Last updated on ***Nov 11, 2025***

The AUTO CDC APIs: Simplify change data capture with pipelines

Lakeflow Spark Declarative Pipelines (SDP) simplifies change data capture (CDC) with the `AUTO CDC` and `AUTO CDC FROM SNAPSHOT` APIs.

> **NOTE**
>
> The `AUTO CDC` APIs replace the `APPLY CHANGES` APIs, and have the same syntax. The `APPLY CHANGES` APIs are still available, but Databricks recommends using the `AUTO CDC` APIs in their place.

The interface you use depends on the source of change data:

- Use `AUTO CDC` to process changes from a change data feed (CDF).
- Use `AUTO CDC FROM SNAPSHOT` (Public Preview, and only available for Python) to process changes in database snapshots.

Previously, the `MERGE INTO` statement was commonly used for processing CDC records on Databricks. However, `MERGE INTO` can produce incorrect results because of out-of-sequence records or requires complex logic to re-order records.

The `AUTO CDC` API is supported in the pipelline SQL and Python interfaces. The `AUTO CDC FROM SNAPSHOT` API is supported in the Python interface. The `AUTO CDC` APIs are not supported by Apache Spark Declarative Pipelines.

Both `AUTO CDC` and `AUTO CDC FROM SNAPSHOT` support updating tables using SCD type 1 and type 2:

- Use SCD type 1 to update records directly. History is not retained for updated records.
- Use SCD type 2 to retain a history of records, either on all updates or on updates to a specified set of columns.

For syntax and other references, see AUTO CDC for pipelines (SQL), AUTO CDC for pipelines (Python), and AUTO CDC FROM SNAPSHOT for pipelines (Python).

> **NOTE**
>
> This article describes how to update tables in your pipelines based on changes in source data. To learn how to record and query row-level change information for Delta tables, see Use Delta Lake change data feed on Databricks.

Requirements

To use the CDC APIs, your pipeline must be configured to use serverless SDP or the SDP `Pro` or `Advanced` editions.

How is CDC implemented with the AUTO CDC API?

By automatically handling out-of-sequence records, the AUTO CDC API ensures correct processing of CDC records and removes the need to develop complex logic for handling out-of-sequence records. You must specify a column in the source data on which to sequence records, which the APIs interpret as a monotonically increasing representation of the proper ordering of the source data. Pipelines automatically handle data that arrives out of order. For SCD type 2 changes, pipelines propagate the appropriate sequencing values to the target table's `__START_AT` and `__END_AT` columns. There should be one distinct update per key at each sequencing value, and NULL sequencing values are unsupported.

To perform CDC processing with `AUTO CDC`, you first create a streaming table and then use the `AUTO CDC ... INTO` statement in SQL or the `create_auto_cdc_flow()` function in Python to specify the source, keys, and sequencing for the change feed. To create the target streaming table, use the `CREATE OR REFRESH STREAMING TABLE` statement in SQL or the `create_streaming_table()` function in Python. See the SCD type 1 and type 2 processing examples.

For syntax details, see the pipelines SQL reference or Python reference.

How is CDC implemented with the `AUTO CDC FROM SNAPSHOT` API?

> **PREVIEW**
>
> The `AUTO CDC FROM SNAPSHOT` API is in Public Preview.

`AUTO CDC FROM SNAPSHOT` is a declarative API that efficiently determines changes in source data by comparing a series of in-order snapshots and then runs the processing required for CDC processing of the records in the snapshots. `AUTO CDC FROM SNAPSHOT` is supported only by the Python pipelines interface.

`AUTO CDC FROM SNAPSHOT` supports ingesting snapshots from multiple source types:

- Use periodic snapshot ingestion to ingest snapshots from an existing table or view. `AUTO CDC FROM SNAPSHOT` has a simple, streamlined interface to support periodically ingesting snapshots from an existing database object. A new snapshot is ingested with each pipeline update, and the ingestion time is used as the snapshot version. When a pipeline is run in continuous mode, multiple snapshots are ingested with each pipeline update on a period determined by the trigger interval setting for the flow that contains the `AUTO CDC FROM SNAPSHOT` processing.
- Use historical snapshot ingestion to process files containing database snapshots, such as snapshots generated from an Oracle or MySQL database or a data warehouse.

To perform CDC processing from any source type with `AUTO CDC FROM SNAPSHOT`, you first create a streaming table and then use the `create_auto_cdc_from_snapshot_flow()` function in Python to specify the snapshot, keys, and other arguments required to implement the processing. See the periodic snapshot ingestion and historical snapshot ingestion examples.

The snapshots passed to the API must be in ascending order by version. If SDP detects an out-of-order snapshot, an error is thrown.

For syntax details, see the pipelines Python reference.

Use multiple columns for sequencing

You can sequence by multiple columns (for example, a timestamp and an ID to break ties), you can use a STRUCT to combine them: it orders by the first field of the STRUCT first, and in the event of a tie, considers the second field, and so on.

Example in SQL:

```sql
SEQUENCE BY STRUCT(timestamp_col, id_col)
```

Example in Python:

```python
sequence_by = struct("timestamp_col", "id_col")
```

Limitations

The column used for sequencing must be a sortable data type.

Example: SCD type 1 and SCD type 2 processing with CDF source data

The following sections provide examples of SCD type 1 and type 2 queries that update target tables based on source events from a change data feed that:

1. Creates new user records.
2. Deletes a user record.
3. Updates user records. In the SCD type 1 example, the last `UPDATE` operations arrive late and are dropped from the target table, demonstrating the handling of out-of-order events.

The following examples assume familiarity with configuring and updating pipelines. See Tutorial: Build an ETL pipeline using change data capture.

To run these examples, you must begin by creating a sample dataset. See Generate test data.

The following are the input records for these examples:

userId	name	city	operation	sequenceNum
124	Raul	Oaxaca	INSERT	1
123	Isabel	Monterrey	INSERT	1
125	Mercedes	Tijuana	INSERT	2
126	Lily	Cancun	INSERT	2
123	null	null	DELETE	6
125	Mercedes	Guadalajara	UPDATE	6
125	Mercedes	Mexicali	UPDATE	5
123	Isabel	Chihuahua	UPDATE	5

If you uncomment the final row in the example data, it will insert the following record that specifies where records should be truncated:

userId	name	city	operation	sequenceNum
null	null	null	TRUNCATE	3

NOTE

All the following examples include options to specify both `DELETE` and `TRUNCATE` operations, but each is optional.

Process SCD type 1 updates

The following example demonstrates processing SCD type 1 updates:

Python SQL

Python

```
from pyspark import pipelines as dp
from pyspark.sql.functions import col, expr

@dp.view
def users():
  return spark.readStream.table("cdc_data.users")

dp.create_streaming_table("target")

dp.create_auto_cdc_flow(
  target = "target",
  source = "users",
  keys = ["userId"],
  sequence_by = col("sequenceNum"),
  apply_as_deletes = expr("operation = 'DELETE'"),
  apply_as_truncates = expr("operation = 'TRUNCATE'"),
  except_column_list = ["operation", "sequenceNum"],
  stored_as_scd_type = 1
)
```

After running the SCD type 1 example, the target table contains the following records:

userId	name	city
124	Raul	Oaxaca
125	Mercedes	Guadalajara
126	Lily	Cancun

After running the SCD type 1 example with the additional `TRUNCATE` record, records `124` and `126` are truncated because of the `TRUNCATE` operation at `sequenceNum=3`, and the target table contains the following record:

userId	name	city
125	Mercedes	Guadalajara

Process SCD type 2 updates

The following example demonstrates processing SCD type 2 updates:

Python SQL

Python

```
from pyspark import pipelines as dp
from pyspark.sql.functions import col, expr

@dp.view
def users():
  return spark.readStream.table("cdc_data.users")

dp.create_streaming_table("target")

dp.create_auto_cdc_flow(
  target = "target",
  source = "users",
  keys = ["userId"],
  sequence_by = col("sequenceNum"),
  apply_as_deletes = expr("operation = 'DELETE'"),
  except_column_list = ["operation", "sequenceNum"],
  stored_as_scd_type = "2"
)
```

After running the SCD type 2 example, the target table contains the following records:

userId	name	city	__START_AT	__END_AT
123	Isabel	Monterrey	1	5
123	Isabel	Chihuahua	5	6
124	Raul	Oaxaca	1	null
125	Mercedes	Tijuana	2	5
125	Mercedes	Mexicali	5	6
125	Mercedes	Guadalajara	6	null
126	Lily	Cancun	2	null

An SCD type 2 query can also specify a subset of output columns to be tracked for history in the target table. Changes to other columns are updated in place rather than generating new history records. The following example demonstrates excluding the `city` column from tracking:

The following example demonstrates using track history with SCD type 2:

Python SQL

Python

```python
from pyspark import pipelines as dp
from pyspark.sql.functions import col, expr

@dp.view
def users():
  return spark.readStream.table("cdc_data.users")

dp.create_streaming_table("target")

dp.create_auto_cdc_flow(
  target = "target",
  source = "users",
  keys = ["userId"],
```

```
  sequence_by = col("sequenceNum"),
  apply_as_deletes = expr("operation = 'DELETE'"),
  except_column_list = ["operation", "sequenceNum"],
  stored_as_scd_type = "2",
  track_history_except_column_list = ["city"]
)
```

After running this example without the additional `TRUNCATE` record, the target table contains the following records:

userId	name	city	__START_AT	__END_AT
123	Isabel	Chihuahua	1	6
124	Raul	Oaxaca	1	null
125	Mercedes	Guadalajara	2	null
126	Lily	Cancun	2	null

Generate test data

The code below is provided to generate an example dataset for use in the example queries present in this tutorial. Assuming that you have the proper credentials to create a new schema and create a new table, you can run these statements with either a notebook or Databricks SQL. The following code is **not** intended to be run as part of a pipeline definition:

SQL

```
CREATE SCHEMA IF NOT EXISTS cdc_data;

CREATE TABLE
  cdc_data.users
AS SELECT
  col1 AS userId,
  col2 AS name,
  col3 AS city,
  col4 AS operation,
  col5 AS sequenceNum
```

```
FROM (
  VALUES
  -- Initial load.
  (124, "Raul",     "Oaxaca",      "INSERT", 1),
  (123, "Isabel",   "Monterrey",   "INSERT", 1),
  -- New users.
  (125, "Mercedes", "Tijuana",     "INSERT", 2),
  (126, "Lily",     "Cancun",      "INSERT", 2),
  -- Isabel is removed from the system and Mercedes moved to Guadalajara.
  (123, null,       null,          "DELETE", 6),
  (125, "Mercedes", "Guadalajara", "UPDATE", 6),
  -- This batch of updates arrived out of order. The above batch at sequenceNum 6 will be
the final state.
  (125, "Mercedes", "Mexicali",    "UPDATE", 5),
  (123, "Isabel",   "Chihuahua",   "UPDATE", 5)
  -- Uncomment to test TRUNCATE.
  -- ,(null, null,      null,          "TRUNCATE", 3)
);
```

Example: Periodic snapshot processing

The following example demonstrates SCD type 2 processing that ingests snapshots of a table stored at `mycatalog.myschema.mytable`. The results of processing are written to a table named `target`.

`mycatalog.myschema.mytable` records at the timestamp 2024-01-01 00:00:00

Key	Value
1	a1
2	a2

`mycatalog.myschema.mytable` records at the timestamp 2024-01-01 12:00:00

Key	Value
2	b2

Key	Value
3	a3

Python

```
from pyspark import pipelines as dp

@dp.view(name="source")
def source():
 return spark.read.table("mycatalog.myschema.mytable")

dp.create_streaming_table("target")

dp.create_auto_cdc_from_snapshot_flow(
 target="target",
 source="source",
 keys=["key"],
 stored_as_scd_type=2
)
```

After processing the snapshots, the target table contains the following records:

Key	Value	__START_AT	__END_AT
1	a1	2024-01-01 00:00:00	2024-01-01 12:00:00
2	a2	2024-01-01 00:00:00	2024-01-01 12:00:00
2	b2	2024-01-01 12:00:00	null
3	a3	2024-01-01 12:00:00	null

Example: Historical snapshot processing

The following example demonstrates SCD type 2 processing that updates a target table based on source events from two snapshots stored in a cloud storage system:

Snapshot at `timestamp`, stored in `/<PATH>/filename1.csv`

Key	TrackingColumn	NonTrackingColumn
1	a1	b1
2	a2	b2
4	a4	b4

Snapshot at `timestamp + 5`, stored in `/<PATH>/filename2.csv`

Key	TrackingColumn	NonTrackingColumn
2	a2_new	b2
3	a3	b3
4	a4	b4_new

The following code example demonstrates processing SCD type 2 updates with these snapshots:

Python

```
from pyspark import pipelines as dp

def exist(file_name):
  # Storage system-dependent function that returns true if file_name exists, false
otherwise

# This function returns a tuple, where the first value is a DataFrame containing the
snapshot
# records to process, and the second value is the snapshot version representing the
logical
# order of the snapshot.
# Returns None if no snapshot exists.
def next_snapshot_and_version(latest_snapshot_version):
  latest_snapshot_version = latest_snapshot_version or 0
  next_version = latest_snapshot_version + 1
```

```
  file_name = "dir_path/filename_" + next_version + ".csv"
  if (exist(file_name)):
    return (spark.read.load(file_name), next_version)
   else:
     # No snapshot available
     return None

dp.create_streaming_live_table("target")

dp.create_auto_cdc_from_snapshot_flow(
  target = "target",
  source = next_snapshot_and_version,
  keys = ["Key"],
  stored_as_scd_type = 2,
  track_history_column_list = ["TrackingCol"]
)
```

After processing the snapshots, the target table contains the following records:

Key	TrackingColumn	NonTrackingColumn	__START_AT	__END_AT
1	a1	b1	1	2
2	a2	b2	1	2
2	a2_new	b2	2	null
3	a3	b3	2	null
4	a4	b4_new	1	null

Add, change, or delete data in a target streaming table

If your pipeline publishes tables to Unity Catalog, you can use data manipulation language (DML) statements, including insert, update, delete, and merge statements, to modify the target streaming tables created by `AUTO CDC ... INTO` statements.

NOTE

- DML statements that modify the table schema of a streaming table are not supported. Ensure that your DML statements do not attempt to evolve the table schema.
- DML statements that update a streaming table can be run only in a shared Unity Catalog cluster or a SQL warehouse using Databricks Runtime 13.3 LTS and above.
- Because streaming requires append-only data sources, if your processing requires streaming from a source streaming table with changes (for example, by DML statements), set the skipChangeCommits flag when reading the source streaming table. When `skipChangeCommits` is set, transactions that delete or modify records on the source table are ignored. If your processing does not require a streaming table, you can use a materialized view (which does not have the append-only restriction) as the target table.

Because Lakeflow Spark Declarative Pipelines uses a specified `SEQUENCE BY` column and propagates appropriate sequencing values to the `__START_AT` and `__END_AT` columns of the target table (for SCD type 2), you must ensure that DML statements use valid values for these columns to maintain the proper ordering of records. See How is CDC implemented with the AUTO CDC API?.

For more information about using DML statements with streaming tables, see Add, change, or delete data in a streaming table.

The following example inserts an active record with a start sequence of 5:

SQL

```
INSERT INTO my_streaming_table (id, name, __START_AT, __END_AT) VALUES (123, 'John Doe',
5, NULL);
```

Read a change data feed from an AUTO CDC target table

In Databricks Runtime 15.2 and above, you can read a change data feed from a streaming table that is the target of `AUTO CDC` or `AUTO CDC FROM SNAPSHOT` queries in the same way that you read a

change data feed from other Delta tables. The following are required to read the change data feed from a target streaming table:

- The target streaming table must be published to Unity Catalog. See Use Unity Catalog with pipelines.
- To read the change data feed from the target streaming table, you must use Databricks Runtime 15.2 or above. To read the change data feed in a different pipeline, the pipeline must be configured to use Databricks Runtime 15.2 or above.

You read the change data feed from a target streaming table that was created in Lakeflow Spark Declarative Pipelines the same way as reading a change data feed from other Delta tables. To learn more about using the Delta change data feed functionality, including examples in Python and SQL, see Use Delta Lake change data feed on Databricks.

NOTE

The change data feed record includes metadata identifying the type of change event. When a record is updated in a table, the metadata for the associated change records typically includes `_change_type` values set to `update_preimage` and `update_postimage` events.

However, the `_change_type` values are different if updates are made to the target streaming table that include changing primary key values. When changes include updates to primary keys, the `_change_type` metadata fields are set to `insert` and `delete` events. Changes to primary keys can occur when manual updates are made to one of the key fields with an `UPDATE` or `MERGE` statement or, for SCD type 2 tables, when the `__start_at` field changes to reflect an earlier start sequence value.

The `AUTO CDC` query determines the primary key values, which differ for SCD type 1 and SCD type 2 processing:

- For SCD type 1 processing and the pipelines Python interface, the primary key is the value of the `keys` parameter in the `create_auto_cdc_flow()` function. For the SQL interface the primary key is the columns defined by the `KEYS` clause in the `AUTO CDC ... INTO` statement.
- For SCD type 2, the primary key is the `keys` parameter or `KEYS` clause plus the return value from the `coalesce(__START_AT, __END_AT)` operation, where `__START_AT` and `__END_AT` are the corresponding columns from the target streaming table.

Get data about records processed by a CDC query in pipelines

> **NOTE**
>
> The following metrics are captured only by `AUTO CDC` queries and not by `AUTO CDC FROM SNAPSHOT` queries.

The following metrics are captured by `AUTO CDC` queries:

- **`num_upserted_rows`**: The number of output rows upserted into the dataset during an update.
- **`num_deleted_rows`**: The number of existing output rows deleted from the dataset during an update.

The `num_output_rows` metric, output for non-CDC flows, is not captured for `AUTO CDC` queries.

What data objects are used for CDC processing in a pipeline?

> **NOTE**
>
> - These data structures apply only to `AUTO CDC` processing, not `AUTO CDC FROM SNAPSHOT` processing.
> - These data structures apply only when the target table is published to the Hive metastore. If a pipeline publishes to Unity Catalog, the internal backing tables are inaccessible to users.

When you declare the target table in the Hive metastore, two data structures are created:

- A view using the name assigned to the target table.
- An internal backing table used by the pipeline to manage CDC processing. This table is named by prepending `__apply_changes_storage_` to the target table name.

For example, if you declare a target table named `dp_cdc_target`, you will see a view named `dp_cdc_target` and a table named `__apply_changes_storage_dp_cdc_target` in the metastore.

Creating a view allows Lakeflow Spark Declarative Pipelines to filter out the extra information (for example, tombstones and versions) required to handle out-of-order data. To view the processed data, query the target view. Because the schema of the `__apply_changes_storage_` table might change to support future features or enhancements, you should not query the table for production use. If you add data manually to the table, the records are assumed to come before other changes because the version columns are missing.

Additional resources

- Tutorial: Build an ETL pipeline using change data capture