Snowflake Streams Explained

Snowflake is a cloud data warehouse offering which is available on multiple cloud platforms, including Azure. To learn more about Snowflake itself, check out the tutorial. To keep track of data changes in a table, Snowflake has introduced the **streams** feature.

A stream is an object you can query, and it returns the inserted or deleted rows from the table since the last time the stream was accessed (well, it's a bit more complicated, but we'll deal with that later). Updates are returned as an insert and a delete, where the insert contains the new values and the update the old values.

Using streams, you can set up a change data capture scenario where you only take new or modified rows – sometimes called "the delta" – into consideration for the remainder of your process. For example, when loading a dimension you can only take the new and updates rows from the source tables, making the process more efficient.

You can create a stream with the following syntax:

```
CREATE OR REPLACE STREAM mySchema.MyStream
ON TABLE mySchema.myTable;
```

For the complete syntax, check out the documentation.

Streams work by keeping track of an **offset**; a pointer which indicates the point in time since you last read the stream. However, the stream itself does not contain data. Every time a table changes, a new version of the table is created (which is the basis for the time travel feature). Using metadata, a stream can track all the changes over these different versions. These versions are not kept around forever, after some time they are discarded. This is referred to as the *data retention* of the table.

Creating Snowflake Streams

Let's illustrate the concept of streams with an example. The following SQL statements create a schema and a table:

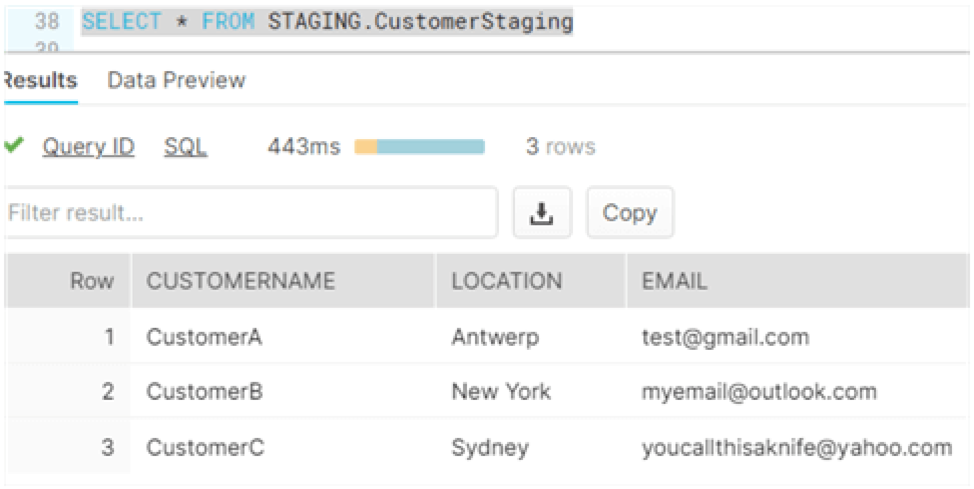
```
CREATE SCHEMA STAGING;
```

```
CREATE OR REPLACE TABLE STAGING.CustomerStaging(
   CustomerName VARCHAR(50) NOT NULL,
   Location VARCHAR(50) NOT NULL,
   Email VARCHAR(50) NULL
);
```

This is the same table we used in the tip Exploring Tasks in Snowflake. In that tip, you can also download CSV files with sample data. We can load such a CSV file with the following COPY INTO statement:

```
COPY INTO STAGING.CustomerStaging
FROM @MSSQLTIPS_STAGE/tasks
    FILE_FORMAT=(TYPE=CSV COMPRESSION=NONE FIELD_DELIMITER=';' SKIP_HEADER=1
TRIM_SPACE=TRUE)
    PATTERN ='.*csv'
    PURGE = TRUE
    ON_ERROR='CONTINUE';
```

The table now contains 3 rows:



```
SELECT * FROM STAGING.CustomerStaging
```

Row	CUSTOMERNAME	LOCATION	EMAIL
1	CustomerA	Antwerp	test@gmail.com
2	CustomerB	New York	myemail@outlook.com
3	CustomerC	Sydney	youcallthisaknife@yahoo.com

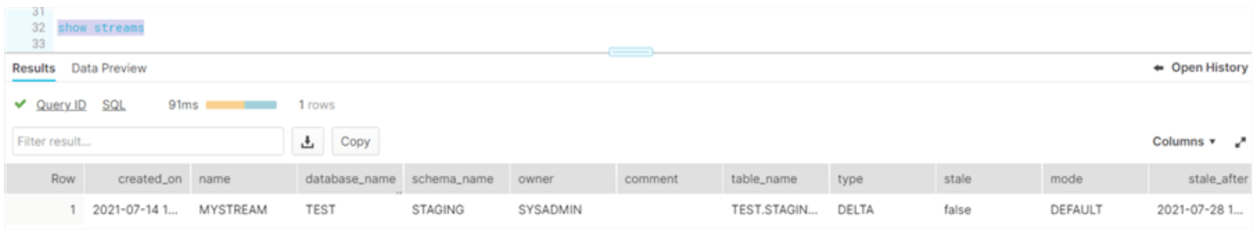
You can also populate the table with your own sample data using the INSERT statement. We can create a stream to monitor the table with the following statement:

```
CREATE OR REPLACE STREAM STAGING.MyStream
ON TABLE STAGING.CustomerStaging
APPEND_ONLY = FALSE
SHOW_INITIAL_ROWS = TRUE;
```

We added two parameters to the stream:

- **APPEND_ONLY** is false. This means the stream will monitor for inserts, updates and deletes. When set to true, the stream will only return new rows.
- **SHOW_INITIAL_ROWS** is true. This means the first time, the stream will return the rows that were present in the table. After this, the stream will return only new and modified rows.

With SHOW STREAMS we can find out which streams have been created and for which you have access:



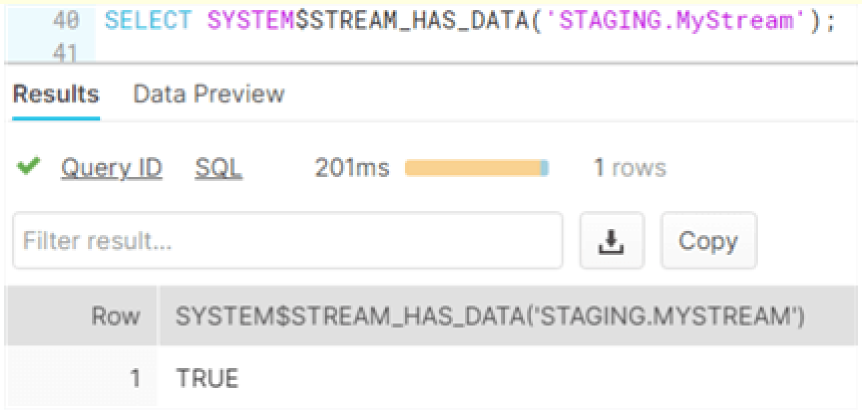
```
show streams
```

Row	created_on	name	database_name	schema_name	owner	comment	table_name	type	stale	mode	stale_after
1	2021-07-14 1...	MYSTREAM	TEST	STAGING	SYSADMIN		TEST.STAGIN...	DELTA	false	DEFAULT	2021-07-28 1...

An important property to take notice is **stale**. A stream can become stale when the offset (the point in time since you last accessed the stream) becomes bigger than the data retention period. If this happens, you would need to recreate the stream. To avoid the stream from becoming stale, you'd need to access it periodically. More information about data retention and staleness can be found here.

We can check if a stream has data (again, the stream itself doesn't contain data, it's all metadata) with the function STREAM_HAS_DATA:

```
SELECT SYSTEM$STREAM_HAS_DATA('STAGING.MyStream');
```

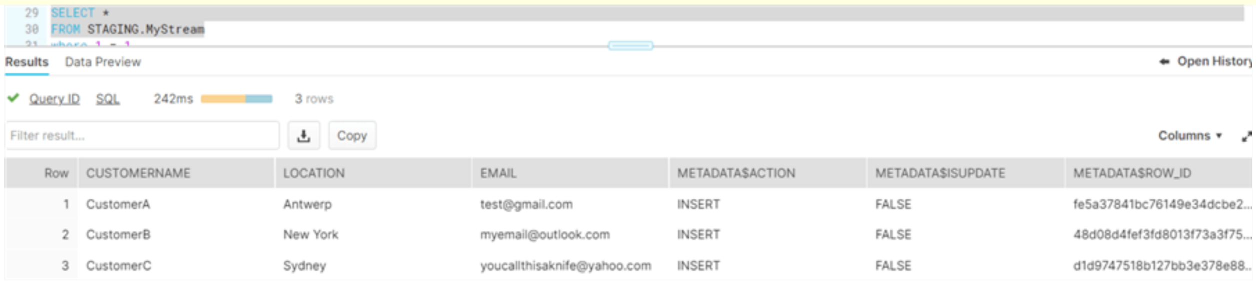


```
SELECT SYSTEM$STREAM_HAS_DATA('STAGING.MyStream');
```

Row	SYSTEM$STREAM_HAS_DATA('STAGING.MYSTREAM')
1	TRUE

We can query the stream as if it is a view or table:

```
SELECT *
FROM STAGING.MyStream;
```



```
SELECT *
FROM STAGING.MyStream
```

Row	CUSTOMERNAME	LOCATION	EMAIL	METADATA$ACTION	METADATA$ISUPDATE	METADATA$ROW_ID
1	CustomerA	Antwerp	test@gmail.com	INSERT	FALSE	fe5a37841bc76149e34dcbe2...
2	CustomerB	New York	myemail@outlook.com	INSERT	FALSE	48d08d4fef3fd8013f73a3f75...
3	CustomerC	Sydney	youcallthisaknife@yahoo.com	INSERT	FALSE	d1d9747518b127bb3e378e88...

The stream returns all of the columns from the table, as well as some metadata columns:

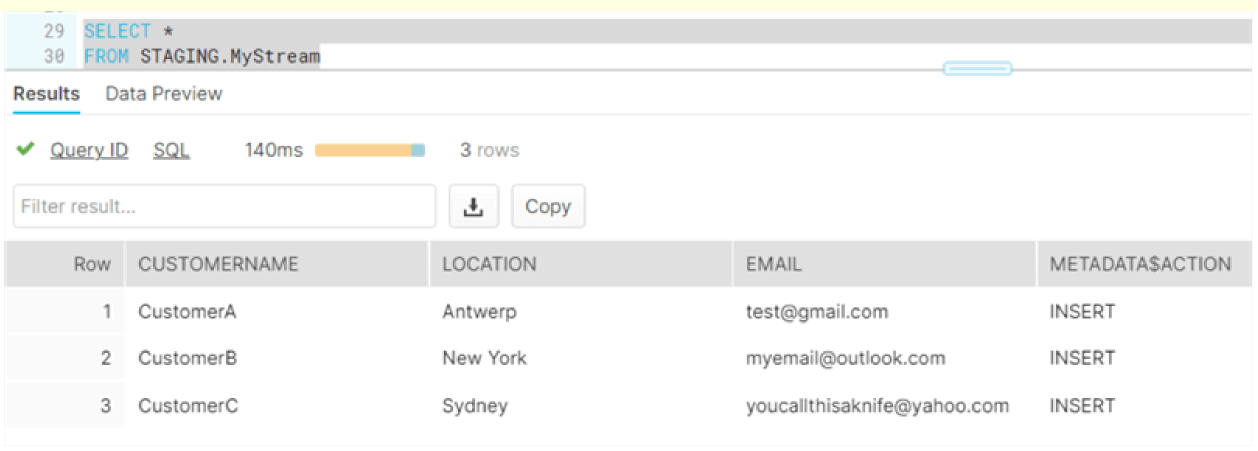
- **ACTION**: This column tells you if a row is an insert or a delete. Remember, updates are split into two rows: one delete and one insert.
- **ISUPDATE**: this Boolean field indicates if a row (insert or delete) was caused by an UPDATE statement.
- **ROW_ID**: an internal ID to uniquely identify a row.

You can select specific columns of a stream and use a WHERE clause, just like in any SELECT statement.

Changing the Offset of a Table

Just reading the data from a stream will not change the offset. If we run the same SELECT statement, the same 3 rows are returned. However, when we insert new data into the table, the new row is not included in the stream output.

```
INSERT INTO STAGING.CustomerStaging
(CustomerName, Location, Email)
SELECT 'Test','TestLoc','test@test.be';
```



```
SELECT *
FROM STAGING.MyStream
```

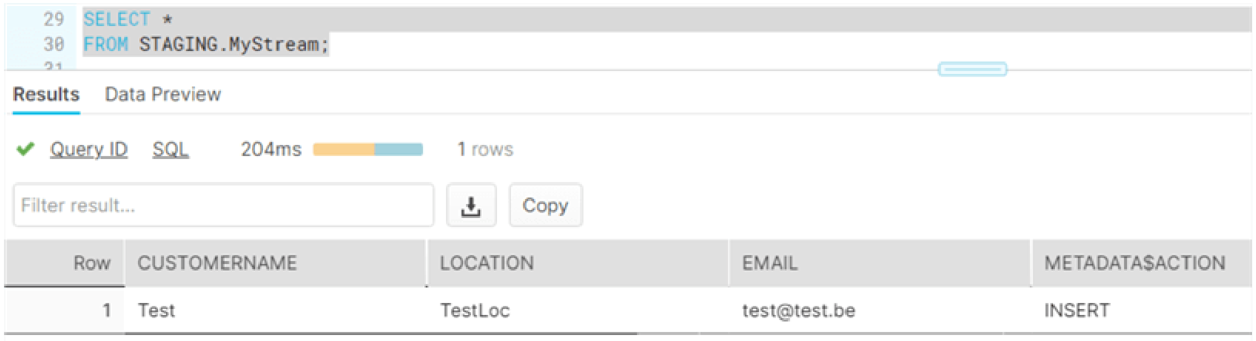
Row	CUSTOMERNAME	LOCATION	EMAIL	METADATA$ACTION
1	CustomerA	Antwerp	test@gmail.com	INSERT
2	CustomerB	New York	myemail@outlook.com	INSERT
3	CustomerC	Sydney	youcallthisaknife@yahoo.com	INSERT

How is this possible? Remember we created our stream with the parameter SHOW_INITIAL_ROWS set to true. This means the initial set of rows will be returned until we have processed them! Selecting rows from the stream is not enough. The offset will only change when a DML statement (insert, update, delete or merge) is used with the result of the stream.

Let's insert those initial 3 rows into a table:

```
CREATE OR REPLACE TABLE STAGING.CustomerDelta(
   CustomerName VARCHAR(50) NOT NULL,
   Location VARCHAR(50) NOT NULL,
   Email VARCHAR(50) NULL,
    METADATA_Action VARCHAR(10) NOT NULL,
    METADATA_IsUpdate BOOLEAN NOT NULL,
    METADATA_ROWID VARCHAR(100) NOT NULL
);
INSERT INTO STAGING.CustomerDelta
(   CustomerName
    ,Location
    ,Email
    ,METADATA_Action
    ,METADATA_IsUpdate
    ,METADATA_ROWID)
SELECT
     CustomerName
    ,Location
    ,Email
    ,METADATA$Action     AS METADATA_Action
    ,METADATA$IsUpdate   AS METADATA_IsUpdate
    ,METADATA$ROW_ID     AS METADATA_ROWID
FROM STAGING.MyStream;
```

With the 3 initial rows being "consumed", the stream will now show the extra row we inserted before:



```
SELECT *
FROM STAGING.MyStream;
```

Row	CUSTOMERNAME	LOCATION	EMAIL	METADATA$ACTION
1	Test	TestLoc	test@test.be	INSERT

Working with initial rows can be tricky. For example, if the table initially had no rows, the stream will return no rows even if we insert new rows into the table! The stream will continue to return no rows (even though the STREAM_HAS_DATA function will return true) until this empty result set is used in a DML statement.

What if you need the output of the stream for multiple statements? For example, you select the changed rows to do an UPDATE on a dimension, and then you need the inserted rows for an INSERT statement. To read the same output multiple times, you need to put the statements inside an explicit transaction. After the transaction has committed successfully, the offset has changed.

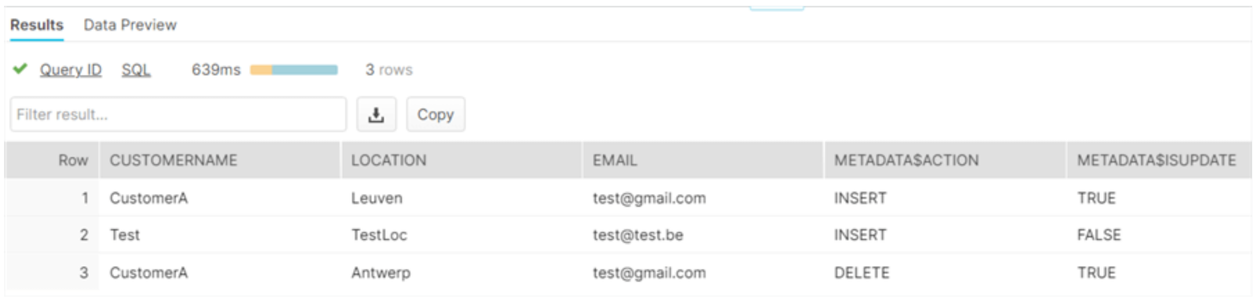
Multiple Changes to a Single Row

What if a row is updated multiple times before we're able to read those out with a stream? In this case, the stream will return only the **net changes**. In other words, only the last value is returned. Let's illustrate with an example. The following statements update the same row multiple times:

```
UPDATE STAGING.CustomerStaging
SET Location = 'Brussels'
WHERE customerName = 'CustomerA';

UPDATE STAGING.CustomerStaging
SET Location = 'Leuven'
WHERE customerName = 'CustomerA';
```

The stream returns 3 rows (ignore the row of the inserted Test value, it's there because we haven't processed it yet):



Results | Data Preview

✓ Query ID SQL 639ms 3 rows

Row	CUSTOMERNAME	LOCATION	EMAIL	METADATA$ACTION	METADATA$ISUPDATE
1	CustomerA	Leuven	test@gmail.com	INSERT	TRUE
2	Test	TestLoc	test@test.be	INSERT	FALSE
3	CustomerA	Antwerp	test@gmail.com	DELETE	TRUE

We can see that a row for customerA has been "deleted": this is the old value of the update. And a new row has been inserted with the value of "Leuven" (this is the new value of the update). However, there are no rows with a location of Brussels. It's as if it never happened.

More Info

The CHANGES feature

If you don't want to create an explicit stream and periodically query it to prevent it from going stale, you can use the CHANGES clause. This feature uses the change tracking metadata (using the different table versions) between different transactional start and endpoints. For example, the following SELECT statement will return all changed rows of the last half hour:

```
SELECT *
FROM STAGING.CustomerStaging
CHANGES(information => default)
AT (OFFSET => -30*60);
```

Streams and Tasks

The two features make an ideal combination for an efficient ETL pattern. With a scheduled task, we can read periodically the delta from a table so we can process it into the data warehouse. A task can even be conditionally executed, meaning it only runs if the stream has data. This is done with the optional Boolean WHEN parameter of a task:

```
CREATE TASK myTask
    WAREHOUSE = myWarehouse

   [SCHEDULE = x MINUTE | USING CRON cron_expression]

   [parameters…]

   [AFTER otherTask]

WHEN Boolean_expression
```

```
AS SYSTEM$STREAM_HAS_DATA('STAGING.MyStream')

<some SQL statement>
```

This task will now only run when the function returns true. An advantage of this pattern is that the function is evaluated inside the metadata layer of Snowflake, so it incurs no extra cost. Using streams combined with tasks can thus be cost efficient.