HANDS-ON LAB GUIDE FOR DATA ENGINEERING

To be used with the Snowflake free 30-day trial at:
https://trial.snowflake.com

Works for any Snowflake edition or cloud provider
Approximate duration: 90 minutes. Approximately 7 credits used.

Table of Contents

Lab Overview

This HOL is meant to be used to demonstrate features specifically aligned to “Data Engineering” tasks in the Snowflake platform. This HOL is an extension of the Citibike core demo and uses the “dba_citibike” Role. Ensure you have previously set up your Citibike demo environment based on the latest materials here.

Target Audience

Database and Data Warehouse Administrators and Architects

What you'll learn

This hands-on lab will demonstrate using the following features of the platform:

1. Create a Stage and configure it to automatically call **Snowpipe** using S3 event notifications. NOTE: This will require the user to create an external S3 bucket (or Azure or GCP equivalent) in order to demonstrate.
2. **Streams** - Abstraction over staging table used by Snowpipe to allow one time processing of ingested data.
3. **Tasks** (in conjunction with Stored Procedures) - Regularly scheduled execution of transformation logic over data in the stream.

Prerequisites

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

Module 1: Prepare Your Lab Environment

1.1 AWS and Snowpipe Plumbing

1.1.1 See the docs here for enabling Snowpipe in another cloud provider.
https://docs.snowflake.com/en/user-guide/data-load-snowpipe-auto.html

1.1.2 Create an AWS account if needed.

1.1.3 Create an S3 bucket to be used as an external stage for Snowflake.

1.1.4 Follow instructions in the link to configure Snowflake access to the external stage.
https://docs.snowflake.com/en/user-guide/data-load-s3-config.html#https://docs.snowflake.com/en/user-guide/data-load-s3-config.html#

1.1.5 Create S3 notifications (option 1) to automate loading into Snowpipe.
https://docs.snowflake.com/en/user-guide/data-load-snowpipe-auto-s3.html#option-1-creating-a-new-s3-event-notification-to-automate-snowpipe

Set the context for the session and create all the dependencies that will be used in this Lab.

```
use role dba citibike;
create warehouse if not exists demo_wh with warehouse_size = 'small' auto_suspend =
300 initially suspended = true;
use warehouse demo_wh;

-- create the schema
create or replace schema citibike.raw;
create or replace schema citibike.modelled;
use schema citibike.raw;
```

1.2 Creating an External Stage for Snowpipe

1.2.1 Create the streaming_data STAGE object where the data will land. See section 1.1.4 above.

```
create or replace stage streaming_data
  url = '<URLToYourCloudObjStore>'
  <Security Configuration>
  file_format=utils.json;
```

1.2.2 In order to mimic periodically arriving data, this stored proc trickle-unloads data from the TRIPS table in the RESET DB into JSON files in the stage. The procedure takes a start and stop date range and aggregates records on a daily basis and then writes the files

out to the bucket location in 5 second intervals. When it is run, it will produce a steady flow of data arriving in your defined STAGE object.

```
create or replace procedure stream_data (START_DATE STRING, STOP_DATE STRING)
returns float
language javascript strict
as
$$
var counter = 0;

// list the partition values
var days = snowflake.execute({ sqlText: `
    select
        distinct to_char(date(starttime))
    from citibike_reset_v2.public.trips
    where
        to_date(starttime) >= to_date('` + START_DATE + `')
        and to_date(starttime) <= to_date('` + STOP_DATE + `')
    order by 1;` });


// for each partition
while (days.next())
{
    var day = days.getColumnValue(1);
    var unload_qry = snowflake.execute({ sqlText: `
    copy into @streaming_data/` + day + ` from (
    select object_construct(
        'tripduration', tripduration,
        'starttime', starttime,
        'stoptime', stoptime,
        'start_station_id', start_station_id,
        'start_station_name', ss.station_name,
        'start_station_latitude', ss.station_latitude,
        'start_station_longitude', ss.station_longitude,
        'end_station_id', end_station_id,
        'end_station_name', es.station_name,
        'end_station_latitude', es.station_latitude,
        'end_station_longitude', es.station_longitude,
        'bikeid', bikeid,
        'usertype', usertype,
        'birth_year', birth_year,
        'gender', gender,
        'program_id', t.program_id,
        'program_name', program_name)
        from citibike_reset_v2.public.trips t
        inner join citibike_reset_v2.public.stations ss on t.start_station_id =
ss.station_id
        inner join citibike_reset_v2.public.stations es on t.end_station_id =
es.station_id
```

```
        inner join citibike_reset_v2.public.programs p on t.program_id = p.program_id
        where to_date(starttime) = to_date('` + day + `')
    order by starttime);` });

    counter++;

    // sleep for five seconds
    var wake = new Date();
    var now = new Date();
    wake = Date.now() + 10000;
    do { now = Date.now(); }
      while (now <= wake);
}

return counter;
$$;
```

1.3 Stored Procedure for Cleaning Up Stage Files after Successful Load

1.3.1 In this lab, data being written to the STAGE by the STREAM_DATA stored procedure will be automatically loaded into a table via Snowpipe. The procedure here will be used to clean up files that are successfully loaded. It will compare records entries in the information_schema.copy_history view with the file names still present in the stage. Files that were found to have been loaded are now safely deleted by the procedure.

```
create or replace procedure purge_files (TABLE_NAME STRING, STAGE_NAME STRING,
BUCKET_URL STRING, BUCKET_SUBDIR STRING)
returns real
language javascript strict
execute as caller
as
$$
  var counter = 0;
  var files = snowflake.execute( {sqlText: `
  select
    h.file_name
    from table(information_schema.copy_history( table_name=>'` + TABLE_NAME + `',
start_time=>dateadd(hour, -10, current_timestamp))) h
    inner join (select distinct '` + BUCKET_URL + `' || metadata$filename filename
from @streaming_data) f
        on f.filename = (h.stage_location || h.file_name)
    where h.error_count = 0;`} );

  // for each file
  while (files.next())
  {
    var file = files.getColumnValue(1);
    sqlRemove = "rm " + STAGE_NAME + BUCKET_SUBDIR + file;
```

```
    try {
        var unload_qry = snowflake.execute({ sqlText: sqlRemove });
        counter++;
    }catch(err){
        //logging here
    }

}
  return counter;
$$;
```

1.4 Finalizing Setup

1.4.1 Lastly in the setup step, give the dba_citibike role the authority to execute tasks. These will be used in the next steps to help schedule and automate the processes in our data engineering pipeline.

```
use role accountadmin;
grant execute task on account to role dba_citibike;
use role dba_citibike;
```

1.4.2 Pause to take stock for what has been built so far. An external stage pointing to an S3 bucket was created and two stored procedures have been defined

```
show stages like '%STREAMING%';
show procedures;
```

1.4.3 At any time you can run the list command to see files that have been written to the STAGE

```
list @streaming_data;
```

Module 2: Snowpipe

2.1 Leverage Snowpipe to Populate Table using S3 Event Notifications

2.1.1 Now it's time to create a table that Snowpipe will use to write the incoming data. Recall that the data being loaded is JSON formatted so create a table using the VARIANT data type.

```
create or replace table trips_raw (v variant);
```

2.1.2 Create a PIPE definition to tell Snowpipe to load the JSON records in the STAGE into the table you just created.

```
create or replace pipe trips_pipe auto_ingest=true as copy into trips_raw from
@streaming_data/;

show pipes;
```

If you haven't done so already, you need to complete the configuration in your S3 bucket to automatically invoke Snowpipe when files are written there. This is the mechanism that is used to automatically load the streaming data files. Snowpipe relies on notifications from your S3 bucket to achieve this. Instructions for completing this configuration are found in the Snowflake documentation here.

2.1.3 Confirm the setup is working by running the stream_data stored procedure to load one day's worth of data.

```
call stream_data('2019-01-01', '2019-01-01');
select system$pipe_status('trips_pipe');
```

2.1.4 Within a minute or two you should see data written to the trips_raw target table. Once you have confirmed that everything is working as expected so far, you can truncate data in the trips_raw table.

```
truncate table trips_raw;
```

2.2 Troubleshooting Snowpipe

If data isn't written to the table you can use the following commands to see where the process is broken.

2.2.1 Verify data was in fact written to your STAGE by the stored procedure.

```
list @streaming_data;
```

2.2.2 Next check the status of the PIPE using the system$pipe_status call. Make sure the result shows status ‘RUNNING’ and there are no errors present.

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

2.2.3 Lastly, check the information_schema.copy_history to see if there were any problems encountered while writing records to the trips_raw table. Parsing errors or permissions errors could accidentally affect your load jobs. You can find the details for each of the files you are loading. Play with the dateadd function to adjust the size of the window when looking at processed files.

```
select *
from table(information_schema.copy_history(
  table_name=>'citibike.raw.trips_raw',
  start_time=>dateadd(hour, -1, current_timestamp)));
```

Note the above commands aren’t strictly used for troubleshooting purposes. You can use these anytime to gain insight into the status or jobs and track the velocity of arriving data.

Module 3: Streams

3.1 Creating Trips, Stations, and Program Streams on the Raw Data

3.1.1 After confirming everything is working as expected, create multiple streams on the trips_raw table to track new trips, stations or programs records. Streams provide a convenient way to ensure that you only process new records in the table each time.

```
create or replace stream new_trips on table citibike.raw.trips_raw;
create or replace stream new_stations on table citibike.raw.trips_raw;
create or replace stream new_programs on table citibike.raw.trips_raw;

show streams;
```

3.1.2 Load 1 day of data to test the ingestion

```
call stream_data('2019-01-02', '2019-01-02');
```

3.1.3 Show the contents of the stage

```
list @streaming_data;

select $1 from @streaming_data limit 100;
```

3.1.4 Show the Snowpipe pipe watching for file create events

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

3.1.5 This is the list of files that have been loaded by Snowpipe in the last hour. Change the values in the dateadd function to control the window size for loaded data files.

```
select *
from table(information_schema.copy_history(
  table_name=>'citibike.raw.trips_raw',
  start_time=>dateadd(hour, -1, current_timestamp)))
order by last_load_time desc;
```

3.1.6 Snowpipe copies the data into our raw table…

```
select count(*) from citibike.raw.trips raw;
select * from citibike.raw.trips_raw limit 100;
```

and the insertions are tracked in the stream.

```
select count(*) from new_trips;
select * from new_trips limit 100;
```

3.1.7 Clean up the stage by calling the purge_files function. Note: be sure to leave the trailing slash on your BUCKET_URL. If the data is being written to a subdirectory in your bucket, include that information as well. If your data is not being written to a subdirectory, just provide an empty string.

```
call purge_files('trips_raw', '@streaming_data/', 's3://MyDemoBucket/',
'streaming_trips');
```

3.1.8 The data engineering process will process the streaming trips data so next create tables to store those results.

```
create or replace table trips (
  tripduration integer,
  starttime timestamp_ntz,
  stoptime timestamp_ntz,
  start_station_id integer,
  end_station_id integer,
  bikeid integer,
  usertype string,
  birth_year integer,
  gender integer,
  program_id integer);

create or replace table stations (
  station_id integer,
  station_name string,
  station_latitude float,
  station_longitude float,
  station_comment string
);

create or replace table programs (
  program_id integer,
  program_name string
);
```

The process flow will be:
stream new_trips -> table trips (convert to structured, append new records)
stream new_programs -> table programs (build a dimension table, merge new programs)
stream new_stations -> table stations (build a dimension table, merge new start and end stations)

Module 4: Tasks

At this point, 2 of the 3 pieces of the engineering pipeline are in place. The last step is to create the TASKS that will operate on the STREAMS you just created. Tasks allow us to schedule and orchestrate ETL logic which consumes the records collected by the streams and runs DML transactions (insert, merge) on those records. The completion of the DML transaction resets the stream so we can capture changes going forward from here.

Tasks need a warehouse to execute their DML commands so create a warehouse specifically to handle this. In a production environment you could choose to create a dedicated warehouse like here or use an existing ETL warehouse. In the examples below we will use the existing HOL_WH to execute TASK activities.

4.1 Create and Activate Tasks

4.1.1 Create the push_trips task to read JSON data in the new_trips stream and transform and write it to the trips table you just created. You indicate the warehouse where this task will run and how frequently it will run. You can also add the optimization to only run this task if there is new data in the stream to process. In the event no new data has arrived in the last minute, the task will not kick off and you can potentially save the startup cost for the warehouse.

```
create or replace task push_trips
warehouse = HOL_WH
schedule = '1 minute'
when system$stream_has_data('new_trips')
as
insert into trips
  select v:tripduration::integer,
  v:starttime::timestamp_ntz,
  v:stoptime::timestamp_ntz,
  v:start_station_id::integer,
  v:end_station_id::integer,
  v:bikeid::integer,
  v:usertype::string,
  v:birth_year::integer,
  v:gender::integer,
  v:program_id::integer
  from new_trips;
```

4.1.2 Set up a similar TASK for the new_programs stream but this time perform a merge into the programs table.

```
create or replace task push_programs
```

```
warehouse = HOL_WH
schedule = '1 minute'
when system$stream_has_data('new_programs')
as
merge into programs p
  using (
    select distinct v:program_id::integer program_id,
      v:program_name::string program_name
    from new_programs) np
  on p.program_id = np.program_id
  when not matched then
    insert (program_id, program_name)
    values (np.program_id, np.program_name);
```

4.1.3 Create a TASK to perform a merge into the stations table.

```
create or replace task push_stations
warehouse = HOL_WH
schedule = '1 minute'
when system$stream_has_data('new_stations')
as
merge into stations s
  using (
    select v:start_station_id::integer station_id,
      v:start_station_name::string station_name,
      v:start_station_latitude::float station_latitude,
      v:start_station_longitude::float station_longitude,
      'Station at ' || v:start_station_name::string station_comment
    from new_stations
    union
    select v:end_station_id::integer station_id,
      v:end_station_name::string station_name,
      v:end_station_latitude::float station_latitude,
      v:end_station_longitude::float station_longitude,
      'Station at ' || v:end_station_name::string station_comment
    from new_stations) ns
  on s.station_id = ns.station_id
  when not matched then
    insert (station_id, station_name, station_latitude, station_longitude,
station_comment)
    values (ns.station_id, ns.station_name, ns.station_latitude, ns.station_longitude,
ns.station_comment);
```

4.1.4 You can chain TASKS together to form sophisticated processing operations. Define a TASK to call the purge_files procedure AFTER the push_trips TASK runs.

```
create or replace task purge_files
warehouse = task_wh
```

```
after push_trips
as
  call purge_files('trips_raw', '@streaming_data', 's3://MyDemoBucket/',
'streaming_trips/');
```

4.1.5 Tasks need to be activated after they are created or anytime they are modified in some way. Activate the tasks that were just created.

```
alter task push_trips resume;
alter task push_programs resume;
alter task push_stations resume;
--alter task purge_files resume;

show tasks;
```

4.2 Use Tasks to Stream Data into Stage and then Trips Table

That's it! Let's kick this off and observe the data now flowing through the pipeline end to end for a month's worth of data.

4.2.1 Call the stream_data procedure to drop files into the STAGE which will then get auto loaded into the raw.trips_raw table. Here we'll load a whole months worth of data.

```
call stream_data('2019-01-01','2019-01-31');
```

4.2.2 As the stored procedure is slowly feeding data to our stage. Open a new tab in the UI and set the context so we can observe progress and metrics of the pipeline process:

```
use role dba_citibike;
use warehouse demo_wh;
use schema citibike.raw;
```

4.2.3 The following command will show us the details of each of our tasks over the last five minutes. You will notice the STATE for some of these show 'SKIPPED' because no data has arrived in the STREAM.

```
select * from table(information_schema.task_history())
  where scheduled_time > dateadd(minute, -5, current_time())
  and state <> 'SCHEDULED'
  order by completed_time desc;
```

4.2.4 How long until the next task runs?

```
select timestampdiff(second, current_timestamp, scheduled_time) next_run,
scheduled_time, name, state
  from table(information_schema.task_history())
  where state = 'SCHEDULED' order by completed_time desc;
```

4.2.5 How many files have been processed by the pipeline in the last hour?

```
select count(*)
from table(information_schema.copy_history(
  table_name=>'citibike.raw.trips_raw',
  start_time=>dateadd(hour, -1, current_timestamp)));
```

4.2.6 Here's a convenient query to get an overview of the pipeline including: time to next task run, the number of files in the bucket, number of files pending for loading, total number of files processed in the last hour, and record count metrics across all the tables referenced in the solution.

```
select
  (select min(timestampdiff(second, current_timestamp, scheduled_time))
    from table(information_schema.task_history())
    where state = 'SCHEDULED' order by completed_time desc) time_to_next_pulse,
  (select count(distinct metadata$filename) from @streaming_data/) files_in_bucket,
  (select
parse_json(system$pipe_status('citibike.raw.trips_pipe')):pendingFileCount::number)
pending_file_count,
  (select count(*)
    from table(information_schema.copy_history(
    table_name=>'citibike.raw.trips_raw',
    start_time=>dateadd(hour, -1, current_timestamp)))) files_processed,
  (select count(*) from citibike.raw.trips_raw) trips_raw,
  (select count(*) from citibike.raw.new_trips) recs_in_stream,
  (select count(*) from citibike.modelled.trips) trips_modelled,
  (select count(*) from citibike.modelled.programs) num_programs,
  (select count(*) from citibike.modelled.stations) num_stations,
  (select max(starttime) from citibike.modelled.trips) max_date;
```

Congratulations, you are now done with this lab! Let's wrap things up in the next, and final, section.

Summary & Next Steps

This lab has walked you step by step through the elements that comprise data engineering using Snowflake data pipelines. You created a Stage object and configured it to invoke Snowpipe when files arrive. This allows for automatic ingestion of streaming data sources. Next you created streams on top of the staging table that Snowpipe loads data into. Streams provide a way to continually process data as it arrives but only focus that processing on new data and avoid reprocessing old data. Lastly, you created tasks for scheduled execution of logic to move data from the streams into the final tables of the overall data model. Along the way you also created handy procedures to mimic streaming data and clean up data files that were loaded.

It's easy to see how powerful these tools are and how you can apply them to workloads in your Snowflake account to achieve automated data engineering.

Resetting Your Snowflake Environment

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

```
rm @citibike.raw.streaming_data;
drop schema citibike.raw;
drop schema citibike.modelled;
drop warehouse demo_wh;
drop warehouse task_wh;
```