Auto-ingest Snowpipe on S3

A create, debug and architect example for JSON file ingestion

Snowpipe is an ingestion pipeline, an out-the-box component that is ready to use when you purchase Snowflake data warehouse. From a user's perspective, there is no special interface for Snowpipe. It is created and configured using SnowSQL.

Snowpipe lends itself well to real-time requirements of data, as it loads data based on triggers and can manage vast and continuous loading. Data volumes and the compute/storage resources to load data are managed by the Snowflake cloud, which is why it is promoted as a serverless feature. If it's one less thing to manage, all the better to focus our energies on our own application development!

To start with, your source data is in your S3 bucket and your target is the Snowflake table you've created.There are two ways you can use Snowpipe to move the data from source to target. But which one?

Auto-ingest

Referring to my initial mention of simple and elegant, I believe the auto-ingest feature fits the description well. If you simply want every single object that ever lands in your S3 bucket to be automatically ingested soon after it lands, then this is the solution for you. Your S3 bucket can be configured so that every time a new object is created, it raises a Snowflake SQS event that feeds the Snowpipe. Your S3 bucket event notification serves as the trigger mechanism, no code required.

REST API call

If there are business rules on particular files to load or timing of load, then call a Snowflake API at the preferred timing, to notify of a partifulcar file to ingest. On the Snowflake end, whenever the API receives a post, the payload details are added onto the SQS queue that feeds the Snowpipe. This level of trigger control then involves maintaining the additional code that calls the API.

Create an Auto-ingest Snowpipe

Security plays a major role in the complexity of the set up, but it’s for good reason. Also be aware is that deployment is a bit of a tango between Snowflake components and your own AWS services, as they are inter-dependent. Follow the guide below to get it right first time!

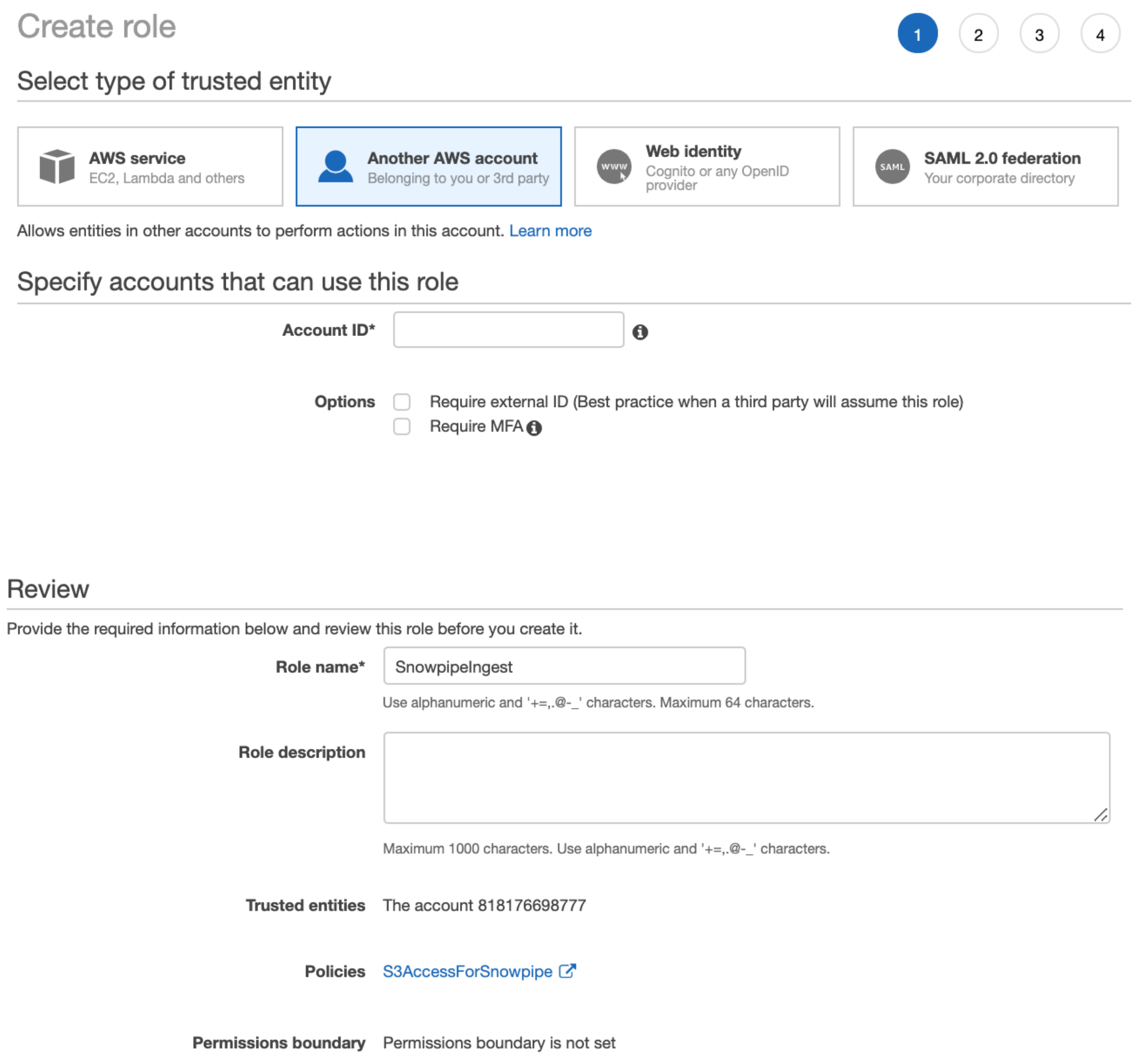
Prerequisites: It is assumed that you already have an S3 bucket and that it is private. Additionally, you know the path/prefix where your data is landing in the bucket. You will need to already have a Snowflake account and have created a landing table in the desired schema and database. For the purposes of the below *example, the table I created is defined as follows to be able to store a JSON object.*

```
CREATE TABLE MYDATA_TARGET_TABLE

(

   MYDATA_JSON variant

);
```

Step 1: In your AWS Account, create a policy to define the access permission for your S3 bucket. Here's my policy called S3AccessForSnowpipe. I have one folder in my bucket, called MyData.

```
{

   "Version": "2012-10-17",

   "Statement": [

       {

           "Effect": "Allow",
           "Action": [
               "s3:GetObject",
               "s3:GetObjectVersion"
           ],
           "Resource":
"arn:aws:s3:::demo-bucket-04042020/MyData/*"
       },
       {
           "Effect": "Allow",
           "Action": "s3:ListBucket",
           "Resource":
"arn:aws:s3:::demo-bucket-04042020",
           "Condition": {
               "StringLike": {
                       "s3:prefix": [
                            "MyData/"
                       ]
                  }
             }
        }
   ]
}
```

Step 2: Create an IAM role and attach the policy to it. Choose **Another AWS Account** as the type of trusted entity. The Account ID for your Snowflake's backend AWS Account will become known once the Snowpipe components are created. Until then, use your own AWS Account ID.


Create role

Select type of trusted entity

Allows entities in other accounts to perform actions in this account. Learn more

Specify accounts that can use this role

Account ID*	
Options	☐ Require external ID (Best practice when a third party will assume this role)
	☐ Require MFA

Review

Provide the required information below and review this role before you create it.

Role name*	SnowpipeIngest
	Use alphanumeric and '+=,.@-_' characters. Maximum 64 characters.
Role description	
	Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.
Trusted entities	The account 818176698777
Policies	S3AccessForSnowpipe
Permissions boundary	Permissions boundary is not set

Create IAM role that a Storage Integration user in the Snowflake AWS Account will assume

Step 3: In Snowflake, create a Storage Integration object. Only the ACCOUNTADMIN role will have permission to do this. Creating this object creates an IAM user on the Snowflake-managed AWS account, which will assume the role you've assigned. Storage integration can apply further control by limiting access to specific S3 paths.

The use of Storage Integration is recommended because the access parameter (i.e. the role arn) is specified once. This Storage Integration object can subsequently be used to authenticate for access to a number of your S3 paths, for multiple Snowpipes; no access keys passed.

Every time you want your Snowpipe to access additional S3 paths, alter STORAGE_ALLOWED_LOCATIONS attribute on the Storage Integration object and ensure that your S3 access policy is updated on your own account.

```
USE role ACCOUNTADMIN;

CREATE STORAGE INTEGRATION st_int

 TYPE = EXTERNAL_STAGE

 STORAGE_PROVIDER = S3

 ENABLED = TRUE
 STORAGE_AWS_ROLE_ARN =
'arn:aws:iam::<your_aws_acc_id>:role/SnowpipeIngest
'
```

```
 STORAGE_ALLOWED_LOCATIONS =
('s3://demo-bucket-04042020/MyData/')
```

Step 4: Gather key information about your Snowflake-managed AWS account

By querying the configuration for the recently created Storage Integration, you will be able to extract key information about the backend Snowflake cloud. Run the below query in Snowflake and save the values of these two parameters: STORAGE_AWS_IAM_USER_ARN and STORAGE_AWS_EXTERNAL_ID

```
DESC INTEGRATION st_int;
```

Step 5: Now that you know what the external trusted AWS account details are, go back to the IAM role you created in Step 2 and update it. Update the Trusted Relationship policy as follows:

```
{
```

```
 "Version": "2012-10-17",
 "Statement": [
   {
     "Sid": "",
     "Effect": "Allow",
     "Principal": {
       "AWS": "<STORAGE_AWS_IAM_USER_ARN>"
     },
     "Action": "sts:AssumeRole",
     "Condition": {
       "StringEquals": {
         "sts:ExternalId":
"<STORAGE_AWS_EXTERNAL_ID>"
       }
```

```
        }
```

```
    }
```

```
  ]
```

```
}
```

Step 6: Milestone moment! Create an external stage in Snowflake. An external stage is like a pointer to your S3 path. It is a window to the files that you've just set up access permissions for! In Snowflake, run this command, specifying the Snowflake role that will manage and maintain the external stage.

```
GRANT create stage ON schema TEST TO role <myrole>;
```

```
GRANT usage ON integration st_int TO role <myrole>;
```

```
USE role <myrole>;
```

```
USE schema TEST;
```

```
CREATE stage TEST.MYDATA_STAGE
```

```
 storage_integration = st_int
```

```
 url = 's3://demo-bucket-04042020/MyData/';
```

Now, check if you can see what's in your S3 bucket, from Snowflake, with the following list command against your stage name prefixed with '@'.

```
ls @MYDATA_STAGE;
```

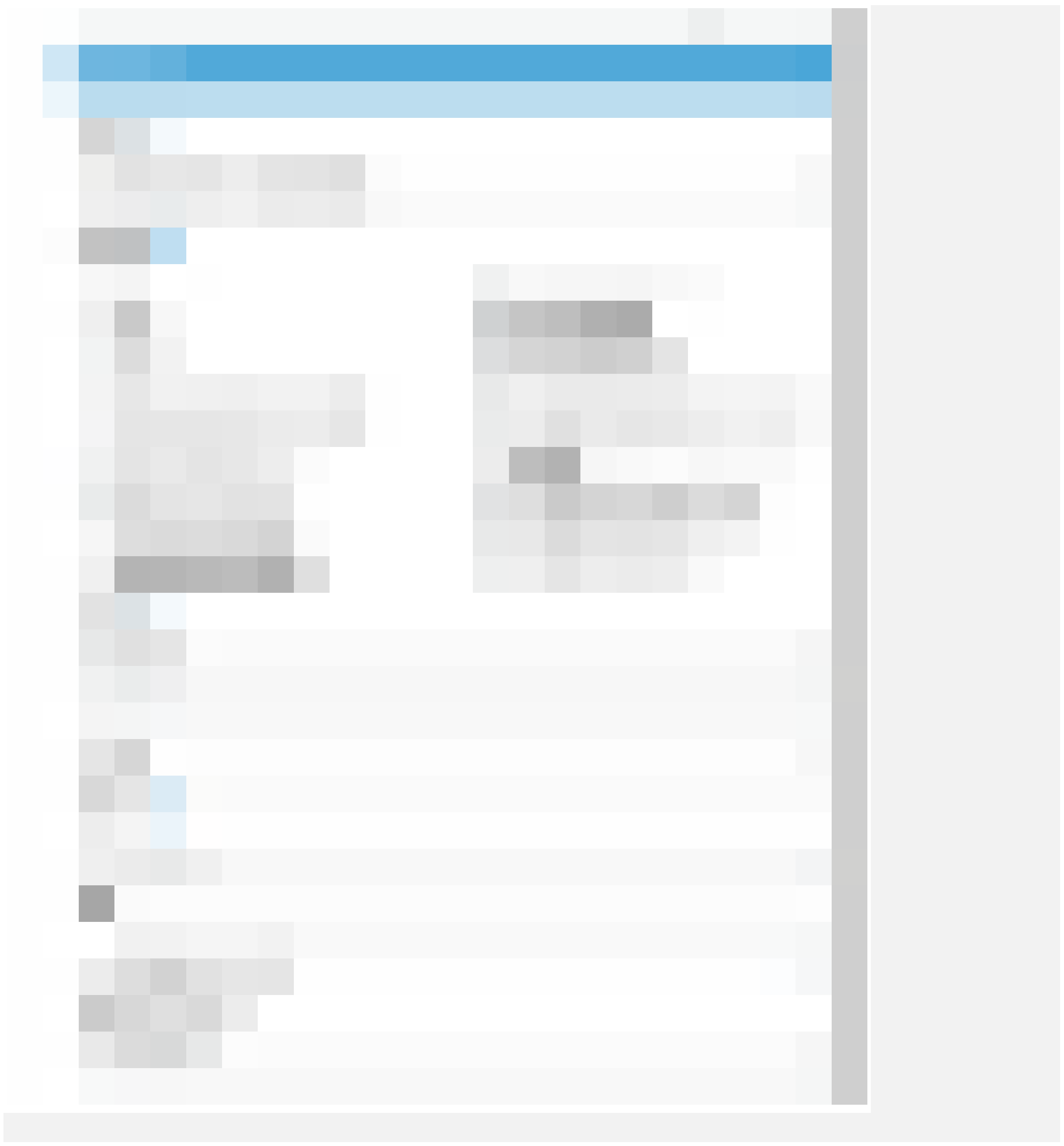
Bravo! Milestone reached if you got to this point with no errors. The next two steps finally establish the auto-ingest Snowpipe. Note that even at this stage, you could employ the COPY command to load in whatever is already in your S3 path. This might just be the one-off deployment step you need, to ingest existing files.

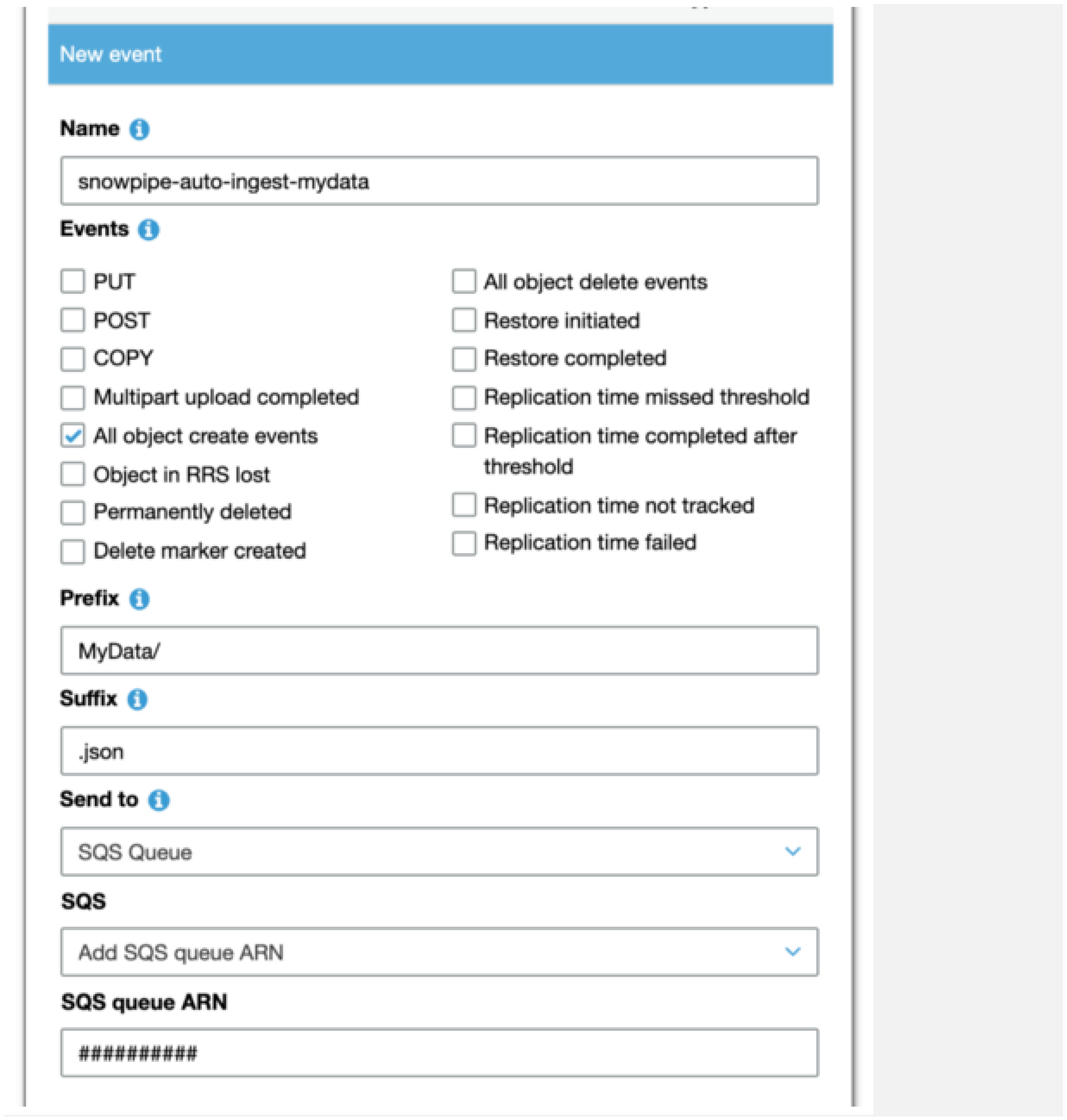
Step 7: Onwards with the Snowpipe, which is actually the easiest part of this whole set up because this is fully managed by Snowflake. The loading mechanism is still the COPY command. The pipe wraps the COPY command with additional capability to process the data load messages in its internal SQS queue. As such, the pipe specifies the copy from an external stage to the target table.

```
CREATE pipe MYDATA_PIPE auto_ingest = true AS

 COPY INTO MYDATA_TARGET_TABLE_NEW

 (

   MYDATA_JSON

 )

 FROM

   @MYDATA_STAGE

file_format = (type = json);
```

If there are multiple JSON objects in a file, consider setting the STRIP_OUTER_ARRAY property in the file format to true.

Step 8: The last step! Set up the event notifications to trigger the automatic ingestion. Go to your S3 bucket properties. Under Events, create a new event notification that sends an SQS message to the Snowflake-managed SQS queue. You'll need to specify the SQS arn. Find this out by running the SHOW pipes command and noting the arn in the **notification channel** field.




New event

Name: snowpipe-auto-ingest-mydata

Events

- [ ] PUT
- [ ] POST
- [ ] COPY
- [ ] Multipart upload completed
- [✓] All object create events
- [ ] Object in RRS lost
- [ ] Permanently deleted
- [ ] Delete marker created
- [ ] All object delete events
- [ ] Restore initiated
- [ ] Restore completed
- [ ] Replication time missed threshold
- [ ] Replication time completed after threshold
- [ ] Replication time not tracked
- [ ] Replication time failed

Prefix: MyData/

Suffix: .json

Send to: SQS Queue

SQS: Add SQS queue ARN

SQS queue ARN: ##########

```
SHOW pipes;
```

Save your event notification. Now drop a few .json files in your S3 path and check your target table about 30 seconds later. (For some handy and delicious JSON data that you can use in your test file, see the end of the article!)

```
SELECT * FROM MYDATA_TARGET_TABLE;
```

That's it, auto-ingest Snowpipe is now set up!

Debugging and monitoring a Snowpipe

So you follow the steps, drop in a file in your S3 path, check the target table, nothing… wait a minute, check again… nothing. At this point, you'd probably drop in another file, and do the checks again to no avail. You may start doubting that the S3 events trigger at all! So, where do you start?

Working through a process of elimination, here's a list of checks you can do.

1. **Potential access issue:** Assume the role that owns the pipe and see if you can list what's in the external stage. (This is only a good test if the IAM role you created has the same restrictions on ListBucket as it does for the GetObject). The SHOW GRANTS command can also be used to ensure that the owner of the pipe has at least usage permission on the stage.

```
ls @MYDATA_STAGE;
```

```
show grants on pipe MYDATA_PIPE;
```

```
show grants on stage MYDATA_STAGE;
```

2. **Pipe in error state:** Check the instantaneous state of the Snowpipe. At the same time you will be able to check other useful information such as, last received message timestamp.

```
SYSTEM$PIPE_STATUS( 'MYDATA_PIPE' );
```

3. **File error:** Check whether the pipe and the copy statement within it produced an error while processing the file. Only first error within the file is logged. It will show the exact filename and path that caused the error.

```
SELECT * FROM table(validate_pipe_load(
```

```
 pipe_name=>'MYDATA_PIPE_NEW',
```

```
 start_time=>dateadd(days, -10,
current_timestamp()),
```

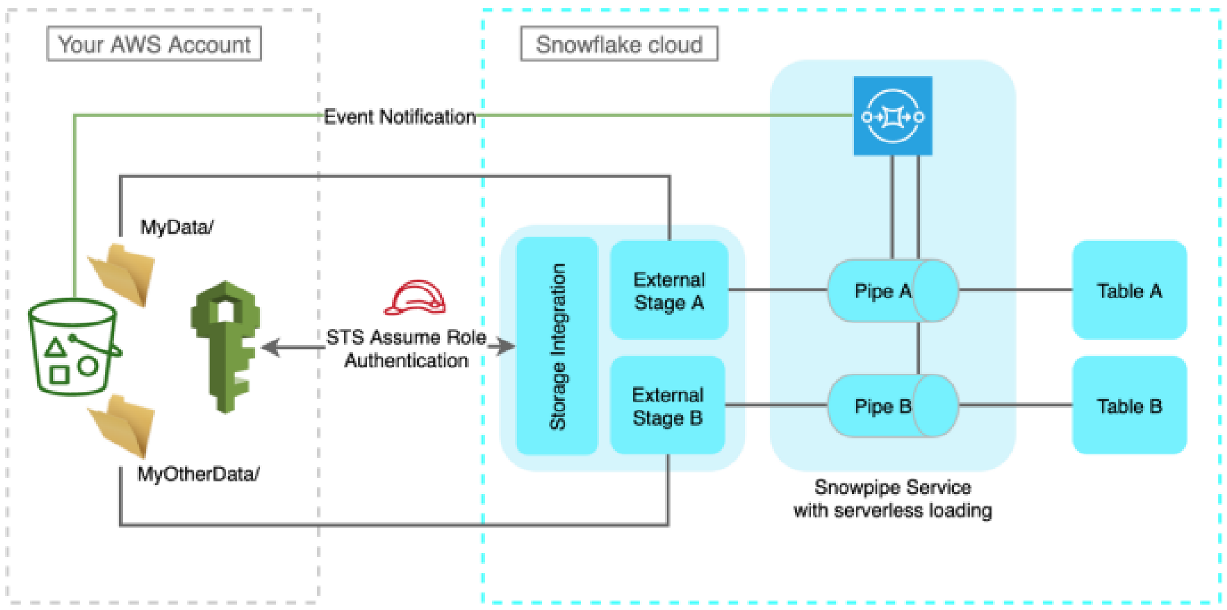
```
 end_time=>current_timestamp()));
```

4. **Check the S3 event notification trigger**: Try remove the file extension filter if you have any and see if that works. You could also create your own test SQS queue and point to that to see if your trigger works.

Architecting multiple Snowpipes

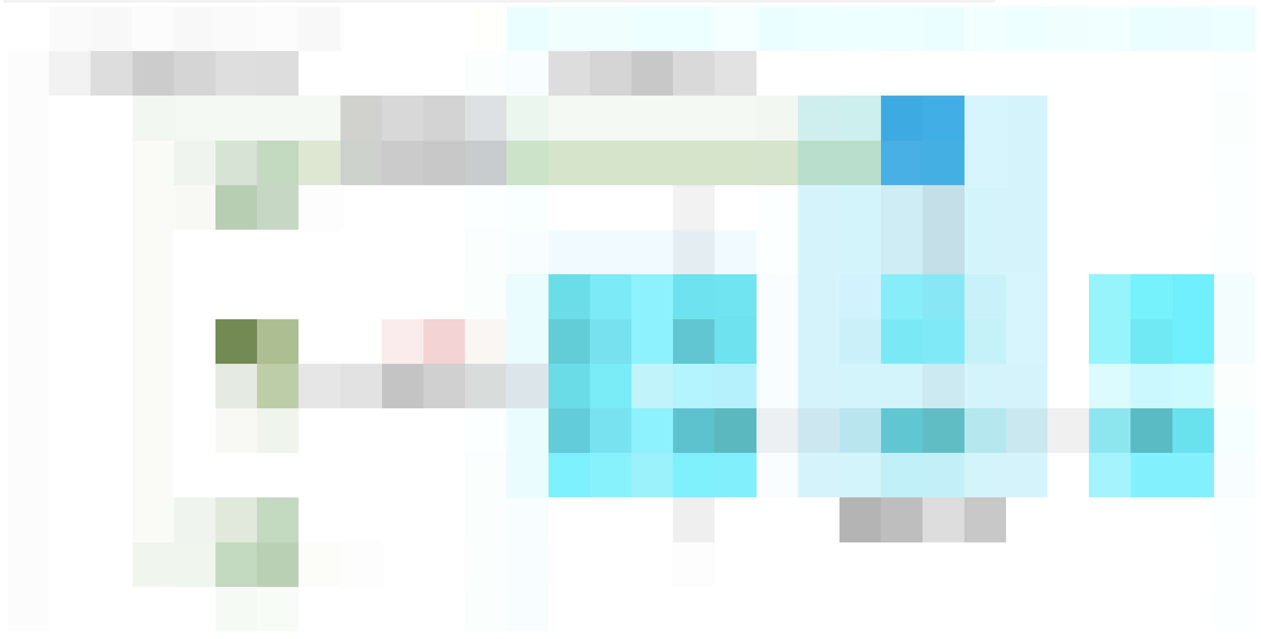
It is entirely probable that you'd want to ingest multiple/different streams of data into your data warehouse. Simply set up a separate stage and pipe for every source location.

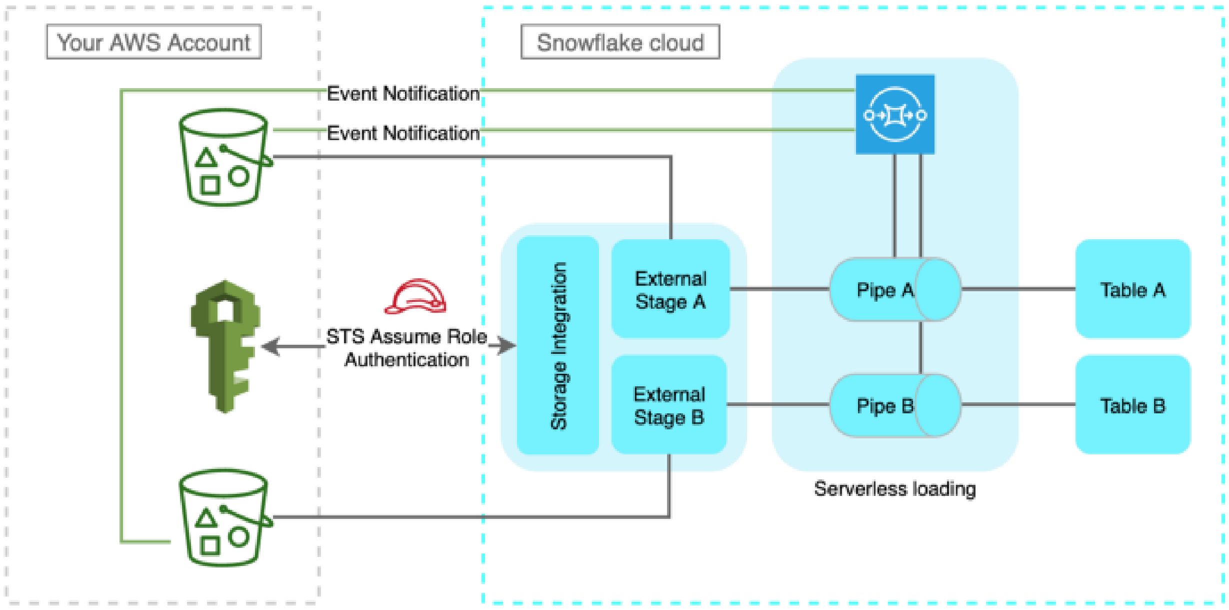
Note, every pipe you set up in your Snowflake account will be listening on exactly the same SQS arn. This may pose an issue when you try to set up event notifications on the same bucket, for multiple prefixes. S3 will not allow you to have multiple events with the same SQS endpoint. Instead, set up a single event notification, that will trigger for any new files in either of the prefixes. Notice the external stages each point to the specific prefixes. (Storage Integration settings would also need to be updated to allow access to the source S3 paths.)



Multiple Snowpipe architecture — Source data in mutiple prefixes in a single S3 bucket

Alternatively, you could arrange your source data across multiple buckets. Each bucket can then raise events against the same Snowflake SQS queue.





Multiple Snowpipe architecture — Source data in multiple S3 buckets

Lastly, should you require multiple source data locations to map to a single target table, that's fine too. In the definition of all the pipes, apply the COPY command to copy into the single target table.

Ingestion audit notes

Typically, data warehouse designers are very interested in the auditing of the ingestion process and subsequent transformation. It is critical to be able to prove process rigour as it affects the quality and usability of the data.

The closest that Snowflake comes to revealing information about Snowpipe ingestion is the COPY_HISTORY table in information_schema. This will also reveal any errors that the COPY SQL statement encountered. While the data is useful, COPY_HISTORY is limiting as it spans only the past 14 days and it is not updated real-time.

```
SELECT * FROM table(

information_schema.copy_history

  (

   table_name=>'MYDATA_TARGET_TABLE',

   start_time=> dateadd(hours, -10,
current_timestamp())

  )

);
```

I always find it useful to record the timestamp of the ingestion, alongside the data. One way is to introduce a timestamp column in the target table and load the CURRENT_TIMESTAMP into that field in the copy command of the pipe. This small design goes a long way to enabling source to target comparisons, but the reality of developing a rigorous and efficient audit framework for Snowpipe is far from simple, and it remains a topic that I am actively exploring.

Do share your constructive comments and questions, for the benefit of everyone. May your data adventures be full of fun and fancy!

Here’s the JSON data I promised; a simple recipe for one of my favourite dishes.

```
{

 "key": 1,

 "recipe_name": "Dal Makhani",

 "ingredients": [

   {"ingredient": "Whole urad dal", "qty": "1",
"uom": "cup"},

   {"ingredient": "Kidney beans", "qty": "0.3",
"uom": "cup"},

   {"ingredient": "Garlic", "qty": "2", "uom":
"cloves"},
   {"ingredient": "Grated ginger", "qty": "0.5",
"uom": "tsp"},

   {"ingredient": "Ghee", "qty": "2", "uom":
"tbsp"},

   {"ingredient": "Garam masala", "qty": "2",
"uom": "tsp"},

   {"ingredient": "Tomato paste", "qty": "3",
"uom": "tbsp"},

   {"ingredient": "Brown onion (large)", "qty":
"1", "uom": "unit"},

   {"ingredient": "Cream", "qty": "2", "uom":
"tbsp"},

   {"ingredient": "Sugar", "qty": "1", "uom":
"tsp"}

   ],

 "method": [

   {"step": "1", "description": "If using dried dal
and beans, soak overnight. Thereafter, cook in a
pressure cooker on high for 7 minutes, with 2.5
cups of water."},
   {"step": "2", "description": "Chop the onion and
garlic very finely. Saute in the ghee for 5
minutes, stirring frequently. Add tomato paste and
ginger, stir and cook until the oil separates."},

   {"step": "3", "description": "Mash half of the
cooked dal and beans with a potato masher. Pour
into the onion mixture, add in garam masala and
keep on a low simmer for 30 minutes. Add water to
reach a slightly runny consistency."},

   {"step": "4", "description": "Stir in sugar and
cream and salt to taste. Serve with chopped
coriander."}

   ],

 "portions": "6"

}
```