**PIVOT** is a SQL operation that can be used to turn unique values from one column of a table into multiple columns in the output, essentially rotating data from rows to columns. It often includes an aggregation function and is useful for generating more readable and organized reports from raw data.

**Basic PIVOT Syntax and Usage**

The basic syntax for a PIVOT operation is as follows:

SELECT <non-pivoted column>,

[first pivoted column] AS <column name>,

[second pivoted column] AS <column name>,

[last pivoted column] AS <column name>

FROM

(<SELECT query that produces the data>)

AS <alias for the source query>

PIVOT

(

<aggregation function>(<column being aggregated>)

FOR

[<column that contains the values that will become column headers>]

IN ( [first pivoted column], [second pivoted column],

... [last pivoted column])

) AS <alias for the pivot table>

<optional ORDER BY clause>;

This structure allows you to transform data from a format with many rows to one with fewer rows and more columns, which can be easier to read and analyze[1](https://www.bing.com/ck/a?!&&p=76f1f57eca9ec0beed02b097f9e652537509e48315b4a7aa9797297ab7837f59JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY29tL2VuLXVzL3NxbC90LXNxbC9xdWVyaWVzL2Zyb20tdXNpbmctcGl2b3QtYW5kLXVucGl2b3Q_dmlldz1zcWwtc2VydmVyLXZlcjE2&ntb=1).

**Practical Example of PIVOT**

Consider a scenario where you have a table of products with a 'DaysToManufacture' column. You might want to find the average standard cost for each 'DaysToManufacture' value. Without PIVOT, you would get a simple two-column table. With PIVOT, you can turn each unique 'DaysToManufacture' value into its own column, with the average costs as the values[1](https://www.bing.com/ck/a?!&&p=76f1f57eca9ec0beed02b097f9e652537509e48315b4a7aa9797297ab7837f59JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY29tL2VuLXVzL3NxbC90LXNxbC9xdWVyaWVzL2Zyb20tdXNpbmctcGl2b3QtYW5kLXVucGl2b3Q_dmlldz1zcWwtc2VydmVyLXZlcjE2&ntb=1).

Here's an example of how you might write this query:

SELECT 'AverageCost' AS Cost\_Sorted\_By\_Production\_Days,

[0], [1], [2], [3], [4]

FROM

(

SELECT DaysToManufacture, StandardCost

FROM Production.Product

) AS SourceTable

PIVOT

(

AVG(StandardCost)

FOR DaysToManufacture IN ([0], [1], [2], [3], [4])

) AS PivotTable;

This query would produce a single row with each 'DaysToManufacture' value as a column header and the corresponding average costs as the row's values[1](https://www.bing.com/ck/a?!&&p=76f1f57eca9ec0beed02b097f9e652537509e48315b4a7aa9797297ab7837f59JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY29tL2VuLXVzL3NxbC90LXNxbC9xdWVyaWVzL2Zyb20tdXNpbmctcGl2b3QtYW5kLXVucGl2b3Q_dmlldz1zcWwtc2VydmVyLXZlcjE2&ntb=1).

**UNPIVOT: The Inverse of PIVOT**

**UNPIVOT** performs the reverse operation of PIVOT by rotating columns of a table-valued expression into column values. This is useful when you need to normalize denormalized data or when preparing data for operations that require a long format.

Here's a basic example of UNPIVOT:

SELECT VendorID, Employee, Orders

FROM

(SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5

FROM pvt) p

UNPIVOT

(Orders FOR Employee IN

(Emp1, Emp2, Emp3, Emp4, Emp5)

)AS unpvt;

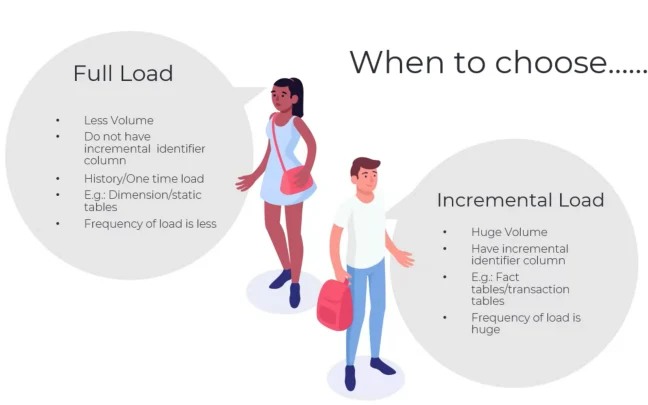
In this example, the columns 'Emp1', 'Emp2', 'Emp3', 'Emp4', and 'Emp5' are transformed into rows in the 'Employee' column, with their values appearing in the 'Orders' column[1](https://www.bing.com/ck/a?!&&p=76f1f57eca9ec0beed02b097f9e652537509e48315b4a7aa9797297ab7837f59JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY29tL2VuLXVzL3NxbC90LXNxbC9xdWVyaWVzL2Zyb20tdXNpbmctcGl2b3QtYW5kLXVucGl2b3Q_dmlldz1zcWwtc2VydmVyLXZlcjE2&ntb=1).

**Considerations When Using PIVOT**

* **Aggregation**: PIVOT requires an aggregation function because it's possible for multiple rows to be combined into a single row in the output. Common aggregation functions include SUM, AVG, COUNT, MIN, and MAX.
* **NULL Values**: If the column being aggregated contains NULL values, they will not be considered in the aggregation[1](https://www.bing.com/ck/a?!&&p=76f1f57eca9ec0beed02b097f9e652537509e48315b4a7aa9797297ab7837f59JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY29tL2VuLXVzL3NxbC90LXNxbC9xdWVyaWVzL2Zyb20tdXNpbmctcGl2b3QtYW5kLXVucGl2b3Q_dmlldz1zcWwtc2VydmVyLXZlcjE2&ntb=1).
* **Dynamic PIVOT**: For scenarios where the column values are not known beforehand or can change, dynamic SQL can be used to construct and execute a PIVOT query[2](https://www.bing.com/ck/a?!&&p=1492417ad1845c4d7a3c5fddf35c6e52f8e137977517a57e578b0b881d0be652JmltdHM9MTcyOTQ2ODgwMA&ptn=3&ver=2&hsh=4&fclid=05ebcdbb-d927-68c3-087d-d9ddd8bc699e&u=a1aHR0cHM6Ly93d3cuc3Fsc2VydmVydHV0b3JpYWwubmV0L3NxbC1zZXJ2ZXItYmFzaWNzL3NxbC1zZXJ2ZXItcGl2b3Qv&ntb=1).

PIVOT is a powerful tool in SQL that can greatly enhance the readability and analytical value of your data. It's particularly useful in reporting and data analysis scenarios where data needs to be summarized in a cross-tabular format.

Incremental data load refers to the selective movement of data from one system to another, focusing on identifying the data that was created or modified since the last load process ran. It allows organizations to handle large amounts of data efficiently by loading only the necessary sections in small, incremental steps.



**Snowflake Dynamic Table**

A diagram of a diagram

Description automatically generated

* Snowflake has introduced**Dynamic Tables** as a preview feature, which is **now available to all accounts**.
* **Dynamic tables** are tables that materialize the results of a specified query. Rather than creating a separate target table and writing code to modify and update the data in that table, dynamic tables allow you to designate the target table as dynamic and define an SQL statement to perform the transformation.
* These tables automatically update the materialized results through regular and often incremental refreshes, eliminating the need for manual updates. Dynamic tables provide a convenient and automated way to manage data transformations and keep the target table up-to-date with the latest query results.

***How to create Dynamic Tables?***

* To create a dynamic table, use the ***CREATE DYNAMIC TABLE*** command, specifying the query to use, the target lag of the data, and the warehouse to use to perform the refreshes.
* ***Syntax:***

CREATE [ OR REPLACE ] DYNAMIC TABLE <name>  
 TARGET\_LAG = { '<num> { seconds | minutes | hours | days }' | DOWNSTREAM }  
 WAREHOUSE = <warehouse\_name>  
 AS <query>

**TARGET\_LAG** = { *num* { seconds | minutes | hours | days } | DOWNSTREAM }

Specifies the lag for the dynamic table:

***'*num*seconds | minutes | hours | days'***

* The **TARGET\_LAG** parameter specifies the maximum allowed time lag between updates to the base tables and the content of the dynamic table.
* The minimum allowed value is 1 minute.
* ***Example:***
* CREATE OR REPLACE DATABASE DYNAMIC\_TABLE\_DB;  
    
  CREATE OR REPLACE SCHEMA DYNAMIC\_TABLE\_SCH;  
    
  CREATE OR REPLACE TABLE EMPLOYEE(EMP\_ID INT, EMP\_NAME VARCHAR,EMP\_ADDRESS VARCHAR);  
    
  INSERT INTO EMPLOYEE VALUES(1,'AGAL','INDIA');  
  INSERT INTO EMPLOYEE VALUES(2,'KINNU','INDIA');  
  INSERT INTO EMPLOYEE VALUES(3,'SHUKESH','AUSTRALIA');  
  INSERT INTO EMPLOYEE VALUES(4,'SUPREET','UAE');  
    
  SELECT \* FROM EMPLOYEE;  
    
  CREATE OR REPLACE TABLE EMPLOYEE\_SKILL(  
  SKILL\_ID NUMBER,  
  EMP\_ID NUMBER,  
  SKILL\_NAME VARCHAR(50),  
  SKILL\_LEVEL VARCHAR(50)  
  );  
    
  INSERT INTO EMPLOYEE\_SKILL VALUES(1,1,'SNOWFLAKE','ADVANCE');  
  INSERT INTO EMPLOYEE\_SKILL VALUES(2,1,'PYTHON','BASIC');  
  INSERT INTO EMPLOYEE\_SKILL VALUES(3,1,'SQL','INTERMEDIATE');  
  INSERT INTO EMPLOYEE\_SKILL VALUES(1,2,'SNOWFLAKE','ADVANCE');  
  INSERT INTO EMPLOYEE\_SKILL VALUES(1,4,'SNOWFLAKE','ADVANCE');  
    
  SELECT \* FROM EMPLOYEE\_SKILL;

The given script includes the creation and population of two tables: **EMPLOYEE**and **EMPLOYEE\_SKILL**. Here’s a brief description of each table:

***EMPLOYEE Table:***

* **Columns**: EMP\_ID (integer), EMP\_NAME (varchar), EMP\_ADDRESS (varchar)
* **Purpose**: This table stores information about employees, including their unique IDs, names, and addresses.

***EMPLOYEE\_SKILL Table:***

* **Columns**: SKILL\_ID (number), EMP\_ID (number), SKILL\_NAME (varchar), SKILL\_LEVEL (varchar)
* **Purpose**: This table maintains the skills and skill levels of employees. It establishes a relationship with the EMPLOYEE table through the EMP\_ID column, representing the employee’s ID. Each skill entry includes a skill ID, skill name, and skill level.

***Points to remember:***

* As dynamic tables rely on tracking changes in the underlying database objects, it becomes necessary to enable change tracking on all related objects.
* When creating a dynamic table in Snowflake, the platform automatically attempts to enable change tracking on the underlying objects. However, it is important to note that the user creating the dynamic table might not have the necessary privileges to enable change tracking on all the required objects. Therefore, it is advisable to use commands such as **SHOW VIEW**, **SHOW TABLE**, or similar ones to inspect the **CHANGE\_TRACKING**column.
* This will help determine if change tracking is enabled for specific database objects, ensuring smooth and error-free refreshes of dynamic tables.

Now, we will check change tracking for the table which we have created,

SHOW TABLES;

A screenshot of a computer

Description automatically generated

Although change tracking is currently disabled for both the **Employee**and **Employee\_Skill**tables, it’s important to note that when a dynamic table is created on top of these tables, change tracking will be automatically enabled. This ensures that the dynamic table captures and reflects any modifications made to the underlying data.

***Dynamic Table:***

CREATE OR REPLACE DYNAMIC TABLE EMPLOYEE\_DET  
 TARGET\_LAG = '1 MINUTE'  
 WAREHOUSE = COMPUTE\_WH  
 AS  
 SELECT A.EMP\_ID,A.EMP\_NAME,A.EMP\_ADDRESS, B.SKILL\_ID,B.SKILL\_NAME,B.SKILL\_LEVEL  
 FROM EMPLOYEE A, EMPLOYEE\_SKILL B  
 WHERE A.EMP\_ID=B.EMP\_ID  
 ORDER BY B.SKILL\_ID ;

In this scenario:

* The code snippet demonstrates the creation or replacement of a dynamic table named EMPLOYEE\_DET. It utilizes the EMPLOYEE and EMPLOYEE\_SKILL tables to populate the dynamic table.
* The target lag for the dynamic table is set to 1 minute, indicating that the data in the dynamic table should ideally not be more than 1 minute behind the data in the source tables.
* The dynamic table is refreshed automatically, leveraging the compute resources of the **COMPUTE\_WH**warehouse.
* The data in the dynamic table is derived by selecting relevant columns from the **EMPLOYEE**and **EMPLOYEE\_SKILL**tables, performing a join based on the **EMP\_ID**column, and ordering the result by the **SKILL\_ID**column.

*When querying the Dynamic Table****EMPLOYEE\_DET****immediately after its creation, you may encounter an error stating, “Dynamic Table ‘****DYNAMIC\_TABLE\_DB.DYNAMIC\_TABLE\_SCH.EMPLOYEE\_DET****’ is not initialized. Please run a manual refresh or wait for a scheduled refresh before querying.” This error occurs because the table requires a one-minute wait for the Target Lag to be completed. It is necessary to either manually refresh the table or wait until the scheduled refresh occurs before querying the data successfully.*

After a one-minute duration following the execution of the Dynamic table creation process,

A screenshot of a computer

Description automatically generated

SELECT \* FROM EMPLOYEE\_DET;

we will explore the following areas related to dynamic tables:

1. **Working with Dynamic Tables: (Alter / Describe / Drop / Show)**
2. **Dynamic Tables vs. Streams and Tasks:**

* Comparing dynamic tables with streams and tasks
* Understanding their respective use cases and advantages
* Exploring the differences in functionality and behavior

3. **Dynamic Tables vs. Materialized Views:**

* Contrasting dynamic tables with materialized views
* Examining their distinct characteristics and purposes
* Analyzing the benefits and trade-offs of using each approach

4. **Managing Dynamic Tables:**

* Best practices for managing dynamic tables effectively
* Optimizing performance and resource utilization
* Handling dynamic table dependencies and refresh scheduling

5. **Understanding Dynamic Table States:**

* Exploring the different states of dynamic tables
* Interpreting their significance and implications
* Managing and troubleshooting dynamic table states

**Parquet** is an open-source file format built to handle flat columnar storage data formats. Parquet operates well with complex data in large volumes. It is known for its both performant data compression and its ability to handle a wide variety of encoding types.

**What is a Parquet file vs. CSV?**

Parquet file format is a columnar storage format, which means that data for each column is stored together. The storage mechanism enables better compression and typically results in smaller file sizes compared to row-based formats.

CSV is a row-based format, where each row is represented as a separate line in the file.

**How to Load PARQUET File (Unstructured Data) into Snowflake**

R**equirements**

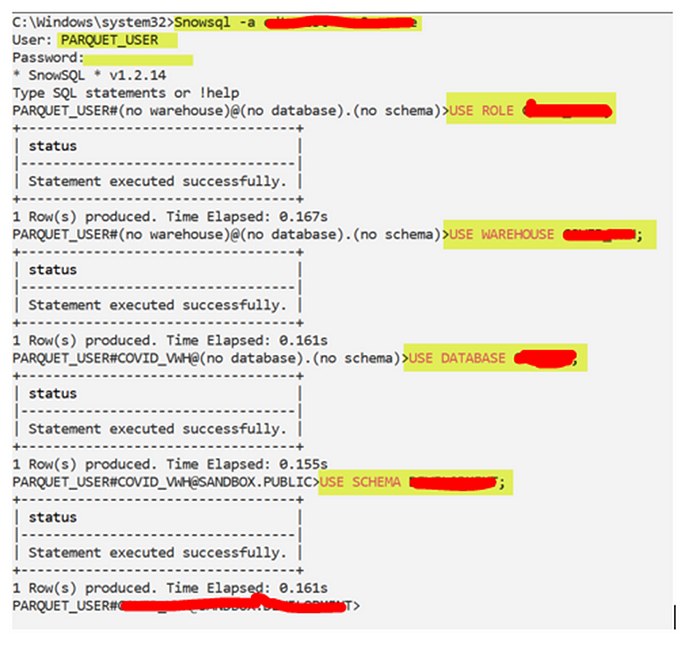
Snowsql installed on your local machine. Make sure PARQUET File is accessible by the SNOWSQL Program.

In ***snowsql command line, run below commands for connecting to snowflake.***

Snowsql –a ACCOUNTNAME.azure (enter)  
User: USER\_NAME (enter)  
Password: PASSWORD (enter)

O***nce connected to Snowflake from SNOWSQL, use the following commands individually to set the context for DB objects***

USE ROLE DBNAME\_ROLE;  
USE WAREHOUSE DBNAME\_VWH;  
USE DATABASE DBNAME;  
USE SCHEMA DEVELOPMENT;



C***opy the files from local system to Snowflake stage.***

Parquet File is USERDATA4.PARQUET  
Internal Stage is PARQUET\_TEST  
***put file://c:/parquet\_pipeline/userdata4.parquet***[***@PARQUET\_TEST***](http://twitter.com/PARQUET_TEST)***auto\_compress=true;***

A screenshot of a computer

Description automatically generated

L***ist the contents of the Stage***

***list @parquet\_test;***



C***reate temporary table in snowflake environment (in Snowflake GUI)***

PARQUET file format can produce only one column of type variant or object or array. Create a temporary table contains only one column.

CREATE table if not exists DB.SCHEMA.PARQUET\_VARIANT  
(src variant);

C***opy files from stage to temporary table (in the Snowflake GUI)***

COPY INTO “SANDBOX”.”DEVELOPMENT”.”PARQUET\_VARIANT”  
FROM [@parquet\_test](http://twitter.com/parquet_test)  
FILE\_FORMAT = ‘“SANDBOX”.”DEVELOPMENT”.”PARQUET”’  
ON\_ERROR = ‘CONTINUE’  
PURGE = TRUE;

V***erify the VARIANT Column***

select \* from development.parquet\_variant;



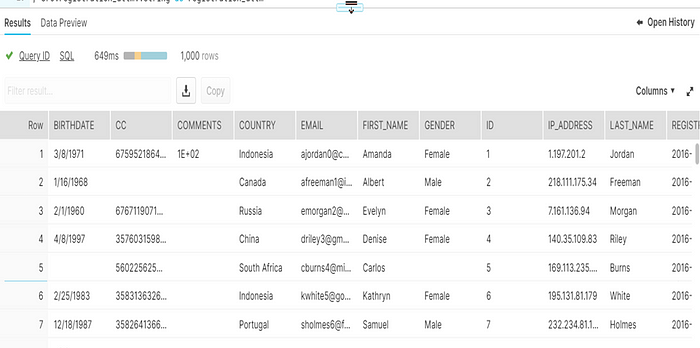
C***reate normal tabular format table with all columns and data.***

create or replace table USERDATA\_PARQUET AS

Select  
Src:birthdate::string as birthdate  
, Src:cc::string as cc  
, Src:comments::string as comments  
, Src:country::string as country  
, Src:email::string as email  
, Src:first\_name::string as first\_name  
, Src:gender::string as gender  
, Src:id::string as id  
, Src:ip\_address::string as ip\_address  
, Src:last\_name::string as last\_name  
, Src:registration\_dttm::string as registration\_dttm  
, Src:salary::string as salary  
, Src:title::string as title  
from development.parquet\_variant.

V***erify the data in Snowflake***

select \* from USERDATA\_PARQUET;



**HOW TO: Load and Query JSON data in Snowflake?**

**Contents**

Introduction

Loading JSON data from local machine into Snowflake Internal Stage

Step-1: Create a Snowflake Internal named Stage

Step-2: Load data from local machine into Internal Stage using SnowSQL

Step-3: Verify the file in Internal Stage using List command

Loading JSON data from Snowflake internal Stage into database table

Step-4: Create a JSON file format in Snowflake

Step-5: Create database table to load JSON data

Step-6: Load data from Internal Stage into database table using COPY command

Querying JSON data from Snowflake database table

Flattening Arrays in JSON data

Summary

**Introduction**

Snowflake supports loading semi structured data like JSON in [database](https://thinketl.com/how-to-load-and-query-json-data-in-snowflake/) tables. Once the data is loaded from stage into a database table, Snowflake has an excellent functionality for directly querying semi-structured data along with flattening it into a columnar structure.

In this article, let us discuss the entire process of loading and [parsing](https://thinketl.com/how-to-load-and-query-json-data-in-snowflake/) JSON data in Snowflake.

**Local Machine → Snowflake Internal Stage → Database table → Querying and Flattening the JSON data**

For the demonstration, we will consider below JSON data containing novel’s information of three authors.

[

{

"AuthorName": "Author-1",

"Category": [{

"CategoryName": "Fiction",

"Genre": [{

"GenreName": "Action and Adventure",

"Novel": [{

"Novel": "Novel-1",

"Sales": "200"

},

{

"Novel": "Novel-2",

"Sales": "850"

}

]

},

{

"GenreName": "Romance",

"Novel": [{

"Novel": "Novel-3",

"Sales": "400"

}]

}

]

},

{

"CategoryName": "NonFiction",

"Genre": [{

"GenreName": "Autobiography",

"Novel": [{

"Novel": "Novel-4",

"Sales": "900"

}]

}]

}

]

},

{

"AuthorName": "Author-2",

"Category": [{

"CategoryName": "Fiction",

"Genre": [{

"GenreName": "Action and Adventure",

"Novel": [{

"Novel": "Novel-5",

"Sales": "340"

}]

},

{

"GenreName": "Crime",

"Novel": [{

"Novel": "Novel-6",

"Sales": "940"

},

{

"Novel": "Novel-7",

"Sales": "540"

}

]

}

]

}

]

},

{

"AuthorName": "Author-3",

"Category": [{

"CategoryName": "Fiction",

"Genre": [{

"GenreName": "Romance",

"Novel": [{

"Novel": "Novel-8",

"Sales": "820"

},

{

"Novel": "Novel-9",

"Sales": "620"

}

]

},

{

"GenreName": "Thriller",

"Novel": [{

"Novel": "Novel-10",

"Sales": "770"

}]

}

]

},

{

"CategoryName": "NonFiction",

"Genre": [{

"GenreName": "History",

"Novel": [{

"Novel": "Novel-11",

"Sales": "450"

}]

},

{

"GenreName": "Travel",

"Novel": [{

"Novel": "Novel-12",

"Sales": "150"

}]

}

]

}

]

}

]

Our goal is to load the above JSON data into Snowflake and flatten it to achieve below columnar format of the data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Author** | **Category** | **Genre** | **Novel** | **Sales** |
| Author1 | Fiction | Action and Adventure | Novel-1 | 200 |
| Author1 | Fiction | Action and Adventure | Novel-2 | 850 |
| Author1 | Fiction | Romance | Novel-3 | 400 |
| Author1 | Non-Fiction | Autobiography | Novel-4 | 900 |
| Author2 | Fiction | Action and Adventure | Novel-5 | 340 |
| Author2 | Fiction | Crime | Novel-6 | 940 |
| Author2 | Fiction | Crime | Novel-7 | 540 |
| Author3 | Fiction | Romance | Novel-8 | 820 |
| Author3 | Fiction | Romance | Novel-9 | 620 |
| Author3 | Fiction | Thriller | Novel-10 | 770 |
| Author3 | Non-Fiction | History | Novel-11 | 450 |
| Author3 | Non-Fiction | Travel | Novel-12 | 150 |

**Loading JSON data from local machine into Snowflake Internal Stage**

Before we load the data, we must choose the database and schema where the data is staged and later loaded into table. For the demonstration we will be using MY\_DB database and MY\_DB.MY\_SCHEMA schema.

USE DATABASE my\_db;

USE SCHEMA my\_schema;

**Step-1: Create a Snowflake Internal named Stage**

Create an [**Internal Named Stage**](https://thinketl.com/types-of-snowflake-stages-data-loading-and-unloading-features/) in Snowflake to hold the data loaded from local machine as shown below.

CREATE OR REPLACE STAGE my\_internal\_stage;

**Step-2: Load data from local machine into Internal Stage using SnowSQL**

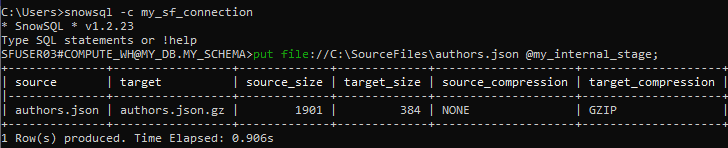
To load data from our local machine into the Snowflake Internal stage, we have to use the Snowflake’s CLI tool which is SnowSQL.

To know more about this, read our detailed article on [**Snowflake SnowSQL**](https://thinketl.com/snowflake-snowsql-command-line-tool-to-access-snowflake/).

1. Login to Snowflake SnowSQL

2. Using PUT command, copy the files from the local folder into snowflake internal stage created in earlier step.

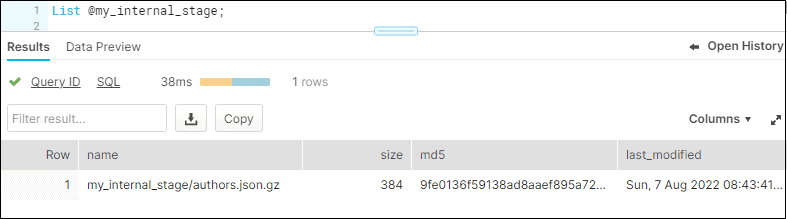
put file://C:\SourceFiles\authors.json @my\_internal\_stage;

Loading JSON file from local machine into Snowflake Internal Stage

**Step-3: Verify the file in Internal Stage using List command**

Use **List** command in Snowflake worksheet to verify the file in internal stage loaded from SnowSQL as shown below.

List @my\_internal\_stage;

Listing files in Snowflake Internal Stage

**Loading JSON data from Snowflake internal Stage into database table**

**Step-4: Create a JSON file format in Snowflake**

Create a [**Snowflake File format**](https://thinketl.com/snowflake-file-formats/) of type JSON which encapsulates information of data files which helps in processing the files from stage.

CREATE OR REPLACE FILE FORMAT my\_json\_format

    type = json

    strip\_outer\_array = true

;

We have used an option called **STRIP\_OUTER\_ARRAY** for this load. It helps in removing the outer set of square brackets **[ ]** when loading the data, separating the initial array into multiple lines. Else the entire JSON data gets loaded into single record instead of multiple records.

**Step-5: Create database table to load JSON data**

Create a table to load JSON data from Snowflake internal stage as shown below. Snowflake stores semi-structured data using the **VARIANT** field type in tables.

CREATE OR REPLACE TABLE Authors (

JSON\_DATA VARIANT

);

Note that the data of each author in the JSON file will be loaded into a single column **JSON\_DATA** of type **VARIANT** in the table named **Authors**.

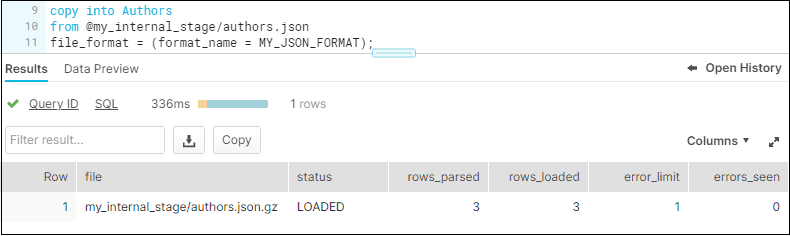
**Step-6: Load data from Internal Stage into database table using COPY command**

**COPY** command in Snowflake helps in loading data from staged files to an existing table. Load the data from a JSON file in internal stage to Authors [database](https://thinketl.com/how-to-load-and-query-json-data-in-snowflake/) table using the **MY\_JSON\_FORMAT** file format as shown below

COPY INTO Authors

FROM @my\_internal\_stage/authors.json

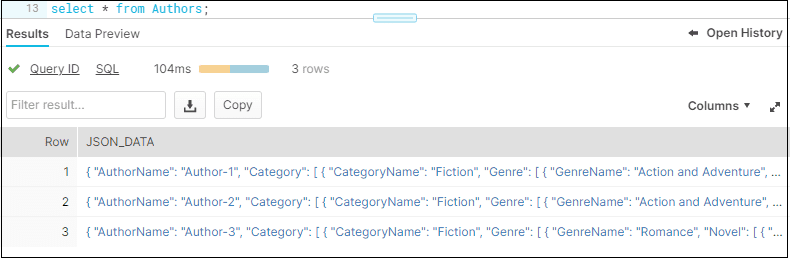
FILE\_FORMAT = (format\_name = MY\_JSON\_FORMAT);

Copying files from Snowflake internal stage into database table

**Querying JSON data from Snowflake database table**

The data from the Authors table can be queried directly as shown below. By using the **STRIP\_OUTER\_ARRAY** option, we were able remove this initial array **[]** and treat each object in the array as a row in Snowflake. Hence each author object loaded as a separate row.

SELECT \* FROM Authors;

Querying data from database table with JSON data

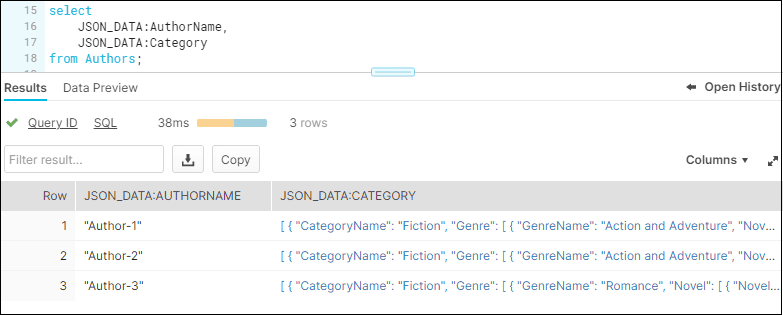
The individual elements in the column JSON\_DATA can be queried using standard **:** notation as shown below.

SELECT

    JSON\_DATA:AuthorName,

    JSON\_DATA:Category

FROM Authors;

Querying individual JSON elements from database table

The data in the Category can be further drilled down and required elements information can be fetched as shown below.

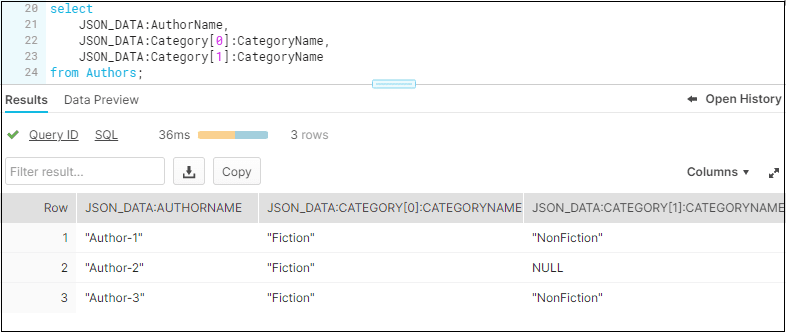
SELECT

    JSON\_DATA:AuthorName,

    JSON\_DATA:Category[0]:CategoryName,

    JSON\_DATA:Category[1]:CategoryName

FROM Authors;

Querying individual JSON elements using index from database table

The novels of the authors are categorized into two different types in the JSON data. The details of each category can be accessed using the index [0], [1].. notation as shown above.

The outer quotes in the column data can be removed by using **::** notation which lets you define the end data type of the values being retrieved. Notice how in this example, the outer quotes “ are removed.

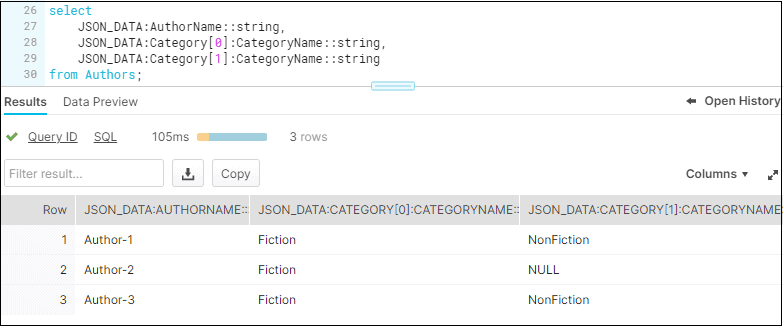
SELECT

    JSON\_DATA:AuthorName::string,

    JSON\_DATA:Category[0]:CategoryName::string,

    JSON\_DATA:Category[1]:CategoryName::string

FROM Authors;

Casting the datatype of fields while querying JSON data

Further more details of author can be drilled down as shown below.

SELECT

    JSON\_DATA:AuthorName::string,

    JSON\_DATA:Category[0]:CategoryName::string,

    JSON\_DATA:Category[0]:Genre[0]:GenreName::string,

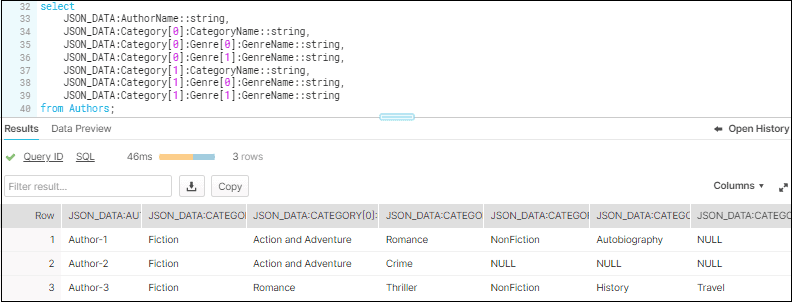
    JSON\_DATA:Category[0]:Genre[1]:GenreName::string,

    JSON\_DATA:Category[1]:CategoryName::string,

    JSON\_DATA:Category[1]:Genre[0]:GenreName::string,

    JSON\_DATA:Category[1]:Genre[1]:GenreName::string

FROM Authors;

Querying details of each category from JSON data

Unfortunately, this approach is not ideal. As the data increases, you need to add additional levels of category and genre in the query statement specifying the index values. Using the **:** and **[]** notation alone is not sufficient to dynamically get every object in an array.

**Flattening Arrays in JSON data**

Flattening is a process of unpacking the semi-structured data into a columnar format by converting arrays into different rows of data.

Using the **LATERAL FLATTEN** function we can explode arrays into individual JSON objects. The **input** for the function is the **array in the JSON structure** that we want to flatten (In the example shown below, the array is Category). The flattened **output** is stored in a **VALUE** column. The individual elements from unpacked array can be accessed through the VALUE column as shown below.

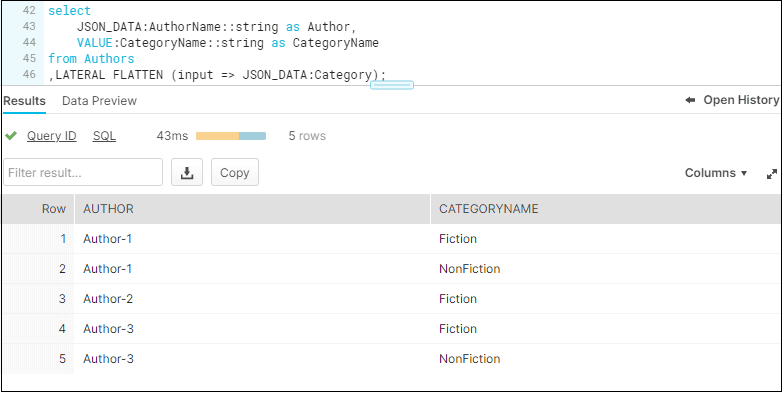
SELECT

    JSON\_DATA:AuthorName::string AS Author,

    VALUE:CategoryName::string AS CategoryName

FROM Authors

,LATERAL FLATTEN (input => JSON\_DATA:Category);

Flattening the Category array from JSON data

When there are multiple arrays which you need to flatten, it is mandatory to pass an **alias** to every input array. The VALUE column also should be used along with the alias you passed to the input array.

In our example, we need to flatten the Category, Genre and Novel arrays to get the desired output. Also note that the Novel array is present inside Genre array which is present inside Category array. So the flattened array output VALUE becomes input for the array present inside it.

The final query to get the desired output is as below.

SELECT

    JSON\_DATA:AuthorName::string AS Author\_Name,

    Flatten\_Category.VALUE:CategoryName::string AS Category\_Name,

    Flatten\_Genre.VALUE:GenreName::string AS Genre\_Name,

    Flatten\_Novel.VALUE:Novel::string AS Novel\_Name,

    Flatten\_Novel.VALUE:Sales:: number AS Sales\_in\_Millions

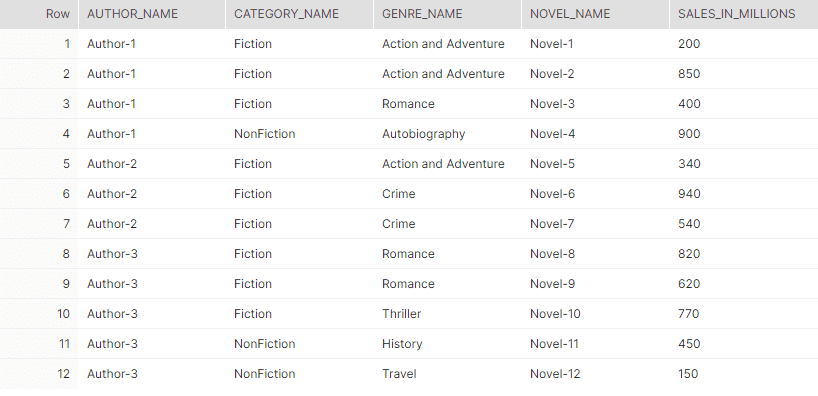
FROM Authors

,LATERAL FLATTEN (input => JSON\_DATA:Category) AS Flatten\_Category

,LATERAL FLATTEN (input => Flatten\_Category.VALUE:Genre) AS Flatten\_Genre

,LATERAL FLATTEN (input => Flatten\_Genre.VALUE:Novel) AS Flatten\_Novel

;

Final output after flattening the Category, Genre and Novel arrays in JSON data

**Summary**

Snowflake supports loading semi structured data files from external and internal stages into [database](https://thinketl.com/how-to-load-and-query-json-data-in-snowflake/) tables. Once the data is loaded into the table, it is important to understand the data structure and identify the arrays to flatten which provides the required output. The transformed data can then be loaded into another database tables with proper field names and data types easily.

This is a good example of Snowflake’s ELT features which is extremely helpful as the semi structured data can be easily transformed once loaded without the help of external ETL tools.

**Snowflake can send emails now**

To send an email notification with Snowflake, you now need only one SQL call:

call system$send\_email(  
 'my\_email\_int',  
 'felipe@example.com, dan@example.com, rajiv@example.com',  
 'This is the subject',  
 'This is the body'  
);

**Pre-requisites:**

1. A [verified Snowflake User's email address](https://docs.snowflake.com/en/user-guide/ui-snowsight-profile#verifying-your-email-address), and a valid [email notification integration object](https://docs.snowflake.com/en/user-guide/email-stored-procedures#creating-a-notification-integration) referencing it (example uses [**test@organization.com**](mailto:test@organization.com) and **MY\_EMAIL\_INTEGRATION**)
2. A [valid table](https://docs.snowflake.com/en/sql-reference/sql/create-table) name (example uses **MY\_TABLE** with three columns and a few rows)

**Example Notification Integration Object:**

The following definitions create an example email notification integration object named **MY\_EMAIL\_INTEGRATION** and a verified Snowflake User account email **address**[**test@organization.com**](mailto:test@organization.com):

/\*\*

Allowed email recipient '[test@organization.com](mailto:test@organization.com)' belongs to

a pre-existing, verified user in the Snowflake account.

To verify a Snowflake User account, follow this document:

<https://docs.snowflake.com/en/user-guide/ui-snowsight-profile#verifying-your-email-address>

\*/

CREATE OR REPLACE **NOTIFICATION INTEGRATION** **MY\_EMAIL\_INTEGRATION**

TYPE=EMAIL

ENABLED=TRUE

ALLOWED\_RECIPIENTS=('[**test@organization.com**](mailto:test@organization.com)');

**Example Table with Data:**

The following example table definition, named **MY\_TABLE**, will carry 3 columns and 2 sample data rows.

CREATE OR REPLACE TABLE **MY\_TABLE** (

COMPANY STRING,

CONTACT STRING,

COUNTRY STRING

);

INSERT INTO **MY\_TABLE** VALUES

('ACME Corporation', 'Wile Ethelbert', 'USA' ),

('Dunder Mifflin', 'David Wallace', 'USA' );

**Example Stored Procedure:**

The example stored procedure (called **MY\_EMAIL\_PROCEDURE**) is provided below in SQL definition form.

CREATE OR REPLACE PROCEDURE **MY\_EMAIL\_PROCEDURE**(

**email\_integration\_name** STRING,

**email\_address** STRING

)

RETURNS STRING

LANGUAGE PYTHON

RUNTIME\_VERSION = 3.10

PACKAGES = ('snowflake-snowpark-python', 'pandas', 'tabulate')

HANDLER = '**main**'

AS $$

import snowflake.snowpark

import pandas

def **main**(

session: snowflake.snowpark.Session,

email\_integration\_name: str,

email\_address: str

) -> str:

table\_pandas\_df: pandas.DataFrame = session.table("MY\_TABLE").to\_pandas()

table\_as\_html: str = table\_pandas\_df.to\_markdown(

  tablefmt="html",

  index=False

  )

  email\_as\_html: str = f"""

  <p>Today's report of companies</p>

  <p>{table\_as\_html}</p>

  """

success: bool = session.call(

"SYSTEM$SEND\_EMAIL",

email\_integration\_name,

email\_address,

'Example email notification in HTML format',

email\_as\_html,

  'text/html'

)

  return "Email sent successfully" if success else "Sending email failed"

$$;

**Example Call:**

The following invokes the stored procedure defined above, referencing the example values created earlier:

CALL **MY\_EMAIL\_PROCEDURE**(

**'MY\_EMAIL\_INTEGRATION'**,

**'**[**test@organization.com**](mailto:test@organization.com)**'**

);

**Example Email:**

The following is a screenshot of the received HTML email in Gmail, an email client app that supports rendering HTML emails:



**Additional Information**

**Caveats:**

* This solution relies on the use of a Pandas library feature. Since Pandas DataFrames are held in-memory, the table or query result data that needs to be sent must be limited to a few rows.
* Email text values must not be large. Restrict the table or query's rows to a small number of only most important rows. Otherwise there may be errors in sending or in delivery of the email.
* The solution uses third-party Python packages ([*pandas*](https://github.com/pandas-dev/pandas), [*tabulate*](https://github.com/astanin/python-tabulate), etc.) from the Snowflake Anaconda channels and [requires its terms and conditions signed](https://docs.snowflake.com/en/developer-guide/udf/python/udf-python-packages#getting-started) before it can be used.
* Email delivery is an asynchronous action, and delivery depends on the recipient's email servers as well as their inbox permitting contents. Email notification deliveries, from the sender's perspective, can be tracked through the [NOTIFICATION\_HISTORY information schema table function](https://docs.snowflake.com/en/sql-reference/functions/notification_history).

Other Method for Email

How to create a notification integration

To create a notification integration, you can run the following command:

CREATE NOTIFICATION INTEGRATION my\_email\_int

TYPE=EMAIL

ENABLED=TRUE

ALLOWED\_RECIPIENTS=('joe.doe@my\_domain.com','another.joe@my\_domain.com');

The **CREATE NOTIFICATION** command has 3 parameters:

* **TYPE:** what kind of notification integration we want (EMAIL, QUEUE)
* **ENABLED:** - TRUE or FALSE
* **ALLOWED\_RECIPIENTS:** up to 10 verified email addresses where we want to deliver the email message

Once we have the notification integration in place, users might want to grant the usage privilege to some other [**roles**](https://select.dev/posts/snowflake-roles) so they can use it for sending emails.

GRANT USAGE ON INTEGRATION my\_email\_int TO ROLE my\_developer\_role;

Once a notification integration object is created with at least one verified email address, users can now send an email.

Manually Triggering Emails from Snowflake

The **SYSTEM$SEND\_EMAIL** procedure is invoked with **CALL** keyword, just like any other stored procedure. Here's an example command:

CALL SYSTEM$SEND\_EMAIL(

'my\_email\_int',

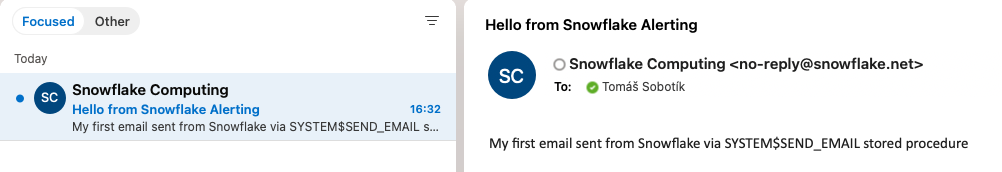
'joe.doe@my\_domain.com,

'Hello from Snowflake Alerting',

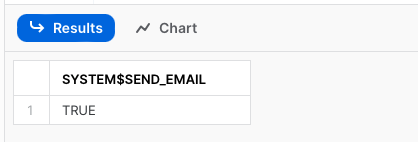
'My first email sent from Snowflake via SYSTEM$SEND\_EMAIL stored procedure'

You can notice that we define the email address again in function call. If an email address that was not in the **ALLOWED\_RECIPIENTS** property defined in the notification integration, no email would be sent.

Here's what the email looks like from the command above:



If the stored procedure executes successfully, it returns **\*\*TRUE\*\***:



Automatically Triggering Emails from Snowflake

Most Snowflake customers will want to have automated alerting in place. How can users achieve this?

A common deployment pattern is to embed the **SYSTEM$SEND\_EMAIL** stored procedure in another procedure. Typically, stored procedure fill first validate if some condition is met, and if it is, trigger the **SYSTEM$SEND\_EMAIL** procedure to send the emails.

What kind of conditions might be checked?

* Monitoring Snowflake tasks by checking if the task has been suspended
* Monitoring Snowflake streams to get notified if a stream has become stale too soon
* Validating specific business roles (i.e. users must be a paying customer in order to access feature X)

Let’s use the Snowflake task monitoring as an example. You might have a task or tree of tasks which is supposed to be running on a given schedule. You want to know if a task has become suspended and therefore stopped running. This can happen whenever a task is modified, since users have to suspend the task prior to making the change, and resume it again.

To achieve the detection of suspended tasks, we'll create a stored procedure called **task\_state\_monitoring** which will be monitoring the state of the given task and if it finds it in suspended state, it will send you an email. This procedure lists all the tasks using the **show tasks** command, then loops through each task and checks if the task's current **state** variable is equal to **'suspended'**. An email will be sent for each task that is suspended.

The code of such procedure can look like this:

create or replace procedure task\_state\_monitor(task\_name string)

returns varchar not null

language SQL

AS

$$

DECLARE

task\_state string;

c CURSOR FOR SELECT "state" from table(result\_scan(last\_query\_id())) where "name" = ?;

BEGIN

show tasks;

open c USING (task\_name);

fetch c into task\_state;

IF(task\_state = 'suspended') THEN

CALL SYSTEM$SEND\_EMAIL(

'my\_email\_int',

'joe.doe@mydomain.com',

'Email alert: Task is suspended!',

'Please check the task state.'

);

RETURN 'Email has been sent.';

ELSE

RETURN 'Task state is ok.';

END IF;

END;

$$

;

To deploy this stored procedure, users can create a [**Snowflake task**](https://docs.snowflake.com/en/user-guide/tasks-intro) which will call this **task\_state\_monitoring** procedure according any desired schedule (i.e. every 5 mins, 2 times per day, etc.).

Snowflake Alerts

Another powerful notification feature is Snowflake Alerts. This is schema level object which allows you to react on different situations in your data, or your Snowflake account in general.

Cost Alerting is common use case for Snowflake Alerts. For example, an alert can be created to detect if an account's credit usage spikes for a single virtual warehouse, or any other Snowflake service. Another use case could be validation of some business rules and getting a notification if the validation fails. Alerts are not limited to just triggering an email. For example, whenever the validation of a given rule does not pass, you add the record into a log table that contains the timestamp, the validation type and a failure description.

Snowflake Alert consists of three components:

* A **condition** that triggers the alert (e. g. query returns no records)
* An **action** that defines what should happen in case condition is met (e.g. send email, insert record into the table)
* A **frequency** setting that defines how often the condition evaluated (e.g. every 24 hours)

How to create Snowflake Alert?

For this exmaple, we'll create a Snowflake Alert to notify us via email if the daily Snowflake credit consumption exceeds 100 credits.

Here is the code for the alert which runs every hour:

CREATE OR REPLACE ALERT credits\_consumption

WAREHOUSE = COMPUTE\_WH

SCHEDULE = 'USING CRON 0 1 \* \* \* UTC'

IF( EXISTS (

select

1

from

snowflake.account\_usage.metering\_history

where

start\_time >= SNOWFLAKE.ALERT.LAST\_SUCCESSFUL\_SCHEDULED\_TIME()

AND SNOWFLAKE.ALERT.SCHEDULED\_TIME()

group by service\_type

having sum(credits\_used) > 100

))

THEN

CALL SYSTEM$SEND\_EMAIL(

'my\_email\_int',

'joe.doe@mydomain.com',

'Credit consumption warning',

'Please be aware that daily credit consumption is out of your specified limit of 100 credits per day'

);

Let’s go through the code and explain it:

* We have to define a virtual warehouse, **COMPUTE\_WH**, which will be used for running the Alert.
* We defined a schedule using the CRON syntax to have the alert run daily at 1AM UTC
* As a condition we have a SQL query which returns a single row containing the value "1" if the Snowflake credit consumption is over 100 in last 24 hours
* As an action we call the SYSTEM$SEND\_EMAIL procedure.

Once the Alert is successfully created we have to resume it, otherwise it will not run:

**alter alert credits\_consumption resume;**

Now the Alert is in operational mode.

Alerts Monitoring

Snowflake offers an [**ALERT\_HISTORY view**](https://docs.snowflake.com/en/sql-reference/account-usage/alert_history) in the **ACCOUNT\_USAGE** which can be used for monitoring of the alerts execution:

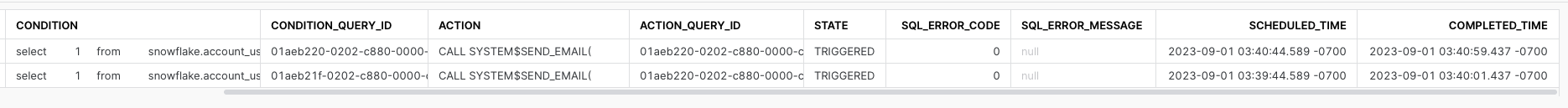
SELECT \*

FROM snowflake.account\_usage.alert\_history

ORDER BY SCHEDULED\_TIME DESC

;

The view contains an overview about each Alert execution together with the **query\_id**s associated with the query run for to validate the alert condition (**CONDITION\_QUERY\_ID**) and the query run for the associated action (**ACTION\_QUERY\_ID**, i.e. the query that triggers the **SYSTEM$SEND\_EMAIL** procedure).



Please note that action and condition queries are not visible in the **QUERY\_HISTORY** and if you want to get them you have to use the **RESULT\_SCAN** function:

select \* from table(result\_scan('<action\_query\_id>'));

How are Snowflake Alerts Billed

There is no extra payment for sending emails or for the Snowflake Alerts themselves. Users only pay for the compute time required to run the validation and action queries on the specified virtual warehouse.

Advantages of Snowflake Alerts

Snowflake Alerts vastly simplify the process of validating business logic and triggering some action on top of that. To trigger email notifications using the **SYSTEM$SEND\_EMAIL** procedure, we had to create a separate stored procedure which explicitly executed some SQL, processed the results and performed the validation checks before triggering the email. Then, we had to create a separate task that invoked the stored procedure on a schedule.

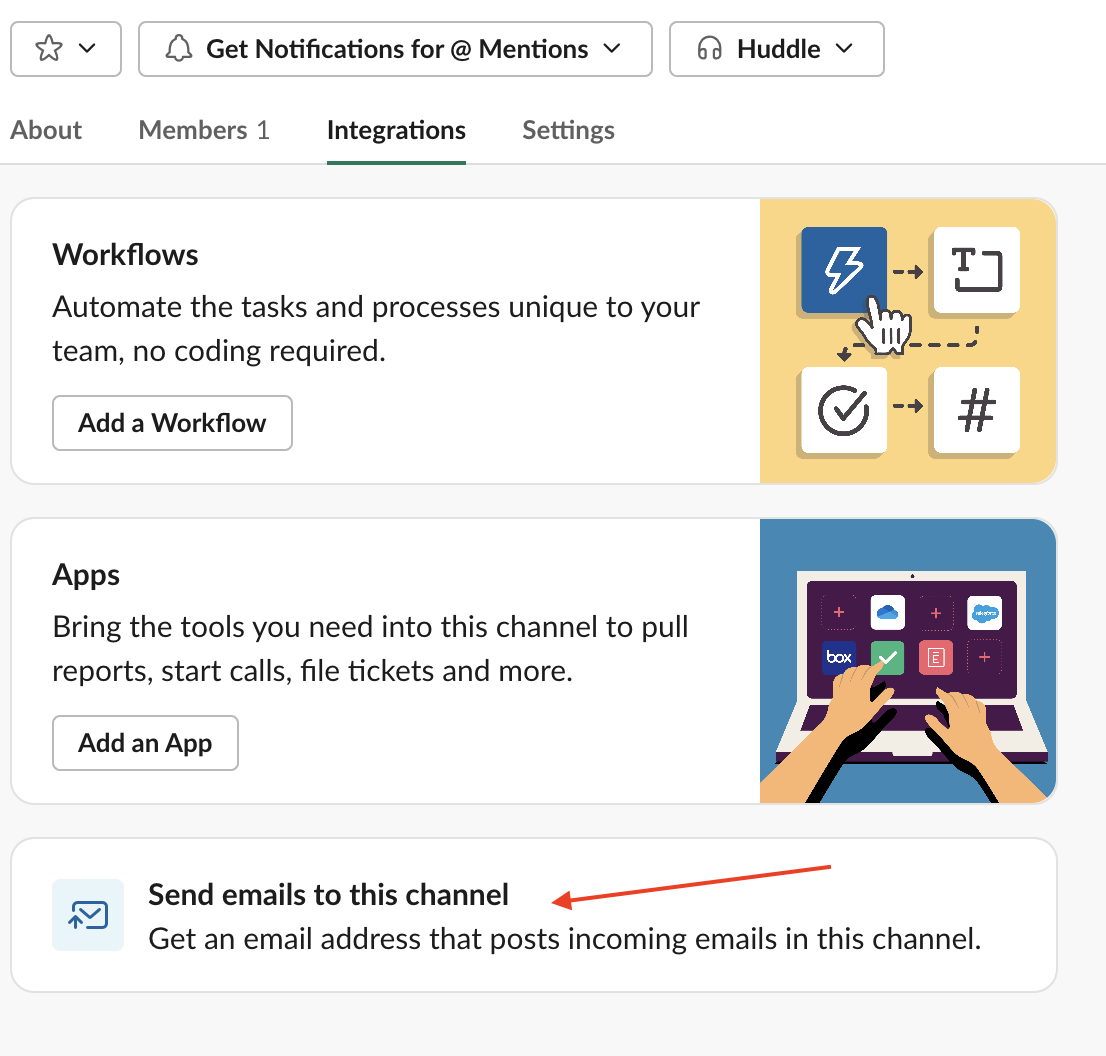
With Snowflake Alerts, all of these steps are abstracted into the creation of a single alert object.

How to send notifications to Slack from Snowflake?

Many users prefer to receive their notifications in a team Slack channel instead of via email. This helps enable quicker collaboration and resolution of the notification. Since Slack supports sending emails to a channel through a specific email address, we can setup our Snowflake notification integration using the email address dedicated to this channel.

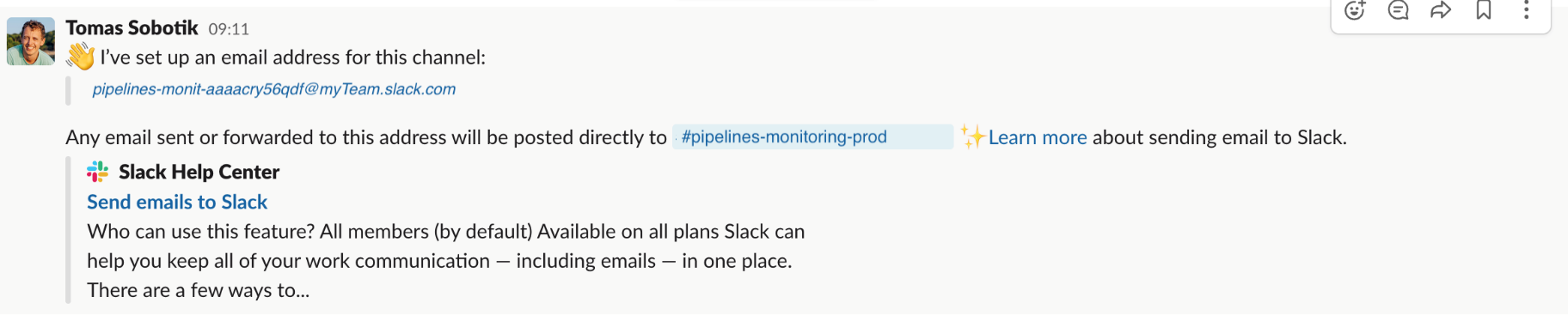
Step 1: Create email address for Slack channel

First, we need to create a new integration in our Slack channel to receive an email address we can use. Under the channel settings, click "Integrations" then "Send emails to this channel":



Step 2: Receive Slack confirmation

After completing this step, Slack will automatically send you a message into your channel:

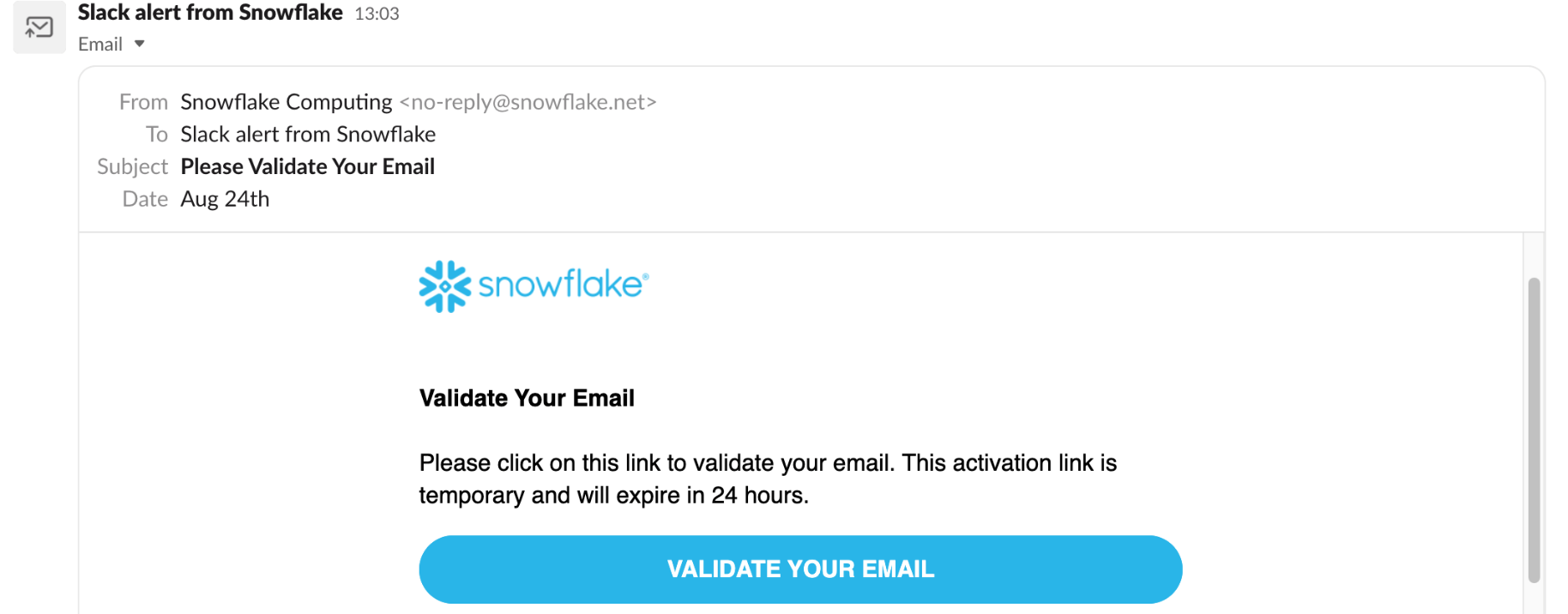


Step 3: Create Snowflake user with Slack email address

Since Snowflake only allows emails to be sent to **verified Snowflake users**, we have to create a new user in Snowflake and assign it this email address from Slack. Then we must verify the email address.

Step 4: Verify the Snowflake email in Slack

The verification email will be sent from Snowflake directly to the Slack channel, where you can click on the link and verify the email address for future usage in the **SYSTEM$SEND\_EMAIL()** stored procedure. Here's what the Snowflake verification email will render in Slack as:



How to trigger other notification types from Snowflake

Previously, when creating the notification integration we set the **TYPE** parameter to have a value of **EMAIL**. But, this parameter also supports **QUEUE** as an option. When set to **QUEUE**, users can send notifications from Snowflake to other cloud messaing services like Amazon SNS, Google Pub/Sub, or Microsoft Azure Event grid. Pub/Sub, or Microsoft Azure Event Grid.

This functionality works well in conjunction with the **ERROR\_INTEGRATION** parameter on Snowflake Tasks or Snowpipe. The **ERROR\_INTEGRATION** parameter accepts a notification integration with a **QUEUE** type. To learn more about this functionality, check out the blog post on [**Error Notifications for Snowflake tasks**](https://select.dev/posts/error-notifications-for-snowflake-tasks).

**Snowpipe**

What is Snowpipe?

Snowpipe is a fully managed data ingestion service provided by Snowflake.

If you’re familiar with batch data loading using **the COPY command**, you can think of Snowpipe as an "automated copy command." Snowpipes are a first-class Snowflake object, meaning you create and manage them via SQL like any other Snowflake object.

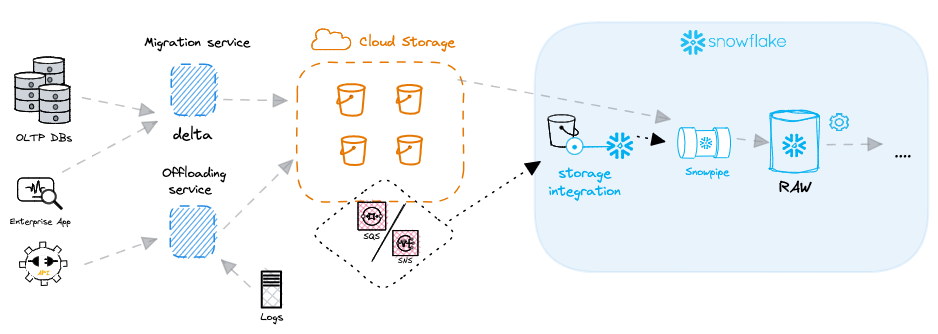
Snowpipe automatically loads files from an external stage based on notifications about newly landed files. The delivered notification triggers processing on the Snowflake side, where Snowflake runs the COPY command defined in the Snowpipe. Notifications are based on the notification service on the cloud provider side, such as AWS SQS/SNS.

What is the difference between Snowpipe and COPY?

The main difference lies in the compute model and automation. Snowpipe is a serverless feature, meaning you don't need to worry about the virtual warehouse for running the Snowpipe code (sizing, resuming, suspending, etc.). Snowflake automatically provides a compute cluster for Snowpipe. In terms of automation, the COPY command requires scheduling, ensuring the command runs at an exact time. In contrast, Snowpipe is triggered automatically based on received notifications, resulting in lower latency.

How to create a Snowpipe?

Before creating a Snowpipe, it's important to understand the overall data loading architecture. The Snowpipe object does not work in isolation. In addition to it, you will also need [**storage integration**](https://select.dev/posts/snowflake-batch-loading#storage-integration), [**stage**](https://select.dev/posts/snowflake-batch-loading#stage), and [**file format**](https://select.dev/posts/snowflake-batch-loading#file-format) definitions. We covered the creation of those objects in the previous post on [**batch data loading**](https://select.dev/posts/snowflake-batch-loading).



Once the necessary stage, storage integration and file format objects are created, a Snowpipe object can be created with the following code:

CREATE OR REPLACE PIPE mypipe

AUTO\_INGEST = TRUE AS

COPY INTO snowpipe\_landing\_table

FROM @my\_s3\_stage/snowpipe/

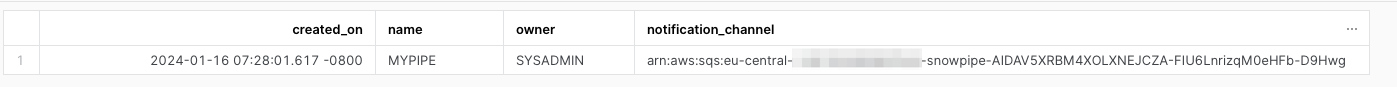
FILE\_FORMAT = csv\_file\_format;

One of the important parameters to note here is **AUTO\_INGEST**, which specifies whether you want to load files based on received notifications (**TRUE**) or call Snowpipe REST API with a list of files for ingestion (**FALSE**).

Configuring event notifications for Snowpipe in AWS

In addition to the Snowpipe object definition, you must also configure a notification integration. In order to be able to automatically load files, Snowpipe needs to receive notifications from the cloud provider about new files.

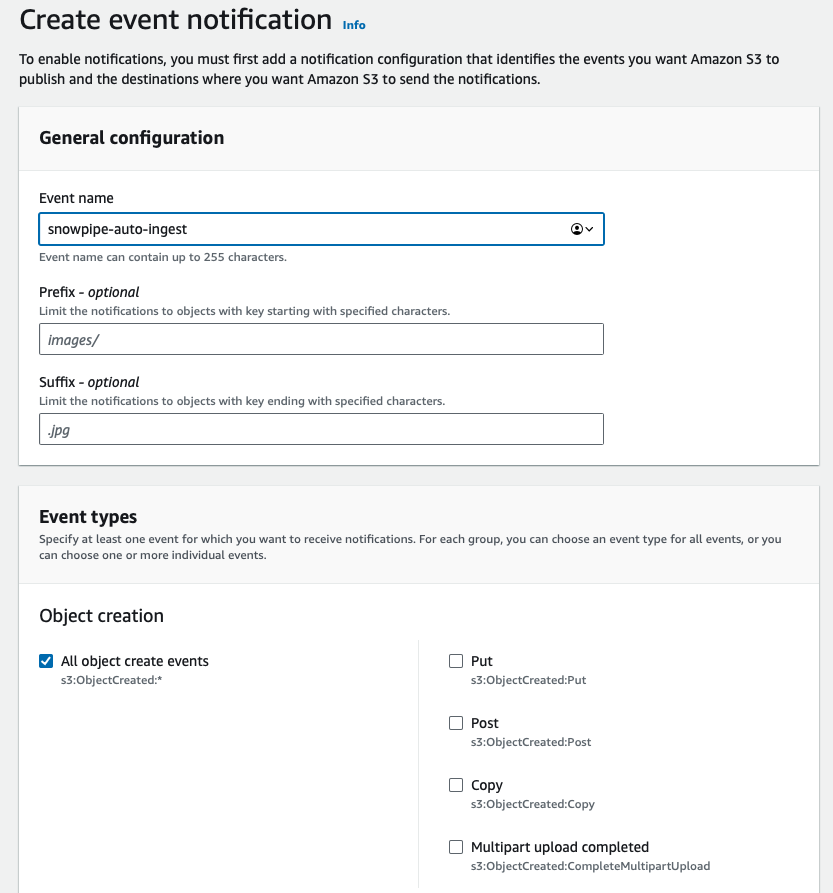
Once you create a Snowpipe object with **AUTO\_INGEST = TRUE**, Snowflake automatically assigns a notification channel to it. If you are using Amazon Web Services (AWS), Snowflake uses [**Amazon Simple Queue Service (SQS)**](https://aws.amazon.com/sqs/) for receiving notifications. The SQS ID value can be found in the **notification\_channel** column in the output of **DESC PIPE mypipe**.



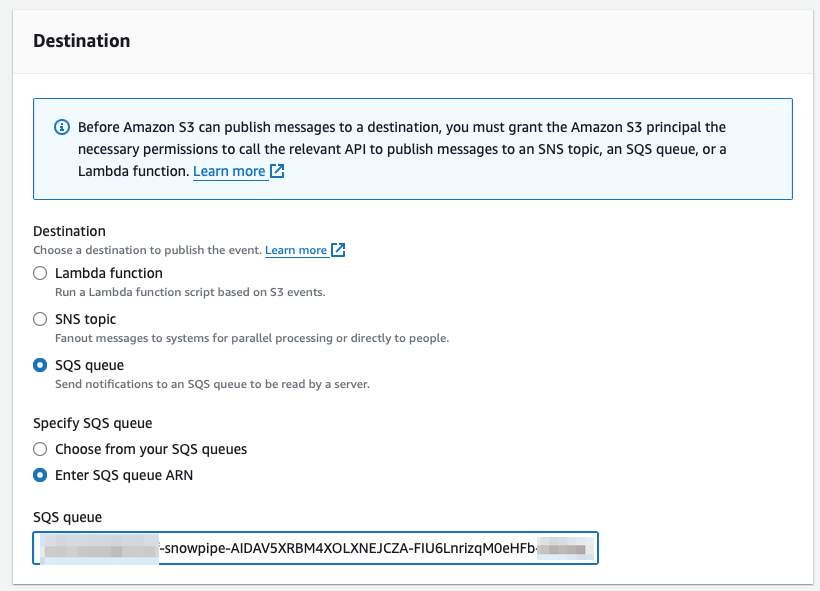
For Snowpipe auto ingest to work, notifications about new files need to be sent to this queue. Luckily for us, this system can be built using the event notification feature for S3 to send a messages to the queue.

Create event notification

Open the S3 bucket containing the files you want to load with Snowpipe. Go to the properties tab and find the configuration for event notification. Create a new one with the desired name, and possibly configure the prefix to limit the notification to relevant files. In case you have multiple directories and Snowpipe should load files only from a single one, define it to [**reduce cost**](https://select.dev/posts/snowflake-cost-optimization), latency, and event noise.



Scroll to the bottom of the screen where the destination configuration is. Select SQS and paste the value you got from **DESC PIPE mypipe** command shown above.



And that’s it! Now whenever there is a new file landed in S3, Snowpipe will be notified and can trigger the **COPY** command automagically 🪄.

Managing Snowpipes

There are a bunch of helpful commands to help you manage Snowpipes. Let’s deep dive into them

Listing & Describing Snowpipes

We have already used **DESC PIPE mypipe** above, providing basic information about the given pipe. The same output, but not filtered for a single pipe, can be provided by the **SHOW PIPES** command.

Snowpipe Status

A the **PIPE\_STATUS** system function provides an overview of the current pipe state. The output includes several values such as the current state, information about the last ingested file, and whether there are any pending files.

To get such an overview, run **SELECT SYSTEM$PIPE\_STATUS('mypipe')**;

The function output is a [**JSON structure**](https://select.dev/posts/snowflake-json-data-parsing):

{

"executionState": "RUNNING",

"lastForwardedFilePath": "source\_bucket/ingestion/snowpipe/orders-3.csv",

"lastForwardedMessageTimestamp": "2024-01-16T15:45:00.262Z",

"lastIngestedFilePath": "orders-3.csv",

"lastIngestedTimestamp": "2024-01-16T15:45:00.101Z",

"lastPulledFromChannelTimestamp": "2024-01-26T11:01:14.901Z",

"lastReceivedMessageTimestamp": "2024-01-16T17:52:29.645Z",

"notificationChannelName": "arn:aws:sqs:eu-central-1:XXXXXXXXXXXX:sf-snowpipe-AIDAV5XuBMEXOLXcEJCZA-FIU6Lnri5qM0eVFb-D9Hwm",

"numOutstandingMessagesOnChannel": 0,

"pendingFileCount": 0

}

Pausing a pipe

A pipe has an execution state. When you create a Snowpipe, it automatically gets a **RUNNING** state. There's no need to activate it like you do with a [**Snowflake task**](https://select.dev/posts/snowflake-tasks). However, there might be situations when you would like to pause the pipe for a while:

* Changing the ownership of the pipe
* Manipulating with the files in the source directory
* Testing the upstream process generating the files
* To pause the execution of the pipe, Snowflake has the parameter **PIPE\_EXECUTION\_PAUSED**.

Pause the execution of a Snowpipe with the ALTER statement:

ALTER PIPE MYPIPE SET PIPE\_EXECUTION\_PAUSED = TRUE;

This ALTER statement changes the state of the pipe to a **PAUSED** state. New files could still be delivered to the stage directory, but they won’t be processed until the pipe is resumed. Be aware that a pipe could become stale if it is paused for longer than the retention period for received event messages (14 days by default). To resume a pipe, change the parameter back to **FALSE**:

ALTER PIPE MYPIPE SET PIPE\_EXECUTION\_PAUSED = FALSE;

Snowpipe Error Notifications

Snowpipe can also be integrated with cloud messaging services (e.g. AWS SNS) and send notifications in case of any failures. The operations team can react to such notifications and resolve the issue before it would be spotted by business users. To enable error integration, there are several configuration steps on both the Snowflake and cloud provider sides:

* Create AWS SNS Topic
* Create AWS IAM Policy
* Create AWS IAM Role
* Create the Notification Integration (Snowflake side)
* Grant Snowflake access to the SNS topic
* Enable error Notification in Snowpipe

To enable error notification in Snowpipe, you can use the ALTER PIPE command:

ALTER PIPE mypipe SET ERROR\_INTEGRATION = my\_notification\_int;

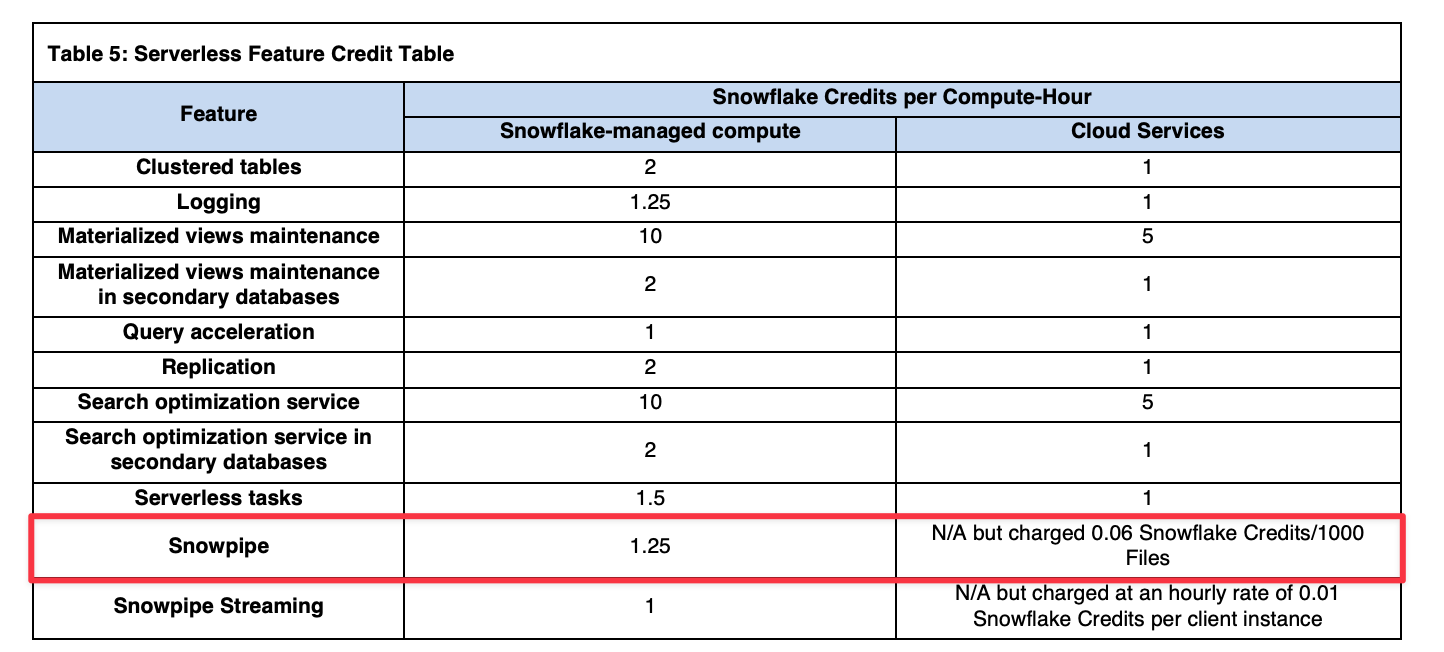
For a deep dive into how to set this up, you can review our previous posts on [**creating error notifications for Snowflake Tasks**](https://select.dev/posts/error-notifications-for-snowflake-tasks) or our more general overview on [**Snowflake Alerting**](https://select.dev/posts/snowflake-alerts).

Snowflake Snowpipe Costs

With [**Snowflake cost management**](https://select.dev/posts/snowflake-cost-management) being an important priority for all Snowflake customers, it's important to understand how Snowpipe is billed.

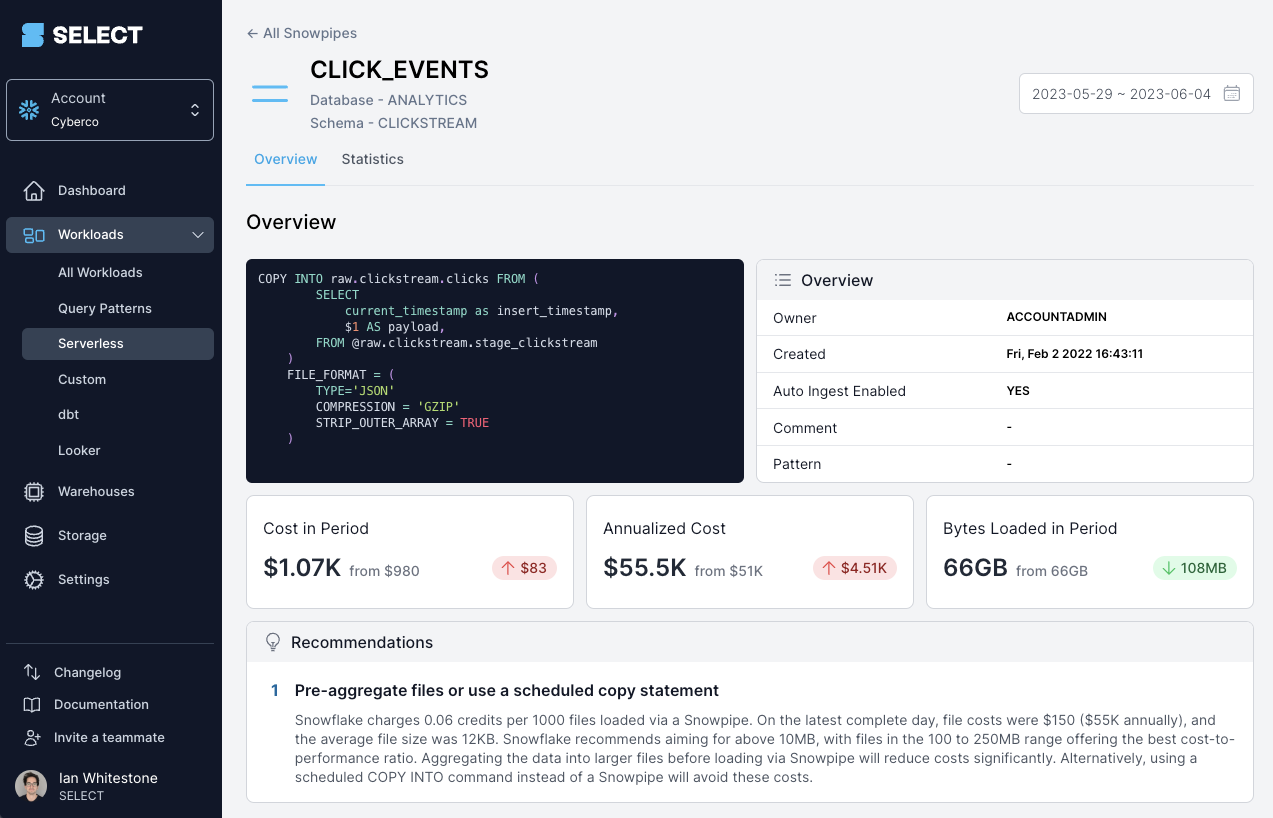
Snowpipe is a serverless feature, meaning you do not need to worry about providing and sizing the compute cluster for Snowpipe loading. In Snowflake, each serverless feature uses a different pricing model than the model used for virtual warehouses.

Snowpipe compute costs are 1.25x the regular virtual warehouse compute costs (reference: [**Snowflake Credit Consumption Table**](https://www.snowflake.com/legal-files/CreditConsumptionTable.pdf)). For example, if Snowflake uses the equiavelent of an X-Small warehouse to load your data, you will be charged 1.25 credits instead of 1 credit per compute-hour.



In addition to the Snowflake-managed compute costs, there is an additional file overhead fee: a charge of 0.06 credits/1000 files loaded. As a result, it's very important to ensure [**your files are optimally sized**](https://select.dev/posts/snowflake-batch-loading#optimal-file-sizes-for-loading) when using Snowpipe.

If you're using a tool like [**SELECT**](https://select.dev/), you'll be able to esily see when there are significant Snowpipe cost optimization opportunities based on your file sizes:



Alternatively, you can identify this yourself using the metadata views discussed below.

How to monitor Snowpipe costs?

Snowflake keeps detailed metadata about your Snowpipe usage. You can use this data to get an overview of your costs. Let’s have a look at some **ACCOUNT\_USAGE** views providing Snowpipe data. If you would like to get details about all your Snowpipes, you can use the **PIPES** view and the following query:

SELECT

PIPE\_ID,

PIPE\_NAME AS NAME,

PIPE\_SCHEMA\_ID AS SCHEMA\_ID,

PIPE\_SCHEMA AS SCHEMA\_NAME,

PIPE\_CATALOG\_ID AS DATABASE\_ID,

PIPE\_CATALOG AS DATABASE\_NAME,

IS\_AUTOINGEST\_ENABLED,

NOTIFICATION\_CHANNEL\_NAME,

PIPE\_OWNER,

DEFINITION,

CREATED,

LAST\_ALTERED,

COMMENT,

PATTERN,

DELETED,

OWNER\_ROLE\_TYPE

FROM SNOWFLAKE.ACCOUNT\_USAGE.PIPES;

This provides a complete list, including deleted pipes. You can add a condition **WHERE DELETED IS NULL** to get only currently existing pipes.

Using the Snowpipe History View

To calculate Snowpipe cost, you can use **PIPE\_USAGE\_HISTORY**. Custom calculations are required to include credits charged as an additional files overhead fee.

SELECT

START\_TIME,

PIPE\_ID,

COALESCE(PIPE\_NAME, 'External table refreshes') AS NAME, *-- External table refreshes do not have a pipe name*

FILES\_INSERTED,

BYTES\_INSERTED,

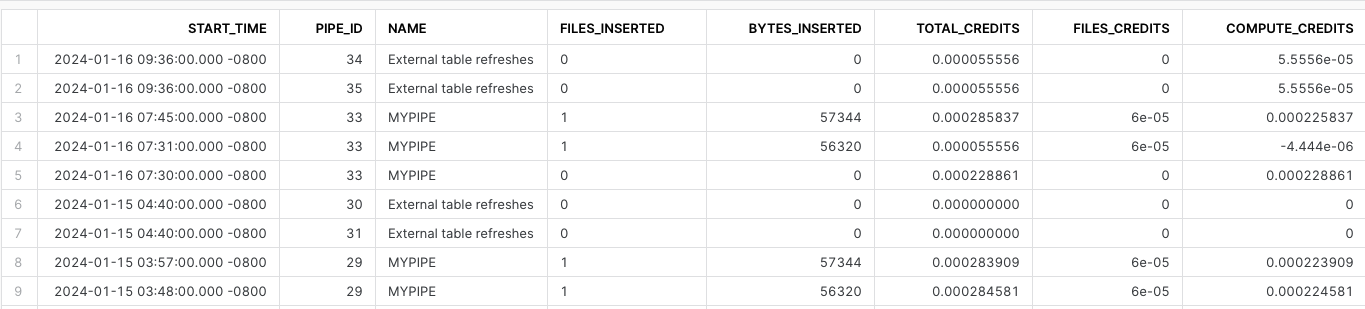
CREDITS\_USED AS TOTAL\_CREDITS,

0.06 \* FILES\_INSERTED / 1000 AS FILES\_CREDITS, *-- 0.06 credits per 1000 files*

TOTAL\_CREDITS - FILES\_CREDITS AS COMPUTE\_CREDITS

FROM SNOWFLAKE.ACCOUNT\_USAGE.PIPE\_USAGE\_HISTORY

ORDER BY START\_TIME DESC;



Snowpipe best practices

Let’s highlight a few best practices related to using Snowpipes. As discussed earlier, the most important factor for Snowpipe is to ensure your files are sized correctly. Snowpipe can be very ineffective and costly when ingesting a lot of small files. We can demonstrate that on simple example with loading of 100 GB data daily:

| Data size | idividual file size | number of files | credits/day (file overhead fee) | cost/year (file overhead fee) |
| --- | --- | --- | --- | --- |
| 100 GB | 25 KB | 4.2 million | 251 | $275k |
| 100 GB | 250 MB | 410 | 0.06 | $66 |

As you can see, loading a huge amount of very small files can lead to high costs related only to the file overhead fee. When we size files according to the recommendation (100 - 250 MB), the yearly cost is insignificant.

A similar recommendation is related to using compressed file formats (e.g., gzip) over uncompressed (e.g., CSV). Snowflake performs better on compressed formats, and you have other benefits like transferring significantly lower amounts of data over the network + storage requirements are also lower.

I would also mention implementing filters for file notifications to limit sent notifications only to relevant files and not everything happening inside your bucket.

Advantages and Disadvantages of Snowpipe

To wrap up, let's discuss the main benefits of using Snowpipe over the COPY command.

Snowpipe offers simplicity and reduces management overhead. It brings automation and near-real-time data ingestion. With the serverless model, it can save you from challenges related to right-sizing the compute cluster. It reacts better to changing needs in workload size. For most use cases, Snowpipe will be a better option than the **COPY** command. Where it won’t work well, especially from a cost point of view, is when the files are not correctly sized.

Check out our previous post for a more detailed comparison of the different **data loading options in Snowflake**.

**Complex SQL Interview Questions :**

**1.How to display 1 to 100 Numbers with query?**

**Answer:**

Select level from dual connect by level <=100;

**Tip:** User needs to know the concept of Hierarchical queries.

**2.How to remove duplicate rows from table?(100% asked in Complex SQL Interview Questions )**

**Answer:**

**First Step:**Selecting Duplicate rows from table

**Tip:**Use concept of max (rowid) of table.

Select rollno FROM Student WHERE ROWID <>

(Select max (rowid) from Student b where rollno=b.rollno);

**Step 2:**  Delete duplicate rows

Delete FROM Student WHERE ROWID <>

(Select max (rowid) from Student b where rollno=b.rollno);

**3.How to find count of duplicate rows? (95% asked complex sql interview questions )**

**Answer:**

Select rollno, count (rollno) from Student

Group by rollno

Having count (rollno)>1

Order by count (rollno) desc;

**4.How to find Third highest salary in Employee table using self-join?(90% asked Complex SQL Interview Questions )**

**Answer:**

Select \* from Employee a Where 3 = (Select Count (distinct Salary) from Employee where a.salary<=b.salary;

**5.How to Show the Max marks and min marks together from student table?**

**Answer:**

Select max (marks) from Student

Union

Select min (marks) from Student;

**Tip :** Use the concept of union to show the max and min marks together.

**6.How to display following using query?**

**\***

**\*\***

**\*\*\***

**Answer:**

We cannot use dual table to display output given above. To display output use any table. I am using Student table.

SELECT lpad (‘\*’, ROWNUM,’\*’) FROM Student WHERE ROWNUM <4;

**7.How to display Date in DD-MON-YYYY table?**

**Answer:**

Select to\_date (Hire\_date,’DD-MON-YYYY’) Date\_Format from Employee;

**8.If marks column contain the comma separated values from Student table. How to calculate the count of that comma separated values?**

|  |  |
| --- | --- |
| Student Name | Marks |
| Amit | 30,130,20,4 |
| Sukruta | 100,20,30 |
| Sonali | 140,10 |

**Want to display output like :**

|  |  |
| --- | --- |
| Student Name | Marks Count |
| Amit | 4 |
| Sukruta | 3 |
| Sonali | 2 |

**Answer:**

Select Student\_name, regexp\_count (marks,’,’) + As “Marks Count” from Student;

**Tip:** In real scenarios, lot of times developer needs to calculate the number of commas in the column then regexp\_count function is used.

**9.How to create the Student\_1 table, which is exact replica of Student table?**

**Answer:**

Create Table Student\_1 as select \* from Student;

**10.What is Query to drop all user tables from Oracle?**

**Answer:**

To Drop all tables user needs to write simple PLSQL block

Begin

For I In

(Select \* from Tabs)    —Tabs is system table in which user get the different user defined table names.

Loop

Execute immediate (‘Drop Table  ‘||i.table\_name||’cascade constraints’);

End loop;

End;

**11.How to get number of Weekends of current month?**

**Answer:**

SELECT count (\*) AS Weekends FROM

(SELECT TRUNC (SYSDATE,’mm’) +LEVEL-1 Current\_dt

FROM Dual

CONNECT BY LEVEL <= last\_day (SYSDATE) – TRUNC (SYSDATE,’mm’) +1

)

Where TO\_CHAR (Current\_dt,’dy’) IN (‘sat’,’sun’);

Let us Fragment the Query for Understanding,

**Step 1:  Try running internal query**

SELECT TRUNC (SYSDATE,’mm’) +LEVEL-1 Current\_dt

FROM Dual

CONNECT BY LEVEL <= last\_day (SYSDATE) – TRUNC (SYSDATE,’mm’) +1;

The query will give the all the dates from first to last of current date.

**Step 2: To count the weekends.**

From all the month, we need to calculate the weekends. Weekends means the Saturdays and Sundays from the month. So here, we need to use To\_char function and ‘dy’ attribute of that function to calculate days. Therefore, we have used Where **TO\_CHAR (Current\_dt,’dy’) IN (‘sat’,’sun’);** condition.

**Therefore, Final Query will be,**

SELECT count(\*) AS Weekends FROM

(SELECT TRUNC (SYSDATE,’mm’) +LEVEL-1 Current\_dt

FROM Dual

CONNECT BY LEVEL <= last\_day (SYSDATE) – TRUNC (SYSDATE,’mm’) +1

)

Where TO\_CHAR (Current\_dt,’dy’) IN (‘sat’,’sun’);

**12.What is query to fetch last day of previous month in oracle?**

**Answer:**

Select LAST\_DAY (ADD\_MONTHS (SYSDATE,-1)) from dual;

**13.How to display the String vertically in Oracle?**

**Answer:**

SELECT SUBSTR (‘AMIET’, LEVEL, 1) FROM dual

Connect by level <= length (‘AMIET’);

Output :

A

M

I

E

T

**14.Write query to find the repeated characters from your name?**

**Answer:**

Select regexp\_count (‘AmitA’,’A’) as Repeated\_character from dual;

**15.How to display departmentwise and monthwise maximum salary?**

**Answer:**

Select Department\_no, TO\_CHAR (Hire\_date,’Mon’) as Month from Employee group by Department\_no, TO\_CHAR (Hire\_date,’mon’);

**16.How to get DDL of table in Oracle?**

**Answer:**

To get DDL user needs to use dbms\_metadata package and its get\_ddl procedure,

Select dbms\_metadata.get\_ddl (TABLE,’table\_name’) from dual;

**17.How to convert seconds in to time format?**

**Answer:**

SELECT

TO\_CHAR (TRUNC (2700/3600),’FM9900′) || ‘:’ ||

TO\_CHAR (TRUNC (MOD (2700, 3600)/60),’FM00′) || ‘:’ ||

TO\_CHAR (MOD (2700, 60),’FM00′)

FROM DUAL;

Where 2700 is seconds.

**Output:**

00:45:00

**18.How to calculate number of rows in table without using count function?**

**Answer:**

Select table\_name, num\_rows from user\_tables where table\_name=’Employee’;

**Tip:**User needs to use the system tables for the same. So using user\_tables user will get the number of rows in the table.

**19.How to fetch common records from two different tables which has not any joining condition.**

**Answer:**

Select \* from Table1

Intersect

Select \* from Table2;

**Tip:**Use Intersect keyword for fetching common records.

**20.Display 4 to 7 records from Employee table.**

**Answer:**

Select \* from (Select rownum as ‘No\_of\_Row’, E.\* from Employee E)

Where No\_of\_Row between 4 and 7;

**21.Display 10 to 15 records from Employee table.**

**Answer:**

Select \* from (Select rownum as ‘No\_of\_Row’, E.\* from Employee E)

Where No\_of\_Row between 10 and 15;

**22.What is query to fetch last record of table?**

**Answer :**

Select \* from Customer where Rowid= select max(Rowid) from Customer;

23.How to Find Companies Start with ‘W’ letter?

**Answer :**

SELECT \* FROM companies WHERE companies\_name %STARTSWITH ‘W’;