columns: 'Row' and 'status'.

Row	status
1	Table STUDENT successfully created.

Temporary Table:

Snowflake supports creating temporary tables to store transient, non-permanent data. Temporary tables exist only within the session. They are created and persist only for the session remainder. They are not visible to other sessions or users and don't support standard features like cloning.

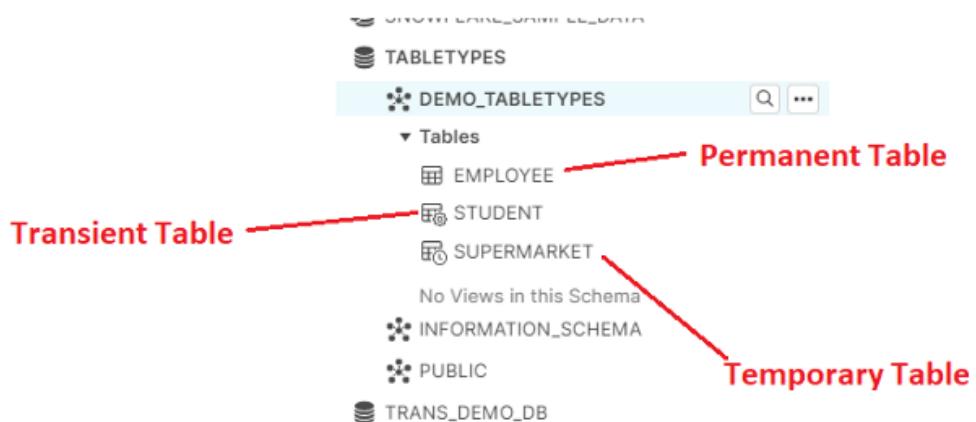
Therefore the data stored in the system is cleaned entirely and is not recoverable either by the user-created table or Snowflake.

```
-- Temporary Table  
create temporary table Supermarket (Pono number, BatchName varchar(50));
```

The screenshot shows the Snowflake UI interface. In the code editor, lines 13 through 15 are visible, defining a temporary table named 'Supermarket' with columns 'Pono' and 'BatchName'. Below the code, the 'Results' tab is selected, showing a success message: 'Table SUPERMARKET successfully created.' The results table has two columns: 'Row' and 'status'.

Row	status
1	Table SUPERMARKET successfully created.

PFB the icon displayed for **different tables**,



Though Temporary tables are dropped at the end of the session, Snowflake recommends explicitly dropping these tables once they are no longer needed to prevent any unexpected storage changes when working with large temporary tables.

If logout the particular session and re login then particular temporary table will not be displayed.

The screenshot shows the Snowflake UI with the following structure:

- Find database objects**
- Starting with...**
- Tables**
 - EMPLOYEE
 - STUDENT
- No Views in this Schema
- INFORMATION_SCHEMA
- PUBLIC
- TRANS_DEMO_DB

As shown in **Figure 1.1**, the temporary table (SUPERMARKET) has been dropped after relogging in.

Next, we will cover how to differentiate Temporary Tables from Non-Temporary Tables.

Types of Table in Snowflake

In this section, we see about **How Snowflake behaves when we create temporary table that has the same name as an existing table in the same schema?**

Before answering this question let's see how to find type of table present in database.

SHOW TABLES --- Lists the tables for which you have access privileges, including dropped tables that are still within the Time Travel retention period and, therefore, can be undropped.

Syntax:

```
SHOW TABLES;
```

Example:

```
SHOW TABLES;
```

Output:

Name	created_on	name	database_name	schema_name	kind	comment	created_by	rows	bytes	owner	retention_days	automatic_clustering	change_tracking	search_optimization	user_credential_id	search_optimization_external
1	2023-09-09 23:1	COMPANY	TABLETYPES	DEMO_TABLETYPE	TABLE			3	1536	ACCOUNTADMIN	1	OFF	OFF	OFF	NULL	N
2	2023-09-23 09:0	EMPLOYEE	TABLETYPES	DEMO_TABLETYPE	TABLE			6	1024	ACCOUNTADMIN	1	OFF	OFF	OFF	NULL	N
3	2023-09-31 09:3	DEMO_TABLE	TABLETYPES	DEMO_TABLETYPE	TABLE			NULL	0	ACCOUNTADMIN	1	OFF	OFF	OFF	NULL	Y
4	2023-09-31 09:4	DEMO_TABLE2	TABLETYPES	DEMO_TABLETYPE	TABLE			NULL	0	ACCOUNTADMIN	1	OFF	OFF	OFF	NULL	Y
5	2023-09-23 09:5	STUDENT	TABLETYPES	DEMO_TABLETYPE	TRANSIENT			6	1024	ACCOUNTADMIN	1	OFF	OFF	OFF	NULL	N

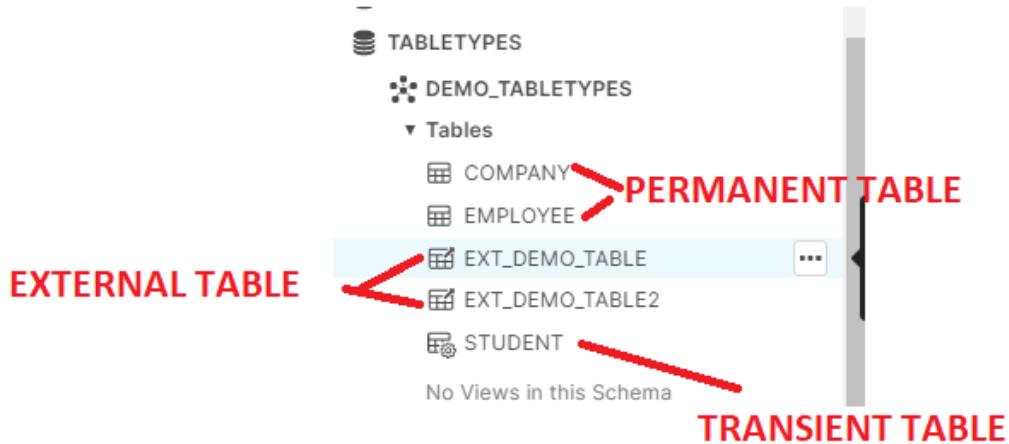
Scenario 1:

When you create a temporary table that has the same name as an existing table in the same schema ?

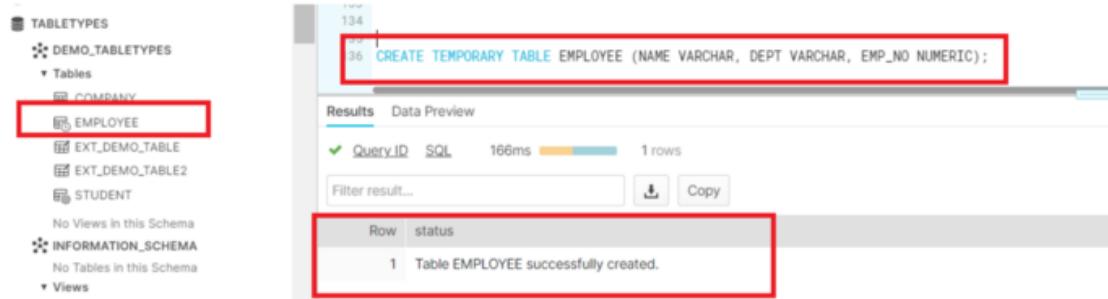
Snowflake supports creating a temporary table that has the same name as an existing permanent/transient table in the same schema. However, note that the temporary table takes precedence in the session over any other table with the same name in the same schema.

Temporary table take precedence and hides the existing non-temporary table.

Screenshot 1: Before creating temporary table in my database.



Screenshot 2: After creating temporary table in my database,



In above screenshot Temporary table take precedence and hides the existing permanent table.

Scenario 2:

When you create a table that has the same name as an existing temporary table in the same schema ?

The newly-created table is hidden by the temporary table.

Screenshot 1: Before creating permanent table same name like temporary table name in my database.

```

115
116 CREATE TEMPORARY TABLE ACCOUNT (NAME VARCHAR, ACCOUNT_NUMBER NUMERIC, BANK_NAME VARCHAR);
117
118

```

Results Data Preview

Query ID SQL 544ms 1 rows

Filter result... Row status

1 Table ACCOUNT successfully created.

TEMPORARY TABLE

PERMANENT TABLE

EXTERNAL TABLE

TRANSIENT TABLE

Screenshot 2: After creating permanent table (Account) same name like temporary table (Account) name in my database.

```

118
119 CREATE TABLE ACCOUNT (NAME VARCHAR, ACCOUNT_NUMBER NUMERIC, BANK_NAME VARCHAR, BALANCE NUMERIC);
120
121

```

Results Data Preview

Query ID SQL 133ms 1 rows

Filter result... Row status

1 Table ACCOUNT successfully created.

In above screenshot Temporary table take precedence and hides the newly created permanent table.

All queries and other operations performed in the session on the table effect only the temporary table.

Comparison of Snowflake Table Types

The below table summarizes the differences between the three table types, particularly with regard to their impact on Time Travel and Fail-safe:

Table Type	Availability	Time-Travel Retention period in days	Fail-Safe period in days
Temporary	Remainder of session	0 or 1 (Default is 1)	0
Transient	Until explicitly dropped	0 or 1 (Default is 1)	0
Permanent (Standard Edition)	Until explicitly dropped	0 or 1 (Default is 1)	7
Permanent (Enterprise and higher Edition)	Until explicitly dropped	0 to 90 (default is configurable)	7

Lab 14: Loading JSON data into Snowflake

Have you ever faced any use case or scenario where you've to **load JSON data into the Snowflake**? We better know JSON data is one of the common data format to store and exchange information between systems. JSON is a relatively concise format. If we are implementing a database solution, it is very common that we will come across a system that provides data in JSON format. Snowflake has a very straight forward approach to load JSON data. In this blog, we will understand this approach in a step-wise manner.

1. Stage the JSON data

The JSON data looks like:

```
{  
  "Name": "Aman Gupta",  
  "family_detail": [  
    {  
      "Name": "Avinash Gupta",  
      "Relationship": "Father",  
    },  
    {  
      "Name": "Lata Gupta",  
      "Relationship": "Mother",  
    },  
    {  
      "Name": "Shrishti Gupta",  
      "Relationship": "Sister",  
    },  
    {  
      "Name": "Bobin Gupta",  
      "Relationship": "Brother",  
    }  
}
```

Create `family.json` file with above content and save it. Also, update file path in below command.

In snowflake Staging the data means, make the data available in Snowflake **stage**(intermediate storage) it can be **internal or external**. Staging JSON data in Snowflake is similar to staging any other files. Let's Staging JSON data file from a **local file system**.

```
CREATE OR REPLACE STAGE my_json_stage file_format = (type = json);  
PUT file:///home/knoldus/Desktop/family.json @my_json_stage;
```

```
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>CREATE OR REPLACE STAGE my_json_stage file_format = (type = json);
+-----+
| status
|-----|
| Stage area MY_JSON_STAGE successfully created.
|-----+
1 Row(s) produced. Time Elapsed: 0.236s
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>PUT file:///home/knoldus/Desktop/family.json @my_json_stage;
family.json_c.gz(0.00MB): [#####] 100.00% Done (0.297s, 0.00MB/s).
+-----+-----+-----+-----+-----+-----+-----+-----+
| source | target | source_size | target_size | source_compression | target_compression | status | message |
+-----+-----+-----+-----+-----+-----+-----+-----+
| family.json | family.json.gz | 353 | 166 | NONE | GZIP | UPLOADED | |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

2. Load JSON data as raw into temporary table

To load the JSON data as raw, first, create a table with a column of VARIANT type. VARIANT can contain any type of data so it is suitable for loading JSON data.

```
CREATE TABLE relations_json_raw (
    json_data_raw VARIANT
);
```

Now let's **copy** the JSON file into **relations_json_raw** table.

```
COPY INTO relations_json_raw from @my_json_stage;
```

Note that a file format does not need to be specified because it is included in the stage definition.

```
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>COPY INTO relations_json_raw from @my_json_stage;
+-----+-----+-----+-----+-----+-----+-----+-----+
| file | status | rows_parsed | rows_loaded | error_limit | errors_seen | first_error | first_error_line | first_error_character | first_error_column_name |
|-----+-----+-----+-----+-----+-----+-----+-----+
| my_json_stage/family.json.gz | LOADED | 1 | 1 | 1 | 0 | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

3. Analyze and prepare raw JSON data

The next step would be **to analyze** the **loaded raw JSON data**. Determining what information needs to be extracted from JSON data. For example, in our case, we are interested to extract the name key and from the family_detail array object, we want to extract the name and relationship key from each JSON object. The below query will do that.

```
SELECT
    json_data_raw:Name,
    VALUE:Name::String,
    VALUE:Relationship::String
FROM
    relations_json_raw
    , lateral flatten( input => json_data_raw:family_detail );
```

The above query using **lateral join** and a **flatten function**. The flatten function returns a row for each JSON object from the family_detail array. and the lateral modifier joins the data with any information outside of the object, in our example candidate name that we are extracting with **json_data_raw: Name**.

```
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>SELECT *
  FROM
    relations_json_raw
  , lateral flatten( input => json_data_raw:family_detail );
+-----+-----+-----+-----+-----+-----+
| JSON_DATA_RAW | SEQ | KEY | PATH | INDEX | VALUE |
+-----+-----+-----+-----+-----+-----+
| { "Name": "Aman Gupta", "family_detail": [ { "Name": "Avinash Gupta", "Relationship": "Father" }, { "Name": "Lata Gupta", "Relationship": "Mother" }, { "Name": "Shrishti Gupta", "Relationship": "Sister" }, { "Name": "Bobin Gupta", "Relationship": "Brother" } ] } | 1 | NULL | [0] | 0 | { "Name": "Avinash Gupta", "Relationship": "Father" } |
| { "Name": "Aman Gupta", "family_detail": [ { "Name": "Avinash Gupta", "Relationship": "Father" }, { "Name": "Lata Gupta", "Relationship": "Mother" }, { "Name": "Shrishti Gupta", "Relationship": "Sister" }, { "Name": "Bobin Gupta", "Relationship": "Brother" } ] } | 1 | NULL | [1] | 1 | { "Name": "Lata Gupta", "Relationship": "Mother" } |
+-----+-----+-----+-----+-----+-----+
| THIS |
+-----+
| [ { "Name": "Avinash Gupta", "Relationship": "Father" }, { "Name": "Lata Gupta", "Relationship": "Mother" }, { "Name": "Shrishti Gupta", "Relationship": "Sister" }, { "Name": "Bobin Gupta", "Relationship": "Brother" } ] |
| [ { "Name": "Avinash Gupta", "Relationship": "Father" } ] |

```

```
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>SELECT
  json_data_raw:Name,
  VALUE:Name::String,
  VALUE:Relationship::String
  FROM
    relations_json_raw
  , lateral flatten( input => json_data_raw:family_detail );
+-----+-----+-----+
| JSON_DATA_RAW:NAME | VALUE:NAME::STRING | VALUE:RELATIONSHIP::STRING |
+-----+-----+-----+
| "Aman Gupta" | Avinash Gupta | Father |
| "Aman Gupta" | Lata Gupta | Mother |
| "Aman Gupta" | Shrishti Gupta | Sister |
| "Aman Gupta" | Bobin Gupta | Brother |
+-----+-----+-----+
```

Load Data into target table

Now we have analyzed and extracted information. We can load the extracted data into the target table.

```
CREATE OR REPLACE TABLE candidate_family_detail AS
SELECT
  json_data_raw:Name AS candidate_name,
  VALUE:Name::String AS relation_name,
  VALUE:Relationship::String AS relationship

FROM
  relations_json_raw
, lateral flatten( input => json_data_raw:family_detail );
```

```

kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>CREATE OR REPLACE TABLE candidate_family_detail AS
  SELECT
    json_data_raw:Name AS candidate_name,
    VALUE:Name::String AS relation_name,
    VALUE:Relationship::String AS relationship
  FROM
    relations_json_raw
    , lateral flatten( input => json_data_raw:family_detail);
+-----+
| status
+-----+
| Table CANDIDATE_FAMILY_DETAIL successfully created.
+-----+
1 Row(s) produced. Time Elapsed: 1.798s
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>SELECT * FROM CANDIDATE_FAMILY_DETAIL;
+-----+-----+-----+
| CANDIDATE_NAME | RELATION_NAME | RELATIONSHIP |
+-----+-----+-----+
| "Aman Gupta"  | Avinash Gupta | Father      |
| "Aman Gupta"  | Lata Gupta   | Mother      |
| "Aman Gupta"  | Shrishti Gupta | Sister      |
| "Aman Gupta"  | Bobin Gupta  | Brother     |
+-----+-----+-----+

```

If you don't want to do a "create table as", you can pre-create a table and then insert the JSON data into the table.

```

kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>CREATE TABLE candidate_family_detail (
  candidate_name STRING,
  relation_name STRING,
  relationship STRING
);
+-----+
| status
+-----+
| Table CANDIDATE_FAMILY_DETAIL successfully created.
+-----+
1 Row(s) produced. Time Elapsed: 0.408s

```

```

kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>INSERT INTO candidate_family_detail
  SELECT
    json_data_raw:Name AS candidate_name,
    VALUE:Name::String AS relation_name,
    VALUE:Relationship::String AS relationship
  FROM
    relations_json_raw
    , lateral flatten( input => json_data_raw:family_detail );
+-----+
| number of rows inserted |
+-----+
|        4                 |
+-----+
4 Row(s) produced. Time Elapsed: 2.193s
kundan59#COMPUTE_WH@INGEST_JSON_DATA.PUBLIC>SELECT * FROM candidate_family_detail;
+-----+-----+-----+
| CANDIDATE_NAME | RELATION_NAME | RELATIONSHIP |
+-----+-----+-----+
| Aman Gupta    | Avinash Gupta | Father      |
| Aman Gupta    | Lata Gupta   | Mother      |
| Aman Gupta    | Shrishti Gupta | Sister      |
| Aman Gupta    | Bobin Gupta  | Brother     |
+-----+-----+-----+
4 Row(s) produced. Time Elapsed: 1.244s

```

Getting Started with Snowpipe

Overview

When building data applications, your users count on seeing the latest. Stale data is less actionable and could lead to costly errors. That's why continuously generated data is essential. Snowflake provides a data loading tool to drive updates, ensuring your databases are accurate by updating tables in micro-batches.

Let's look into how Snowpipe can be configured for continual loading. Then, we can review how you can efficiently perform basic management tasks. But first, If you're unfamiliar with Snowflake or loading database objects, check out these resources to get familiar with the topics ahead.

Prerequisites

- AWS S3 Service [Documentation](#)

What You'll Learn

- Snowpipe staging methods
- Configure security access for Snowflake and AWS
- Automate Snowpipe with AWS S3 event notifications
- Manage Snowpipe and remove
- Next steps with database automation

What You'll Need

- Create a [Snowflake account](#) with an **ACCOUNTADMIN** role
- AWS [Account](#) with access to a Snowflake supported [region](#)

What You'll Build

- Automated data loading with Snowpipe between AWS S3 bucket and Snowflake database.

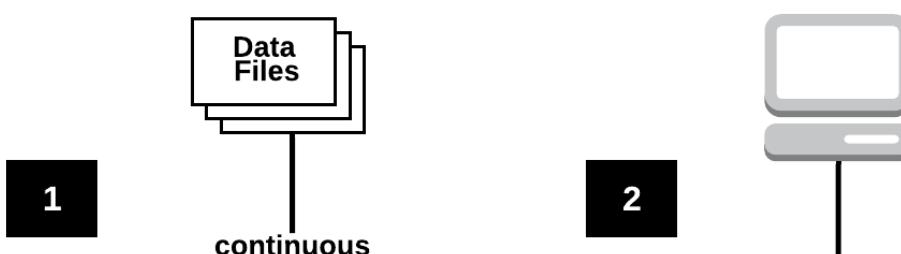
After ensuring the prerequisites detailed in this section, jump into the queueing data integration options with Snowpipe.

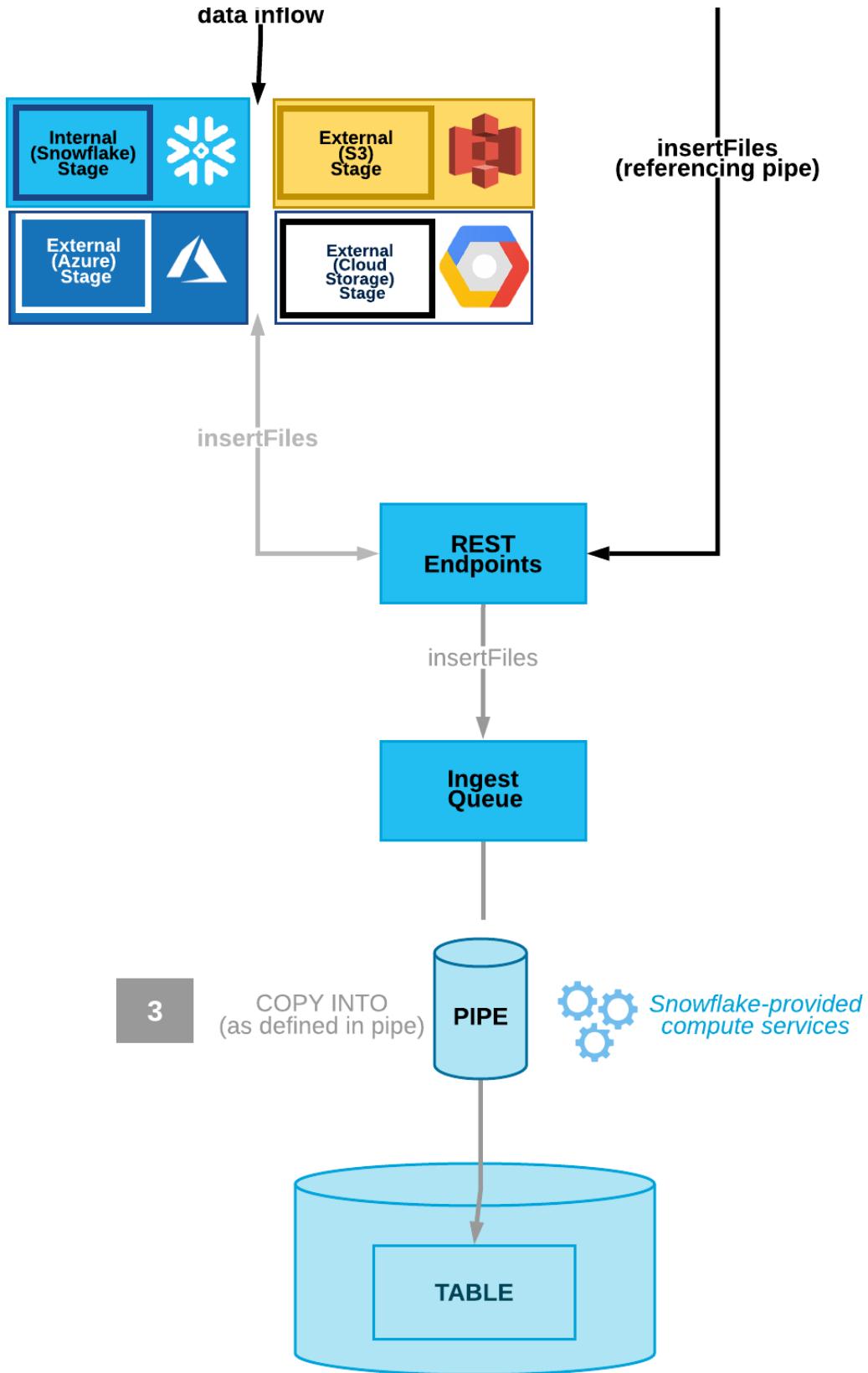
Choose the Data Ingestion Method

Snowpipe is an event based data ingest tool. Snowpipe provides two main methods for triggering a data loading event. This trigger could be a cloud storage notification (i.e. AWS S3 `ObjectCreated` event) or by directly calling the Snowpipe `insertFiles` REST API.

1. Cloud Storage Event Notifications (AWS S3, GCP CS, Azure Blob)
2. Snowpipe's REST API

This tutorial follows option 1, automating continuous data loading with cloud event notifications on AWS S3. In the next section, we'll configure your cloud notification preferences.





The image above reveals the two Snowpipe workflows. Option one shows continuous data loading with cloud storage event notifications. Option two illustrates queueing data with a REST API call to the `insertFiles` endpoint.

Configure Cloud Storage Event Notifications

Notifications from your cloud storage infrastructure are a straight-forward way to trigger Snowpipe for continuous loading.

Cloud Storage Platforms Snowpipe Supports

- Google Cloud Storage
- Microsoft Azure Blob Storage
- AWS S3

Choosing AWS S3 with Snowpipe integration allows you to decide between using an S3 event notification or Amazon's Simple Notification Service(SNS) to stage data for integration. Check if you currently have S3 event notifications that may conflict with a new notification. If so, you'll want to opt for SNS notifications. If not, the simple choice is to trigger continuous integration with S3 event notifications. Be aware that SNS network traffic travels outside of Amazon's VPC. If the potential security risk is an issue, consider employing AWS's PrivateLink service.

Let's go over the access requirements needed to begin using S3 event notifications to load new data seamlessly in micro-batches.

Configure Cloud Storage Permissions

To begin using AWS storage notifications for Snowpipe processing, you'll follow these steps within your AWS and Snowflake account to set up the security conditions.

1. Create IAM Policy for Snowflake's S3 Access

Snowflake needs IAM policy permission to access your S3 with `GetObject`, `GetObjectVersion`, and `ListBucket`. Log into your AWS console and navigate to the IAM service. Within the **Account settings**, confirm the **Security Token Service** list records your account's region as **Active**.

Navigate to **Policies** and use the JSON below to create a new IAM policy named 'snowflake_access'.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": "arn:aws:s3:::<bucket>/<prefix>/*"
        },
        {
            "Effect": "Allow",
            "Action": "s3>ListBucket",
            "Resource": "arn:aws:s3:::<bucket>",
            "Condition": {
                "StringLike": {
                    "s3:prefix": [
                        "
```

```
    " * "
    ]
}
}
]
}
```

Don't forget to replace the `<bucket>` and `<prefix>` with **your** AWS S3 bucket name and folder path prefix.

2. New IAM Role

On the AWS IAM console, add a new IAM role tied to the 'snowflake_access' IAM policy. Create the role with the following settings.

- **Trusted Entity:** Another AWS account
- **Account ID:**
- **Require External ID:** [x]
- **External ID:** 0000
- **Role Name:** snowflake_role
- **Role Description:** Snowflake role for access to S3 `<bucket>`
- **Policies:** snowflake_access

Create the role, then click to see the role's summary and record the **Role ARN**.

3. Integrate IAM user with Snowflake storage.

Within your Snowflake web console, you'll run a `CREATE STORAGE INTEGRATION` command on a worksheet.

```
CREATE OR REPLACE STORAGE INTEGRATION S3_role_integration
TYPE = EXTERNAL_STAGE
STORAGE_PROVIDER = S3
ENABLED = TRUE
STORAGE_AWS_ROLE_ARN = "arn:aws:iam::<role_account_id>:role/snowflake_role"
STORAGE_ALLOWED_LOCATIONS = ("s3://<bucket>/<path>/");
```

Be sure to change the `<bucket>`, `<prefix>` and `<role_account_id>` is replaced with *your* AWS S3 bucket name, folder path prefix, and IAM role account ID.

```

1 CREATE OR REPLACE STORAGE INTEGRATION S3_ROLE_INTEGRATION
2   TYPE = EXTERNAL_STAGE
3   STORAGE_PROVIDER = S3
4   ENABLED = TRUE
5   STORAGE_AWS_ROLE_ARN = "arn:aws:iam::██████████:role/snowflake_role"
6   STORAGE_ALLOWED_LOCATIONS = ("s3://██████████/████/");

Results Data Preview
Query ID SQL 179ms 1 rows
Filter result... Copy Columns ▾
Row status
1 Integration S3_ROLE_INTEGRATION successfully created.

```

Note in the figure above the **ACCOUNTADMIN** role and status message of a successful creation.

4. Run storage integration description command.

```
desc integration S3_ROLE_INTEGRATION;
```

Run the above command to display your new integration's description.

```

1 desc integration S3_ROLE_INTEGRATION;

Results Data Preview
Query ID SQL 653ms 7 rows
Filter result... Copy Columns ▾
Row property property_type property_value property_default
1 ENABLED Boolean true false
2 STORAGE_PROVIDER String S3
3 STORAGE_ALLOWED_LOCATIONS List s3://██████████/
4 STORAGE_BLOCKED_LOCATIONS List
5 STORAGE_AWS_IAM_USER_ARN String arn:aws:iam::██████████:user/██████████
6 STORAGE_AWS_ROLE_ARN String arn:aws:iam::██████████:role/snowflake_role
7 STORAGE_AWS_EXTERNAL_ID String ██████████SFRole-██████████

```

Record the property values displayed for `STORAGE_AWS_IAM_USER_ARN` and `STORAGE_AWS_EXTERNAL_ID`.

5. IAM User Permissions

Navigate back to your AWS IAM service console. Within the **Roles**, click the 'snowflake_role'. On the **Trust relationships** tab, click **Edit trust relationship** and edit the file with the `STORAGE_AWS_IAM_USER_ARN` and `STORAGE_AWS_EXTERNAL_ID` retrieved in the previous step.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "<STORAGE_AWS_IAM_USER_ARN>"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "<STORAGE_AWS_EXTERNAL_ID>"
        }
      }
    }
  ]
}
```

Update Trust Policy after replacing the string values for your `STORAGE_AWS_IAM_USER_ARN` and `STORAGE_AWS_EXTERNAL_ID`.

After completing this section, your AWS and Snowflake account permissions are ready for Snowpipe. The next section provides the steps to perform automated micro-batching with cloud notifications triggering Snowpipe.

Create a pipe in Snowflake

Now that your AWS and Snowflake accounts have the right security conditions, complete Snowpipe setup with S3 event notifications.

1. Create a Database, Table, Stage, and Pipe

On a fresh Snowflake web console worksheet, use the commands below to create the objects needed for Snowpipe ingestion.

Create Database

```
create or replace database S3_db;
```

Execute the above command to create a database called 'S3_db'. The **Results** output will show a status message of `Database S3_DB successfully created.`

Create Table

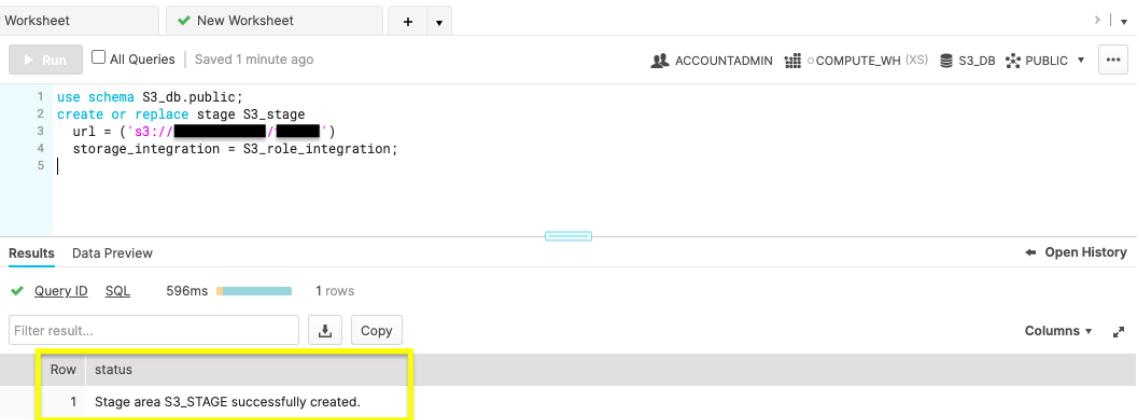
```
create or replace table S3_table(files string);
```

This command will make a table by the name of 'S3_table' on the S3_db database. The **Results** output will show a status message of `Table S3_TABLE successfully created.`

Create Stage

```
use schema S3_db.public;
create or replace stage S3_stage
url = ('s3://<bucket>/<path>/')
storage_integration = S3_role_integration;
```

To make the external [stage] needed for our S3 bucket, use this command. Be mindful to replace the <bucket> and <path> with your S3 bucket name and file path.



The screenshot shows a Snowflake Worksheet interface. At the top, there are tabs for 'Worksheet' and 'New Worksheet'. Below the tabs, there's a toolbar with a 'Run' button, a 'Saved 1 minute ago' indicator, and user account information ('ACCOUNTADMIN', 'COMPUTE_WH (XS)', 'S3_DB', 'PUBLIC'). A code editor window contains the following SQL command:

```
1 use schema S3_db.public;
2 create or replace stage S3_stage
3 url = ( s3://[REDACTED]/[REDACTED] )
4 storage_integration = S3_role_integ;
5 |
```

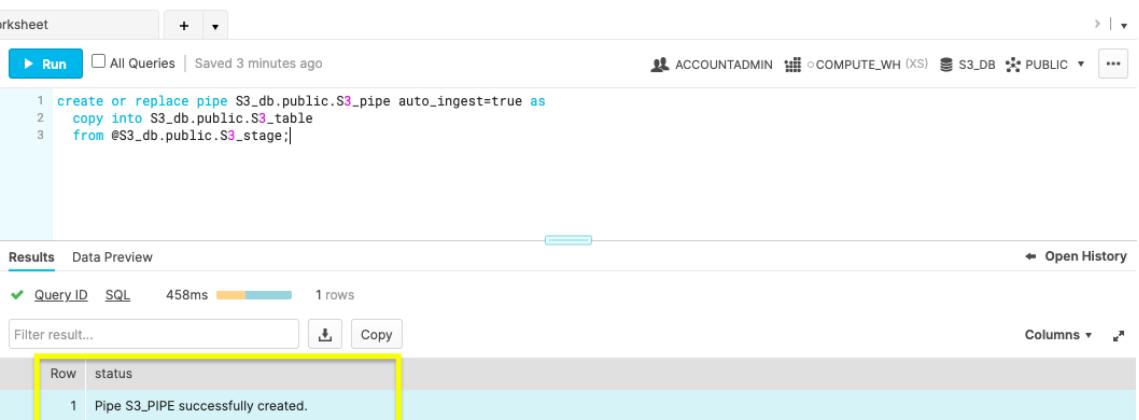
Below the code editor is a results table with two columns: 'Row' and 'status'. The first row shows the message 'Stage area S3_STAGE successfully created.' This row is highlighted with a yellow box.

The figure above shows *Results* reading 'Stage area S3_STAGE successfully created'.

Create Pipe

The magic of automation is in the [create pipe] parameter, `auto_ingest=true`. With `auto_ingest` set to true, data staged will automatically integrate into your database.

```
create or replace pipe S3_db.public.S3_pipe auto_ingest=true as
  copy into S3_db.public.S3_table
  from @S3_db.public.S3_stage;
```



The screenshot shows a Snowflake Worksheet interface. At the top, there are tabs for 'Worksheet' and 'New Worksheet'. Below the tabs, there's a toolbar with a 'Run' button, a 'Saved 3 minutes ago' indicator, and user account information ('ACCOUNTADMIN', 'COMPUTE_WH (XS)', 'S3_DB', 'PUBLIC'). A code editor window contains the following SQL command:

```
1 create or replace pipe S3_db.public.S3_pipe auto_ingest=true as
2   copy into S3_db.public.S3_table
3   from @S3_db.public.S3_stage;
```

Below the code editor is a results table with two columns: 'Row' and 'status'. The first row shows the message 'Pipe S3_PIPE successfully created.' This row is highlighted with a yellow box.

Confirm you receive a status message of, 'Pipe S3_PIPE successfully created'.

2. Configure Snowpipe User Permissions

To ensure the Snowflake user associated with executing the Snowpipe actions had sufficient permissions, create a unique role to manage Snowpipe security privileges. Do not employ the user account you're currently utilizing, instead create a new user to assign to Snowpipe within the web console.

```

-- Create Role
use role securityadmin;
create or replace role S3_role;

-- Grant Object Access and Insert Permission
grant usage on database S3_db to role S3_role;
grant usage on schema S3_db.public to role S3_role;
grant insert, select on S3_db.public.S3_table to role S3_role;
grant usage on stage S3_db.public.S3_stage to role S3_role;

-- Bestow S3_pipe Ownership
grant ownership on pipe S3_db.public.S3_pipe to role S3_role;

-- Grant S3_role and Set as Default
grant role S3_role to user <username>;
alter user <username> set default_role = S3_role;

```



The screenshot shows a Snowflake worksheet interface. At the top, there's a toolbar with 'Worksheet' and a '+' button. Below it is a navigation bar with 'Run' (highlighted in blue), 'All Queries', and a timestamp 'Saved 37 seconds ago'. To the right are account and session details: ACCOUNTADMIN, COMPUTE_WH (XS), S3_DB, PUBLIC, and a three-dot menu.

The main area contains the 16-line SQL script from above. Below the script is a results table:

Row	status
1	Statement executed successfully.

The 'status' column for the first row is highlighted with a yellow box.

Create a new role named `S3_role` with `SECURITYADMIN` access. Give the `S3_role` usage permissions to the database objects, insert permission for the `S3_table` and ownership of `S3_pipe`. Lastly, set the `S3_role` as a Snowflake user's default. Confirm the statement was successful before creating the S3 event notification.

3. New S3 Event

Run the `show pipes` command to record the ARN listed in the 'notification_channel' column.

```
show pipes;
```

```

1 show pipes;

```

Results Data Preview

Query_ID SQL 51ms 1 rows

database_name	schema_name	definition	owner	notification_channel
S3_DB	PUBLIC	copy into S3...	ACCOUNTA...	arn:aws:sqs:us-east-1:...:sf-snowpipe-...

Copy the ARN because you'll need it to configure the S3 event notification.

Sign in to your AWS account and navigate to the S3 service console. Select the bucket being used for Snowpipe and go to the **Properties** tab. The **Events** card listed will allow you to **Add notification**. Create a notification with the values listed.

- **Name:** Auto-ingest Snowflake
- **Events:** All object create events
- **Send to:** SQS Queue
- **SQS:** Add SQS queue ARN
- **SQS queue ARN:**

Name	Events	Filter	Type
Auto-ingest Snowflake	All object create events		SQS

1 Active notifications

Cancel **Save**

Snowpipe's automated micro-batching is now active. Learn how to manage database integration in the next step.

Manage and Remove Pipes

Snowpipe is configured and now updates the Snowflake database when any object is created in the AWS S3 bucket. Let's review a few common commands to manage and remove Snowpipe.

ALTER PIPE

- Pause Snowpipe

```
ALTER PIPE ... SET PIPE_EXECUTION_PAUSED = true
```

Before making configuration changes with `ALTER PIPE`, stop the Snowpipe process by pausing it by setting `PIPE_EXECUTION_PAUSED` to true. Once you're ready to start the pipe again, set this parameter to false.

- Check Snowpipe Status

```
SYSTEM$PIPE_STATUS
```

To check the status of the pipe, run the above command.

DROP PIPE

```
drop pipe S3_db.public.S3_pipe;
```

The `drop` command will delete your Snowpipe once you are finished with this tutorial.

SHOW PIPE

```
show pipes
```

Confirm the pipe was removed by displaying all of the pipes.

After wrapping up this section, you're ready to look ahead to using Snowpipe on your applications.

Conclusion

You've learned the benefits of continuous data loading and the different ways Snowpipe can be triggered to load new data. Plus, you can now configure cloud storage event notifications and manage Snowpipe.

Review the various ways to implement Snowpipe. If you're contemplating migrating an active database, select the staging method best for your architecture, and test the workflow on a small dataset. Experimenting will allow for an accurate cost-benefit analysis.

What we've covered

- Snowpipe Continuous Data Loading vs SnowPipe REST API
- Configuring AWS S3 event notifications for Snowpipe
- Creating, managing, and deleting pipes in Snowflake