Microsoft Azure Delta Lake

DELTA LAKE

Cognitive Convergence is Subject Matter Expert in Azure - Data Factory, Azure - Data Bricks, Azure - Data Lake, Azure – Delta Lake, HD insight, Synapse.
We offer Delta lake consulting services covering solution architecture ACID transactions, scalable metadata handling, streaming and batch data processing, to uncover insights hidden within data and enhance data exploration.

Contents

OBJECTIVES

The objective of this paper is to explore Azure Delta Lake, what is its solution architecture and why to use azure Delta Lake. Lighten the reliability of data lakes. Delta Lake's core objectives like ACID transactions, scalable metadata handling, and unified streaming and batch data processing. Delta Lake links with Data Lake and how it is fully compatible with Apache Spark APIs.

BACKGROUND

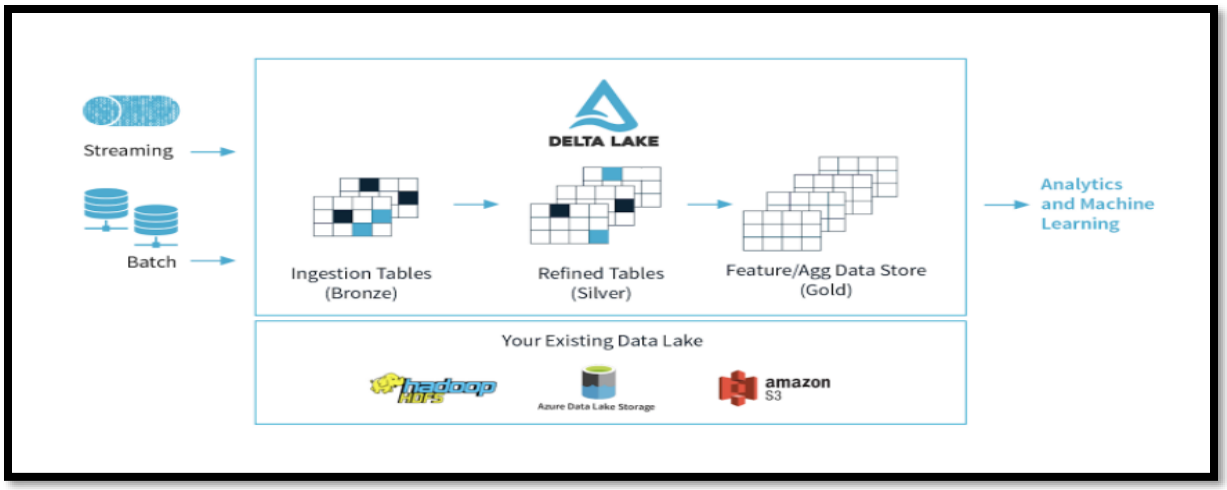
In practice, the need to deliver data in an understandable practice that provides actionable insights extends beyond the requirements of Data Engineers and Scientists. With that in mind, how can marketers, salespeople, and business executives understand and use comprehensive analytics platforms like Azure Databricks for day-to-day tasks? Fortunately, Delta Lake assists in transactional activity without missing data.

DELTA LAKE

Delta Lake is an open format storage layer that provides your data lake with dependability, security, and performance for both streaming and batch operations. Delta Lake is the foundation of a cost-effective, highly scalable lake house by replacing data silos with a single home for structured, semi-structured, and unstructured data.

Delta Lake is an open-source storage layer that improves data lake reliability. Delta Lake supports ACID transactions, and scalable metadata management, and combines streaming and batch data processing. Delta Lake is fully compatible with Apache Spark APIs and runs on top of your existing data lake. Delta Lake on Azure Databricks allows tailoring Delta Lake to your specific workload patterns.

Azure Databricks enhances Delta Lake with optimized layouts and indexes for fast interactive queries.



AZURE DATABRICKS FOR DATASCIENTISTS

Delta Lake takes away a few obstacles faced by Data Scientists and Data Engineers.

Data Skipping

With the Delta file, need not scan the entire data. As new data is inserted into a Databricks Delta table, file-level min/max statistics are collected for all columns, which helps filter files effectively.

Data Skipping

When write data into a Delta table, data skipping information is automatically collected. At query time, Delta Lake on Azure Databricks uses this information (minimum and maximum values) to provide faster queries. There is no need to configure data skipping; the feature is enabled whenever it is applicable. Its effectiveness, however, is dependent on the layout of your data. Use Z-Ordering for the best results.

Z-order

In addition to data skipping, Z-order enables data skipping at multi-dimensional.

Z-Ordering

Z-Ordering is a technique for storing related data in the same set of files. Delta Lake's Azure Databricks data-skipping algorithms use this co-locality automatically. This behavior reduces the amount of data that Delta Lake on Azure Databricks must-read significantly.

If anticipate that a column will be frequently used in query predicates and it has a high cardinality (that is, a large number of distinct values), then use ZORDER BY.

Effective caching

The Delta cache accelerates data reads by creating copies of remote files in nodes' local storage using a fast-intermediate data format.

Scalable Metadata Handling

In the big data world, even meta-data have the characteristics of big data. Delta treats meta-data as big data (i.e., it enables spark to process meta-data in a distributed manner).

Time travel

When write data into a Delta table, every operation is automatically versioned and can be access any version of data. This allows to travel back to a different version of the current delta table.

This time-traveling can be achieved using 2 approaches:

- Using a version number
- Using a timestamp

Time Travel Use Cases

Delta Lake time travel allows us to query an older snapshot of a Delta Lake table. Time travel has many use cases, including:

- Time travel makes it easy to do rollbacks in case of bad writes, playing an important role in fixing mistakes in our data.
- It helps in re-creating analysis, reports, or outputs (for example, the output of a machine learning model). This could be useful for debugging or auditing, especially in regulated industries.
- It also simplifies time-series analytics. For instance, in finding out how many new customers were added over the last week.

Unified platform for batch and streaming

Table in the delta lake can ingest (also handle) batch and streaming data.

OPTIMIZATION

Azure Databricks offers Delta Lake optimizations that accelerate data lake operations, supporting a wide range of workloads from large-scale ETL processing to ad-hoc, interactive queries. Many of these optimizations occur automatically; simply reap their benefits by using Azure Databricks for your data lakes.

TUNE FILE SIZE

- Set a target size
- Autotune based on workload
- Autotune based on table size

Set a target size

Set the table property delta.targetFileSize to the desired size to tune the size of files in your Delta table. If this property is set, all data layout optimization operations will use their best efforts to create files of the specified size. Optimize with Compaction (bin-packing) or Z-Ordering (multi-dimensional clustering), Auto Compaction, and Optimized Writes are some examples.

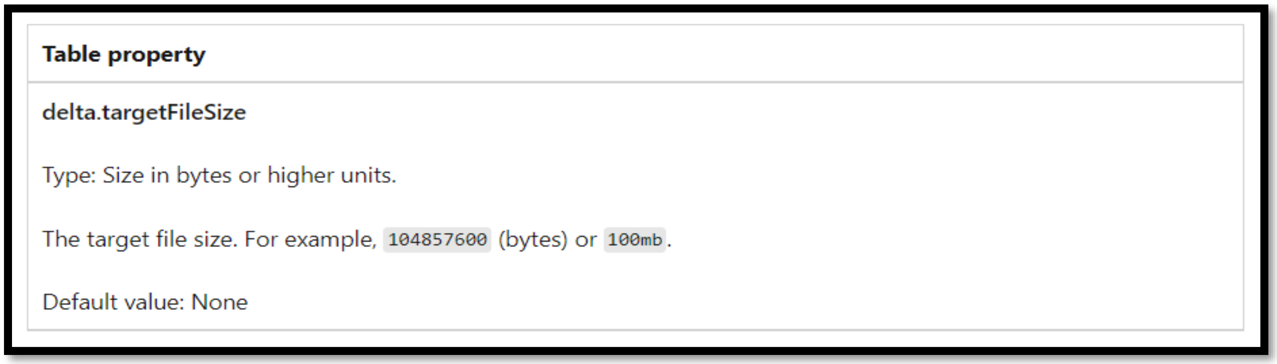


Table property
delta.targetFileSize Type: Size in bytes or higher units. The target file size. For example, `104857600` (bytes) or `100mb`. Default value: None

Autotune based on workload

Azure Databricks can automatically tune the file size of Delta tables based on workloads running on the table to reduce the need for manual tuning. If a Delta table has frequent MERGE operations that rewrite files, Azure Databricks can automatically detect this and may choose to reduce the size of rewritten files in anticipation of future file rewrites. For example, if 9 of the previous 10 operations on the table were also MERGES, then the Optimized Writes and Auto

Compaction used by MERGE (if enabled) will generate smaller file sizes than it would otherwise. This contributes to shorter MERGE operations in the future.

After a few rewrite operations, Autotune is activated. If anticipate that a Delta table will be subjected to frequent MERGE, UPDATE, or DELETE operations and want to tune file sizes for rewrites right away, can explicitly tune file sizes for rewrites by setting the table property delta. tuneFileSizesForRewrites. Set this property to true to ensure that all data layout optimization operations on the table always use smaller file sizes. Set it to false to never tune to smaller file sizes, effectively disabling auto-detection.

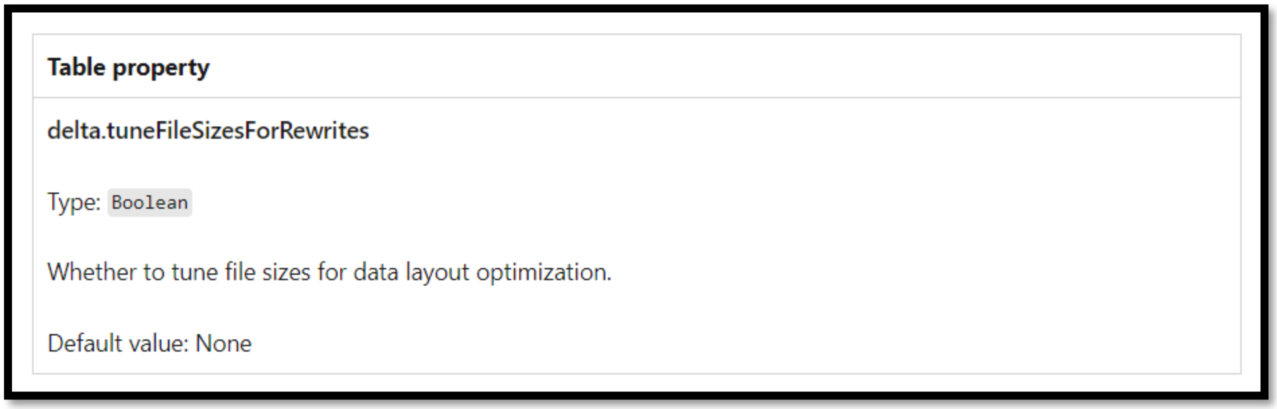


Table property
delta.tuneFileSizesForRewrites Type: `Boolean` Whether to tune file sizes for data layout optimization. Default value: None

Autotune based on table size

Azure Databricks automatically tunes the file size of Delta tables based on table size to reduce the need for manual tuning. Azure Databricks will use smaller file sizes for smaller tables and larger file sizes for larger tables to avoid having too many files in the table. Tables that have tuned with a specific target size or based on a workload with frequent rewrites are not autotuned by Azure Databricks.

The target file size is determined by the size of the Delta table at the time. The autotuned target file size for tables smaller than 2.56 TB is 256 MB. For tables with capacities ranging from 2.56 TB to 10 TB, the target size will increase linearly from 256 MB to 1 GB. The target file size for tables larger than 10 TB is 1 GB.

When a table is incrementally written, the target file sizes and file counts will be close to the numbers below, based on table size. This table's file counts are just an example. The actual results will vary depending on a variety of factors.

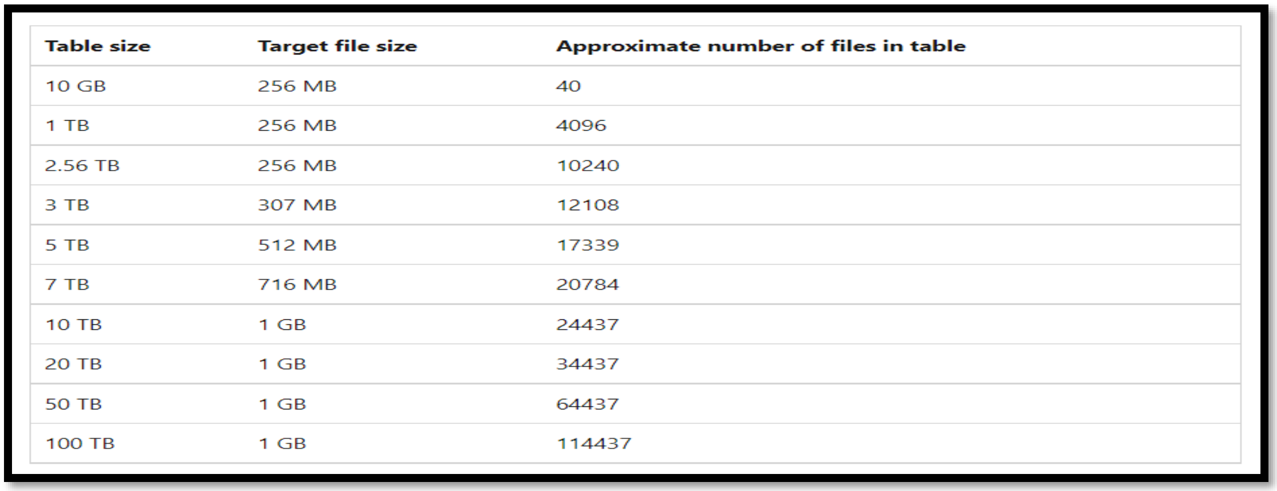


Table size	Target file size	Approximate number of files in table
10 GB	256 MB	40
1 TB	256 MB	4096
2.56 TB	256 MB	10240
3 TB	307 MB	12108
5 TB	512 MB	17339
7 TB	716 MB	20784
10 TB	1 GB	24437
20 TB	1 GB	34437
50 TB	1 GB	64437
100 TB	1 GB	114437

IMPROVE INTERACTIVE QUERY PERFORMANCE

- Manage data recency
- Enhance checkpoint for low-latency queries

Manage data recency

Delta tables auto-update to the most recent version of the table at the start of each query. When the command status reports: This process can be seen in notebooks: The Delta table's state is being updated. However, when performing historical analysis on a table, may not require up-to-the-minute data, especially if streaming data is frequently ingested. In these cases, queries can be run on stale Delta table snapshots. This method can reduce the time it takes to get query results.

Can specify how stale your table data is by setting the Spark session configuration spark.databricks.delta.stalenessLimit to a time string value, such as 1h, 15m, 1d for 1 hour, 15 minutes, and 1 day. Because this configuration is session-specific, it simply prevents a query from waiting for the table to update.

Enhance checkpoints for low-latency queries

Delta Lake writes checkpoints at an optimized frequency as an aggregate state of a Delta table. These checkpoints serve as the starting point for calculating the table's most recent state. To compute the state of a table without checkpoints, Delta Lake would have to read a large collection of JSON files ("delta" files) representing commits to the transaction log. In addition, the checkpoint stores the column-level statistics that Delta Lake uses to skip data.

INGEST DATA INTO DELTA LAKE

Azure Databricks provides a number of options for ingesting data into Delta Lake.

- ***Upload CSV files***
  - Using Databricks SQL's Create Table UI, and can securely create tables from CSV file
- ***Create Table in Databricks SQL***
  - Make a table in Databricks. SQL allows to quickly create a Delta table by uploading a CSV file.
- ***Target table types***
  - Make a table in Databricks. SQL can generate managed Delta tables in either the Unity Catalog or the Hive Metastore.

Requirement

- To use Create table in Databricks SQL with Unity Catalog, need a metastore, catalog, and schema. For more information on how to create these.
  - For Unity Catalog, must also have the USAGE permission on the parent catalog of the selected schema.
  - If have Unity Catalog enabled on your workspace, can still create tables under schemas in the Hive metastore.
- Need USAGE and CREATE permissions on the schema want to create a table in.
- Must have a running SQL Endpoint.

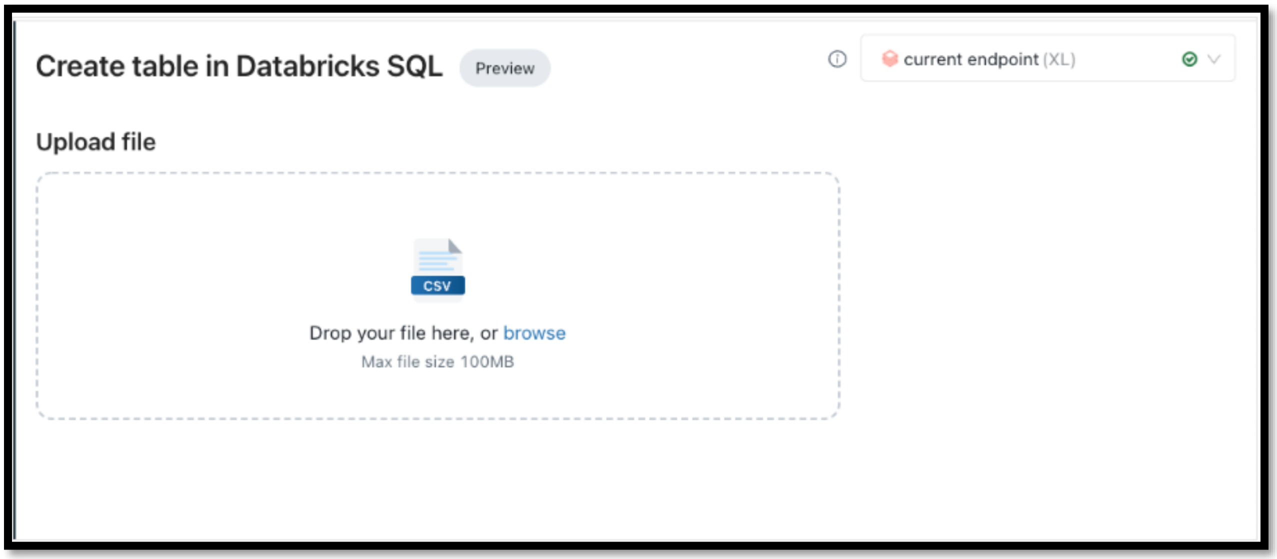
Create a table using CSV upload

Use the UI to create a Delta table by importing small CSV files to Databricks SQL from your local machine.

- The upload UI supports uploading a single file at a time under 100 megabytes.
- The file must be a CSV and have an extension ".csv".
- Compressed files such as zip and tar files are not supported.

Upload the file

1. Navigate to the SQL persona by using the persona switcher.
    - To change the persona, click the icon below the Databricks logo, and select a persona.
2. Click Create in the sidebar and select Table from the menu.
3. The Create table in Databricks SQL page appears.



4. To start an upload, click the file browser button or drag-and-drop files directly on the drop zone.

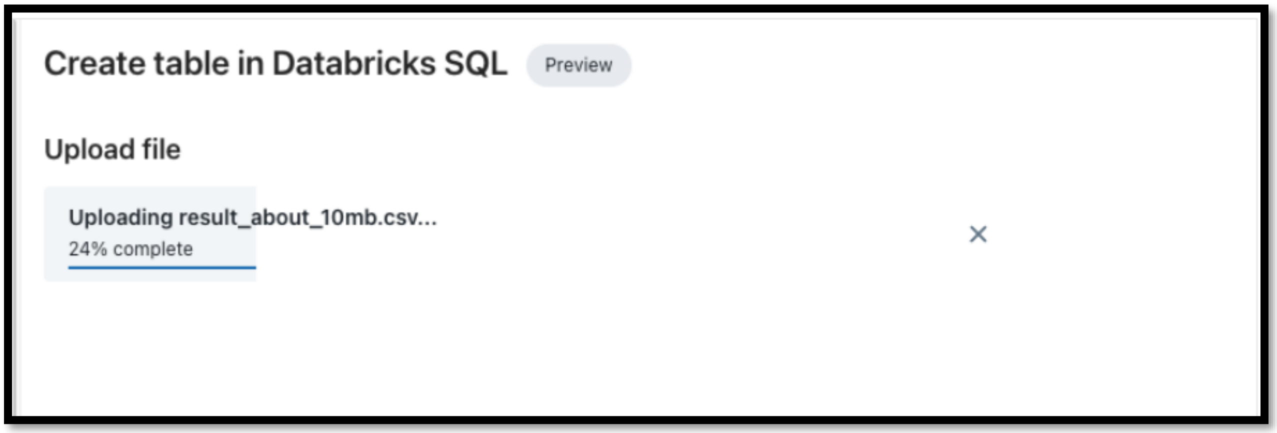
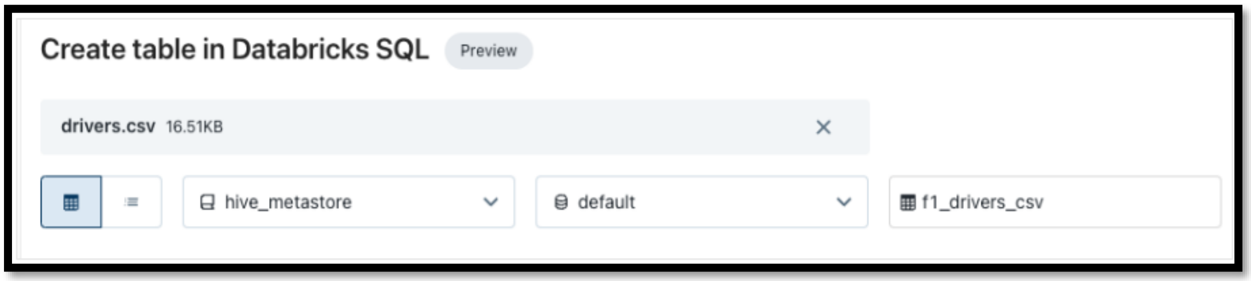

Create table in Databricks SQL
Preview
Upload file
Uploading result_about_10mb.csv...
24% complete

Table name selection

Upon completion of upload, can select the destination for your data.


Create table in Databricks SQL Preview

drivers.csv 16.51KB

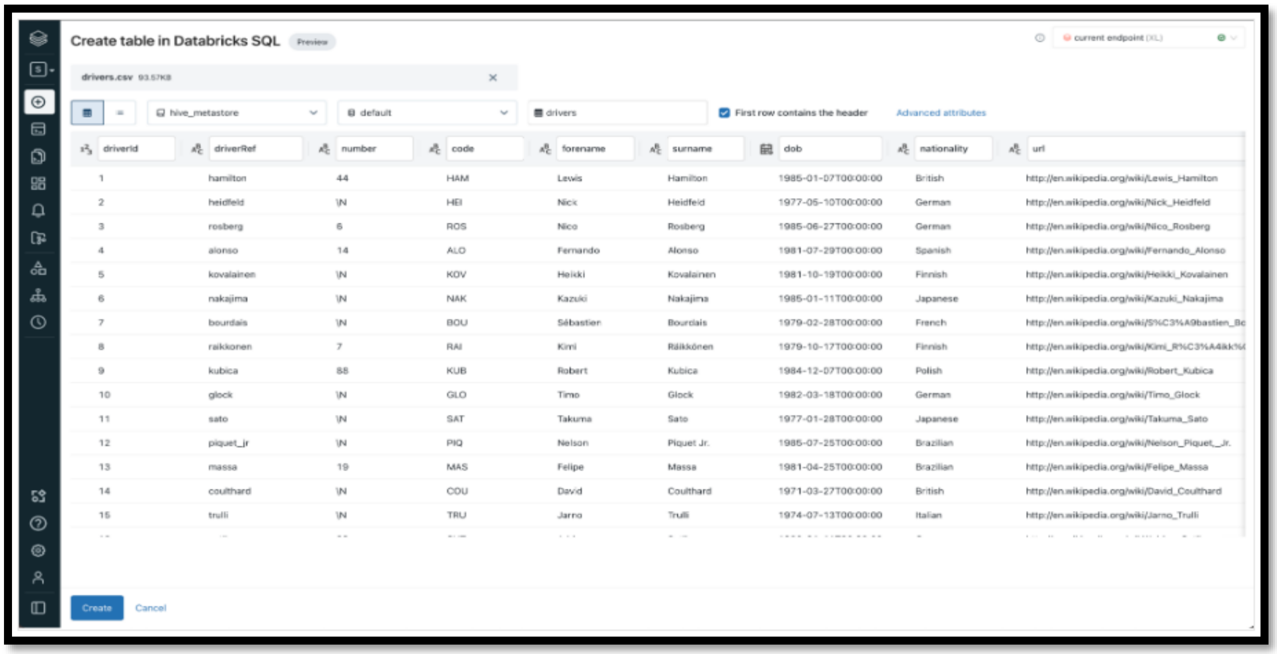
hive_metastore | default | f1_drivers_csv

1. Under Unity Catalog enabled workspaces, can select a catalog. If Unity Catalog is not enabled in your workspace, the destination catalog will be hidden, and schemas will be loaded from the Hive metastore.
    - To use the Hive metastore in a Unity Catalog-enabled workspace, select hive_metastore in the catalog selector.
2. Select a schema.
3. By default, the UI converts the file name to a valid table name. Can edit the table name.

Data preview

After the file upload is complete, can preview the data (limit of 50 rows).

- After the upload, the UI tries to start the endpoint selected in the top right. Can switch endpoints at any time, but the preview and table creation require an active endpoint. If your endpoint is not active yet, it starts automatically. This may take some time. The preview starts when your endpoint is running.


Create table in Databricks SQL Preview

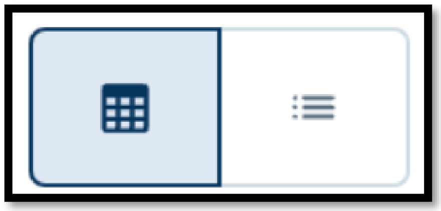
current endpoint (XL)

drivers.csv 93.57KB

hive_metastore | default | drivers | ✓ First row contains the header | Advanced attributes

driverId	driverRef	number	code	forename	surname	dob	nationality	url
1	hamilton	44	HAM	Lewis	Hamilton	1985-01-07T00:00:00	British	http://en.wikipedia.org/wiki/Lewis_Hamilton
2	heidfeld	\N	HEI	Nick	Heidfeld	1977-05-10T00:00:00	German	http://en.wikipedia.org/wiki/Nick_Heidfeld
3	rosberg	6	ROS	Nico	Rosberg	1985-06-27T00:00:00	German	http://en.wikipedia.org/wiki/Nico_Rosberg
4	alonso	14	ALO	Fernando	Alonso	1981-07-29T00:00:00	Spanish	http://en.wikipedia.org/wiki/Fernando_Alonso
5	kovalainen	\N	KOV	Heikki	Kovalainen	1981-10-19T00:00:00	Finnish	http://en.wikipedia.org/wiki/Heikki_Kovalainen
6	nakajima	\N	NAK	Kazuki	Nakajima	1985-01-11T00:00:00	Japanese	http://en.wikipedia.org/wiki/Kazuki_Nakajima
7	bourdais	\N	BOU	Sébastien	Bourdais	1979-02-28T00:00:00	French	http://en.wikipedia.org/wiki/S%C3%A9bastien_Bo
8	raikkonen	7	RAI	Kimi	Räikkönen	1979-10-17T00:00:00	Finnish	http://en.wikipedia.org/wiki/Kimi_R%C3%A4ikk%C
9	kubica	88	KUB	Robert	Kubica	1984-12-07T00:00:00	Polish	http://en.wikipedia.org/wiki/Robert_Kubica
10	glock	\N	GLO	Timo	Glock	1982-03-18T00:00:00	German	http://en.wikipedia.org/wiki/Timo_Glock
11	sato	\N	SAT	Takuma	Sato	1977-01-28T00:00:00	Japanese	http://en.wikipedia.org/wiki/Takuma_Sato
12	piquet_jr	\N	PIQ	Nelson	Piquet Jr.	1985-07-25T00:00:00	Brazilian	http://en.wikipedia.org/wiki/Nelson_Piquet,_Jr.
13	massa	19	MAS	Felipe	Massa	1981-04-25T00:00:00	Brazilian	http://en.wikipedia.org/wiki/Felipe_Massa
14	coulthard	\N	COU	David	Coulthard	1971-03-27T00:00:00	British	http://en.wikipedia.org/wiki/David_Coulthard
15	trulli	\N	TRU	Jarno	Trulli	1974-07-13T00:00:00	Italian	http://en.wikipedia.org/wiki/Jarno_Trulli

- There are two ways to preview the data, vertically or horizontally. To switch between preview options, click the toggle button above the table



Format options

Depending on the file format uploaded, different options are available.
Common format options appear in the header bar, while less commonly used options are available in the Advanced attributes modal.

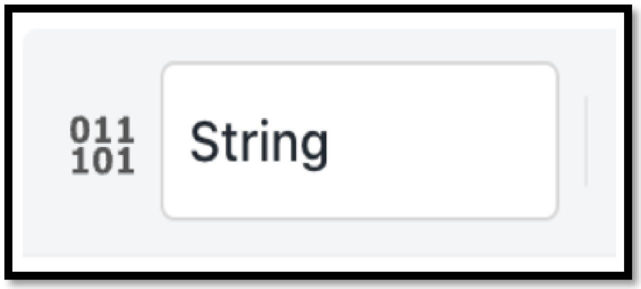
- For CSV, the following options are available.
  - *The first row contains the header* (enabled by default): This option specifies whether the CSV file contains a header.
  - *Column delimiter* the separator character between columns. Only a single character is allowed, and the backslash is not supported. This defaults to a comma for CSV files.
  - *Automatically detect column types* (enabled by default): Automatically detect column types from the file content. Can edit types in the preview table. If this is set to false, all column types are inferred as STRING.
  - *Rows span multiple lines* (disabled by default): Whether a column's value can span multiple lines in the file.

The data preview updates automatically when edit format options.

Column headers and types

Can edit column header names and types.

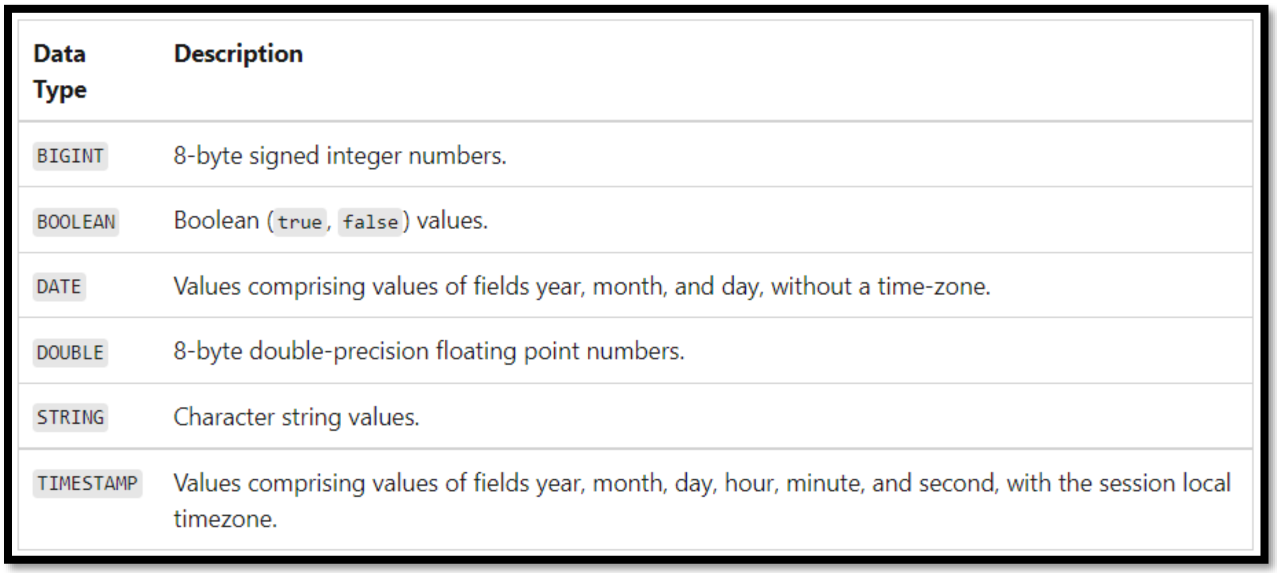
- To edit types, click the icon with the type.
- To edit the column name, click the input box at the top of the column.
  - Column names do not support commas, backslashes, or unicode characters (such as emojis).



For CSV files, the column data types are inferred by default.
Can interpret all columns as STRING type by disabling **Advanced attributes** > **Automatically detect column types**.

Supported data types

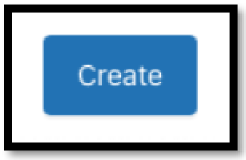
Create table using CSV upload supports the following data types. For more information about individual data types.



Data Type	Description
`BIGINT`	8-byte signed integer numbers.
`BOOLEAN`	Boolean (`true`, `false`) values.
`DATE`	Values comprising values of fields year, month, and day, without a time-zone.
`DOUBLE`	8-byte double-precision floating point numbers.
`STRING`	Character string values.
`TIMESTAMP`	Values comprising values of fields year, month, day, hour, minute, and second, with the session local timezone.

Creating table

To create the table, click **Create** at the bottom of the page.



After create the table using Create table in Databricks SQL, the Data Explorer page for the Delta table under the designated catalog and schema appears.

Partner integrations

Databricks partner integrations enable to easily load data into Azure Databricks. These integrations enable low-code, easy-to-implement, and scalable data ingestion from a variety of sources into Azure Databricks.

Copy INTO SQL Command

The COPY INTO SQL command lets load data from a file location into a Delta table. This is a re-triable and idempotent operation; files in the source location that have already been loaded are skipped.

Use the COPY INTO SQL command instead of Auto Loader when:

- You want to load data from a file location that contains files in the order of thousands or fewer.
- Your data schema is not expected to evolve frequently.
- You plan to load subsets of previously uploaded files.

The following shows how to create a Delta table and then use the COPY INTO SQL command to load sample data from Azure Databricks datasets into the table. Can run the example Python, R, Scala, or SQL code from within a notebook attached to an Azure Databricks cluster. Can also run the SQL code from within a query associated with a SQL endpoint in Databricks SQL.

Read Table

Load Delta table as a DataFrame by specifying a table name or a path

SQL

```
SELECT*FROM default.people10m –query table in the metastore
SELECT*FROM delta.'/tmp/delta/people10m' –query table by path
```

Python

```
Spark.table("default.people10m") #query table in the metastore
Spark.read.format("delta").load("/tmp/delta/people10m") #query table by path
```

Table Streaming reads and writes

Delta Lake is deeply integrated with Spark Structured Streaming through readStream and writeStream. Delta Lake overcomes many of the limitations typically associated with streaming systems and files, including:

- Coalescing small files produced by low latency ingest
- Maintaining "exactly-once" processing with more than one stream (or concurrent batch jobs)
- Efficiently discovering which files are new when using files as the source for a stream

Delta Table as a source

When load a Delta table as a stream source and use it in a streaming query, the query processes all of the data present in the table as well as any new data that arrives after the stream is started.

Load both paths and tables as a stream.

Python

```
spark.readStream.format("delta")
  .load("/tmp/delta/events")
import io.delta.implicits._
spark.readStream.delta("/tmp/delta/events")
or
import io.delta.implicits._
spark.readStream.format("delta").table("events")
```

- Limit input rate
- Ignore updates and deletes
- Specify initial position

Limit input rate

The following options are available to control micro-batches:

maxFilesPerTrigger: How many new files to be considered in every micro-batch. The default is 1000.
maxBytesPerTrigger: How much data gets processed in each micro-batch. This option sets a "soft max", meaning that a batch processes approximately this amount of data and may process more than the limit in order to make the streaming query move forward in cases when the smallest input unit is larger than this limit. If use Trigger.Once for your streaming, this option is ignored. This is not set by default.

If use maxBytesPerTrigger in conjunction with maxFilesPerTrigger, the micro-batch processes data until either the maxFilesPerTrigger or maxBytesPerTrigger limit is reached.

Ignore updates and deletes

Structured Streaming does not handle input that is not an append and throws an exception if any modifications occur on the table being used as a source. There are two main strategies for dealing with changes that cannot be automatically propagated downstream:

- Delete the output and checkpoint and restart the stream from the beginning.
- Set either of these two options:
    - ignoreDeletes: ignore transactions that delete data at partition boundaries.
    - ignoreChanges: re-process updates if files had to be rewritten in the source table due to a data changing operation such as UPDATE, MERGE INTO, DELETE (within partitions), or OVERWRITE. Unchanged rows may still be emitted, therefore your downstream consumers should be able to handle duplicates. Deletes are not propagated downstream. ignoreChanges subsumes ignoreDeletes. Therefore if use ignoreChanges, your stream will not be disrupted by either deletions or updates to the source table.

Specify initial position

Use the following options to specify the starting point of the Delta Lake streaming source without processing the entire table.

- startingVersion: The Delta Lake version to start from. All table changes starting from this version (inclusive) will be read by the streaming source. Can obtain the commit versions from the version column of the DESCRIBE HISTORY command output. In Databricks Runtime 7.4 and above, to return only the latest changes, specify latest.
- startingTimestamp: The timestamp to start from. All table changes committed at or after the timestamp (inclusive) will be read by the streaming source. One of:
    - A timestamp string. For example, "2019-01-01T00:00:00.000Z".
    - A date string. For example, "2019-01-01".

You cannot set both options at the same time; can use only one of them. They take effect only when starting a new streaming query. If a streaming query has started and the progress has been recorded in its checkpoint, these options are ignored.

DELTA TABLE AS A SINK

Write data into a Delta table using Structured Streaming. The transaction log enables Delta Lake to guarantee exactly-once processing, even when there are other streams or batch queries running concurrently against the table.

- Metrics
- Append Mode
- Complete Mode
- Idempotent multi-table writes

Metrics

Find out the number of bytes and number of files yet to be processed in a streaming query process as the numBytesOutstanding and numFilesOutstanding metrics. If are running the stream in a notebook, can see these metrics under the Raw Data tab in the streaming query progress dashboard:

```
{
  "sources" : [
    {
      "description" : "DeltaSource[file:/path/to/source]",
      "metrics" : {
        "numBytesOutstanding" : "3456",
        "numFilesOutstanding" : "8"
      },
    }
  ]
}
```

Append Mode

By default, streams run in append mode, which adds new records to the table. Can use the path method:

Python

```
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/tmp/delta/_checkpoints/")
  .start("/delta/events")
```

Scala

```
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/tmp/delta/events/_checkpoints/")
  .start("/tmp/delta/events")

import io.delta.implicits._
events.writeStream
  .outputMode("append")
  .option("checkpointLocation", "/tmp/delta/events/_checkpoints/")
  .delta("/tmp/delta/events")
```

Complete Mode

Use Structured Streaming to replace the entire table with every batch. One example use case is to compute a summary using aggregation:

Python

```
(spark.readStream
  .format("delta")
  .load("/tmp/delta/events")
  .groupBy("customerId")
  .count()
  .writeStream
  .format("delta")
  .outputMode("complete")
  .option("checkpointLocation",
"/tmp/delta/eventsByCustomer/_checkpoints/")
  .start("/tmp/delta/eventsByCustomer")
)
```

Scala

```
spark.readStream
  .format("delta")
  .load("/tmp/delta/events")
  .groupBy("customerId")
  .count()
  .writeStream
  .format("delta")
  .outputMode("complete")
  .option("checkpointLocation",
"/tmp/delta/eventsByCustomer/_checkpoints/")
  .start("/tmp/delta/eventsByCustomer")
```

For applications with more lenient latency requirements, can save computing resources with one-time triggers. Use these to update summary aggregation tables on a given schedule, processing only new data that has arrived since the last update.

IDEMPOTENT MULTI-TABLE WRITES

Multiple tables using the foreachBatch command, foreachBatch allows the output of every micro-batch in the streaming query to be written to multiple target destinations. However, foreachBatch does not make those writes idempotent as those write attempts lack the information of whether the batch is being re-executed or not. For example, rerunning a failed batch could result in duplicate data writes.

To address this, Delta tables support the following DataFrameWriter options to make the writes idempotent:

- txnAppId: A unique string that can pass on each DataFrame write. For example, can use the StreamingQuery ID as txnAppId.
- txnVersion: A monotonically increasing number that acts as transaction version.

Delta Lake uses the combination of txnAppId and txnVersion to identify duplicate writes and ignore them.

If a batch write is interrupted with a failure, rerunning the batch uses the same application and batch ID, which would help the runtime correctly identify duplicate writes and ignore them. Application ID (txnAppId) can be any user-generated unique string and does not have to be related to the stream ID.

Python

```
app_id = ... # A unique string that is used as an application ID.

def writeToDeltaLakeTableIdempotent(batch_df, batch_id):
  batch_df.write.format(...).option("txnVersion",
batch_id).option("txnAppId", app_id).save(...) # location 1
  batch_df.write.format(...).option("txnVersion",
batch_id).option("txnAppId", app_id).save(...) # location 2
```

Scala

```
val appId = ... // A unique string that is used as an application ID.
streamingDF.writeStream.foreachBatch { (batchDF: DataFrame,
batchId: Long) =>
  batchDF.write.format(...).option("txnVersion",
batchId).option("txnAppId", appId).save(...)  // location 1
  batchDF.write.format(...).option("txnVersion",
batchId).option("txnAppId", appId).save(...)  // location 2
}
```

Performing stream-static joins

Rely on the transactional guarantees and versioning protocol of Delta Lake to perform stream-static joins. A stream-static join joins the latest valid version of a Delta table (the static data) to a data stream using a stateless join.

When Azure Databricks processes a micro-batch of data in a stream-static join, the latest valid version of data from the static Delta table joins with the records present in the current micro-batch. Because the join is stateless, do not need to configure watermarking and can process results with low latency. The data in the static Delta table used in the join should be slowly-changing.

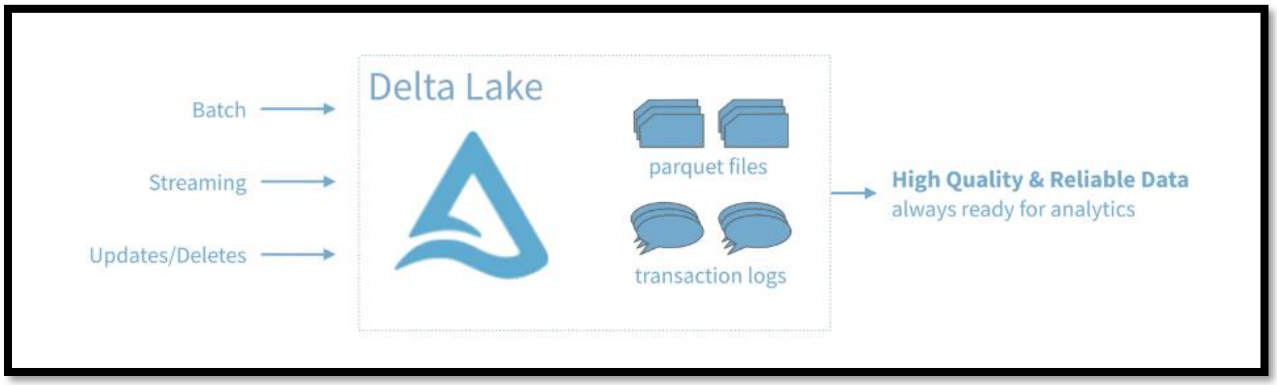
Python

```
streamingDF = spark.readStream.table("orders")
staticDF = spark.read.table("customers")

query = (streamingDF
  .join(staticDF, streamingDF.customer_id==staticDF.id, "inner")
  .writeStream
  .option("checkpointLocation", checkpoint_path)
  .table("orders_with_customer_info")
)
```

TABLE DELETES, UPDATES, AND MERGES

Delta Lake supports several statements to facilitate deleting data from and updating data in Delta tables.



DELETE FROM A TABLE

Remove data that matches a predicate from a Delta table. For instance, in a table named people10m or a path at /tmp/delta/people-10m, to delete all rows corresponding to people with a value in the birthDate column from before 1955, can run the following:

SQL

```
DELETE FROM people10m WHERE birthDate < '1955-01-01'
DELETE FROM delta.`/tmp/delta/people-10m` WHERE birthDate <
'1955-01-01'
```

Python

```
from delta.tables import *
from pyspark.sql.functions import *

deltaTable = DeltaTable.forPath(spark, '/tmp/delta/people-10m')

# Declare the predicate by using a SQL-formatted string.
deltaTable.delete("birthDate < '1955-01-01'")

# Declare the predicate by using Spark SQL functions.
deltaTable.delete(col('birthDate') < '1960-01-01')
```

UPDATE TABLE

Update data that matches a predicate in a Delta table. For example, in a table named people10m or a path at /tmp/delta/people-10m, to change an abbreviation in the gender column from M or F to Male or Female, can run the following:

SQL

```
UPDATE people10m SET gender = 'Female' WHERE gender = 'F';
UPDATE people10m SET gender = 'Male' WHERE gender = 'M';
UPDATE delta.`/tmp/delta/people-10m` SET gender = 'Female' WHERE
gender = 'F';
UPDATE delta.`/tmp/delta/people-10m` SET gender = 'Male' WHERE
gender = 'M';
```

Python

```python
from delta.tables import *
from pyspark.sql.functions import *

deltaTable = DeltaTable.forPath(spark, '/tmp/delta/people-10m')

# Declare the predicate by using a SQL-formatted string.
deltaTable.update(
  condition = "gender = 'F'",
  set = { "gender": "'Female'" }
)

# Declare the predicate by using Spark SQL functions.
deltaTable.update(
  condition = col('gender') == 'M',
  set = { 'gender': lit('Male') }
)
```

UPSERT INTO A TABLE USING MERGE

Upsert data from a source table, view, or DataFrame into a target Delta table by using the MERGE SQL operation. Delta Lake supports inserts, updates, and deletes in MERGE, and it supports extended syntax beyond the SQL standards to facilitate advanced use cases. Suppose have a source table named people10mupdates or a source path at /tmp/delta/people-10m-updates that contains new data for a target table named people10m or a target path at /tmp/delta/people-10m. Some of these new records may already be present in the target data. To merge the new data, want to update rows where the person’s id is already present and insert the new rows where no matching id is present. Can run the following:

SQL

```sql
MERGE INTO people10m
USING people10mupdates
ON people10m.id = people10mupdates.id
WHEN MATCHED THEN
  UPDATE SET
    id = people10mupdates.id,
    firstName = people10mupdates.firstName,
    middleName = people10mupdates.middleName,
    lastName = people10mupdates.lastName,
    gender = people10mupdates.gender,
    birthDate = people10mupdates.birthDate,
    ssn = people10mupdates.ssn,
    salary = people10mupdates.salary
```

```
WHEN NOT MATCHED
  THEN INSERT (
    id,
    firstName,
    middleName,
    lastName,
    gender,
    birthDate,
    ssn,
    salary
  )
  VALUES (
    people10mupdates.id,
    people10mupdates.firstName,
    people10mupdates.middleName,
    people10mupdates.lastName,
    people10mupdates.gender,
    people10mupdates.birthDate,
    people10mupdates.ssn,
    people10mupdates.salary
  )
```

Python

```python
from delta.tables import *
deltaTablePeople = DeltaTable.forPath(spark, '/tmp/delta/people-10m')
deltaTablePeopleUpdates = DeltaTable.forPath(spark, '/tmp/delta/people-10m-updates')
dfUpdates = deltaTablePeopleUpdates.toDF()
deltaTablePeople.alias('people') \
  .merge(
    dfUpdates.alias('updates'),
    'people.id = updates.id'
  ) \
  .whenMatchedUpdate(set =
    {
      "id": "updates.id",
      "firstName": "updates.firstName",
      "middleName": "updates.middleName",
      "lastName": "updates.lastName",
      "gender": "updates.gender",
      "birthDate": "updates.birthDate",
      "ssn": "updates.ssn",
      "salary": "updates.salary"
    }
  ) \
  .whenNotMatchedInsert(values =
    {
      "id": "updates.id",
      "firstName": "updates.firstName",
      "middleName": "updates.middleName",
      "lastName": "updates.lastName",
      "gender": "updates.gender",
      "birthDate": "updates.birthDate",
      "ssn": "updates.ssn",
      "salary": "updates.salary"
    }
  ) \
  .execute()
```

Delta Lake merge operations typically require two passes over the source data. If your source data contains non-deterministic expressions, multiple passes on the source data can produce different rows causing incorrect results. Some common examples of non-deterministic expressions include the current_date and current_timestamp functions. If cannot avoid using non-deterministic functions, consider saving the source data to storage, for example as a temporary Delta table. Caching the source data may not address this issue, as cache invalidation can cause the source data to be recomputed partially or completely (for example when a cluster loses some of it, executors, when scaling down).

SCHEMA VALIDATION

Merge automatically validates that the schema of the data generated by insert and update expressions are compatible with the schema of the table. It uses the following rules to determine whether the merge operation is compatible: For update and insert actions, the specified target columns must exist in the target Delta table. For updateAll and insertAll actions, the source dataset must have all the columns of the target Delta table. The source dataset can have extra columns and they are ignored. For all actions, if the data type generated by the expressions producing the target columns are different from the corresponding columns in the target Delta table, merge tries to cast them to the types in the table.

AUTOMATIC SCHEMA EVOLUTION

By default, updateAll and insertAll assign all the columns in the target Delta table with columns of the same name from the source dataset. Any columns in the source dataset that don’t match columns in the target table are ignored. However, in some use cases, it is desirable to automatically add source columns to the target Delta table. To automatically update the table schema during a merge operation with updateAll and insertAll (at least one of them), can set the Spark session configuration spark.databricks.delta.schema.autoMerge.enabled to true before running the merge operation.

MERGE

Merges a set of updates, insertions, and deletions based on a source table into a target Delta table.

Data deduplication when writing into Delta tables

A common ETL use case is to collect logs into the Delta table by appending them to a table. However, often the sources can generate duplicate log records and downstream reduplication steps are needed to take care of them. With the merge, can avoid inserting the duplicate records.

SQL

```sql
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
  THEN INSERT *
```

Python

```
deltaTable.alias("logs").merge(
    newDedupedLogs.alias("newDedupedLogs"),
    "logs.uniqueId = newDedupedLogs.uniqueId") \
  .whenNotMatchedInsertAll() \
  .execute()
```

SLOWLY CHANGING DATA (SCD) TYPE 2 OPERATION INTO DELTA TABLES

Another common operation is SCD Type 2, which maintains a history of all changes made to each key in a dimensional table. Such operations require updating existing rows to mark previous values of keys as old, and inserting the new rows as the latest values. Given a source table with updates and the target table with the dimensional data, SCD Type 2 can be expressed with the merge. When a customer's address needs to be updated, have to mark the previous address as not the current one, update its active date range, and add the new address as the current one.

WRITE CHANGE DATA INTO A DELTA TABLE

Similar to SCD, another common use case, often called change data capture (CDC), is to apply all data changes generated from an external database into a Delta table. In other words, a set of updates, deletes, and inserts applied to an external table needs to be applied to a Delta table. Can do this using merge as follows.

Languages

- Python
  - Pandas API on Spark
  - Koalas
- R
  - Scala
- SQL

CONCLUSION

Delta Lake, is an ACID table storage layer over cloud object stores that enables a wide range of DBMS-like performance and management features for data in low-cost cloud storage. Delta Lake is implemented solely as a storage format and a set of access protocols for clients, making it simple to operate and highly available, and giving clients directly, and high-bandwidth access to the object-store. Delta Lake support three major languages python, R, and SQL. Features like Z-Ordering, Time travel, and Data skipping makes it more important to be used.

CONTACT US

Shahzad Sarwar
Entrepreneur/Architect/Consultant
Cognitive Convergence
http://www.cognitiveconvergence.com
Voice: +1 4242530744

Skype: Shahzad.Sarwar.Online
shahzad@cognitiveconvergence.com