**Delta Lake Using Azure Data Factory**

**Introduction:**

Delta Lake is an open source storage layer that guarantees data atomicity, consistency, isolation, and durability in the lake. In short, a Delta Lake is ACID compliant. In addition to providing ACID transactions, scalable metadata handling and more, Delta Lake runs on an existing Data Lake and is compatible with Apache Spark APIs. There are a few methods of getting started with Delta Lake. Databricks offers notebooks along with compatible Apache Spark APIs to create and manage Delta Lakes. Alternatively, Azure Data Factory's Mapping Data Flows, which uses scaled-out Apache Spark clusters, can be used to perform ACID compliant CRUD operations through GUI designed ETL pipelines. This article will demonstrate how to get started with Delta Lake using Azure Data Factory's new Delta Lake connector through examples of how to create, insert, update, and delete in a Delta Lake.

## **Why an ACID Delta Lake**

There are many advantages to introducing Delta Lake into a Modern Cloud Data architecture. Traditionally, Data Lakes and Apache Spark are not ACID compliant. Delta Lake introduces this ACID compliance to solve many the following ACID compliance issues.

**Atomicity**: Write either All Data or Nothing. Apache Spark saves mode do not utilize any locking and are not atomic. With this, a failed job may leave an incomplete file and may corrupt data. Additionally, a failing job may remove the old file and corrupt the new file. While this seems concerning, Spark does have in built data frame writer APIs that are not atomic but behaves so for append operations. This however does come with performance overhead for use with cloud storage.

**Consistency**: Data is always in a valid state. If the Spark API writer deletes an old file and creates a new one and the operation is not transactional, then there will always be a period of time when the file does not exist between the deletion of the old file and creation of the new. In that scenario, if the overwrite operation fails, this will result in data loss of the old file. Additionally, the new file may not be created. This is a typical spark overwrite operation issue related to Consistency.

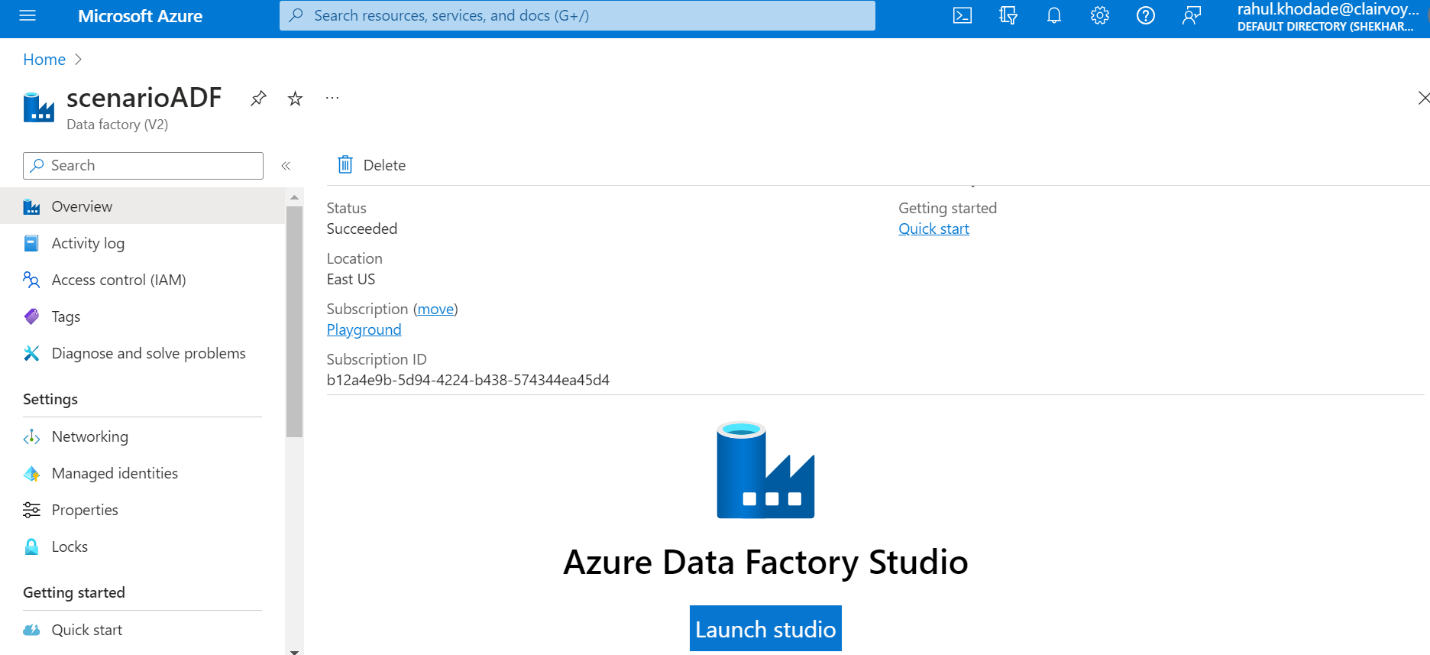
**Isolation**: Multiple transactions occur independently without interference. This means that when writing to a dataset, other concurrent reads or writes on the same dataset should not be impacted by the write operation. Typical transactional databases offer multiple isolation levels. While Spark has task and job level commits, since it lacks atomicity, it does not have isolation types.

**Durability**: Committed Data is never lost. When Spark does not correctly implement a commit, then it overwrites all the great durability features offered by cloud storage options and either corrupts and/or loses the data. This violates data Durability.

**Pre-Requisites­­­­­­­**

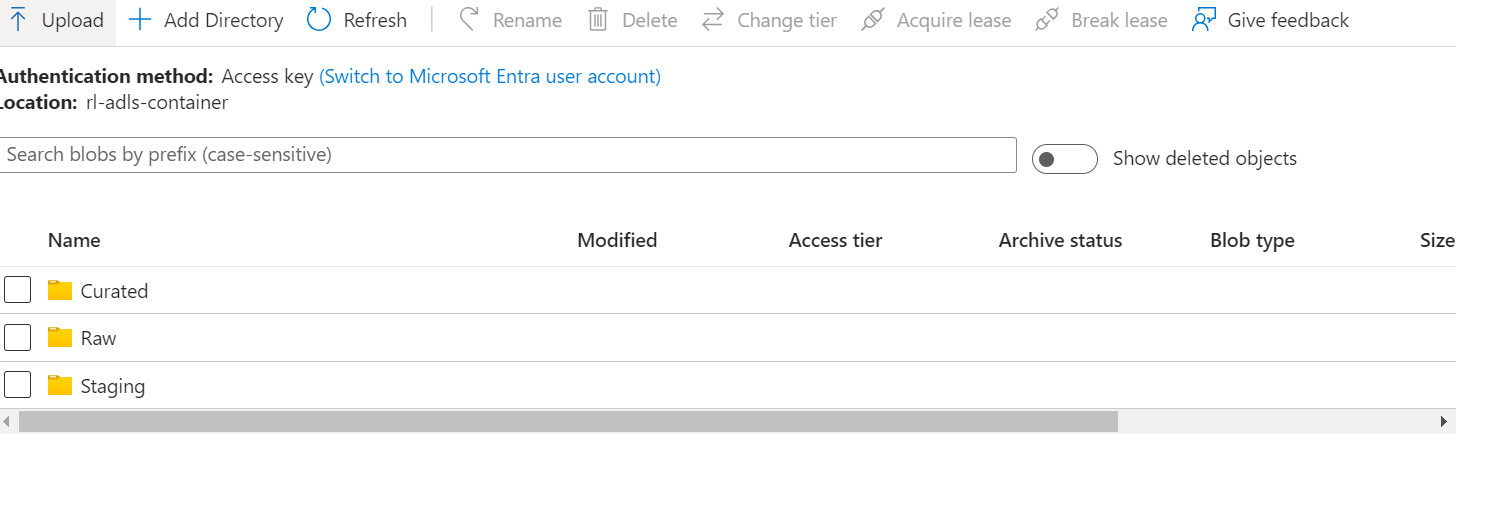
For this Demo, be sure to successfully create the following pre-requisites:

1. **Create a Data Factory V2**: Data Factory will be used to perform the ELT orchestrations.­­­­

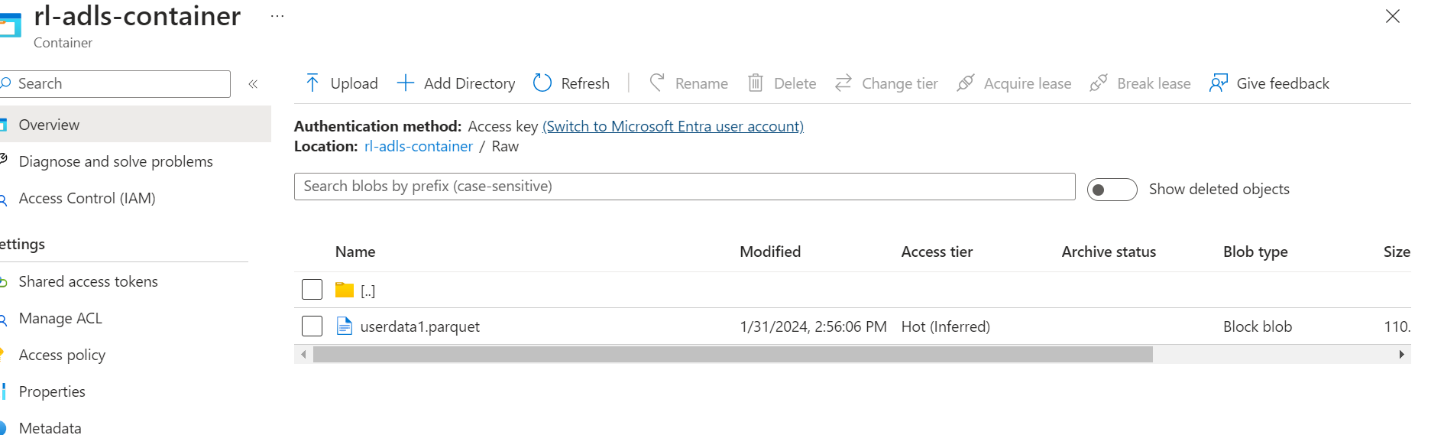


**2) Create a Data Lake Storage Gen2**: ADLSgen2 will be the Data Lake storage on top of which the Delta Lake will be created

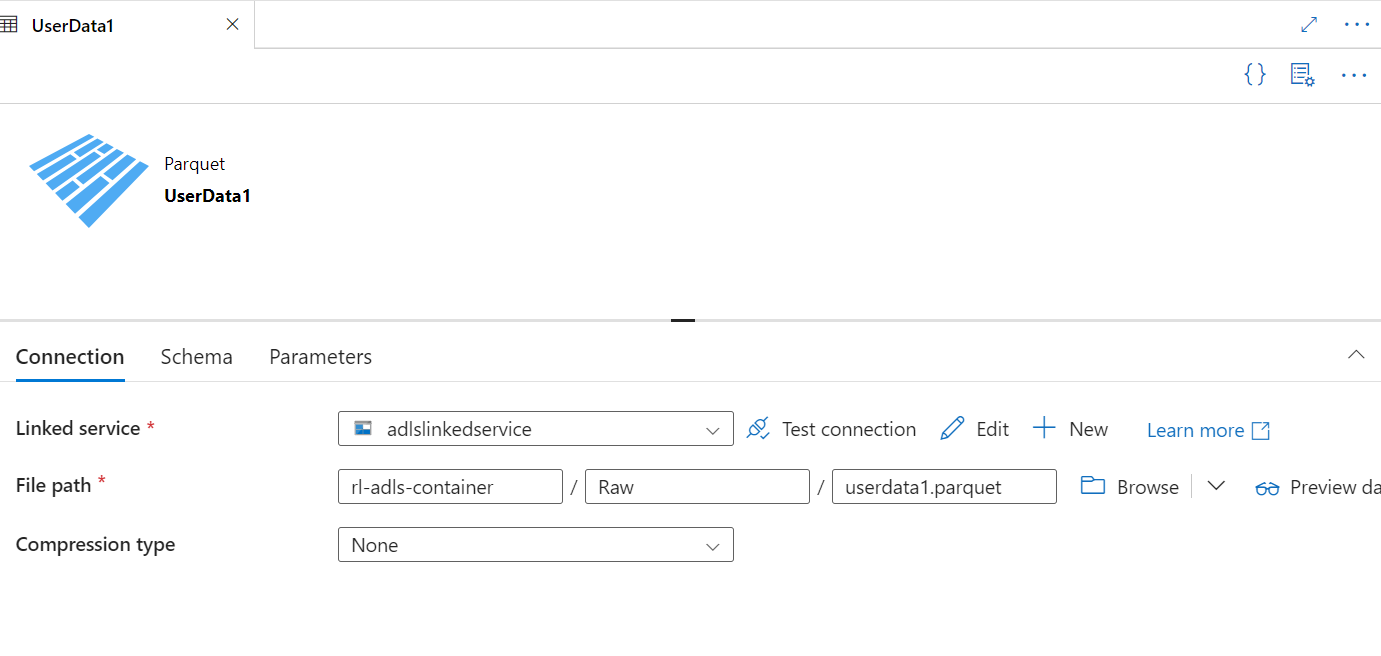
3) **Create Data Lake Storage Gen2 Container and Zones**: This demo will use the **Raw Zone** to store a sample source parquet file. Additionally, the **Staging Zone** will be used for Delta Updates, Inserts, Deletes and additional transformations.



4) **Upload Data to Raw Zone**: Finally, you'll need some data for this demo. The following [GitHub Repo](https://github.com/Teradata/kylo/tree/master/samples/sample-data/parquet) data was used for this demo.



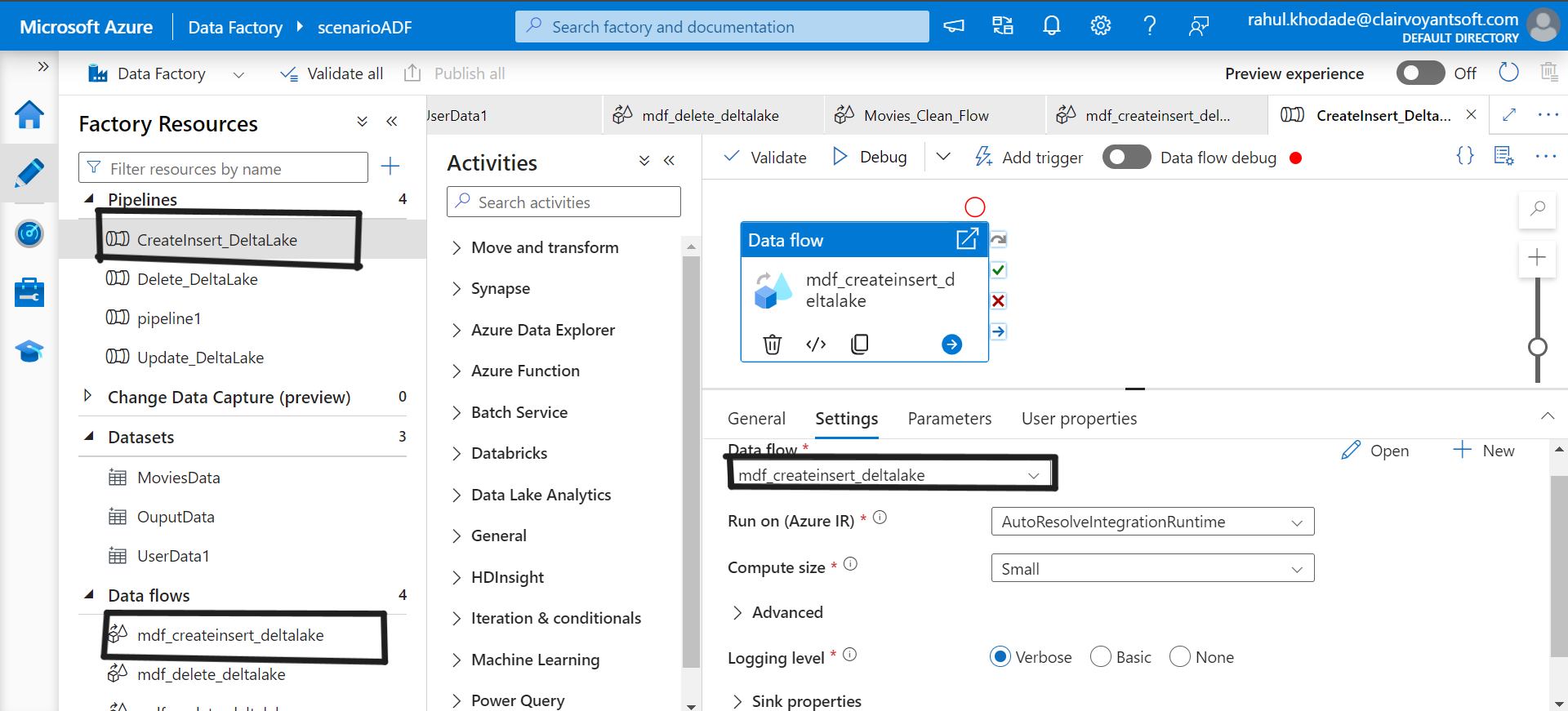
5) **Create a Data Factory Parquet Dataset pointing to the Raw Zone**: The final pre-requisite would be to create a parquet format dataset in the newly created instance of ADF V2 pointing to the sample parquet file stored in the Raw Zone.



**Create and Insert into Delta Lake**

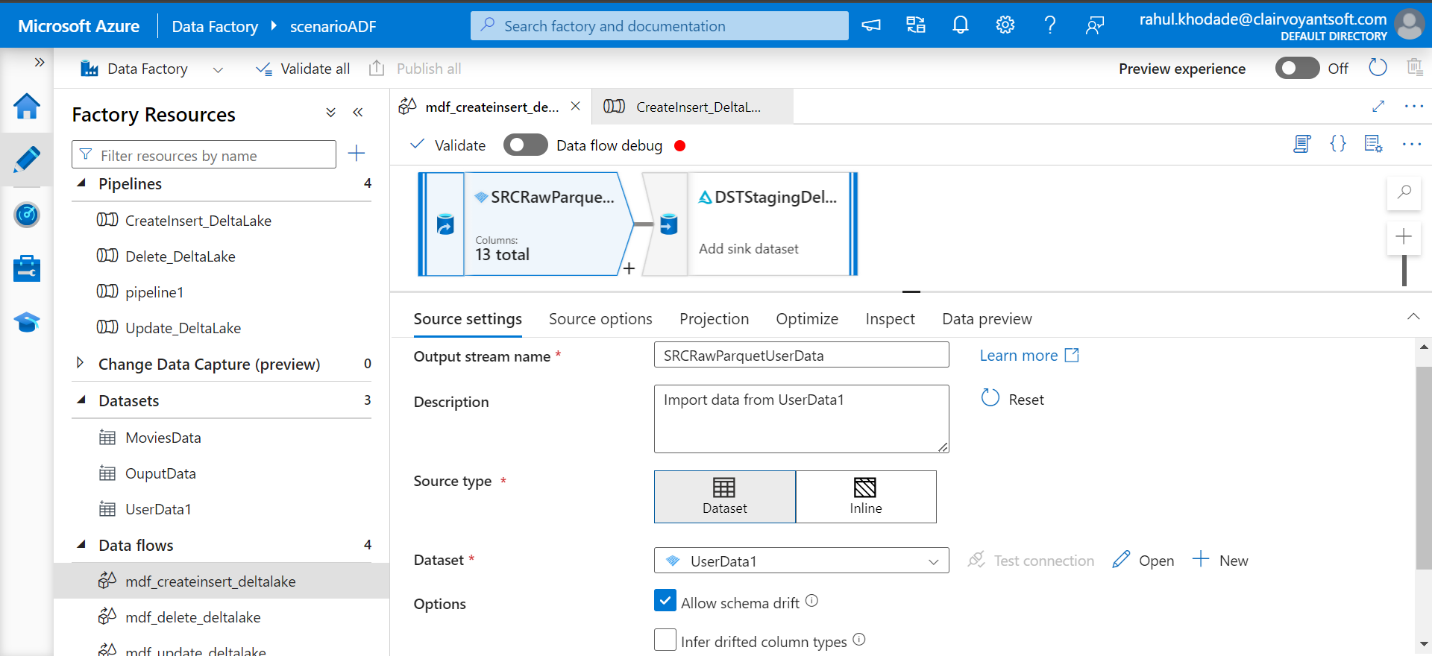
All pre-requisites are in place, we are ready to create the initial delta tables and insert data from our Raw Zone into the delta tables.

Let's begin by creating a new Data Factory pipeline and adding a new 'Mapping Data Flow' to it.

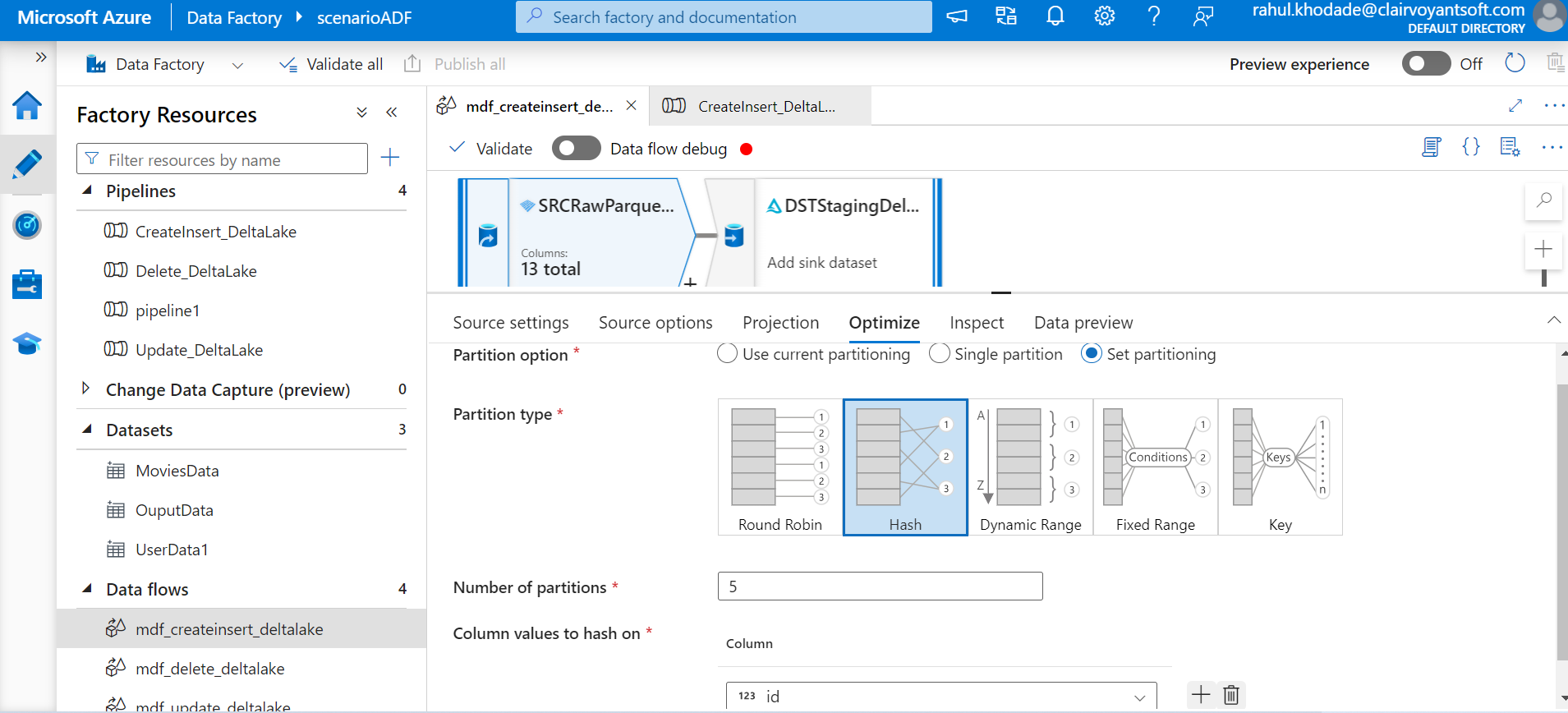


Within the Data Flow, add a source and sink with the following configurations. Schema Drift may be enabled as needed for the specific use case.

Sampling offers a method to limit the number of rows from the source, mainly used for testing and debugging purposes.



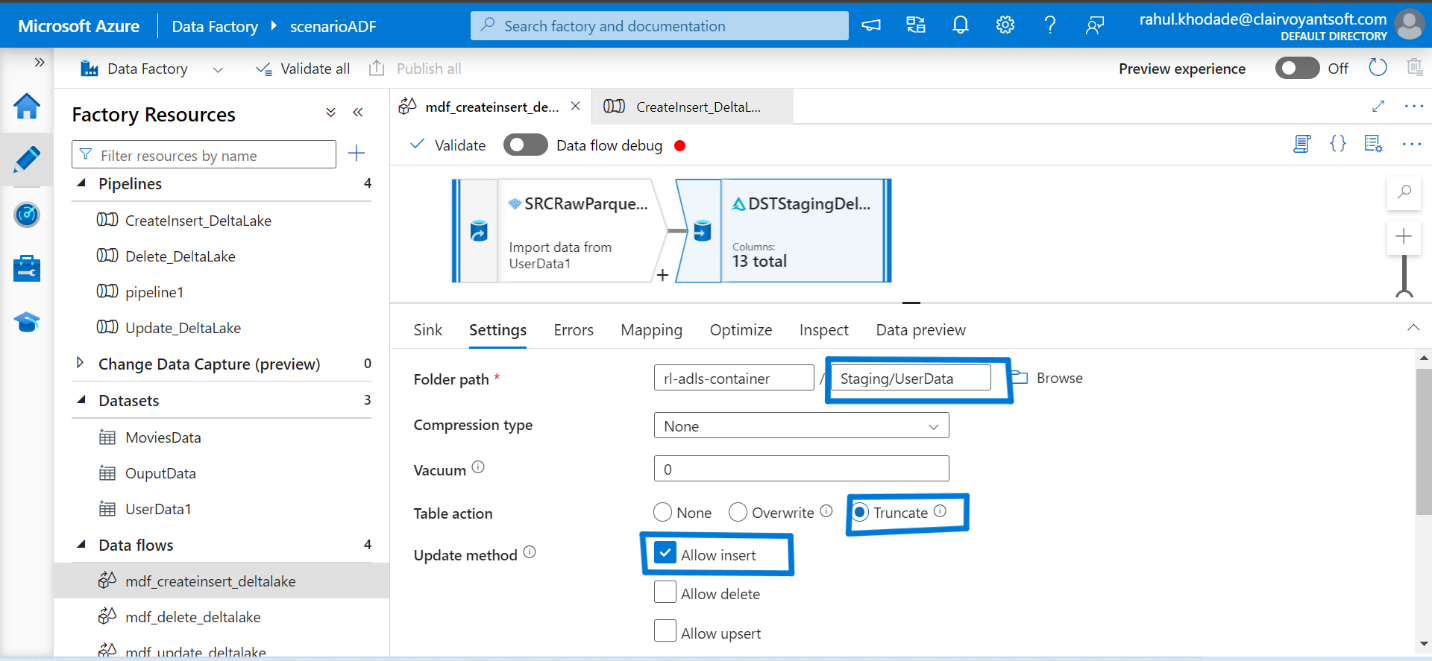
Since Delta Lake leverages Spark's distributed processing power, it is capable of partitioning data appropriately, however, for purposes of demoing the capability of manually setting partitioning, I've configured 5 Hash partitions on the ID column



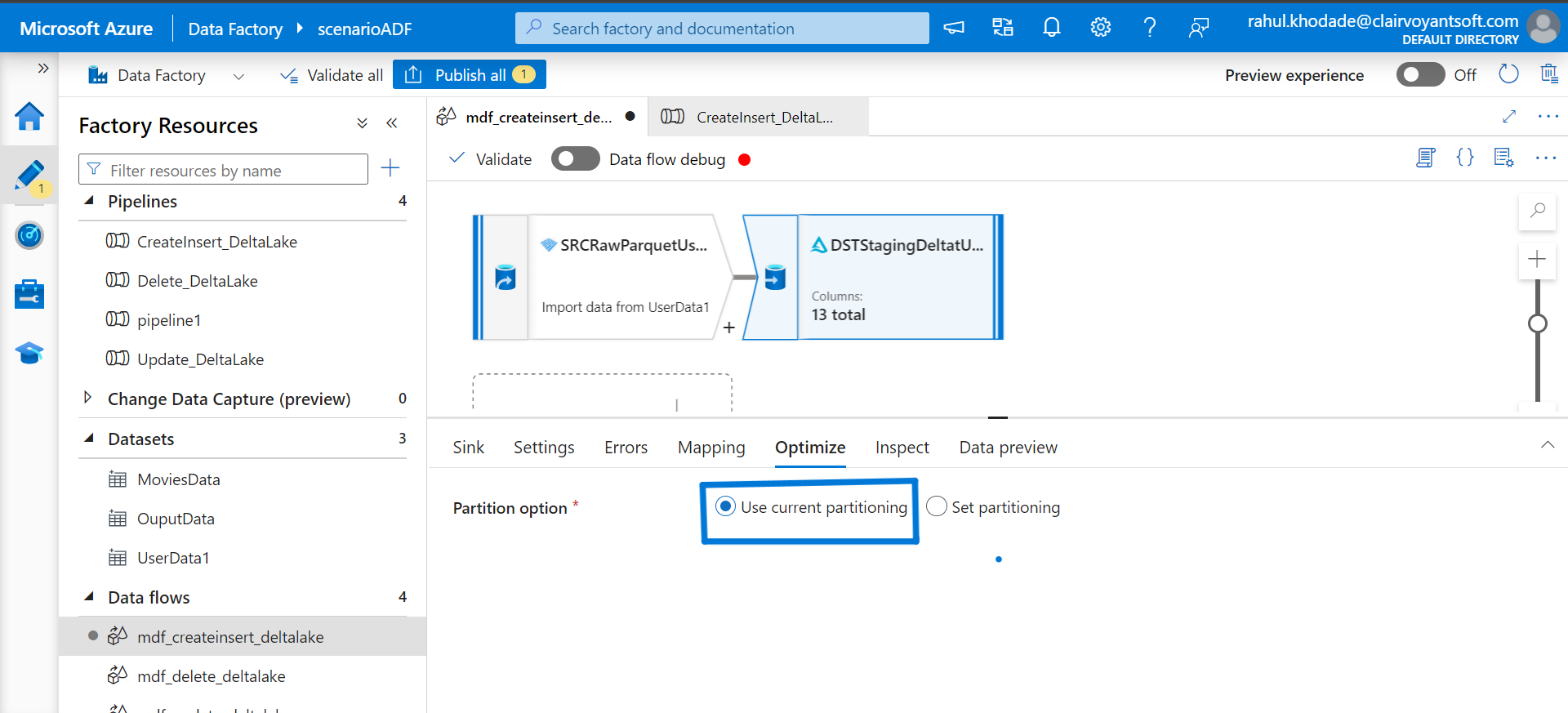
After, adding the destination activity, ensure that the sink type is set to Delta. Note that Delta is available as both a source and sink in Mapping Data Flows. Also, you will be required to select the Linked Service once the sink type of Delta is selected.



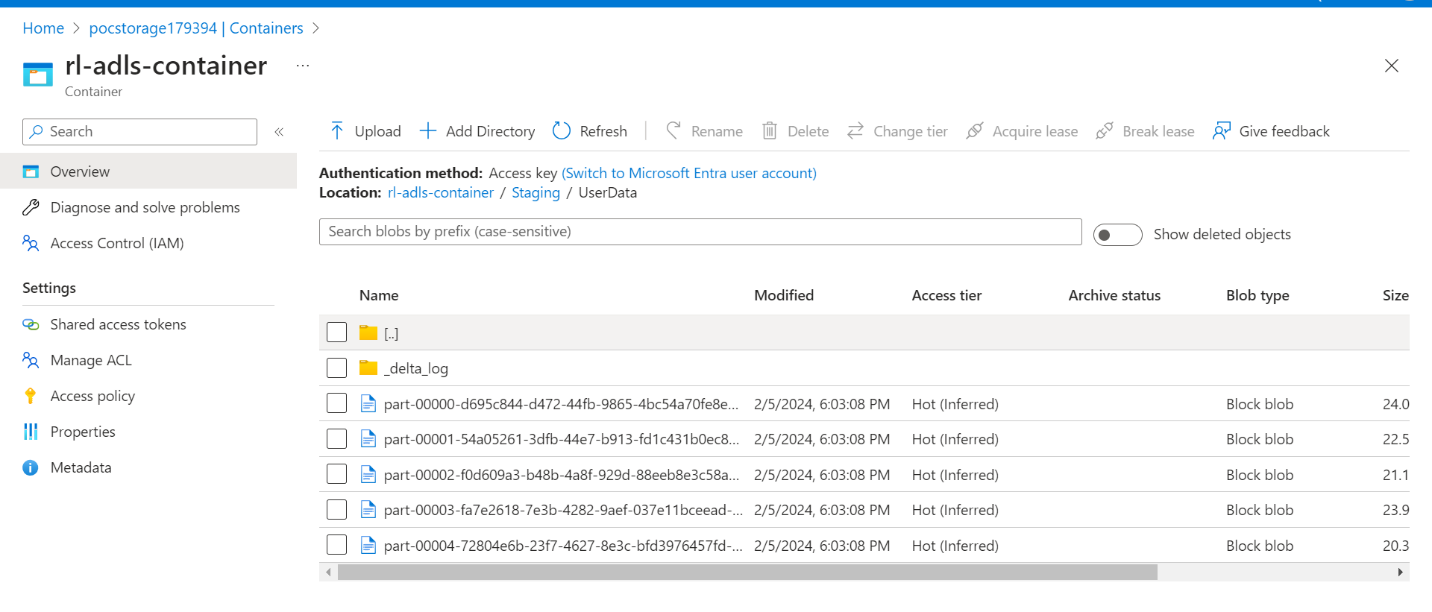
Under the Settings tab, ensure that the Staging folder is selected and select Insert for the Update Method. Also, select Truncate table if there is a need to truncate the Delta Table before loading it.



Finally, within the Optimize tab, simply use the current partitioning since the source partitioning with flow downstream to the sink.



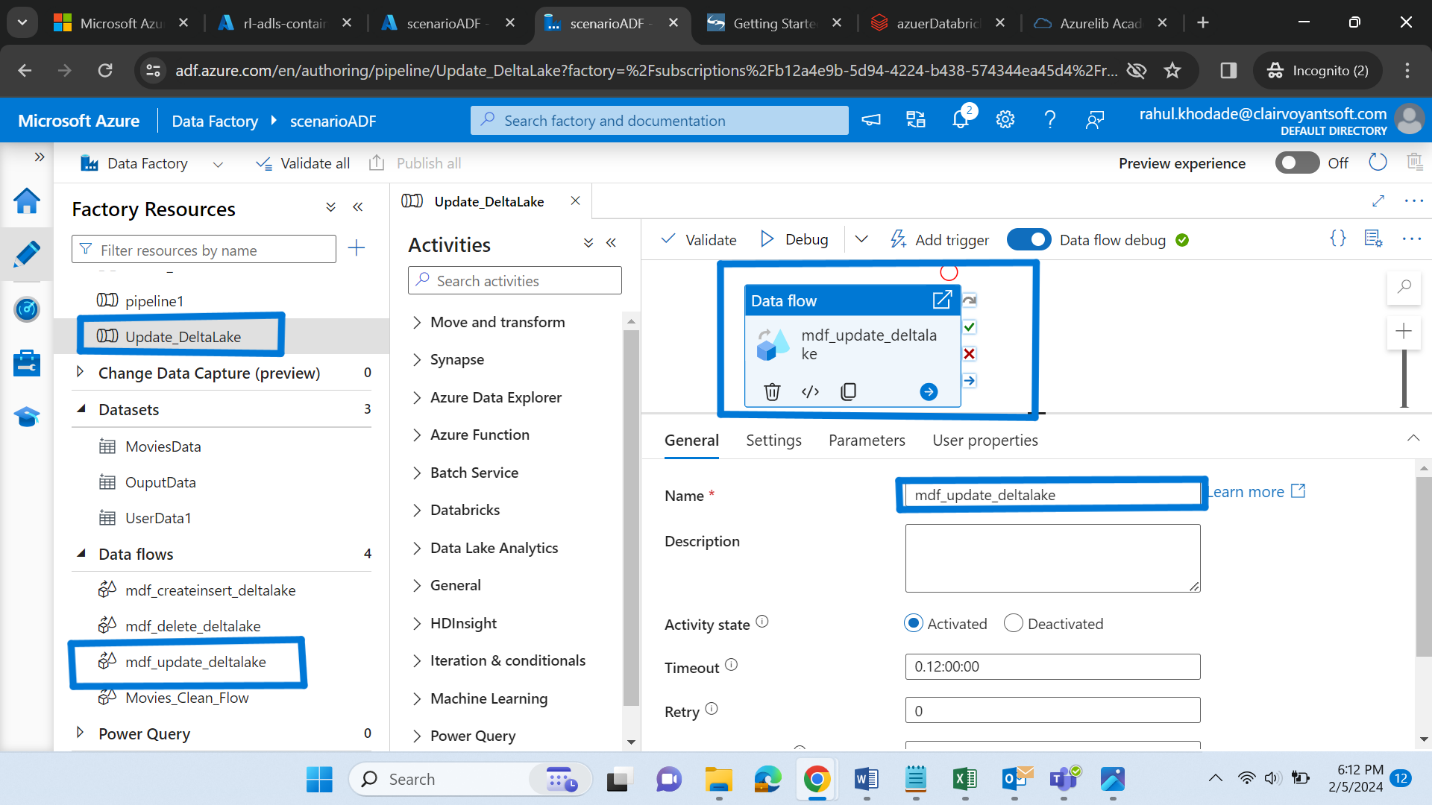
While looking at the ADLS2 staging folder, we see that a delta\_log folder along with 5 snappy compressed parquet files have been created.



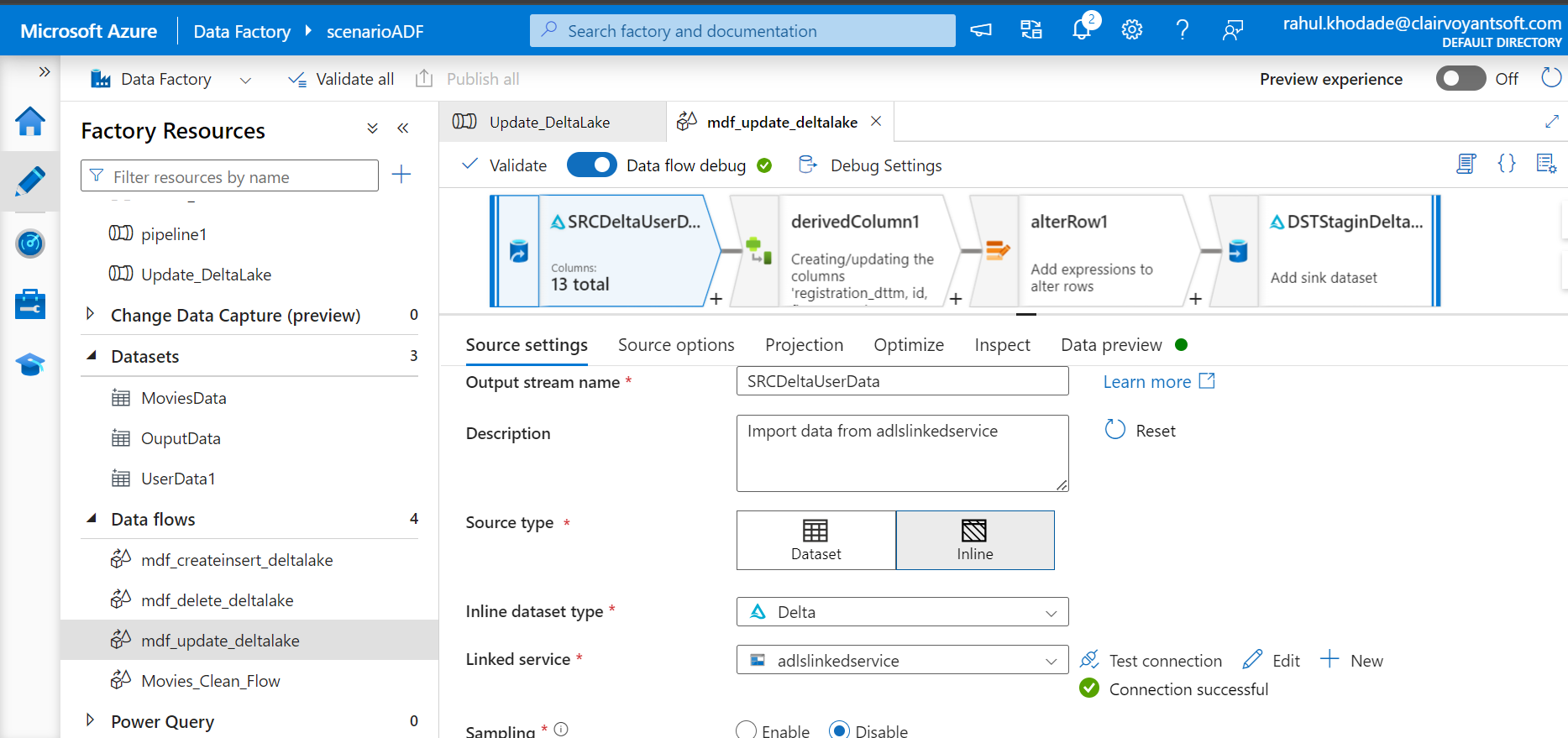
**Update Delta Lake**

Next, let's take a look at how Data Factory can handle Updates to our delta tables.

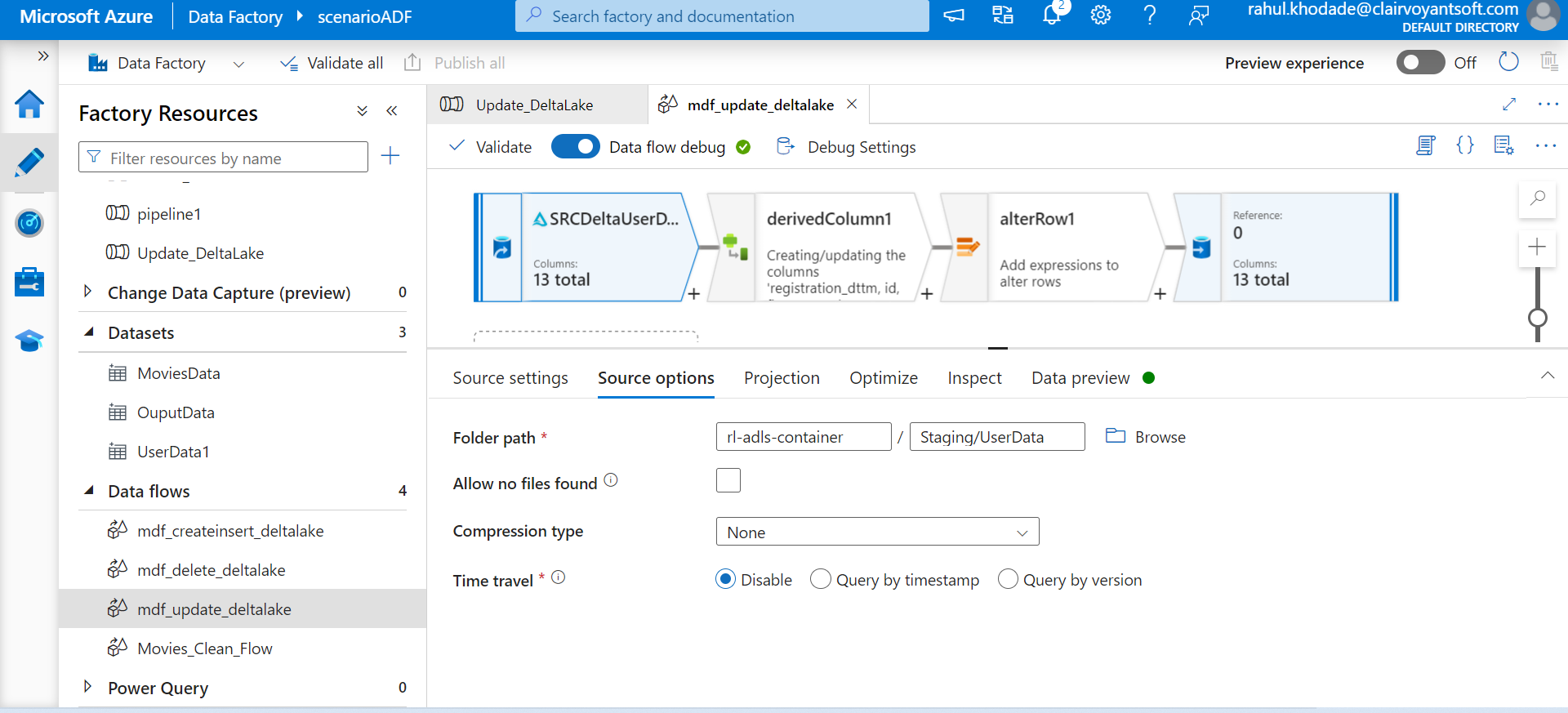
Similar to inserts, create a new ADF pipeline with a mapping data flow for Updates.



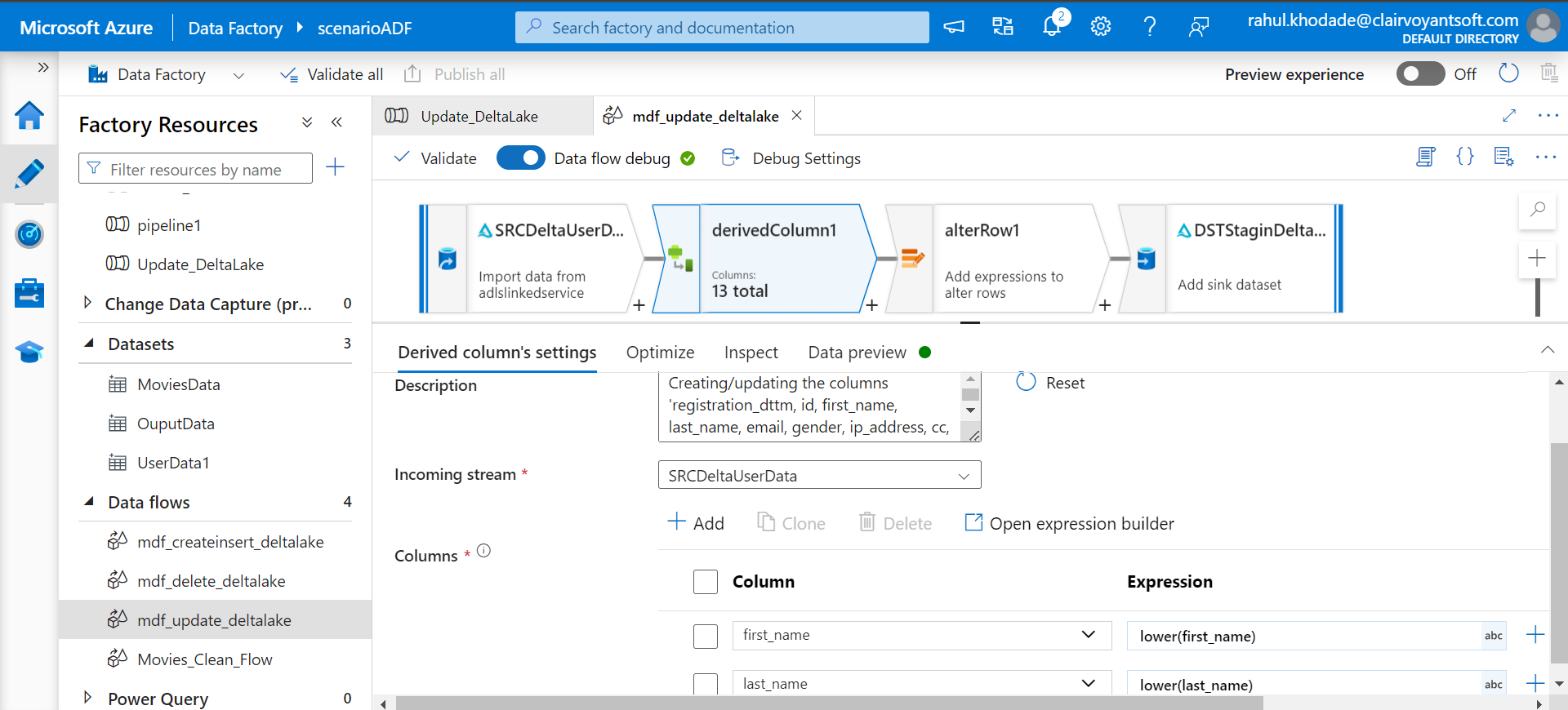
 let's update the first and last name of the user and convert it to lower case. To do this, we add a Derived Columns and Alter Row transform activity to the Update Mapping Data Flow canvas.



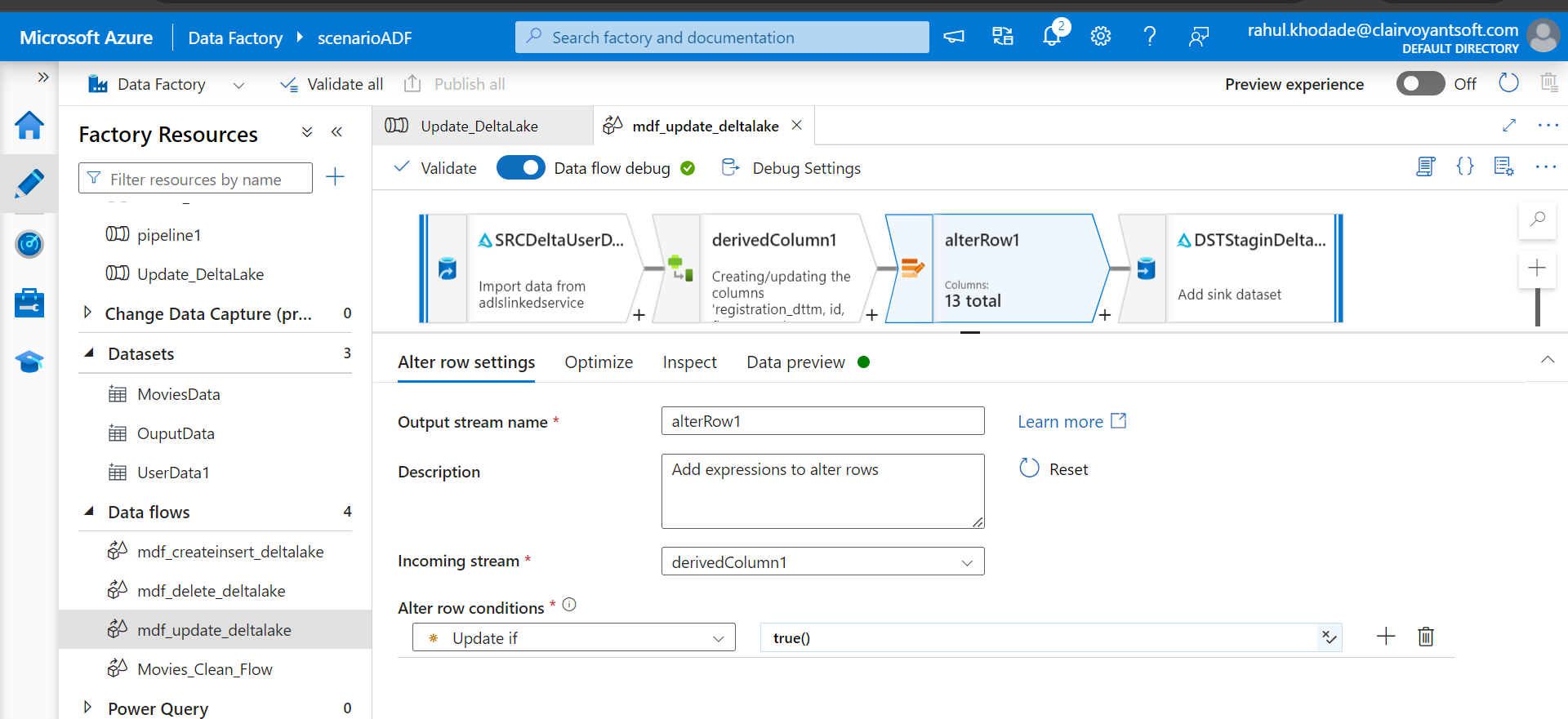
The source data is still our Staging Delta Lake that was also configured for the Inserts



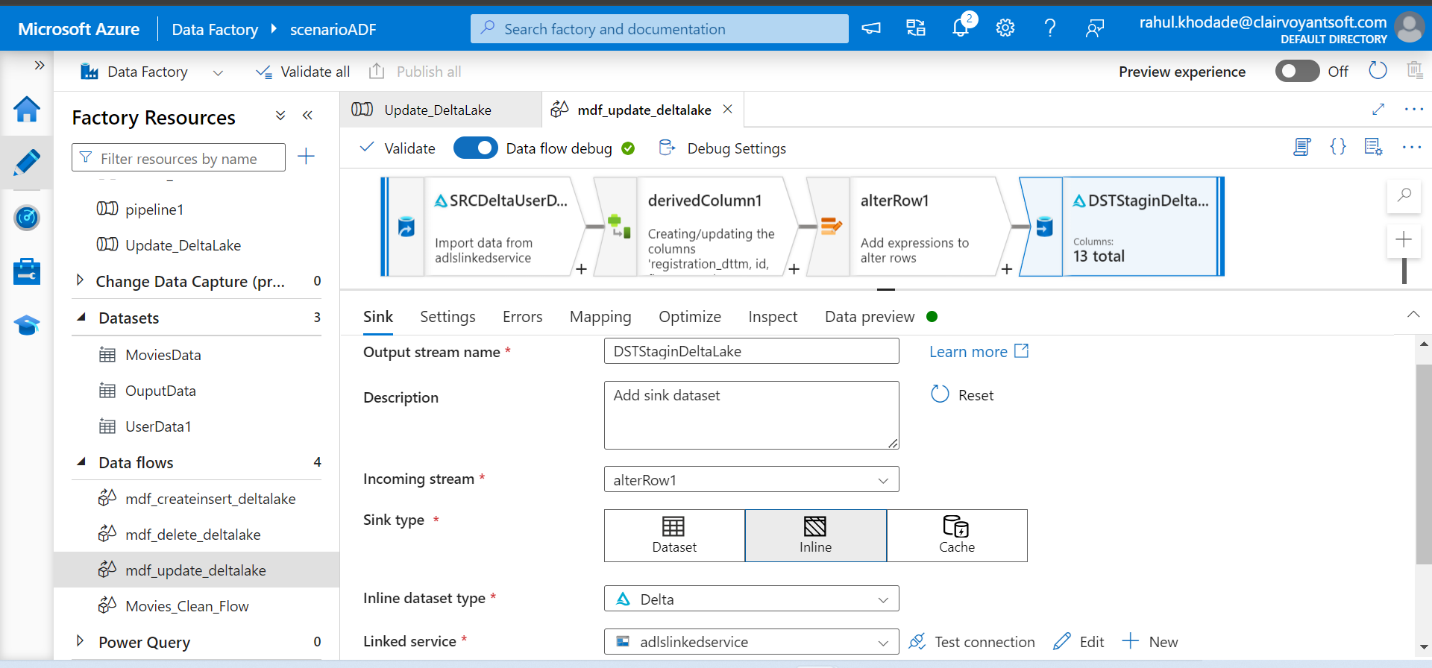
The derived columns convert first and last name to lower case using the following expressions. Mapping Data Flows is capable of handling extremely complex transformations in this stage.



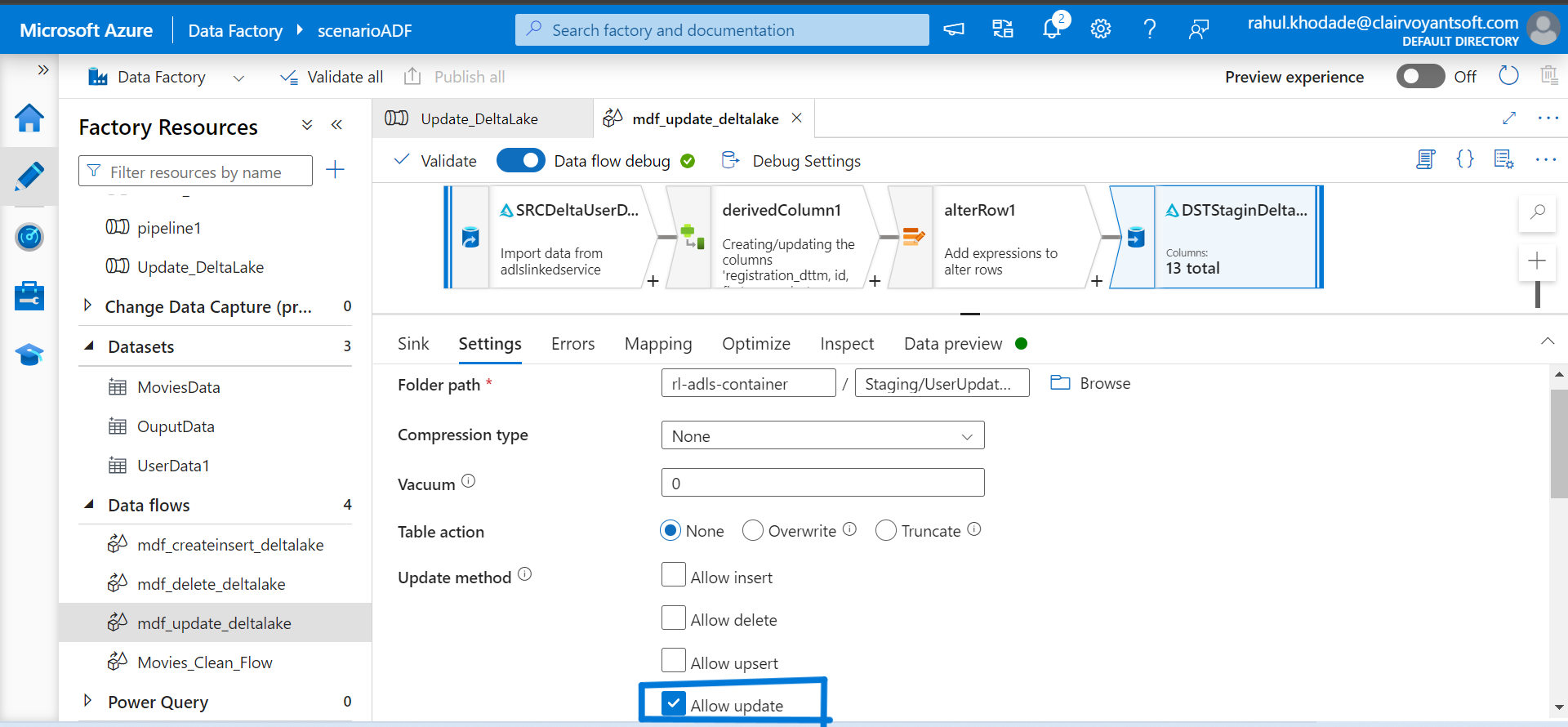
For the alter row settings, we need to specify an Update if condition of true() to update all rows that meet the criteria.



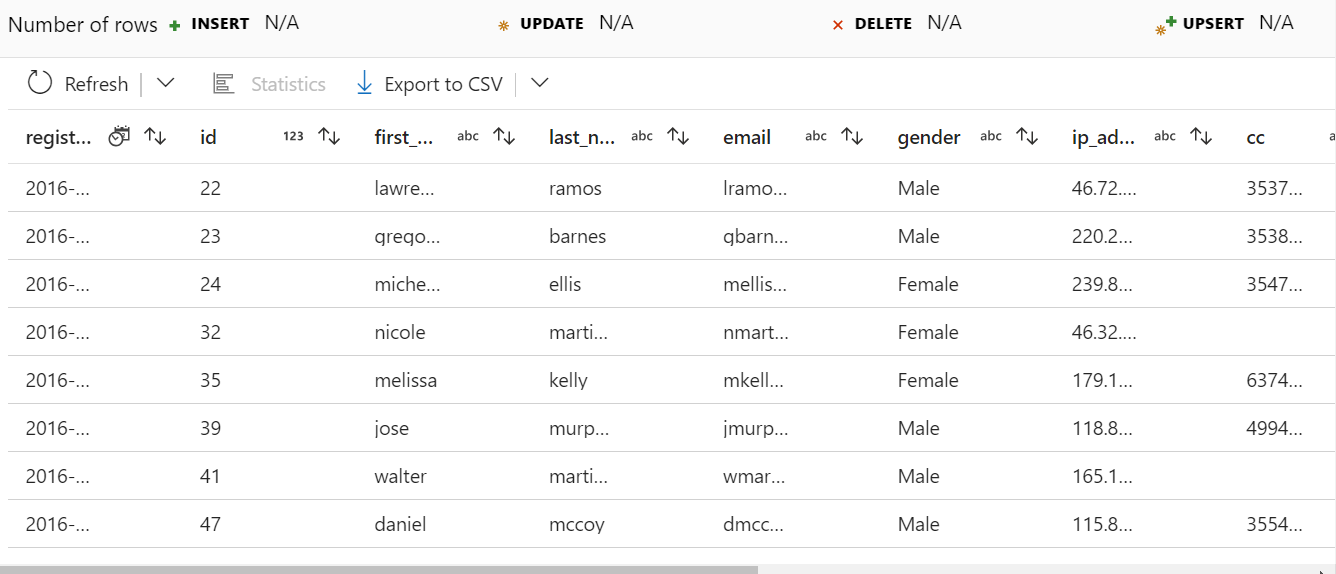
Verify the sink configurations.



Ensure that the sink is still pointing to the Staging Delta Lake data. Also, select Allow Update as the update method. To show that multiple key columns can be simultaneously selected, there are 3 columns selected



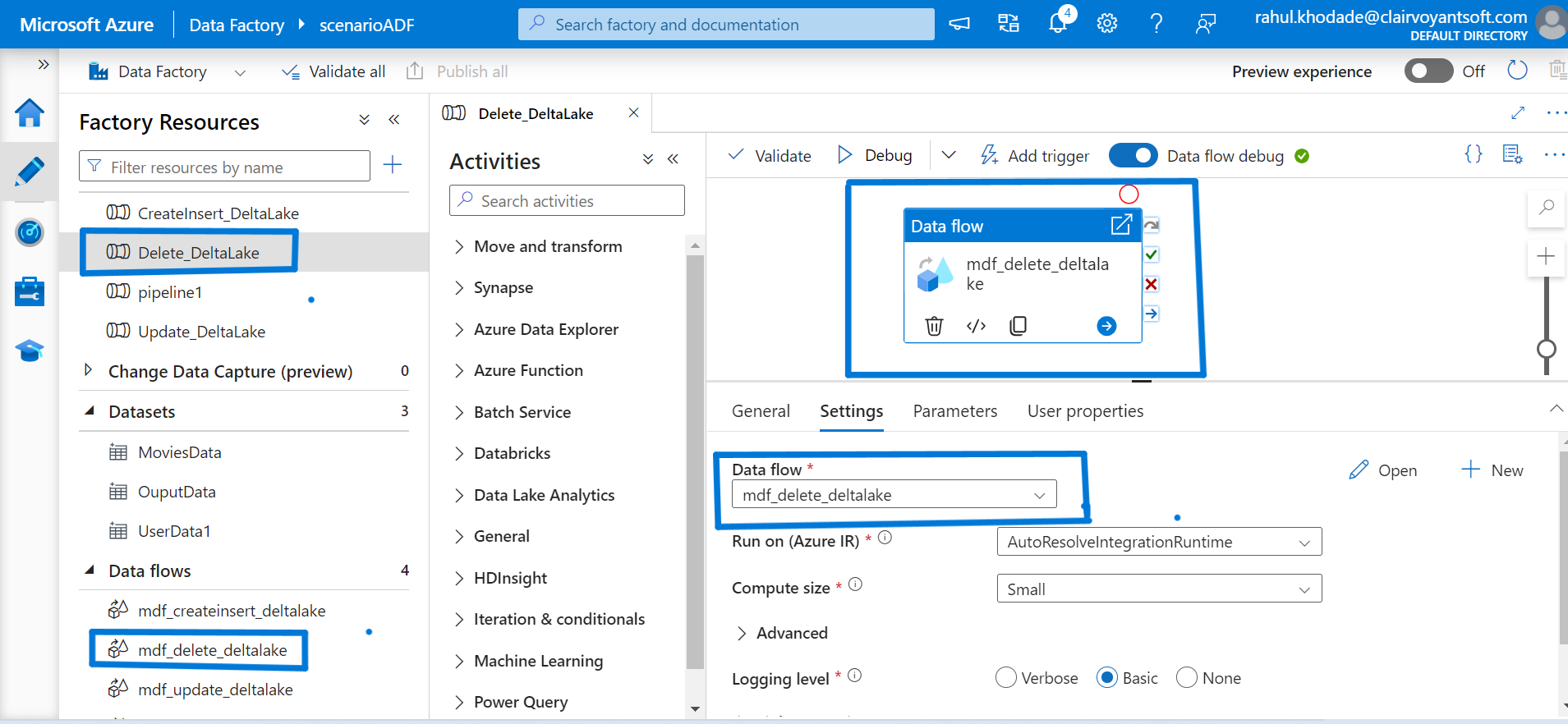
After the pipeline is saved and triggered, we can see that the results reflect the first and last names have been updated to lower case values.



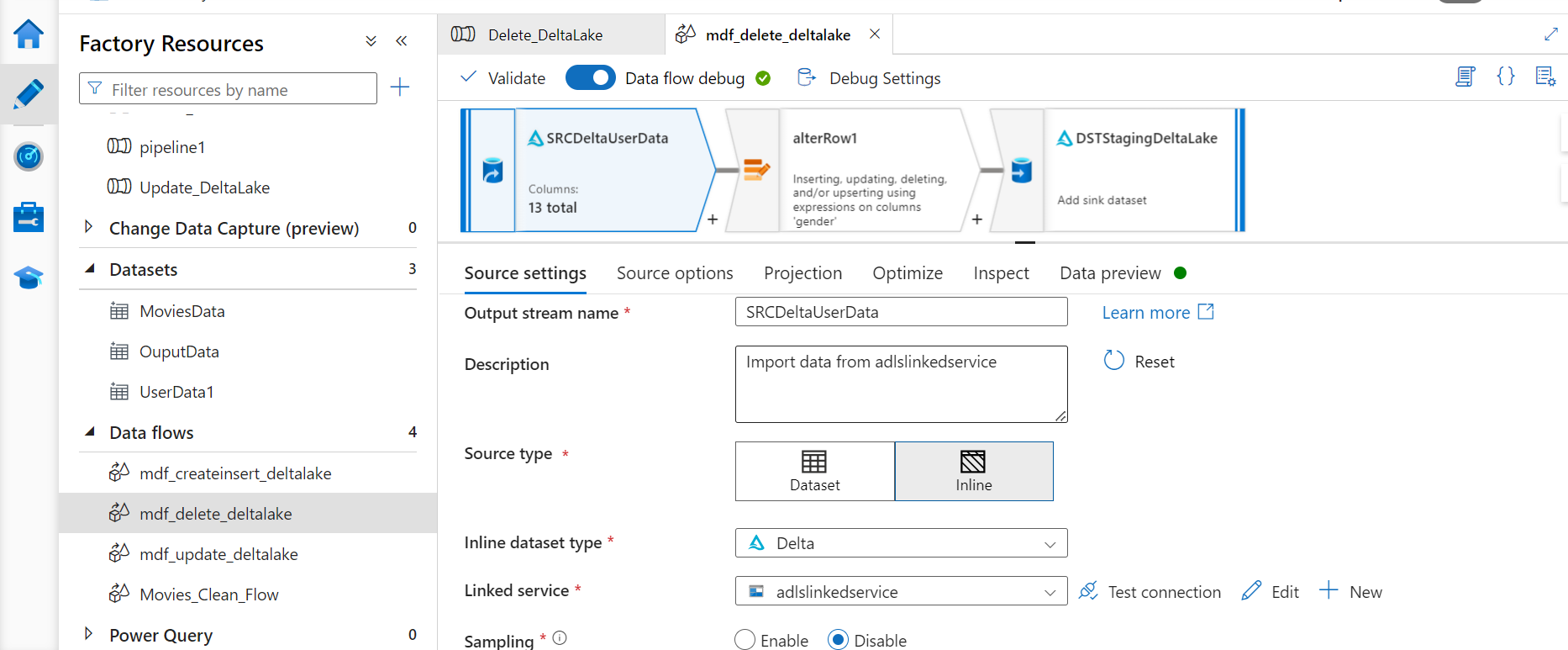
**Delete from Delta Lake**

Next, let's look at an example of how Mapping Data Flows handles deletes in Delta Lake.

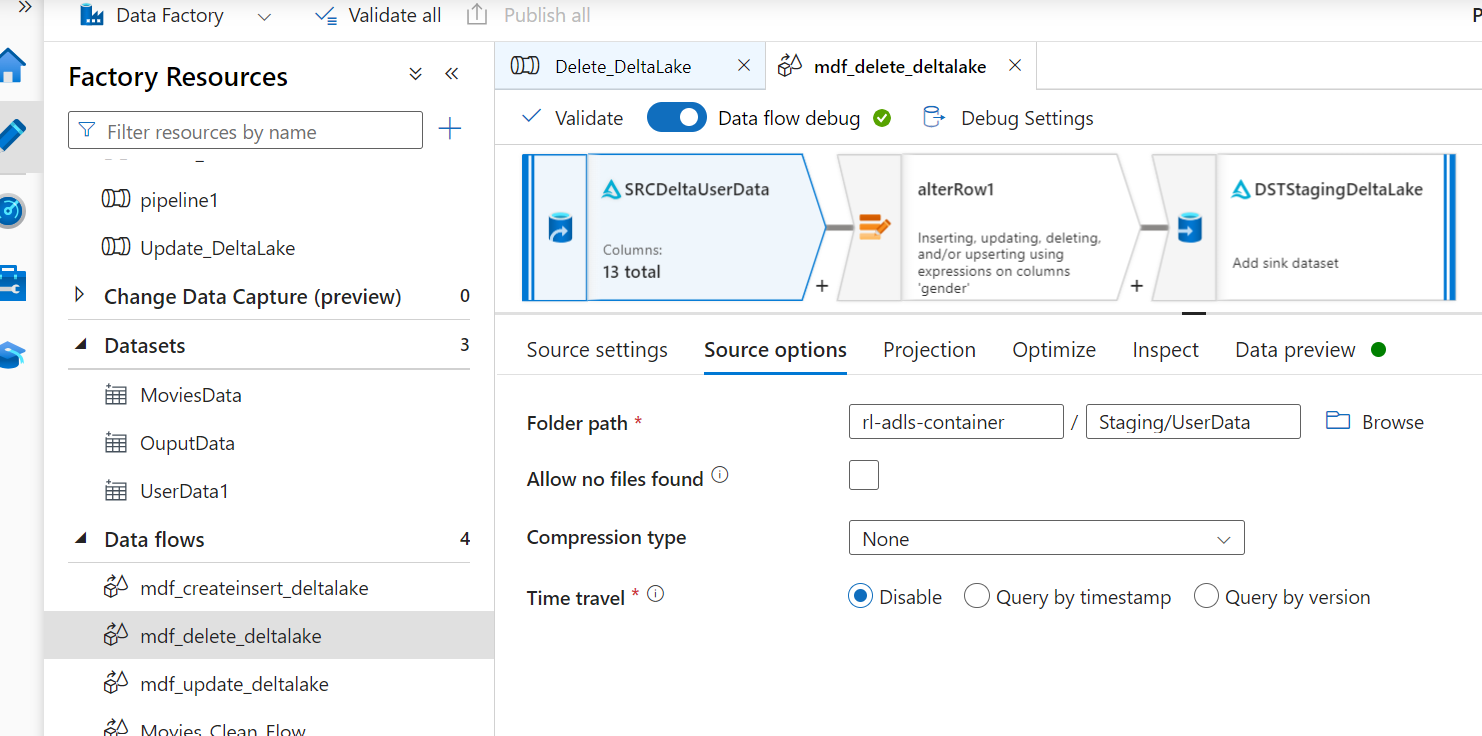
Similar to the Inserts and Updates, create a new Data Factory and Mapping Data Flow.



Configure the Delta source settings.

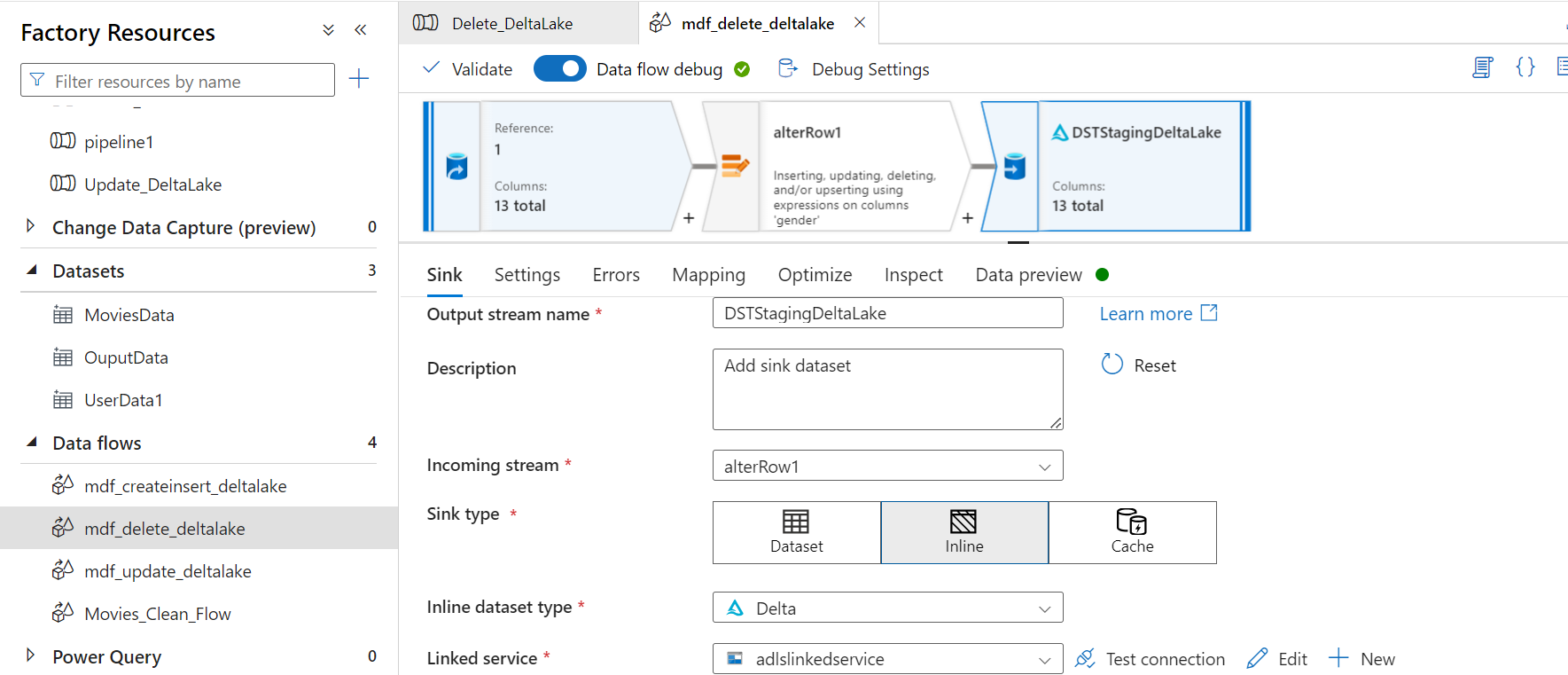


 we are still working with the same Staging Delta Lake, these source settings will be configured similar to Inserts and Updates.

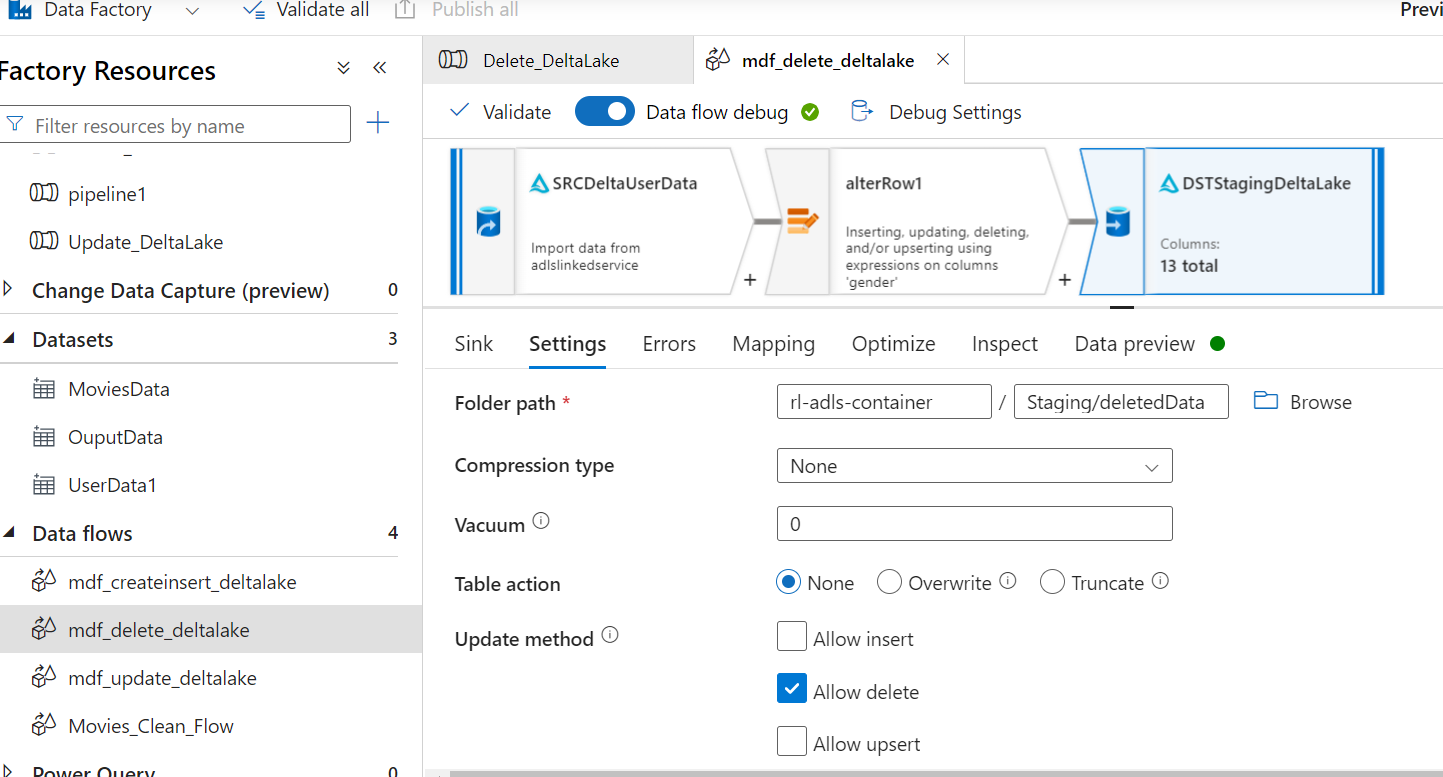


For this example, let's delete all records were gender = male. To do this, we need to configure the alter row conditions to **Delete if gender == 'Male'**.



Finally, configure the sink delta settings.

Select the Staging Delta Lake for the sink, select 'Allow Delete' and select the desired key columns



While looking at the ADLS2 staging folder, we see that a delta\_log folder along with 3 snappy compressed parquet files have been created.

