OPTIMIZING ETL SQL JOBS FOR INCREMENTAL DATA LOADS FROM STREAMING BIGQUERY ODS TABLES

Tabrez Alam
Expert Data Architect, JB Hunt Transport Inc, Lowell, AR, USA
Email: ***totabrez@gmail.com***

ABSTRACT

This document outlines SQL-based optimization techniques for incremental ETL loads from streaming BigQuery ODS tables. It addresses performance challenges caused by append-only ingestion and proposes solutions such as partition-aware filtering, clustering, metadata-driven logic, deduplication, and a two-step incremental load process. These strategies aim to reduce query costs, improve runtime, and enhance resource efficiency for scalable data processing.

1. INTRODUCTION

Modern data platforms increasingly rely on streaming ingestion to capture real-time business events. Streaming pipelines often land data into BigQuery Operational Data Store (ODS) tables in an append-only fashion. While this ensures data freshness, it creates performance challenges for downstream ETL processes, especially for incremental data loads.

This document explores SQL-based optimization techniques to efficiently load incremental data from streaming BigQuery ODS tables while reducing cost and runtime. The techniques include partition-aware filtering, clustering, metadata-driven logic, deduplication, and a two-step incremental load process.

2. Scenario Overview

- Streaming Ingestion: Data is continuously ingested into BigQuery ODS tables via Kafka, Dataflow, or other streaming pipelines.
- Table Partitioning: ODS tables are partitioned on business-relevant date columns, such as CreateTimestamp, OpenDate, or InvoiceDate.
- ETL Frequency: Jobs run every 2 hours to populate staging tables or data marts.
- Incremental Logic: Currently, ETL uses LastUpdatedTimestamp to filter recently changed rows.

3. Problem Statement

The current ETL design causes:

1. Missed Partition Pruning: Filters on LastUpdatedTimestamp bypass the partition column.
2. Full Table Scans: Billions of rows are scanned unnecessarily.
3. High Query Cost: Full scans increase BigQuery usage charges.
4. Long ETL Runtime: Delays downstream data availability.
5. Inefficient Slot Usage: ETL consumes BigQuery slots unnecessarily, affecting other workloads.

4. Objective

To demonstrate practical optimization techniques that make incremental ETL loads faster, cost-efficient, and resource-efficient. Techniques include partition-aware queries, clustering, metadata tables, deduplication, and a two-step ETL process.

5. Proposed Solutions

5.1 Partition-Aware Incremental Queries

Solution: Filter on the table's partition column in addition to Last Updated Timestamp.

Example SQL:

```
DECLARE last_run_partition DATE DEFAUL
T DATE_SUB(CURRENT_DATE(), INTERV
AL 2 DAY);
SELECT *
FROM ods_table
WHERE CreateTimestamp BETWEEN last_run
_partition AND CURRENT_DATE()
  AND LastUpdatedTimestamp > TIMESTAMP
_SUB(CURRENT_TIMESTAMP(), INTERVA
L 2 HOUR);
```

Benefit: Enables partition pruning, scanning only relevant slices.

5.2 Clustering on Frequently Filtered Columns

Solution: Cluster tables on columns frequently used in filters, such as LastUpdatedTimestamp or primary keys.

Example SQL:

```
CREATE TABLE ods_table_clustered
PARTITION BY DATE(CreateTimestamp)
CLUSTER BY LastUpdatedTimestamp, Primar
yKeyColumn AS
SELECT * FROM ods_table;
```

Benefit: Reduces the number of blocks read, improving performance and lowering cost.

5.3 Metadata-Driven Incremental Loads

Solution: Maintain a metadata table to track maximum LastUpdatedTimestamp per partition.

SQL Example:

```
SELECT o.*
FROM ods_table o
JOIN metadata_table m
  ON DATE(o.CreateTimestamp) = m.Partition
Date
WHERE o.LastUpdatedTimestamp > m.MaxLas
tUpdatedTimestamp;
```

Benefit: Scans only partitions with new or updated data.

5.4 Efficient Change Tracking and Deduplication

Solution: Use window functions over filtered partitions to select the latest row per primary key.

Example SQL:

```
WITH filtered AS (
  SELECT *
  FROM ods_table
  WHERE DATE(CreateTimestamp) BETWEE
N @start_partition AND @end_partition
),
deduped AS (
  SELECT *
  FROM (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION
BY PrimaryKeyColumn ORDER BY LastUpdat
edTimestamp DESC) AS rn
    FROM filtered
  )
  WHERE rn = 1
)
SELECT * FROM deduped;
```

Benefit: Eliminates duplicates efficiently without scanning the full table.

5.5 Two-Step Partition-Aware Incremental Load

Problem: Current ETL query:

```
SELECT *
FROM ods_table
WHERE LastUpdatedTimestamp >= TIMESTA
MP_SUB(CURRENT_TIMESTAMP(), INTER
VAL 2 HOUR);
```

- Scans the full table.
- Very expensive on large ODS tables.

Solution: Implement a two-step process leveraging the partition column:

Step 1 – Identify the Minimum Partition to Scan:

```
SELECT MIN(CreateTimestamp) AS INCRCreateTimestamp
FROM ods_table
WHERE LastUpdatedTimestamp >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 2 HOUR);
```

Step 2 – Load Incremental Data Efficiently:

```
SELECT *
FROM ods_table
WHERE CreateTimestamp >= @INCRCreateTimestamp
  AND LastUpdatedTimestamp >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 2 HOUR);
```

Benefit: - Scans only relevant partitions. - Retains incremental logic. - Reduces ETL cost by at least 50%. - Improves runtime and resource utilization.

5.6 ETL Scheduling and Partition Windowing

- Run ETL over recent partitions only (e.g., last 2–3 days).
- Combine partition filters with LastUpdatedTimestamp for hybrid incremental load.
- Avoid full-table scans by restricting processing to partitions with new updates.

6. Expected Outcomes

Implementing these strategies provides: - Reduced Query Costs - Faster ETL Execution - Efficient BigQuery Resource Usage - Scalable Architecture for Streaming Tables

7. Conclusion

Optimizing incremental ETL for streaming BigQuery ODS tables requires aligning filters with partition columns, leveraging clustering and metadata tables, and implementing efficient incremental load logic. The proposed two-step ETL approach is highly effective in reducing cost and runtime while maintaining data accuracy. By adopting these techniques, organizations can efficiently process billions of streaming rows with minimal cost and maximum performance.

References

1. Google Cloud BigQuery Documentation: https://cloud.google.com/bigquery/docs
2. Best Practices for ETL in BigQuery: https://cloud.google.com/architecture/bigquery-etl-patterns
3. Streaming Data into BigQuery: https://cloud.google.com/bigquery/streaming-data-into-bigquery
4. Partitioned Tables in BigQuery: https://cloud.google.com/bigquery/docs/partitioned-tables
5. Clustering Tables in BigQuery: https://cloud.google.com/bigquery/docs/clustered-tables