Deferred Merge (for high-churn tables)

Some DML patterns cause poor DML performance in the data pipeline that, in some cases, can result in extremely high data latency and other performance problems. This article explains why the problem occurs and provides potential solutions.

By Darren Gardner

> Note: The title of this article has been renamed from "Hybrid Merge" to "Deferred Merge" to avoid confusion with hybrid tables.

Problem definition

Some DML patterns do not perform well in Snowflake. One potentially challenging pattern that some customers have encountered is with a MERGE or UPDATE against a random subset of the rows in the target table. Even if only an extremely small *percentage* of the rows in the target table are updated, if those rows are distributed across a high number of micro-partitions, then *all rows within* these micro-partitions (since micro-partitions are *immutable*) will need to be completely rewritten to new micro-partitions. At a minimum, this can result in poor DML performance in the data pipeline, and in some cases cause extremely high data latency. This can also adversely affect the clustering of the table, negatively impacting query performance against the target table due to poor pruning.

Deferred merge solution

> Note: In this article, we consider the case where only the current (final) version of each logical entity (keyed by an `ID` column) needs to be maintained in a target table. But it should be possible to extend it to cover other common scenarios (such as type-2 SCD tables with full versioning), as well.

The basis of the proposed "deferred" solution is to replace the original existing table (that the DMLs are being applied to) with *three* objects:

1. A "base" table that stores a consistent "snapshot" of the data at a specific point in time (not necessarily "current" to the latest set of data). It has an identical column structure as the original table, but with a different name.
2. A "delta" table that acts as an append-only buffer for rows before they are applied to the base table. Note that this can often simply be the staging table at the start of the data pipeline, in which case there is no need for a new physical table. It typically has an identical column structure as the original table, plus an additional timestamp column that indicates the effective time of the row. This can be an existing column in the table, or it can be computed via a context function (*e.g.* `CURRENT_TIMESTAMP`) at the time the data *would* have been applied (typically via a MERGE).
3. A view that has an identical column structure as the original table with the same name as the original table. This view coalesces data from the base and delta tables "on the fly" to present a "current" (final) view of the data that matches what we would see in the original table if we had applied the MERGE.

Now for the workflow:

Assume that we start with all tables empty. When the system encounters new rows, it appends them to the delta table. If a query is issued against the view before the delta data is merged into the base table, the system uses the delta data to compute the "final state." Otherwise, the system combines the base table and the delta table data to compute the final state.

An automated job merges the delta table data into the base table. This job runs on a schedule, so data flowing into the delta table is not a factor. When the job runs, delta table data is briefly present in both the delta table and the base table. The view therefore needs to avoid double processing these rows and ensure that the current state snapshot is correct. Once the delta rows are applied to the base table, the job purges the rows from the delta table.

> Note: It would NOT be appropriate to accumulate *all* historical data in the delta table indefinitely. The fundamental underpinning of the deferred approach is that applying dynamic logic (typically, window functions) to a **subset** of data and blending with the accumulated (base) data can be done efficiently (and sidestep the issue of costly merges being continuously applied, by "batching" them at intervals that reduce micro-partition churn).

Sample SQL script

Initial Configuration (sample tables)

```
CREATE OR REPLACE TRANSIENT TABLE T_BASE (
  ID     INTEGER
 ,COL_X  VARCHAR
 ,COL_Y  VARCHAR
)
;
CREATE OR REPLACE TRANSIENT TABLE T_DELTA (
  ID     INTEGER
 ,TS     TIMESTAMP_LTZ
 ,COL_X  VARCHAR
 ,COL_Y  VARCHAR
)
;
```

Preferred Approach: Stream-based source using UNION ALL with NOT EXISTS

Overview:

This approach uses a UNION ALL to blend the delta data with the base, with a NOT EXISTS added to ensure that a given key (ID) is *only* emitted from the base data if it is *not* emitted by the delta data. One significant performance benefit of this approach is that any filters applied against the view will be "pushed down" into each branch of the UNION ALL, allowing for pruning against the base table, which we expect can grow to be quite large over time. Attaching an *append-only* stream object to the delta table and using it as our source for the delta data that is blended with the base data is recommended, since it efficiently emits only those rows in the delta table that have not yet been applied to the base, which eliminates the need to aggressively truncate data from the delta table that might otherwise be required to achieve reasonable performance. Also, to prevent the delta table from growing

without bound indefinitely, periodic purges of delta data would need to be performed, and using an append-only stream reduces the complexity of eliminating race conditions when purging delta data is done while new data is being loaded into the delta table. In fact, rows from the delta data can be purged even *before* the stream has been consumed, and the stream will still emit the rows, since an append-only stream ignores delete operations. Overall, this is typically the simplest and most efficient approach to the deferred merge strategy.

View Definition:

```
CREATE OR REPLACE STREAM STRM_DELTA ON TABLE T_DELTA APPEND_ONLY=TRUE
;

CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_NET_DELTA AS (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM STRM_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  )
  SELECT B.ID
        ,B.COL_X
        ,B.COL_Y
    FROM T_BASE B
   WHERE NOT EXISTS (
           SELECT 1
             FROM CTE_NET_DELTA D
            WHERE D.ID = B.ID
         )
  UNION ALL
  SELECT D.ID
        ,D.COL_X
        ,D.COL_Y
    FROM CTE_NET_DELTA D
;
```

MERGE Statement (used to periodically apply delta data to base table)

```
MERGE INTO T_BASE B
  USING (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM STRM_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  ) D
  ON D.ID = B.ID
    WHEN NOT MATCHED THEN INSERT (ID, COL_X, COL_Y) VALUES (D.ID, D.
COL_X, D.COL_Y)
    WHEN MATCHED THEN UPDATE SET COL_X = D.COL_X, COL_Y = D.COL_Y
;
```

Test Harness (to demonstrate logic and validate correctness)

```
-- Ensure tables are empty (if re-executing tests)
TRUNCATE TABLE T_BASE
;

TRUNCATE TABLE T_DELTA
;

-- Verify that initial result set is empty...
SELECT *
  FROM V_MERGED
;

-- Acquire an initial delta (batch)... but do not yet apply it to the
base table...
INSERT INTO T_DELTA (ID, TS, COL_X, COL_Y)
  VALUES
    (1, '2020-01-01 01:00:00', 'X1A', 'Y1A')
   ,(2, '2020-01-01 01:00:00', 'X2A', 'Y2A')
   ,(2, '2020-01-01 01:01:00', 'X2B', 'Y2A')
;

-- View the "final" result set (computed from just the delta stream,
since the base table is empty)...
SELECT *
  FROM V_MERGED
;

-- Purge all rows from the delta table...
```

```
-- Note that there is a NO potential race condition here, even if new
rows are being continuously loaded into delta table.
-- Also note that we have not yet consumed the rows, but they will
still be emitted by the stream later on
TRUNCATE TABLE T_DELTA
;

-- Confirm that newly added rows that were purged from the delta table
are still emitted from the stream...
SELECT *
  FROM STRM_DELTA
;

-- View the "final" result set (computed from just the delta stream,
since the base table is empty)...
SELECT *
  FROM V_MERGED
;

-- Now apply the delta stream to the base table...
-- Note that the QUALIFY filter ensures that only the LATEST VERSION of
each key (ID) is applied
MERGE INTO T_BASE B
  USING (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM STRM_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  ) D
  ON D.ID = B.ID
    WHEN NOT MATCHED THEN INSERT (ID, COL_X, COL_Y) VALUES (D.ID, D.
COL_X, D.COL_Y)
    WHEN MATCHED THEN UPDATE SET COL_X = D.COL_X, COL_Y = D.COL_Y
;

-- Confirm that the stream is now empty since the rows have been
applied to the base table via the MERGE...
SELECT *
  FROM STRM_DELTA
;

-- Again, view the "final" result set (blending data from delta stream
and base table), and confirm that it matches the previous result set...
SELECT *
  FROM V_MERGED
;

-- Acquire another delta (batch)... but do not yet apply it to the base
table...
```

```
INSERT INTO T_DELTA (ID, TS, COL_X, COL_Y)
  VALUES
    (1, '2020-01-01 02:00:00', 'X1A', 'Y1B')
   ,(2, '2020-01-01 02:00:00', 'X2C', 'Y2A')
   ,(2, '2020-01-01 02:01:00', 'X2C', 'Y2B')
   ,(3, '2020-01-01 02:00:00', 'X3A', 'Y3A')
   ,(3, '2020-01-01 02:01:00', 'X3B', 'Y3B')
   ,(3, '2020-01-01 02:02:00', 'X3C', 'Y3C')
;

-- Purge all rows from the delta table...
TRUNCATE TABLE T_DELTA
;

-- Again, view the "final" result set (blending data from delta stream
and base table), and confirm that it reflects the latest batch...
SELECT *
  FROM V_MERGED
;

-- Apply the new batch in the delta stream to the base table...
MERGE INTO T_BASE B
  USING (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM STRM_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  ) D
  ON D.ID = B.ID
    WHEN NOT MATCHED THEN INSERT (ID, COL_X, COL_Y) VALUES (D.ID, D.
COL_X, D.COL_Y)
    WHEN MATCHED THEN UPDATE SET COL_X = D.COL_X, COL_Y = D.COL_Y
;

-- Again, view the "final" result set (blending data from delta stream
and base table), and confirm that it matches the previous result set...
SELECT *
  FROM V_MERGED
;
```

Alternative Approach: Table-based source using UNION ALL with NOT EXISTS

Overview:

This approach is almost identical to the previous approach, except it does not incorporate a stream. Since the delta table will grow over time, it is expected that frequent purges of delta data from the table, and/or a more advanced technique (such as the incorporation of a high-water mark filter) will be required to achieve reasonable performance. Also, without streams, there is the potential for race conditions if purges of delta data occur at the

same time as new rows are being added to the delta table, which often increases operational complexity. So, we present it here as an option, but in general recommend the stream-based approach over it.

View Definition:

```
CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_NET_DELTA AS (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM T_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  )
  SELECT B.ID
        ,B.COL_X
        ,B.COL_Y
    FROM T_BASE B
   WHERE NOT EXISTS (
           SELECT 1
             FROM CTE_NET_DELTA D
            WHERE D.ID = B.ID
         )
  UNION ALL
  SELECT D.ID
        ,D.COL_X
        ,D.COL_Y
    FROM CTE_NET_DELTA D
;
```

MERGE Statement (used to periodically apply delta data to base table)

```
MERGE INTO T_BASE B
  USING (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM T_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  ) D
  ON D.ID = B.ID
    WHEN NOT MATCHED THEN INSERT (ID, COL_X, COL_Y) VALUES (D.ID, D.
COL_X, D.COL_Y)
    WHEN MATCHED THEN UPDATE SET COL_X = D.COL_X, COL_Y = D.COL_Y
;
```

Optional enhancements (schema change required: addition of timestamp to base table)

Note: this is a prerequisite for each of the alternative approaches that are presented below, none of which uses a stream object.

Modified Schema for Base Table

```
CREATE OR REPLACE TRANSIENT TABLE T_BASE (
  ID     INTEGER
 ,TS     TIMESTAMP_LTZ  -- New timestamp column added here
 ,COL_X  VARCHAR
 ,COL_Y  VARCHAR
)
;
```

Modified MERGE statement (to manage new timestamp column)

```
MERGE INTO T_BASE B
  USING (
    SELECT ID, TS, COL_X, COL_Y
      FROM T_DELTA
     QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  ) D
  ON D.ID = B.ID
  WHEN NOT MATCHED THEN INSERT (ID, TS, COL_X, COL_Y) VALUES (D.ID, D.
TS, D.COL_X, D.COL_Y)
  WHEN MATCHED AND D.TS >= B.TS THEN UPDATE SET TS = D.TS, COL_X = D.
COL_X, COL_Y = D.COL_Y
;
```

Approach: Use MAX(<timestamp>) in base table as High-Water-Mark

An alternative to using a stream would be to incorporate "high-water mark" logic via a timestamp column in the base table. Assuming that the data in the delta table is naturally chronologically clustered, this filter condition (that uses a fast *metadata-only* query against the T_BASE table) should prune away all historical rows and avoid a table scan of the delta table.

```
CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_NET_DELTA AS (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM T_DELTA
     WHERE TS > (SELECT IFNULL(MAX(TS), TO_TIMESTAMP(0)) FROM T_BASE)
-- New filter condition against timestamp
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  )
  SELECT B.ID
        ,B.COL_X
        ,B.COL_Y
    FROM T_BASE B
   WHERE NOT EXISTS (
           SELECT 1
             FROM CTE_NET_DELTA D
            WHERE D.ID = B.ID
         )
  UNION ALL
  SELECT D.ID
        ,D.COL_X
        ,D.COL_Y
    FROM CTE_NET_DELTA D
;
```

Approach: Handle Out-Of-Sequence (Late-Arriving) Data

All of the implementations above assume that the latest data (for an `ID`) in the delta data set is *always* chronologically greater than or equal to the latest data (for that same `ID`) in the base table. So, if the base table is current as of T2, the latest data set in the delta table for an `ID` will never be T1 (< T2). The `IFF(...)` logic in the code is dependent on this being true. However, this logic can be adjusted to account for late-arriving delta data, if we track the base data with a timestamp as follows:

```
CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_NET_DELTA AS (
    SELECT ID
          ,TS
          ,COL_X
          ,COL_Y
      FROM T_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  )
  SELECT B.ID
        ,B.COL_X
        ,B.COL_Y
    FROM T_BASE B
   WHERE NOT EXISTS (
           SELECT 1
             FROM CTE_NET_DELTA D
            WHERE D.ID = B.ID
              AND D.TS >= B.TS  -- Modified logic here
         )
  UNION ALL
  SELECT D.ID
        ,D.COL_X
        ,D.COL_Y
    FROM CTE_NET_DELTA D
   WHERE NOT EXISTS (
           SELECT 1
             FROM T_BASE B
            WHERE B.ID = D.ID
              AND B.TS > D.TS  -- Modified logic here
         )
;
```

- *This approach can also be adapted to use the delta stream or HWM enhancements described above.*
- *If you use this enhancement, modify the MERGE in the test script to manage the TS column in the base table. See MERGE example provided above.*

Alternative approaches considered (but NOT RECOMMENDED)

Alternative Approach: FULL OUTER JOIN

```
CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_NET_DELTA AS (
    SELECT ID
          ,COL_X
          ,COL_Y
      FROM T_DELTA
    QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
  )
  SELECT IFF(D.ID IS NOT NULL, D.ID, B.ID) AS ID
        ,IFF(D.ID IS NOT NULL, D.COL_X, B.COL_X) AS COL_X
        ,IFF(D.ID IS NOT NULL, D.COL_Y, B.COL_Y) AS COL_Y
    FROM T_BASE B
         FULL OUTER JOIN CTE_NET_DELTA D
           ON D.ID = B.ID
;
```

Alternative Approach: UNION ALL inside window function

```
CREATE OR REPLACE VIEW V_MERGED
AS
  WITH CTE_UNION_ALL AS (
    SELECT ID
          ,COL_X
          ,COL_Y
          ,TS
      FROM T_DELTA

    UNION ALL

   SELECT ID
          ,COL_X
          ,COL_Y
          ,'1900-01-01'
      FROM T_BASE
  )
  SELECT ID
        ,COL_X
        ,COL_Y
    FROM CTE_UNION_ALL
  QUALIFY 1 = ROW_NUMBER() OVER (PARTITION BY ID ORDER BY TS DESC)
;
```

Comparison of approaches

We have determined that the **UNION ALL with NOT EXISTS** approach will usually perform much better than the **FULL OUTER JOIN** or **UNION ALL inside window function** approaches. The primary reason is that it supports far better pruning against the base table. The **FULL OUTER JOIN** approach does not allow predicate pushdown (a significant prerequisite for effective pruning) due to the coalescing operation that is used to derive the column expressions. Similarly, the **UNION ALL inside window function** approach requires that all data (base and delta) pass through a window function to determine the final values, and predicate pushdown is only possible for columns that are included in the PARTITION BY clause. In this case, that would be the ID column. Filtering against any other column(s) would first need to apply the window function to all of the data, and then filter afterwards. By contrast, the **UNION ALL with NOT EXISTS** approach allows predicate pushdown against the base table, which can provide tremendous performance benefits over the other two approaches.