Normalized vs Denormalized Data

Optimizing SQL for Analytics in the Cloud

Objectives

1. Understand characteristics of normalized vs denormalized data
2. Compare advantages and disadvantages in PostgreSQL vs BigQuery
3. Understand how schema design affects cost and performance
4. Work through examples
5. Apply best practices for PostgreSQL (OLTP) and BigQuery (OLAP)

Characteristics of Normalized Data

- Organized into multiple related tables
- Minimizes redundancy, enforces data integrity
- Queries often require joins
- Best suited for OLTP systems like PostgreSQL

Normalized Data

The CMS data we've worked with is a good example of normalized data.

- Data in multiple related tables
- Each table is centered around a single concept
- Minimal data duplication
- Joins required

Characteristics of Denormalized Data

- Combines data into fewer, wider tables
- Increases redundancy (data storage) but simplifies queries
- Fewer joins, optimized for analytics (OLAP)
- May increase storage and risk of inconsistency

Denormalized Data

BigQuery's INFORMATION_SCHEMA tables are a good example of denormalized data

- Schema includes multiple array fields
- Array fields are dynamic to include information relevant to that row (even if it's not included on other rows)

BigQuery vs PostgreSQL

PostgreSQL:

- Excels with normalized schemas
- Joins are efficient with indexes

BigQuery:

- Prefers denormalized or nested schemas
- Joins are costly and increase scanned data
- Query cost optimization is preferred over minimizing storage cost

Cost and Performance Comparison

PostgreSQL:

- Normalized: low storage, fast joins
- Denormalized: simple queries, slower writes

BigQuery:

- Normalized: expensive joins, higher query costs
- Denormalized: fewer joins, lower query cost
- Nested: efficient balance, best practice for BigQuery

Types of Denormalized Datatypes in BigQuery

- Nested Fields (STRUCT): Think of a table inside a cell. A single field (like error_result) contains its own sub-fields (like reason, message)
- Repeated Fields (ARRAY): Think of a “list of items inside a cell”. A single field (eg labels) can hold multiple values, and each of those can be a STRUCT.
- JSON: store semi-structured data in a single field without a pre-defined schema for internal elements; provides flexibility for data where the schema might evolve or be inconsistent.

Performance: This structure avoids JOINs, making analysis extremely fast. The data is “pre-joined” by nesting it.

Querying Nested Fields

To access sub-fields in a STRUCT: use dot (.) notation

Just like accessing a column from a table alias in postgres, but you're accessing a "column within a column".

Querying Nested Fields - Example

```
SELECT creation_time,
  job_id,
  error_result.message AS error_message,
  error_result.reason AS error_reason
FROM `region-us`.INFORMATION_SCHEMA.JOBS
WHERE error_result.reason IS NOT NULL;
```

Querying Repeated Fields

- Repeated fields (arrays) require a different approach since they often have repeated keys.
- Use the UNNEST() operator in your FROM clause to create a new row for each item in the array.

```
SELECT creation_time,
  job_id,
  t.project_id,
  t.dataset_id,
  t.table_id
FROM `region-us`.INFORMATION_SCHEMA.JOBS AS j,
UNNEST(referenced_tables) AS t
```

Querying JSON Fields

To access internal parts within a JSON field, use BigQuery's JSON functions.

JSON_QUERY_ARRAY extracts items in a JSON to an array of JSON values which can be unnested and used in the SELECT.

```
SELECT person_id,
  exposure_json,
  JSON_VALUE(exposure_json, "$.drug_source_concept_id") AS
drug_source_concept_id
FROM
`advanced-sql-class-470811.omop_synthetic_data.drug_exposure_json` AS t,
UNNEST(JSON_QUERY_ARRAY(t.drug_exposures_json, '$')) AS
exposure_json
```

Using INFORMATION_SCHEMA to Track Usage

```
-- This query analyzes BigQuery job metadata to calculate daily total_bytes_billed
-- and a 30-day rolling sum of billed bytes.

WITH daily_billing AS (
  -- 1. Aggregate total_bytes_billed by day
  SELECT
    -- Use job creation date for billing tracking
    DATE(creation_time) AS query_date,

    -- Sum the bytes billed for all relevant jobs on that day
    SUM(total_bytes_billed) / POW(1024, 4) AS tb_daily
  FROM
    -- Reference the regional INFORMATION_SCHEMA.JOBS view.
    `region-us`.INFORMATION_SCHEMA.JOBS
  GROUP BY 1
)
-- 2. Calculate the 30-day rolling total using a window function
SELECT
  query_date,
  tb_daily,
  -- Window function to sum the daily billed bytes over the current row
  -- and the 29 days preceding it (totaling 30 days).
  SUM(tb_daily)
    OVER (
      ORDER BY UNIX_DATE(query_date)
      RANGE BETWEEN 29 PRECEDING AND CURRENT ROW
    ) AS rolling_30_day_tb
FROM daily_billing
ORDER BY query_date DESC
```

Additional Resources

- BigQuery Array Functions
- Working With Arrays (includes information on UNNEST)
- BigQuery JSON Functions
- BigQuery INFORMATION_SCHEMA.JOBS view