eBook

Low-Code Apache Spark™ and Delta Lake

A Low-Code Lakehouse Guide

Authors: **Abhinav Jain, Scott Lyons, Maciej Szpakowski**

Data Lakehouse

Data Lakehouse is a relatively new term in the Big Data space that, like many things in the Data ecosystem, is a combination of two existing concepts: Data Warehouses and Data Lakes.

Data Warehouses (DWs) store large volumes of processed and highly-structured data while enabling efficient and performant querying on top of said data. Most DW implementations are based on MPP architectures and allow users to execute common SQL operations for standard databases such as inserts, updates, and even merges. Some are even ACID compliant. However, with the ever-increasing quantities of data and the rise of unstructured data, warehouses become obsolete, expensive, and difficult to maintain.

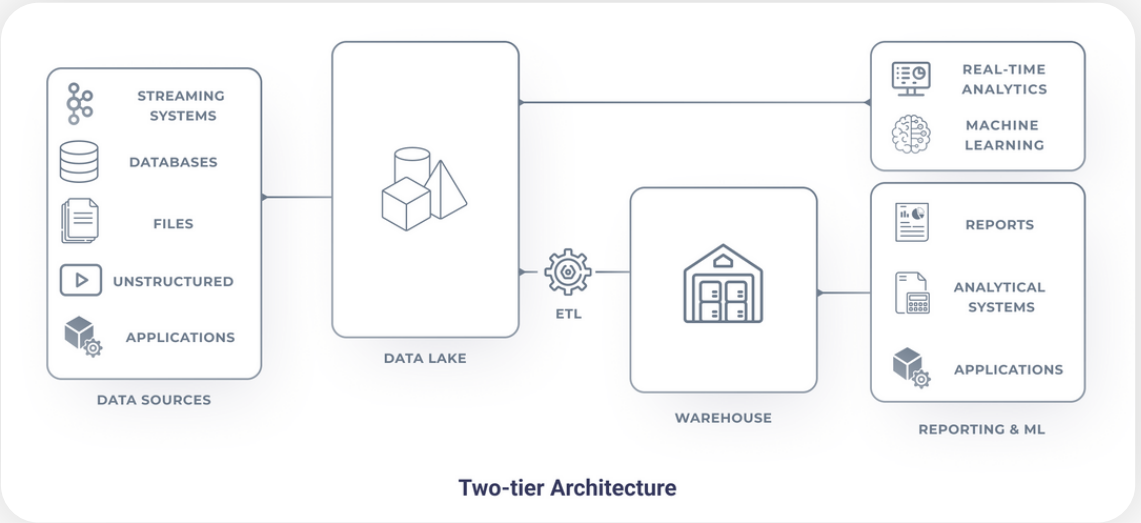
Data Lakes (DLs), while sharing similar underlying concepts, such as high-volume storage and efficient querying, generally tend to be more unstructured by nature. They're frequently used as the pre-processing area by data scientists and engineers to prepare data sourced from a variety of upstream sources for loading into a Data Warehouse.

Data Lakehouse aims to have the best of both worlds. DLs support both Batch and Streaming paradigms and any number of diverse workloads on top of them.

Apache Spark is the ETL/Computation engine of choice that has enabled many companies to perform efficient and scalable data engineering on the cloud without breaking the bank.

The Architecture

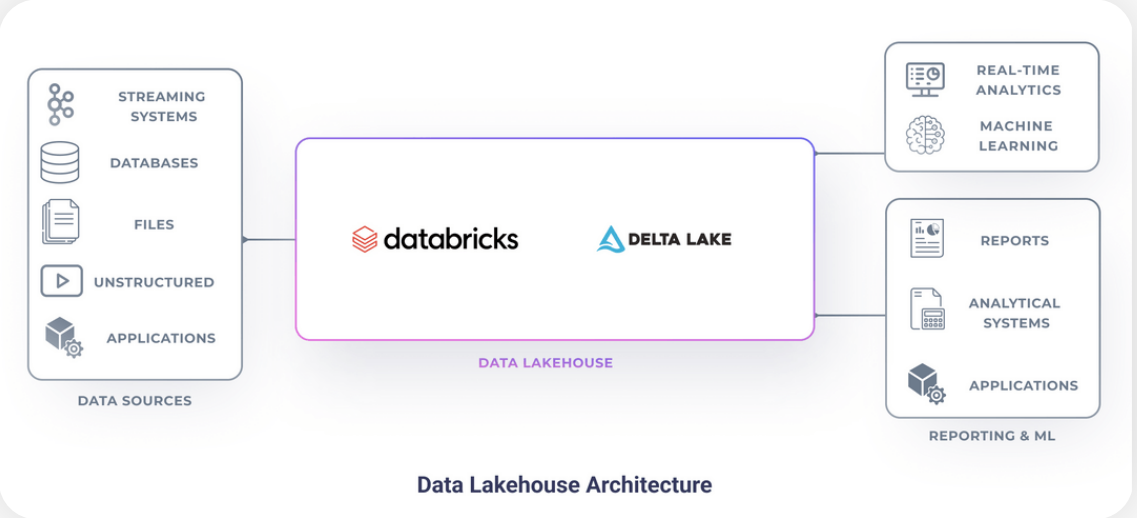
Classically, the need for cost-effective storage, performant processing, and low-latency querying birthed a two-tier architecture. Data Lake for raw storage of unstructured data and Data Warehouse on top of it for high-performance reporting and querying. To integrate those layers ETL batch processes are used.



Two-tier Architecture

Even though this is one of the most commonly used architectures today, its use can result in a number of problems, such as:

- Increased number of bugs due to a larger number of pipelines
- Stale data stored within the data warehouse
- Lack of ability to store unstructured information - SQL is often not enough.

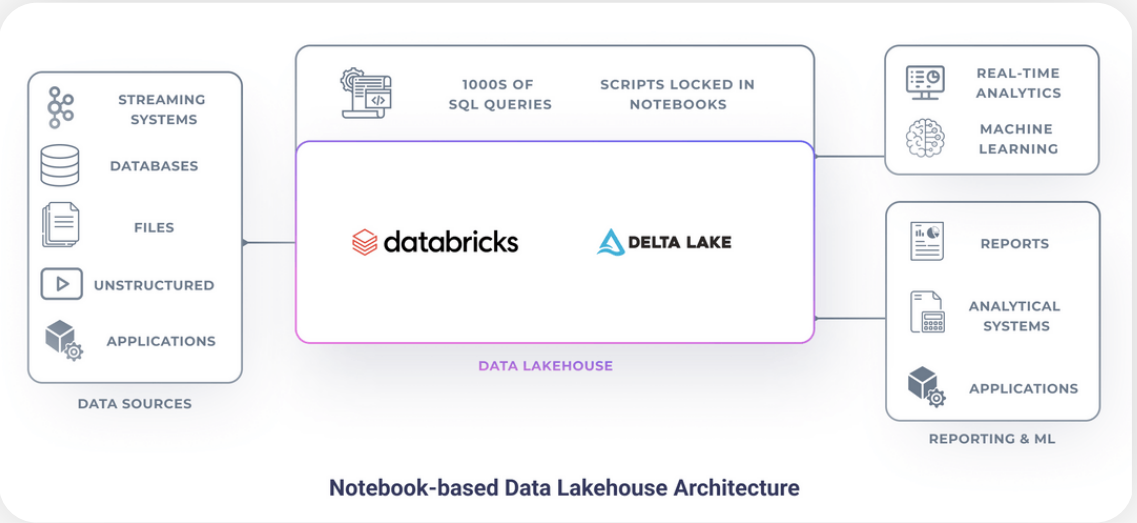


Data Lakehouse Architecture

The introduction of a Lakehouse architecture, using Delta as the underlying storage format and Spark as the querying engine, attempts to solve the two-tiered architecture by unifying it into a single layer. Delta enables the simplicity of a data warehouse with advanced update operations and ACID guarantees while supporting the large majority of Cloud vendor storage offerings. This makes data more accessible, cheaper to query, and reduces the number of buggy data pipelines.

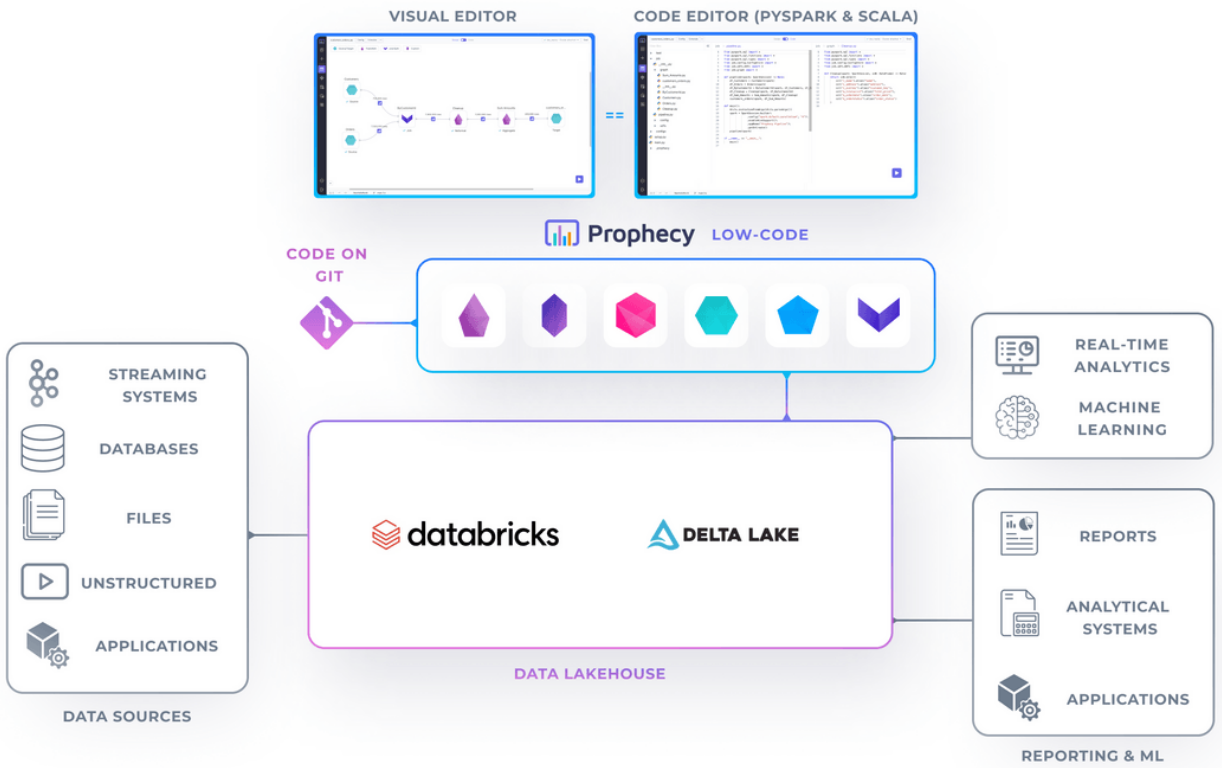
Making Data Lakehouse even easier

Even though Spark and Delta are the perfect basis for your future data pipeline, and the Data Lakehouse architecture allows us to enable a tremendous amount of new use cases, the usability and productivity still remain a challenge. Delta is yet another technology that your team has to learn, requires you to know how to write code in an optimal way, and common use cases like performing slowly changing dimensions require quite a few lines of repeatable code.



Notebook-based Data Lakehouse Architecture

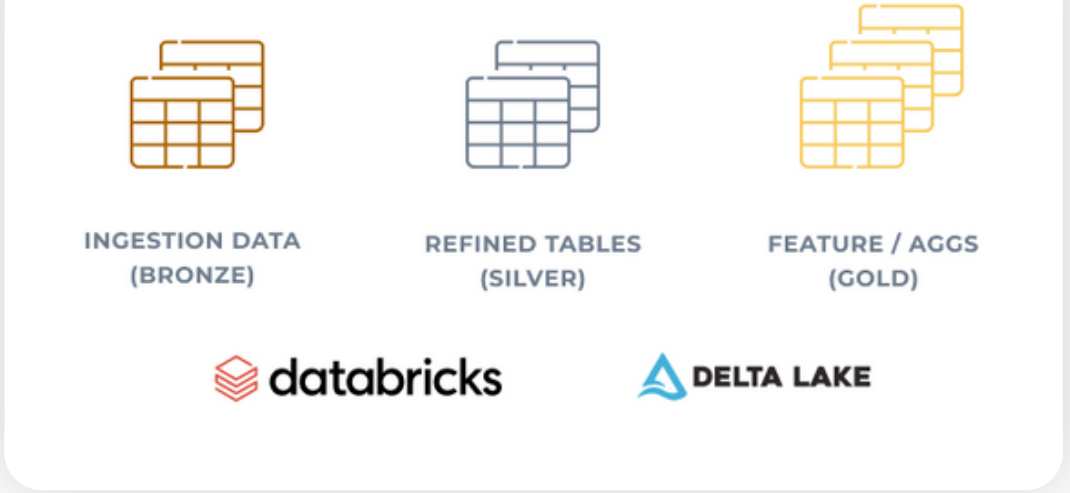
Along comes Prophecy. Gone are the chaotic collection of disconnected ETL notebooks and haphazard orchestration scripts, replaced with a visually built data flow that is turned into well-engineered and efficient Spark or PySpark code stored directly into Git. You can even start leveraging all the best practices that have been classically available only for traditional software engineers, including Unit Testing and CI/CD. Once you're confident that the pipeline will produce the Data you're looking for, you can then use Prophecy to schedule the jobs.



Data Lakehouse Architecture with Prophecy

Bronze, Silver, and Gold layers

With the Data Lakehouse architecture, all your data flows from the operational systems directly to your object store. However, this often results in poor data management and governance, which leads to the creation of a data swamp. To prevent this it's paramount to organize your Lakehouse well. Proper governance, lineage, and taxonomies are also essential (but we will leave that for a future blog).



A common architecture is based on the organization pattern that uses tables that correspond to different quality levels of data:

- **Bronze tables** - raw data directly available after data ingestion, often used as historical store data and a single source of truth
- **Silver tables** - clean transformed data (e.g. clean training data for machine learning - the feature store)
- **Gold tables** - aggregated datasets, ready for consumption by downstream customers (e.g. reports or model predictions)

Building such architecture allows anyone to directly leverage data at different levels, depending on their needs. However, building it often requires multiple data pipelines at each stage that have to be often collaboratively worked upon. Using a low-code tool can make this job significantly easier.

Pipeline example

To demonstrate how this architecture can be built, we're going to use two datasets:

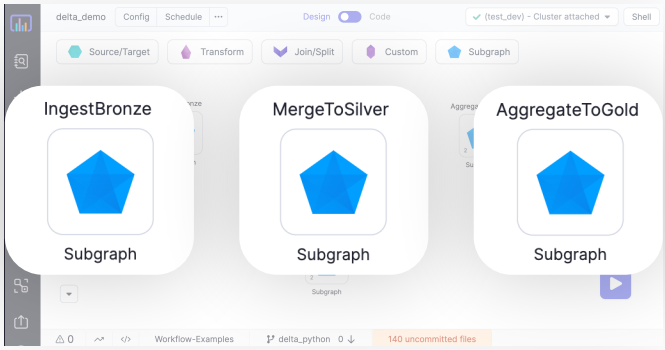
1. Orders - Table that contains a new order for every row. This table is queried directly from an operational system, so every time we query this table it might contain different information for the order_id. As an example, once the order is delivered the order_delivered_dt changes from null to a valid date.

order_id string	customer_id string	order_dt string	order_amount string	order_status string	order_approved_dt string	order_delivered_dt string
1	A	2022-01-12	500	delivered	2022-01-12	2022-01-14
2	A	2022-01-14	1000	approved	2022-01-14	null
3	B	2022-01-12	200	delivered	2022-01-12	2022-01-14
4	B	2022-01-14	200	approved	2022-01-14	null
5	C	2022-01-14	500	approved	2022-01-14	null

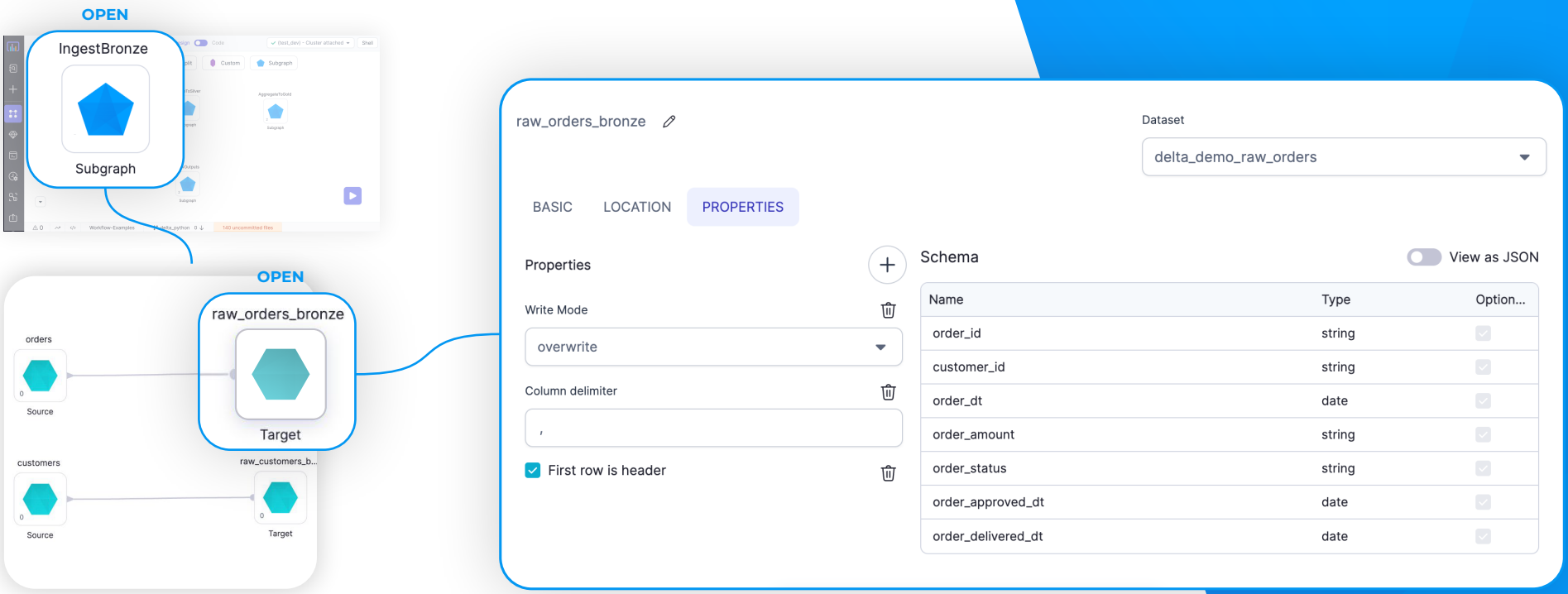
2. Customers - Table that contains a customer per row, with only occasional inserts and updates (e.g. when a customer changes their address).

customer_id string	customer_zip_code string	customer_city string	customer_state string	updated_dt string
A	10000	City1	State1	2022-01-01
B	10010	City2	State1	2022-01-02
C	20000	City3	State2	2022-01-11
D	20010	City4	State2	2022-01-12

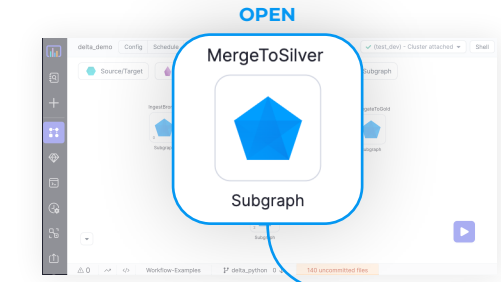
Our reporting pipeline is going to consist of three separate pieces:

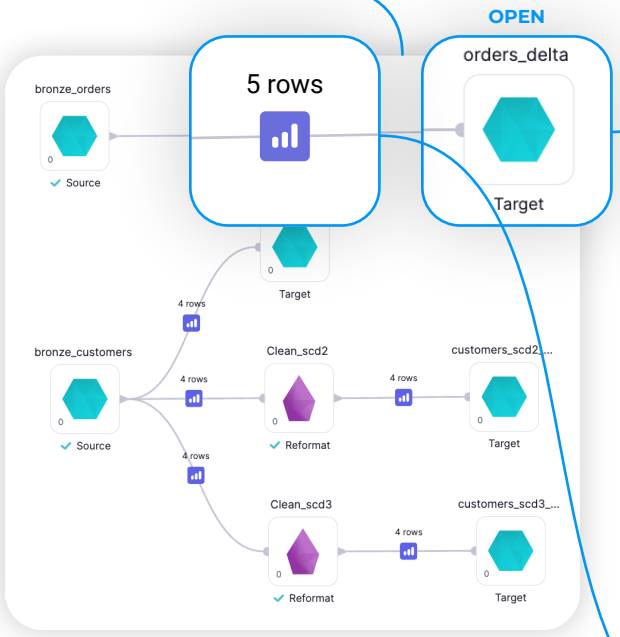


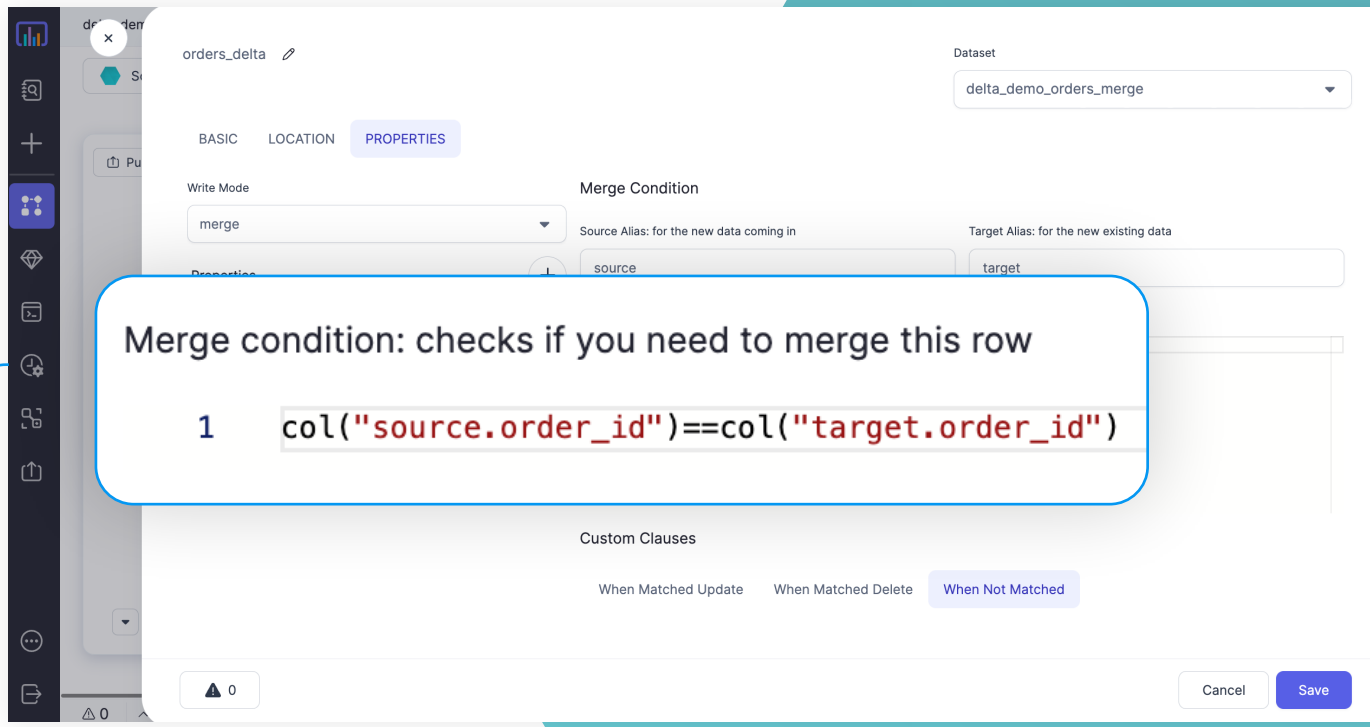
a. **IngestBronze** - ingestion of source data



b. **MergeToSilver** - cleansing the source data and creation of delta tables







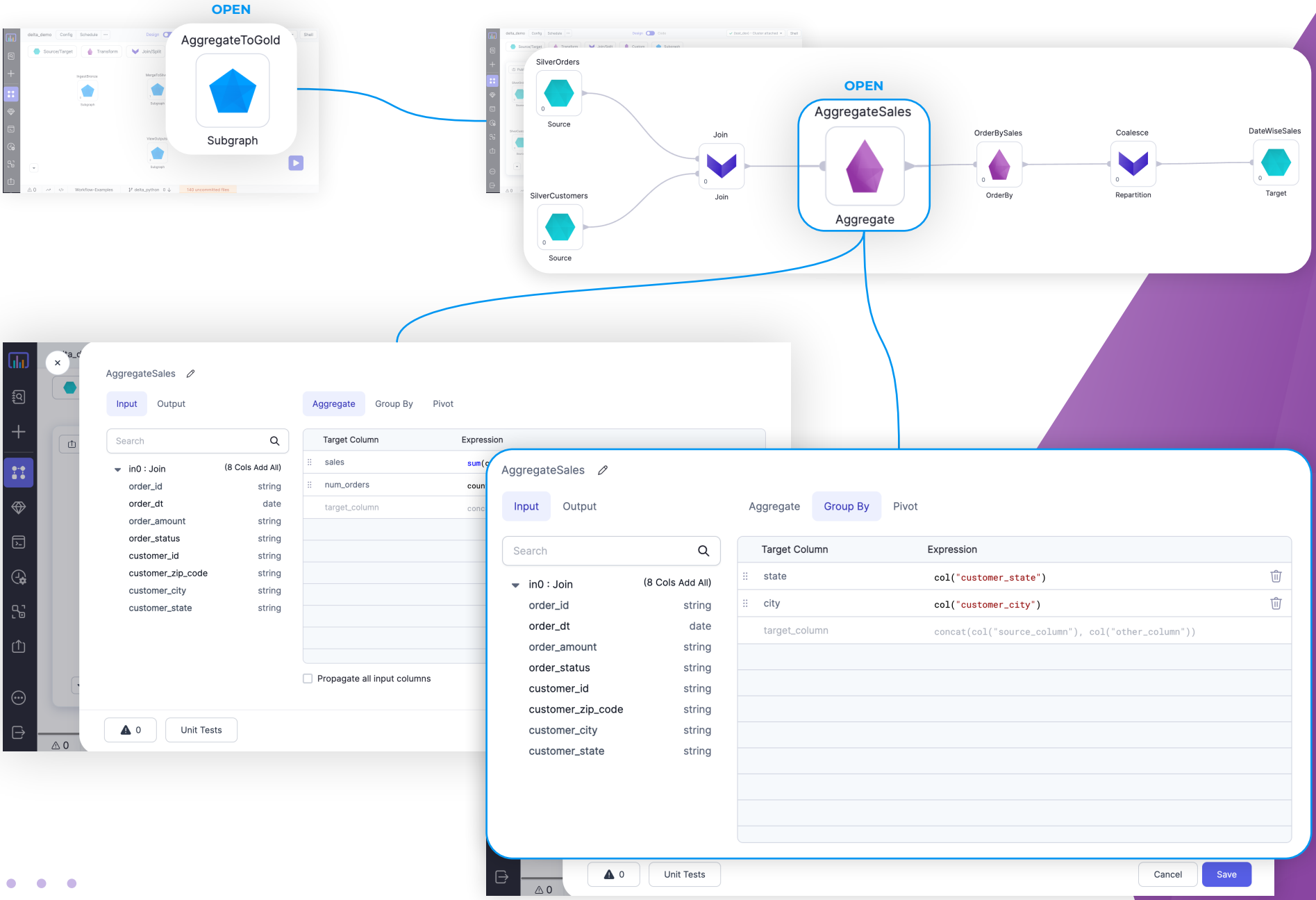
Data samples for bronze_orders (out)

Showing first 5 rows out of 5 rows in 1 Partition

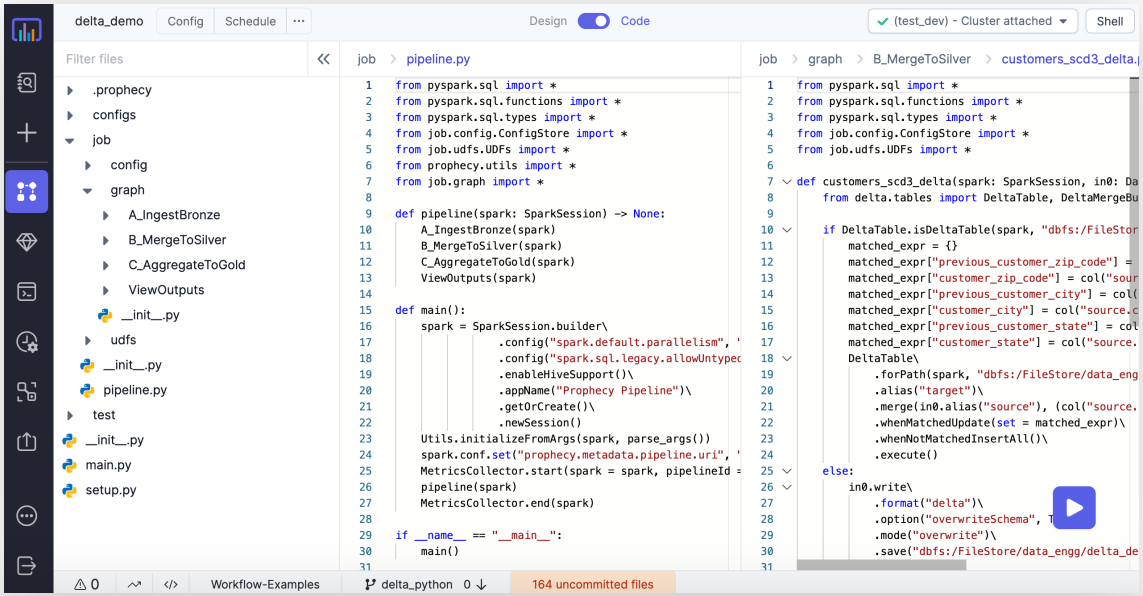
7/7 Columns | Filter Columns

order_id string	customer_id string	order_dt date	order_amount string	order_status string	order_approved_dt date
2	A	2022-01-14	1000	delivered	2022-01-14
4	B	2022-01-14	200	delivered	2022-01-14
6	D	2022-01-16	700	approved	2022-01-16
7	A	2022-01-16	400	approved	2022-01-16
8	C	2022-01-16	800	approved	2022-01-16

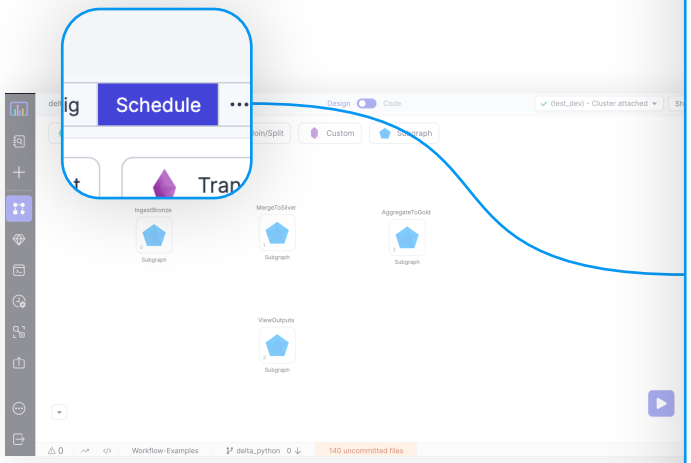
c. **AggregateToGold** - the creation of a business report

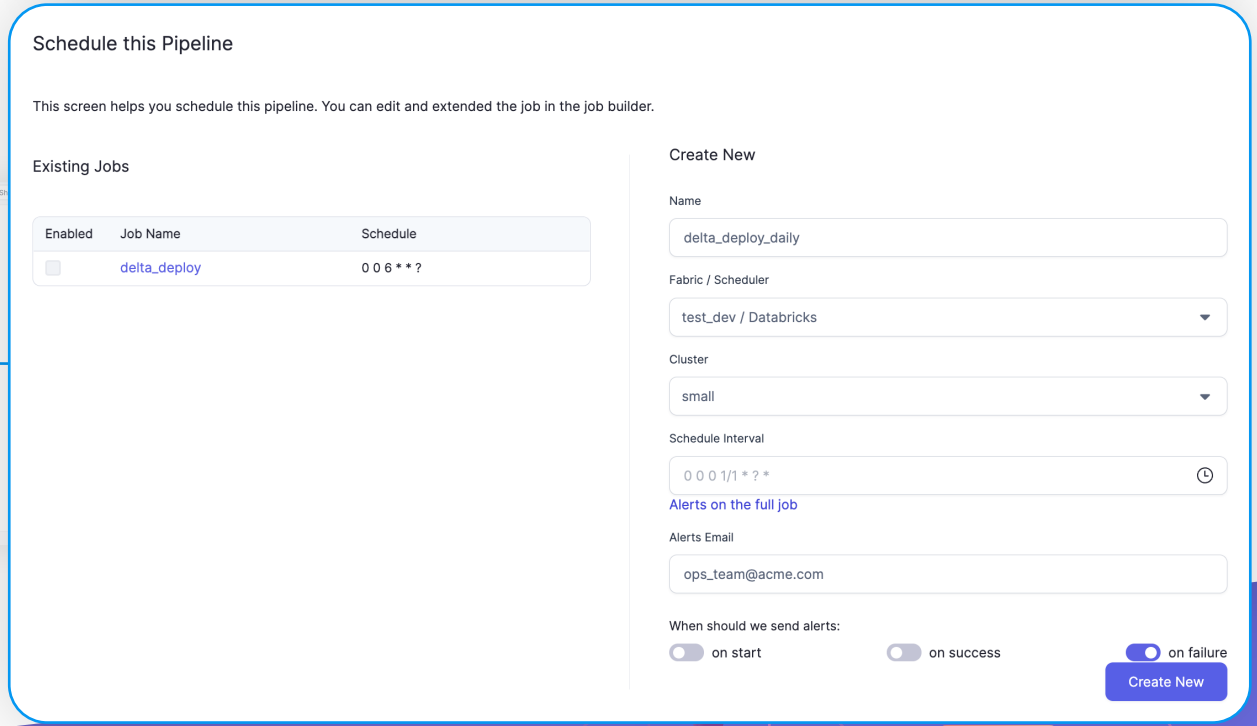


After every change the code for our entire pipeline is automatically generated in the backend in PySpark.



In just a few minutes we've built our report while following the best practices in terms of structuring the code. Now it's time to schedule it.





Slowly Changing Dimensions (SCD)

So far we've built a set of pipelines that produce our report without taking into account any historical data. Historical information, however, can contribute a tremendous value to the business, as it allows us to draw patterns over time.

A commonly followed pattern in the data warehouse space has been called "Slowly changing dimensions". It allows us to store and manage both current and historical data over time directly within our tables. This used to be very difficult to do in a traditional Data Lake due to the lack of table merge and updates. However, Delta makes it possible.

There are three most popular types of SCD:

- **Type 1** - Overwriting
- **Type 2** - Creating dimension records
- **Type 3** - Creating historical columns

Consider a simple example where our initial data contains a Customer table in the operational database. After a Customer changes their address, the row with their corresponding customer_id is updated.

If we're ingesting the data from our operational database using tools like Fivetran or AWS DMS, we'd likely see two files created for us.

customer_id	zip_code	state
123	94105	San Francisco

customers/2022-01-22T00:00:00.parquet

And, after the customer changes their address:

customer_id	zip_code	state
123	10001	New York

customers/2022-05-22T13:23:23.parquet

SCD Type 1

The standard operational databases store data using the SCD1 method to limit the amount of data present in the database at any point in time and speed up the querying.

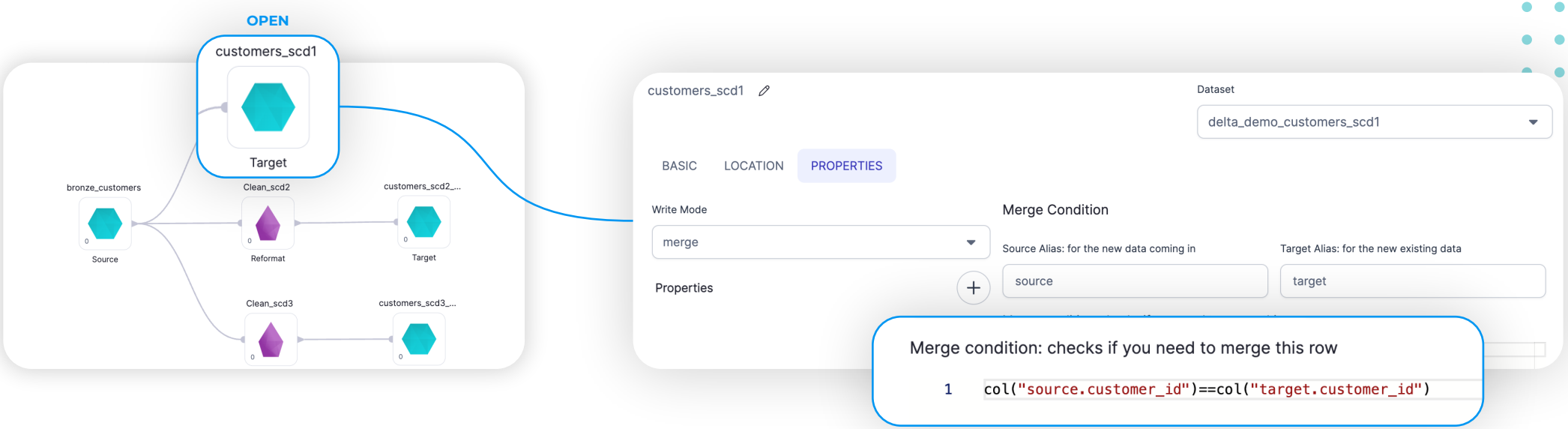
In our Lakehouse, we'd like to store the information using the Type 1 SCD which would simply mean updating and overwriting the row in the existing silver customers table.

This would result in the following table

customer_id string	customer_zip_code string	customer_city string	customer_state string	updated_dt date
A	10001	City1	State1	2022-01-15

customers_silver_scd1

SCD1 doesn't maintain the historical information, thus it's only useful when the change of information over time is not important. Building a pipeline that updates the existing customers, based on the latest record is very simple:

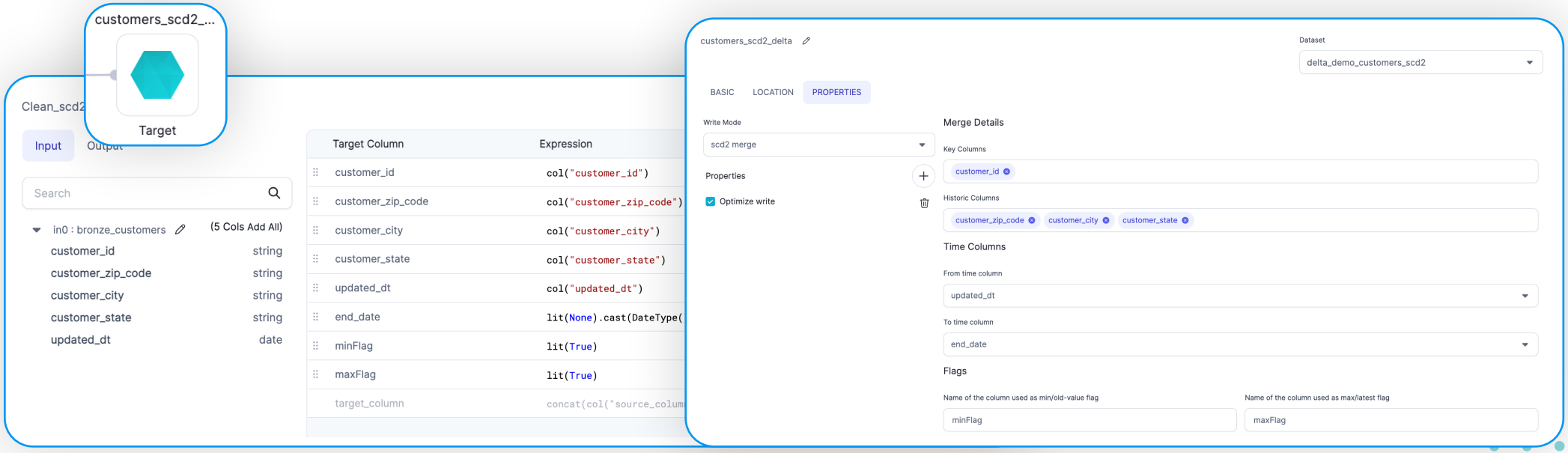


SCD Type 2

SCD2 is another approach to handle slowly changing dimensions. In this case, current and historical data is stored in a data warehouse. The purpose of an SCD2 is to preserve the history of changes. If a customer changes their address, for example, or any other attribute, SCD2 allows analysts to link facts back to the customer and their attributes in the state they were at the time of the fact event.

customer_id string	customer_zip_code string	customer_city string	customer_state string	updated_dt date	end_date date	minFlag boolean	maxFla boolear
A	10000	City1	State1	2022-01-01	2022-01-15	true	false
A	10001	City1	State1	2022-01-15	null	false	true

customers_silver_scd2



Here we have created 3 extra columns which would make it easy to consume this data in several downstream pipelines:

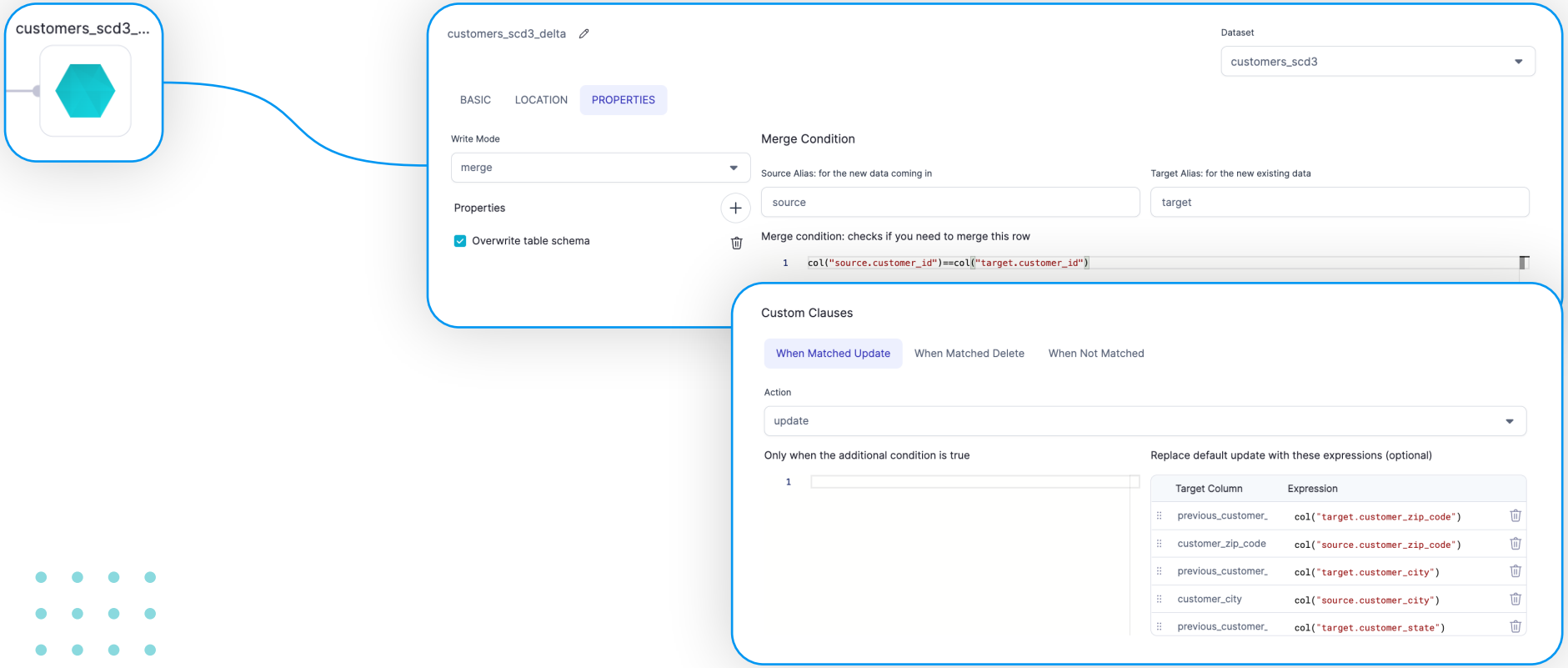
- **end_date** - the date until the row was active
- **minFlag** - if true, the oldest record for that *customer_id*
- **maxFlag** - if true, the latest record for that *customer_id*

SCD Type 3

A Type 3 SCD stores two versions of values for certain selected level attributes. Each record stores the previous value and the current value of the selected attribute. When the value of any of the selected attributes changes, the current value is stored as the old value and the new value becomes the current value.

customer_id string	previous_customer_zip_... string	customer_zip_code string	previous_customer_city string	customer_city string	previous_customer_state string	customer_state string	update date
A	10000	10001	City1	City1	State1	State1	2022-0

customers_silver_scd3



I've configured my slowly changing dimension gems, but how do I know if they are working?
Let's compare SCD1, SCD2, and SCD3 silver tables.

Using SCD1 merge, we don't retain any history. The customer records represent the latest changes, so we know that Customer A lives in zip code 10001.

customer_id string	customer_zip_code string	customer_city string	customer_state string	updated_dt date
A	10001	City1	State1	2022-01-15
B	10010	City2	State1	2022-01-02
C	30000	City3	State2	2022-01-15
D	20010	City4	State2	2022-01-12
E	30001	City5	State3	2022-01-15

customers_silver_scd1

Using SCD2 merge, we retain some interesting history. Customers A and C have moved across zip codes while customers B, D, and E have not. We can see the current zip code for each customer by checking the end_date field. Customer A has moved from 10000 to 10001.

customer_id string	customer_zip_code string	customer_city string	customer_state string	updated_dt date	end_date date	minFlag boolean	maxFlag boolean
A	10000	City1	State1	2022-01-01	2022-01-15	true	false
A	10001	City1	State1	2022-01-15	null	false	true
B	10010	City2	State1	2022-01-02	null	true	true
C	20000	City3	State2	2022-01-11	2022-01-15	true	false
C	30000	City3	State2	2022-01-15	null	false	true
D	20010	City4	State2	2022-01-12	null	true	true
E	30001	City5	State3	2022-01-15	null	true	true

customers_silver_scd2

Using SCD3 merge, we can see Customer A's history in a slightly different format. We retain just one record for each customer, and we can see the current (eg 10001) and previous (eg 10000) zip codes as fields for each record.

customer_id string	previous_customer_zip_... string	customer_zip_code string	previous_customer_city string	customer_city string	previous_customer_state string	customer_state string	updated date
A	10000	10001	City1	City1	State1	State1	2022-0
B	null	10010	null	City2	null	State1	2022-0
C	20000	30000	City3	City3	State2	State2	2022-0
D	null	20010	null	City4	null	State2	2022-0
E	null	30001	null	City5	null	State3	2022-0

customers_silver_scd3

Phew that was a lot! We saw how to organize data in layers our Delta Lake with an example Pipeline in Prophecy. Then we got into the details of merging slowly changing data using SCD1, SCD2, and SCD3 options. Prophecy's low-code environment makes it easy for anyone to build this use case and contribute the resulting 100% open source Spark code to their git repo.

How can I try Prophecy?

Prophecy is available as a SaaS product where you can add your Databricks credentials and start using it with Databricks. You can use an Enterprise Trial with Prophecy's Databricks account for a couple of weeks to kick the tires with examples. We also support installing Prophecy in your network (VPC or on-prem) on Kubernetes.

SIGN UP FOR A TRIAL