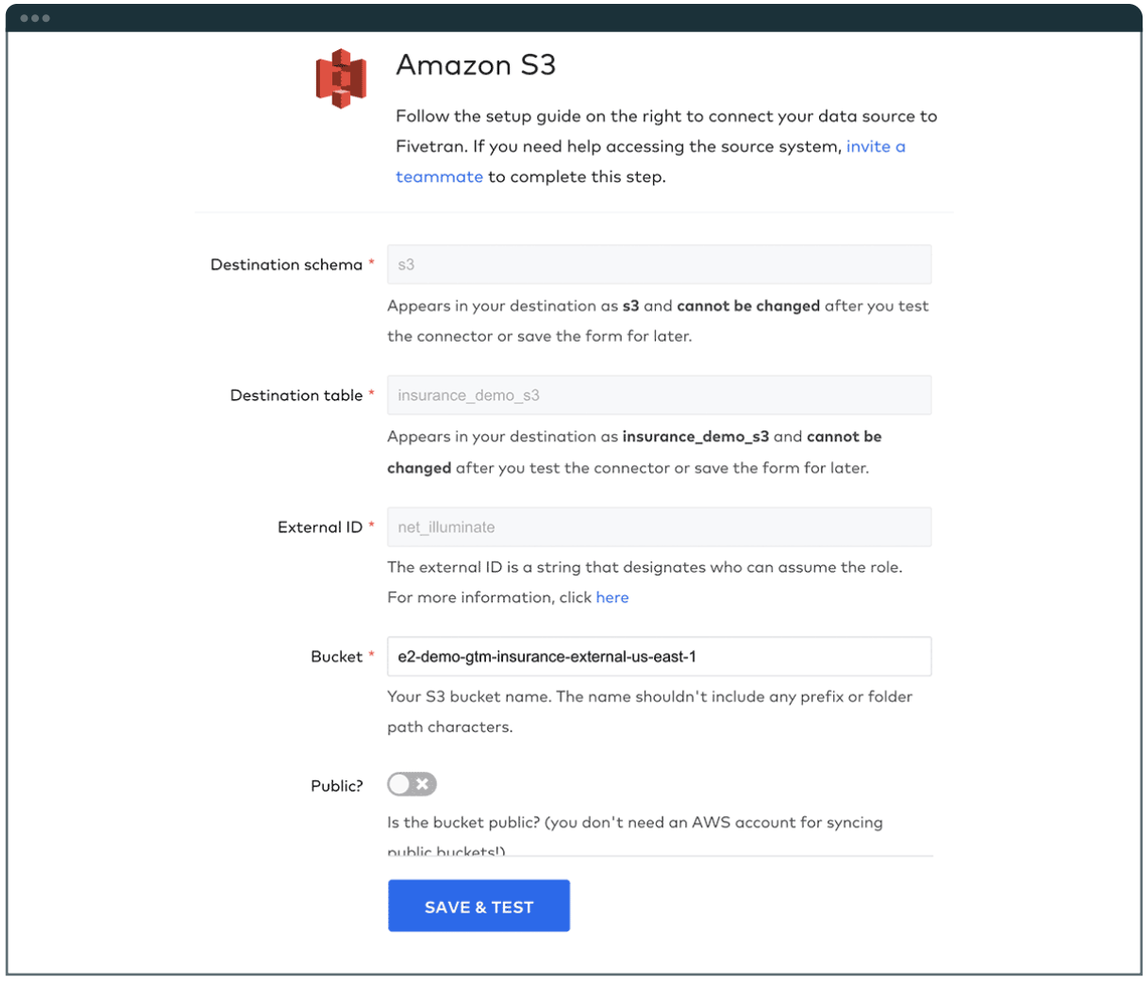

Figure 6: Fivetran interface for configuring a data source connection

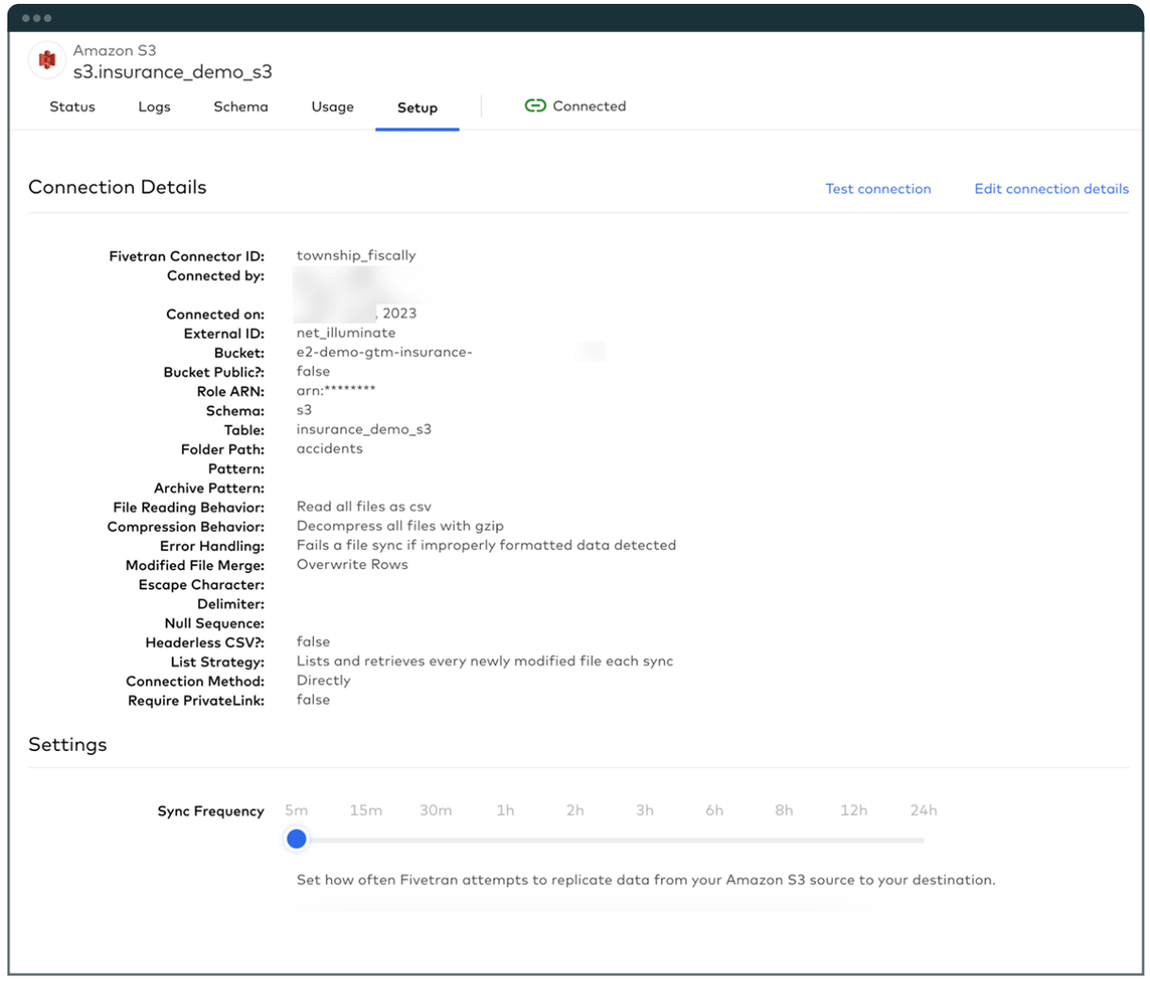

Figure 7: Overview of configuration for a Fivetran connect

Connections can further be configured once they have been validated. One important option to set is the frequency with which Fivetran will interrogate the source system for new data. In Figure 7, we can see how easy Fivetran has made it to set the sync frequency with intervals ranging from 5 minutes to 24 hours.

Fivetran will immediately interrogate and ingest data from source systems once a connection is validated. Data is stored as Delta tables and can be viewed from within Databricks through the DB SQL Data Explorer. By default, Fivetran will store all data under the Hive metastore. A new schema is created for each new connection, and each schema will contain at least two tables: one containing the data and another with logs from each attempted ingestion cycle (see Figure 8).

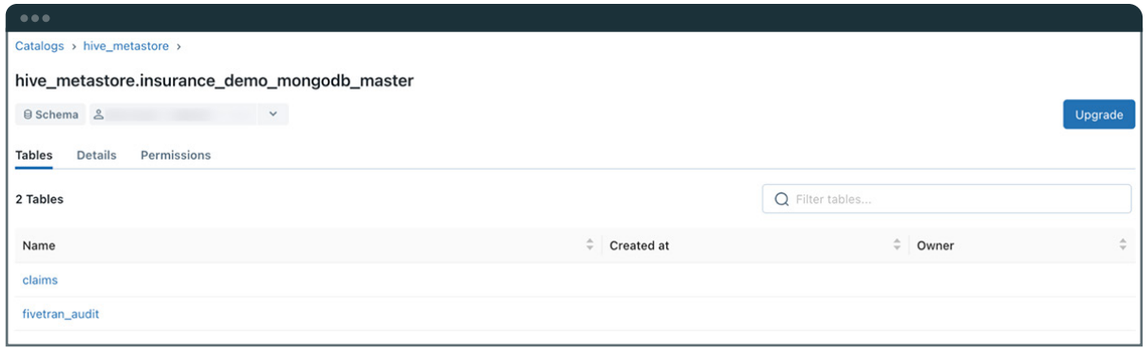

Figure 8: Summary of tables created by Fivetran in the Databricks Warehouse for an example connection

Having the data stored in Delta tables is a significant advantage. Delta Lake natively supports granular data versioning, meaning we can time travel through each ingestion cycle (see Figure 9). We can use DB SQL to interrogate specific versions of the data to analyze how the source records evolved.

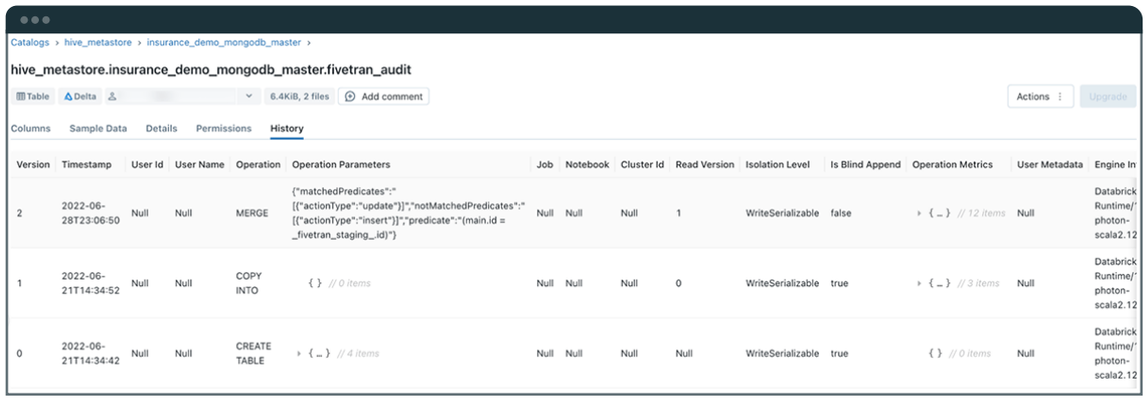

Figure 9: View of the history showing changes made to the Fivetran audit table

It is important to note that if the source data contains semi-structured or unstructured values, those attributes will be flattened during the conversion process. This means that the results will be stored in grouped text-type columns, and these entities will have to be dissected and unpacked with DLT in the curation process to create separate attributes.

STEP 2: AUTOMATING THE WORKFLOW

With the data in the Databricks Data Intelligence Platform, we can use Delta Live Tables (DLT) to build a simple, automated data engineering workflow. DLT provides a declarative framework for specifying detailed feature engineering steps. Currently, DLT supports APIs for both Python and SQL. In this example, we will use Python APIs to build our workflow.

The most fundamental construct in DLT is the definition of a table. DLT interrogates all table definitions to create a comprehensive workflow for how data should be processed. For instance, in Python, tables are created using function definitions and the `dlt.table` decorator (see example of Python code below). The decorator is used to specify the name of the resulting table, a descriptive comment explaining the purpose of the table, and a collection of table properties.

```
@dlt.table(
   name             = "curated_claims",
   comment          = "Curated claim records",
   table_properties = {
       "layer": "silver",
       "pipelines.autoOptimize.managed": "true",
       "delta.autoOptimize.optimizeWrite": "true",
       "delta.autoOptimize.autoCompact": "true"
   }
)
def curate_claims():
   # Read the staged claim records into memory
   staged_claims = dlt.read("staged_claims")
   # Unpack all nested attributes to create a flattened table structure
   curated_claims = unpack_nested(df = staged_claims, schema = schema_claims)

   ...
```

Instructions for feature engineering are defined inside the function body using standard PySpark APIs and native Python commands. The following example shows how PySpark joins claims records with data from the policies table to create a single, curated view of claims.

```
...

   # Read the staged claim records into memory
   curated_policies = dlt.read("curated_policies")
   # Evaluate the validity of the claim
   curated_claims = curated_claims \
       .alias("a") \
       .join(
           curated_policies.alias("b"),
           on  = F.col("a.policy_number") == F.col("b.policy_number"),
           how = "left"
       ) \
       .select([F.col(f"a.{c}") for c in curated_claims.columns] + [F.col(f"b.{c}").
alias(f"policy_{c}")  for c in ("effective_date", "expiry_date")]) \
       .withColumn(
           # Calculate the number of months between coverage starting and the
claim being filed
           "months_since_covered", F.round(F.months_between(F.col("claim_date"),
F.col("policy_effective_date")))
       ) \
       .withColumn(
           # Check if the claim was filed before the policy came into effect
           "claim_before_covered", F.when(F.col("claim_date") < F.col("policy_
effective_date"), F.lit(1)).otherwise(F.lit(0))
       ) \
       .withColumn(
           # Calculate the number of days between the incident occurring and the
claim being filed
           "days_between_incident_and_claim", F.datediff(F.col("claim_date"),
F.col("incident_date"))
       )

   # Return the curated dataset
   return curated_claims
```

One significant advantage of DLT is the ability to specify and enforce data quality standards. We can set expectations for each DLT table with detailed data quality constraints that should be applied to the contents of the table. Currently, DLT supports expectations for three different scenarios:

DECORATOR	DESCRIPTION
expect	Retain records that violate expectations
expect_or_drop	Drop records that violate expectations
expect_or_fail	Halt the execution if any record(s) violate constraints

Expectations can be defined with one or more data quality constraints. Each constraint requires a description and a Python or SQL expression to evaluate. Multiple constraints can be defined using the expect_all, expect_all_or_drop, and expect_all_or_fail decorators. Each decorator expects a Python dictionary where the keys are the constraint descriptions, and the values are the respective expressions. The example below shows multiple data quality constraints for the retain and drop scenarios described above.

```python
@dlt.expect_all({
    "valid_driver_license": "driver_license_issue_date > (current_date() -
cast(cast(driver_age AS INT) AS INTERVAL YEAR))",
    "valid_claim_amount": "total_claim_amount > 0",
    "valid_coverage": "months_since_covered > 0",
    "valid_incident_before_claim": "days_between_incident_and_claim > 0"
})
@dlt.expect_all_or_drop({
    "valid_claim_number": "claim_number IS NOT NULL",
    "valid_policy_number": "policy_number IS NOT NULL",
    "valid_claim_date": "claim_date < current_date()",
    "valid_incident_date": "incident_date < current_date()",
    "valid_incident_hour": "incident_hour between 0 and 24",
    "valid_driver_age": "driver_age > 16",
    "valid_effective_date": "policy_effective_date < current_date()",
    "valid_expiry_date": "policy_expiry_date <= current_date()"
})
def curate_claims():
    ...
```

We can use more than one Databricks Notebook to declare our DLT tables. Assuming we follow the medallion architecture, we can, for example, use different notebooks to define tables comprising the bronze, silver, and gold layers. The DLT framework can digest instructions defined across multiple notebooks to create a single workflow; all inter-table dependencies and relationships are processed and considered automatically. Figure 10 shows the complete workflow for our claims example. Starting with three source tables, DLT builds a comprehensive pipeline that delivers thirteen tables for business consumption.

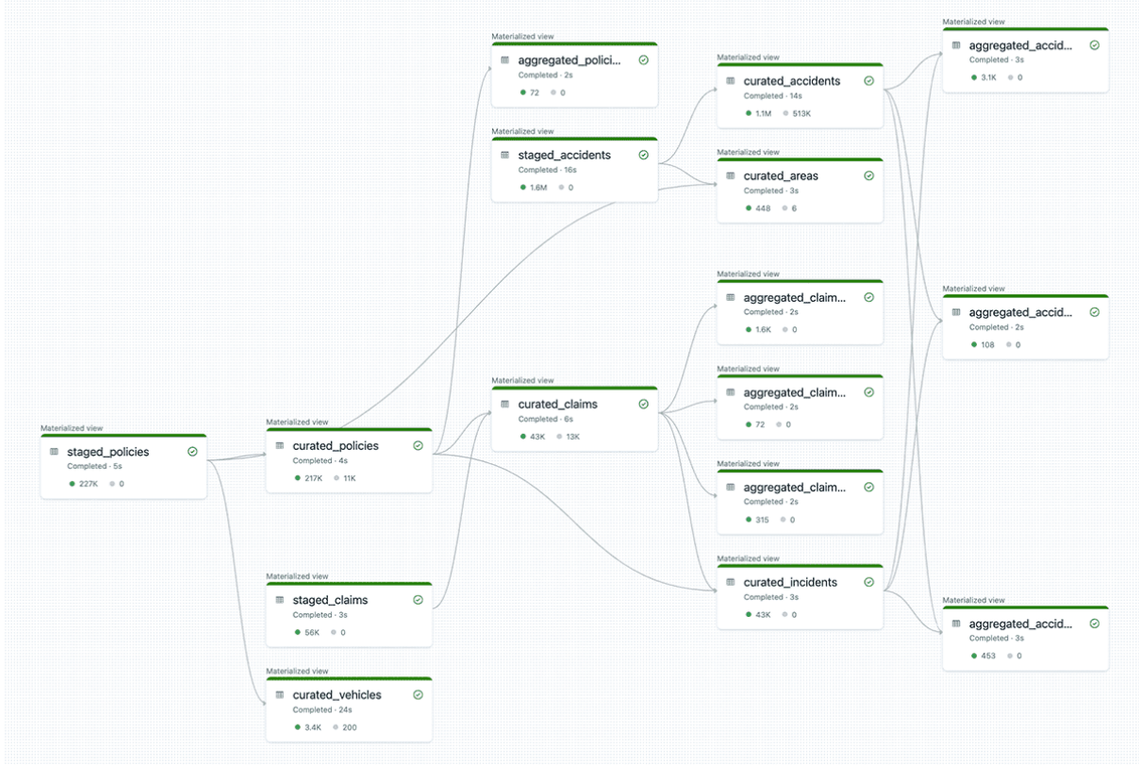

Figure 10: Overview of a complete Delta Live Tables (DLT) workflow

Results for each table can be inspected by selecting the desired entity. Figure 11 provides an example of the results of the curated claims table. DLT provides a high-level overview of the results from the data quality controls:

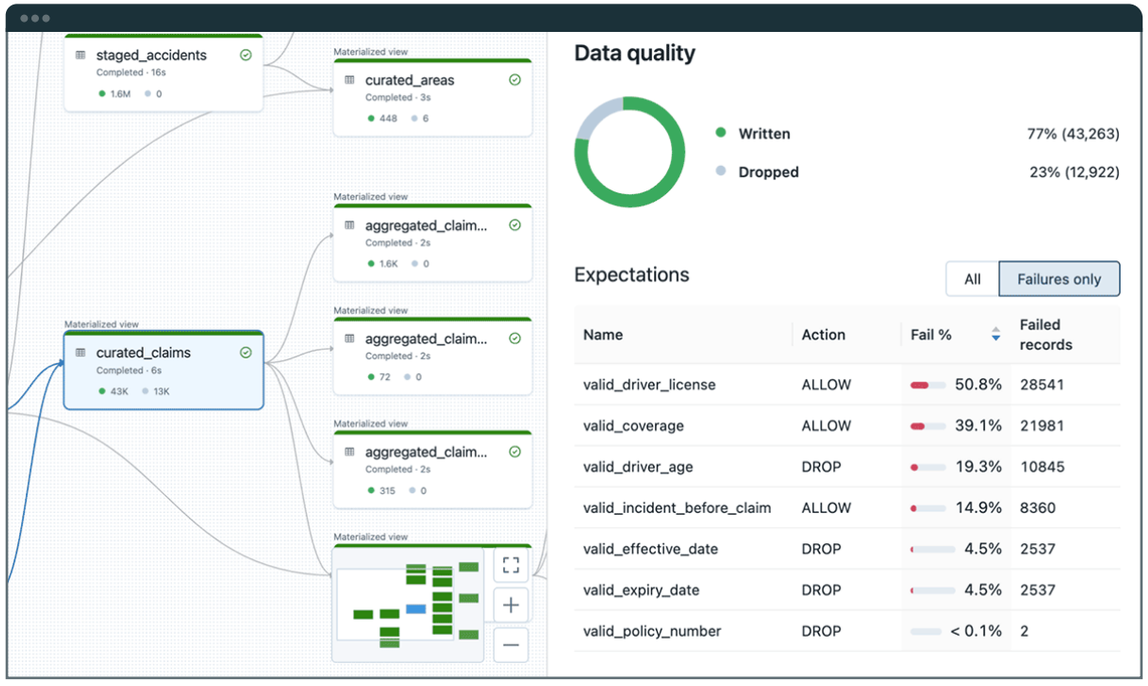

Figure 11: Example of detailed view for a Delta Live Tables (DLT) table entity with the associated data quality report

Results from the data quality expectations can be analyzed further by querying the event log. The event log contains detailed metrics about all expectations defined for the workflow pipeline. The query below provides an example for viewing key metrics from the last pipeline update, including the number of records that passed or failed expectations:

```
SELECT
  row_expectations.dataset AS dataset,
  row_expectations.name AS expectation,
  SUM(row_expectations.passed_records) AS passing_records,
  SUM(row_expectations.failed_records) AS failing_records
FROM
  (
    SELECT
      explode(
        from_json(
          details :flow_progress :data_quality :expectations,
          "array<struct<name: string, dataset: string, passed_records: int, failed_
records: int>>"
        )
      ) row_expectations
    FROM
      event_log_raw
    WHERE
      event_type = 'flow_progress'
      AND origin.update_id = '${latest_update.id}'
  )
GROUP BY
  row_expectations.dataset,
  row_expectations.name;
```

Again, we can view the complete history of changes made to each DLT table by looking at the Delta history logs (see Figure 12). It allows us to understand how tables evolve over time and investigate complete threads of updates if a pipeline fails.

Columns | Sample Data | Details | Permissions | History

Version	Timestamp	User Id	User Name	Operation	Operation Parameters	Job	Notebook	Cluster Id	Read Version	Isolation Level	Is Blind Append	Operation Metrics	User Metadata	Engine Info
12	2023-01-22T19:10:09	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	11	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-1eca0d9-750b289-9ea72db-custom-local
11	2022-12-09T11:48:23	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	10	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-ed5cc83-e81c5c7-17c692e-custom-local
10	2022-11-08T19:48:31	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	9	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-de92f9e-8a33b70-a333f54-custom-local

Figure 12: View the history of changes made to a resulting Delta Live Tables (DLT) table entity

We can further use change data capture (CDC) to update tables based on changes in the source datasets. DLT CDC supports updating tables with slow-changing dimensions (SCD) types 1 and 2.

We have one of two options for our batch process to trigger the DLT pipeline. We can use the Databricks Auto Loader to incrementally process new data as it arrives in the source tables or create scheduled jobs that trigger at set times or intervals. In this example, we opted for the latter with a scheduled job that executes the DLT pipeline every five minutes.

OPERATIONALIZING THE OUTPUTS

The ability to incrementally process data efficiently is only half of the equation. Results from the DLT workflow must be operationalized and delivered to business users. In our example, we can consume outputs from the DLT pipeline through ad hoc analytics or prepacked insights made available through an interactive dashboard.

AD HOC ANALYTICS

Databricks SQL (or DB SQL) provides an efficient, cost-effective data warehouse on top of the Data Intelligence Platform. It allows us to run our SQL workloads directly against the source data with up to 12x better price/performance than its alternatives.

We can leverage DB SQL to perform specific ad hoc queries against our curated and aggregated tables. We might, for example, run a query against the curated policies table that calculates the total exposure. The DB SQL query editor provides a simple, easy-to-use interface to build and execute such queries (see example below).

```
SELECT
  round(curr.total_exposure, 0) AS total_exposure,
  round(prev.total_exposure, 0) AS previous_exposure
FROM
  (
    SELECT
      sum(sum_insured) AS total_exposure
    FROM
      insurance_demo_lakehouse.curated_policies
    WHERE
      expiry_date > '{{ date.end }}'
      AND (effective_date <= '{{ date.start }}'
        OR (effective_date BETWEEN '{{ date.start }}' AND '{{ date.end }}'))
  ) curr
  JOIN
  (
    SELECT
      ...
```

We can also use the DB SQL query editor to run queries against different versions of our Delta tables. For example, we can query a view of the aggregated claims records for a specific date and time (see example below). We can further use DB SQL to compare results from different versions to analyze only the changed records between those states.

```
SELECT
  *
FROM
  insurance_demo_lakehouse.aggregated_claims_weekly TIMESTAMP AS OF '2022-06-05T17:00:00';
```

DB SQL offers the option to use a serverless compute engine, eliminating the need to configure, manage or scale cloud infrastructure while maintaining the lowest possible cost. It also integrates with alternative SQL workbenches (e.g., DataGrip), allowing analysts to use their favorite tools to explore the data and generate insights.

BUSINESS INSIGHTS

Finally, we can use DB SQL queries to create rich visualizations on top of our query results. These visualizations can then be packaged and served to end users through interactive dashboards (see Figure 13).

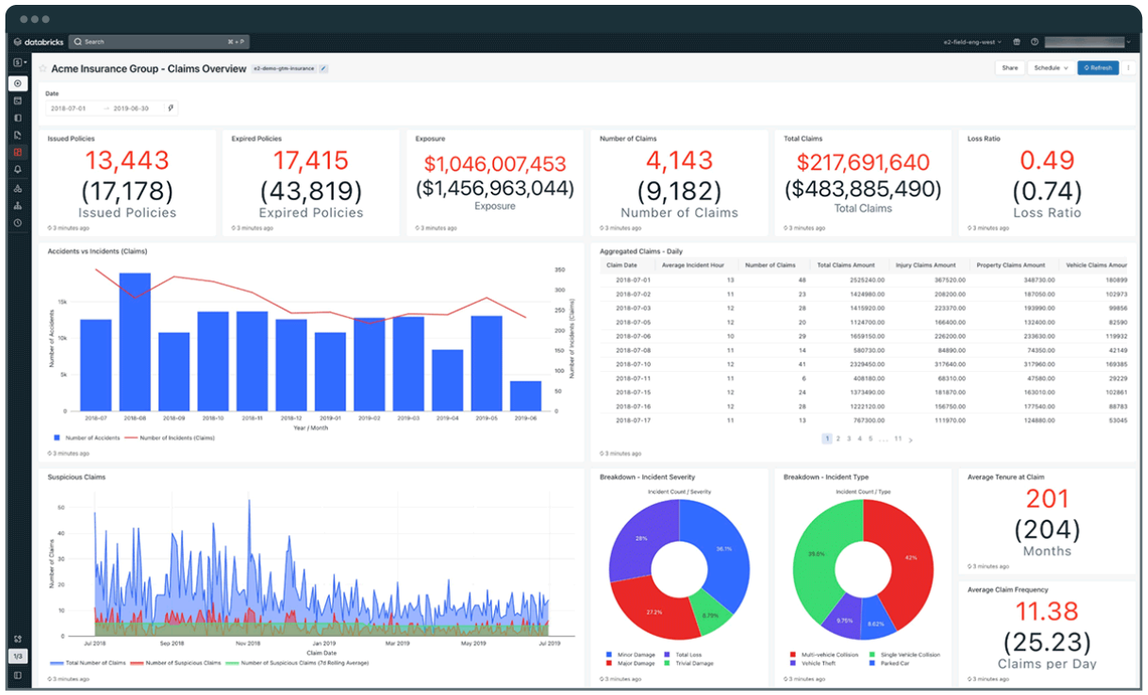

Figure 13: Example operational dashboard built on a set of resulting Delta Live Table (DLT) table entities

For our use case, we created a dashboard with a collection of key metrics, rolling calculations, high-level breakdowns, and aggregate views. The dashboard provides a complete summary of our claims process at a glance. We also added the option to specify specific date ranges. DB SQL supports a range of query parameters that can substitute values into a query at runtime. These query parameters can be defined at the dashboard level to ensure all related queries are updated accordingly.

DB SQL integrates with numerous third-party analytical and BI tools like Power BI, Tableau and Looker. Like we did for Fivetran, we can use Partner Connect to link our external platform with DB SQL. This allows analysts to build and serve dashboards in the platforms that the business prefers without sacrificing the performance of DB SQL and the Databricks Lakehouse.

CONCLUSION

As we move into this fast-paced, volatile modern world of finance, batch processing remains a vital part of the modern data stack, able to hold its own against the features and benefits of streaming and real-time services. We've seen how we can use the Databricks Data Intelligence Platform for Financial Services and its ecosystem of partners to architect a simple, scalable and extensible framework that supports complex batch-processing workloads with a practical example in insurance claims processing. With Delta Live Tables (DLT) and Databricks SQL (DB SQL), we can build a data platform with an architecture that scales infinitely, is easy to extend to address changing requirements and will withstand the test of time.

To learn more about the sample pipeline described, including the infrastructure setup and configuration used, please refer to this GitHub repository or watch this demo video.

How to Set Up Your First Federated Lakehouse

by Mike Dobing

Lakehouse Federation in Databricks is a groundbreaking new capability that allows you to query data across external data sources — including Snowflake, Synapse, many others and even Databricks itself — without having to move or copy the data. This is done by using Databricks Unity Catalog, which provides a unified metadata layer for all of your data.

Lakehouse Federation is a game-changer for data teams, as it breaks down the silos that have traditionally kept data locked away in different systems. With Lakehouse Federation, you can finally access all of your data in one place, making it easier to get the insights you need to make better business decisions.

As always, though, not one solution is a silver bullet for your data integration and querying needs. See below for when Federation is a good fit, and for when you'd prefer to bring your data into your solution and process as part of your lakehouse platform pipelines.

A few of the benefits of using Lakehouse Federation in Databricks are:

- **Improved data access and discovery:** Lakehouse Federation makes it easy to find and access the data you need from your database estate. This is especially important for organizations with complex data landscapes.
- **Reduced data silos:** Lakehouse Federation can help to break down data silos by providing a unified view of all data across the organization.
- **Improved data governance:** Lakehouse Federation can help to improve data governance by providing a single place to manage permissions and access to data from within Databricks.
- **Reduced costs:** Lakehouse Federation can help to reduce costs by eliminating the need to move or copy data between different data sources.

If you are looking for a way to improve the way you access and manage your data across your analytics estate, then Lakehouse Federation in Databricks is a top choice.