Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 2, Issue 3

Open Access | PP: 38-43

DOI: https://doi.org/10.70315/uloap.ulete.2025.0203008

Data Warehousing Techniques for High-Cardinality, High-Frequency Time-Series Analytics

Khrystyna Terletska

Senior Software Engineer at Datadog, NewYork, USA.

Abstract

In the article, modern methods of organizing cloud storage for analytics of high-frequency (HF) and high-cardinality (HC) time series (TS) are examined, driven by the growth of data volumes reaching millions of points per second. The author substantiates the relevance of the study with the example of forecasts for the increase in the number of IoT devices to 41.6 billion by 2025 and the associated needs for HF-HC-TS analysis demanded in real-time financial, telecommunications, and cloud monitoring systems. The primary objective of this study is to review and contrast DWH approaches in light of the swift HF-HC-TS streams. The author proceeds from the systematization of 16 primary sources. Innovation rests in formalizing a set of methods that unite ingest optimization, late event handling via two timestamps, and a trust window. Key results demonstrate that using an append-only model in conjunction with the Capacitor format ensures linear scalability of streaming ingest and minimizes storage costs through delta encoding and dictionary compression. The introduction of two timestamps and watermarks in Dataflow/Apache Beam enables the correct handling of late-arriving points without requiring the rebuilding of historical data. The article demonstrates that combining architectural features of BigQuery with well-conceived operational practices—from mandatory partition filters and ingest lag control to the judicious choice of clustering columns and avoidance of JavaScript UDFs—creates a stable balance between performance and budget. The suggested set of anti-patterns helps identify and rectify unproductive plans promptly, restoring the system to its optimal working state. The mentioned ways ensure meeting SLA needs for delay and price when dealing with data flows that contain billions of records each day. This article is highly relevant to businesses that involve live analysis and monitoring, encompassing finance, IoT reporting, telecommunications, and building management.

Keywords: *High-Frequency Time Series, High Cardinality, BigQuery, Partitioning, Clustering, Capacitor, Append-Only, Watermarks, Surrogate Keys, Change Data Capture.*

INTRODUCTION

High-frequency time series denote streams in which a new point appears with subsecond periodicity, and in financial systems, with submillisecond periodicity. High cardinality refers to hundreds of thousands or millions of independent identifiers by which these points are grouped. Such characteristics are not limited to trading platforms: IDC research forecasts 41.6 billion connected IoT devices by 2025, capable of generating 79.4 zettabytes of data per year, with the majority of this data expected to be high-frequency, high-throughput, and low-latency [1].

The most prominent domains where these series arise naturally are real-time trading systems (tick tapes of stocks and options), telemetry and remote control of IoT fleets, 5G mobile network core signals, as well as detailed event logs in distributed cloud clusters collected by site reliability engineering (SRE) teams for monitoring. In each case, the data arrives continuously but must be analyzed almost instantaneously to ensure, for example, arbitrage trading, adaptive regulation of sensor power, or automatic server scaling.

The primary bottlenecks when working with such streams are well known. First, ingest speed. Even the fundamental limit of the BigQuery Streaming API allows 100,000 rows per second per table and 500,000 rows per project in the US / EU regions [2]. Second, inevitable late and out-of-order events require tolerant semantics and watermarks. Third, scanning petabytes of columns for a second-level chart is expensive. Under BigQuery's on-demand model, billing is based on the number of terabytes scanned, so improper partitioning can easily multiply the bill several times [3]. Finally, complex windowed aggregates must execute within tens of milliseconds, otherwise real-time dashboards lose their relevance. An academic review of time series management systems (TSMS) reveals that specialized time series databases

Citation: Khrystyna Terletska, "Data Warehousing Techniques for High-Cardinality, High-Frequency Time-Series Analytics", Universal Library of Engineering Technology, 2025; 2(3): 38-43. DOI: https://doi.org/10.70315/uloap.ulete.2025.0203008.

(TSDBs) often struggle precisely with the combination of high cardinality and arbitrary analytical queries, as they are optimized for fixed patterns of timestamp-value pairs [4].

Cloud data warehouses, and especially Google BigQuery, remove these limitations through architectural solutions: physical separation of storage and compute allows elastic scaling of slots for peak loads without manual sharding; the columnar Capacitor format automatically applies dictionary and delta encoding, reducing the size of historical tables; streaming ingest via the Storage Write API provides exactly-once delivery even under high queue turbulence. Another critical point is that time-based partitioning and multi-level clustering should be available declaratively in DDL, so engineers do not have to maintain dozens of custom partitions, as in classic TSDBs. Hence, BigQuery can process billions of events per day with a cost-effective scanning and median query latency within the SLA.

MATERIALS AND METHODOLOGY

The storage techniques for HF-HC-TS analytics is drawn from 16 sources: the IDC report on IoT data volumes [1], detailing BigQuery Streaming API and cost estimates for scanning [2, 3], a review TSMS noting that TSDBs have limitations under high cardinality [4], materials on streaming pipelines with Dataflow / Apache Beam [5] and a guide to seamless streaming pipeline updates [6], CDC models in BigQuery[7, 8] partitioning and clustering[9,10] recommendations on compute optimization & use of surrogate keys for Capacitor compression[11] and profiling & managing partition and cluster recommendations[14, 16].

Methodologically, the work combines several elements. First, it compares the append-only ingest model with SCD-2, demonstrating that appends in the Capacitor format minimize costs and simplify sharding under HF conditions [10, 11]. Second, for handling late-arriving data, two timestamps (event_ts and ingest_ts) and a trust window strategy with MERGE-UPSERT are applied on partitions whose ingest_ts is older than watermark-α, as described for Dataflow/Beam [5]. Third, the choice of partition granularity and multi-level clustering for both low- and high-cardinality columns is optimized to reduce the number of bytes scanned and speed up queries [9, 10].

RESULTS AND DISCUSSION

In high-frequency time series with millions of records per second, the best model remains append-only. The new point is added as a separate row. Corrections are published as version 2 of the same business entity. This approach scales exceptionally well because it does not require locks and allows linear sharding of streaming ingestion. Switching to a classic SCD-2 scheme of incremental replacements (update-in-place) only makes sense at extremely low arrival rates, because in BigQuery, it is precisely appends that guarantee the most cost-effective storage, through columnar delta encoding.

The problem of late and out-of-order events is solved by introducing two independent timestamps: event_ts, obtained from the source device, and ingest_ts, assigned by the system upon ingestion. In a streaming pipeline, watermarks advance based on event_ts; at the same time, a trust window of, say, fifteen minutes permits merges performed as a batch MERGE-UPSERT on partitions whose ingest_ts is older than watermark-α. This strategy enables the accounting for almost all late points without rebuilding old partitions. It is formalized in Dataflow/Apache Beam, where an event arriving beyond the watermark is automatically marked as late data [5].

A hopping window is a fixed time interval in the data stream. Hopping windows may overlap, whereas tumbling windows do not overlap. For example, a hopping window may start every thirty seconds and cover a one-minute period of data. The frequency at which hopping windows begin is referred to as the period [5]. In Figure 1, it is shown how elements are divided into one-minute hopping windows with a thirty-second interval. In this example, there is a one-minute window and thirty seconds.

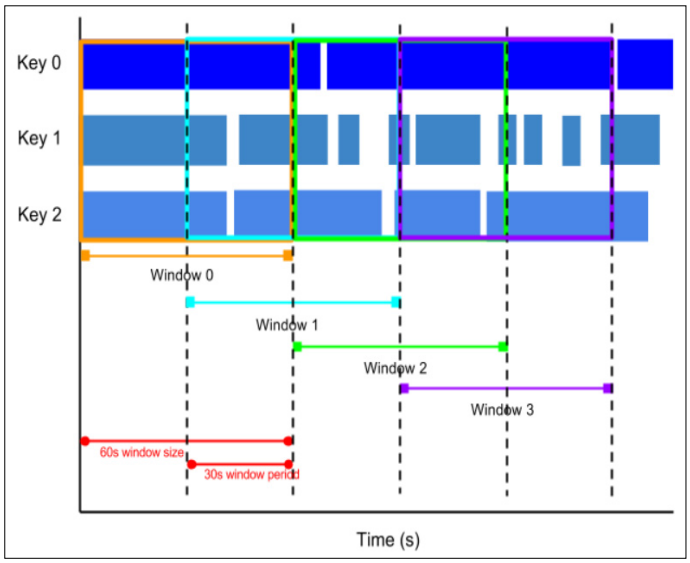

Fig. 1. Distribution of Elements Across Overlapping Hopping Windows [5]

Schema evolution is inevitable in long-lived monitoring systems: a new sensor appears, a field format changes, or a numeric surrogate key is added to string labels. BigQuery allows adding a nullable column online and relaxing a REQUIRED→NULLABLE mode without table rewriting; similar changes are already supported in Pub/Sub schemas, enabling zero downtime even at hundreds of thousands of RPS [6]. More critical changes, such as dropping or renaming a column, are implemented in a side-by-side schema with bidirectional streaming until consumers have migrated.

Likewise, to update a streaming pipeline without downtime, one uses parallel deployment: a new streaming job with updated code is created and run alongside the existing one, using the same windowing strategy. The existing pipeline continues until its watermark exceeds the timestamp

of the earliest complete window processed by the new pipeline. After this, the old pipeline is either drained or canceled, and the new one takes over entirely for processing. Both pipelines write results to different stores, allowing downstream systems, via an abstraction such as a database view, to merge data and eliminate duplicates from the period of simultaneous processing, as shown in Figure 2.

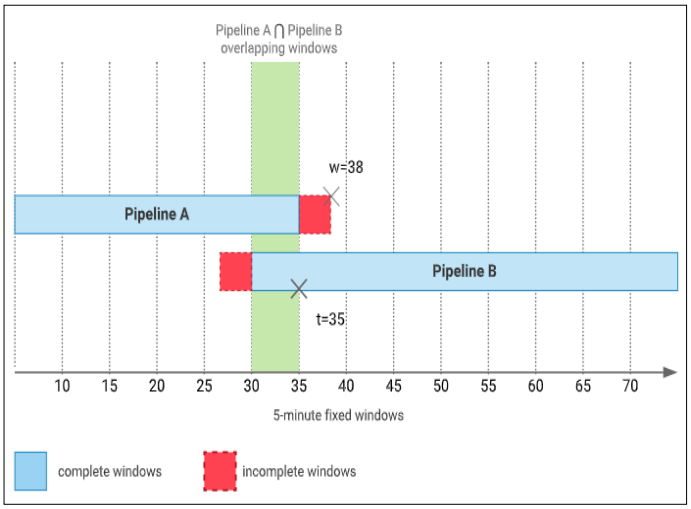

Fig. 2. Parallel Streaming Pipeline Update with Overlapping Five-Minute Tumbling Windows [6]

Finally, to ensure that analytical data marts, machine learning, and regulatory reporting receive only deltas rather than a complete dump of yesterday's data, Change Data Capture is organized on top of the base tables [7]. The Storage Write API supports exactly-once upsert semantics and produces a change log, which is streamed into a Pub/Sub topic. This log is then either materialized into BigQuery via a native CDC subscription or distributed to external services requiring reactive data synchronization [8]. This architectural pattern closes the loop from the immutable raw store to reactive consumers without violating the latency guarantees described in the Introduction.

Choosing the partition granularity suitable for a specific HF-HC dataset begins by matching the analytics window to the data arrival rate. In BigQuery, a table may be divided by hours or days, or rely on the pseudo-column _PARTITIONTIME for ingestion-time partitioning [9]. Hourly partitions are appropriate when query windows rarely exceed twenty-four hours and the daily volume exceeds one billion rows: in this case, the bytes scanned in a typical last-hour query decrease almost proportionally to the number of hourly segments. For retrospective reports spanning months, daily partitioning is more efficient, ensuring that metadata does not grow faster than valid data. Ingestion-time partitioning is convenient when event_ts often lags, as it simplifies the pipeline by avoiding writes to older dates.

After selecting the partitioning scheme, the clustering order is defined. BigQuery sorts blocks on up to four columns, and its engine applies block pruning to exclude irrelevant blocks before actual reading [10]. In practice, the first position is reserved for a low-cardinality filter column (for example, event_type), followed by truly high-cardinality keys such as device_id or user_id. The order is critical: if the WHERE clause filters only on the second column, the savings will be much lower than when filtering on the first column.

To ensure these advantages persist even under continuous streaming of tens of gigabytes per hour, BigQuery runs automatic reclustering. This background process re-sorts new deltas and merges them into the base block without user intervention, eliminating the scheduled VACUUM operations typical of classic data warehouses [10].

The low query cost mentioned in the previous section is achieved not only by proper table partitioning but also by the storage format itself. Inside BigQuery, data is stored in Capacitor: before writing, each column is dictionary-encoded, then run-length compressed, and finally delta-encoded for numeric and timestamp values. This cascade allows repeated string values to be stored as just a few bytes of references, and monotonic timestamps as increments, which in real clusters yields compression ratios multiple times better than file formats such as Parquet or ORC.

If, instead of long string identifiers, surrogate keys of type INT64 are introduced in advance, the dictionary for that column shrinks proportionally to the average length of the original string. On IoT streams, converting to numeric surrogate keys reduces the column size because Capacitor now sees only small numbers, which are ideal for delta encoding. Experience shows that the benefit of this operation pays for the computation of keys at ingestion time through lower storage and subsequent scan costs, the latter of which depend directly on the physical table size [11].

As streaming continues, deltas accumulate over the base block; to prevent fragmentation from growing, BigQuery automatically triggers reclustering, re-sorting fresh fragments until the new baseline reaches a certain threshold. The user only needs to set the storage_partition_period parameter, which specifies the minimum age of fragments eligible for compaction: lowering this value accelerates consolidation but increases instantaneous slot consumption. By combining these techniques, one can precisely calculate savings. For an initial log of 1 TiB per day, the daily storage cost in us-central1 is $23,552 (active logical rate) [12].

Compressed columns also accelerate queries themselves. For the scenario, the last state of an object, it is sufficient to scan only the latest partition and select the earliest element in time within each group. The operator ROW_NUMBER() OVER (PARTITION BY device_id ORDER BY event_ts DESC) together with QUALIFY rn = 1 achieves this without joins, since the filter on _PARTITIONTIME excludes the vast majority of fragments before reading. When analytics require continuous sliding calculations, the new operator TIMELINE_GAP_FILL fills gaps in the series and aggregates values directly over the necessary windows.

For delta analytics, a simple technique is convenient: instead of heavy self-joins on key equality and nearest timestamp,

use LAG() or LEAD(), and when only the last element in a window matters, ARRAY_AGG(value ORDER BY event_ts DESC LIMIT 1) works better. Python engineers can now perform the same calculations using BigQuery DataFrames v2, where enabled partial ordering mode ensures that the engine orders only the used columns, preserving lazy evaluation of the rest [13].

After execution, total_slot_ms and total_bytes_processed remain in INFORMATION_SCHEMA.JOBS from which they can be extracted into a dashboard and used to build a cost trend. This measure-change-measure cycle allows empirical confirmation that the described techniques reduce costs rather than merely promising it on paper [14].

A reliable pipeline for HF-HC data relies on several rules, each of which eliminates unnecessary bytes or milliseconds before they can become costly. First, all queries to partitioned tables must include a constant filter on the partition column; the require partition filter option forces errant dashboards to fail immediately, thereby disciplining developers [15]. Second, ingest lag is monitored via the uploaded_bytes_count metric and a strict SLO. If the streaming delay exceeds a specified threshold, the pipeline automatically switches from Pub/Sub to direct Storage Write API. Third, before performing any JOIN, both sides must be aggregated or reduced to the required time window; otherwise, the shuffle will consume slots and budget [11]. Fourth, each fact table may have no more than four clustering columns, and only in the order in which they most frequently appear in WHERE clauses; the built-in recommender will suggest the optimal depth after one week of operation. Fifth, new fields are added as nullable, while old fields are deprecated through a view layer, allowing the schema to evolve without downtime. Sixth, surrogate keys replace multibyte string identifiers already in the stream worker, because they compress best in Capacitor, reducing storage and scan costs. Seventh, storage_partition_period is set to a minimum interval to prevent the compaction process from competing with live streaming. Eighth, all ETL jobs EXPLAIN and store total_bytes_processed in a separate control table: if volume grows by more than a predefined percentage under the same business logic, an alert opens an incident. Ninth, using partition and cluster recommendations at least once per quarter eliminates the need for manual configuration review, as the algorithm already accounts for changed query patterns [16]. Tenth, stream and batch costs are compared monthly to on-demand scan prices. Sometimes, it is more cost-effective to recompute a data mart hourly rather than maintaining it in real-time, especially considering that the Write API is billed separately from compute slots [12].

Typical anti-patterns are derived from these same rules, but with opposite signs. The absence of a partition filter is instantly diagnosed with the message Cannot query over table without a filter; If an administrator disables this requirement, a sudden jump in total_bytes_processed with an unchanged query count becomes an indirect indicator. A second signal is SELECT * on a wide schema: the plan will show that the entire column is scanned, even though the report uses only a few fields. This is remedied by refactoring queries and creating views with narrow select lists. The third trap is unsystematic clustering: if INFORMATION_SCHEMA.COLUMN_FIELD_PATHS indicates that filtering on the second or third clustering column is triggered less frequently than a certain percentage of queries, resulting in wasted sorting and increased write latency. Fourth, using the legacy API for streaming under stable load: quota logs from jobs_by_project quickly reveal that ingestion speed is insufficient and slot overuse is leading to retries. Fifth, arbitrary JavaScript UDFs: their share of time in the EXPLAIN profile appears as UserDefinedFunctionStage, indicating that business logic should be offloaded to Dataflow or DataFrames v2. Finally, suppose a cross-region query regularly scans tens of terabytes, and latency increases by hundreds of milliseconds. In that case, this is a clear sign that data or compute must be consolidated in a single location. BigQuery guarantees four-fold replication within a region but does not hide network delays between areas. When these symptoms are identified and resolved, the pipeline returns to its SLAs, and budget savings are confirmed by metrics collected from the design phase onward. Figure 3 illustrates common anti-patterns in data-pipeline optimization, including the absence of partition filtering, unsystematic clustering, legacy API usage, excessive JavaScript UDFs, and cross-region queries, which lead to inefficient scanning and increased costs.

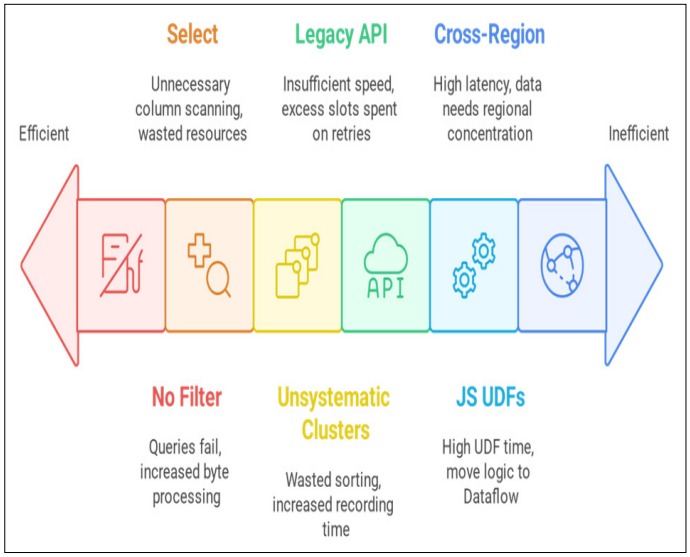

Fig. 3. Patterns Affecting the Efficiency of Data Pipeline Optimization (compiled by author)

As a result of applying the rules described above—from mandatory partition filtering and streaming-lag control to prudent choice of clustering columns, schema evolution via nullable fields, and the replacement of heavy string identifiers with surrogate keys—the HF-HC data pipeline achieves an optimal balance between performance and cost. Timely aggregation before JOINs, adjustment of partitioning periods, continuous monitoring of scanned volumes, and regular updating of partition and cluster recommendations

prevent resource overconsumption and ensure SLAs remain within strict bounds. Understanding and eliminating common anti-patterns—from the absence of partition filtering and unsystematic clustering to the use of legacy streaming APIs, JavaScript UDFs, and excessive cross-region queries—returns the system to a zone of stable operation and cost efficiency.

CONCLUSION

The solutions described in the article demonstrate that combining the architectural features of Google BigQuery with well-considered operational practices enables the reliable processing of HF-HC-TS streams at a rate of millions of records per second. First, using an append-only model, together with the Capacitor format, provides linear ingest scalability and minimal storage costs. Delta encoding and dictionary compression keep table sizes under control, and avoiding update-in-place eliminates locks. Second, introducing two timestamps (event_ts and ingest_ts) together with watermarks in Apache Beam/Dataflow pipelines ensures the correct handling of late events without requiring the complete rebuilding of historical partitions, thereby preserving consistency during analytics within a trust window.

Key to low query latency is declarative time-based partitioning and multi-level clustering: choosing appropriately among hourly, daily, and ingestion-time partitions, as well as ordering low- and high-cardinality keys (event_type, device_id, user_id) in sequence, allows block pruning to exclude unnecessary data before reading. Background reclustering operations and automatic delta merges eliminate the need for manual VACUUM procedures, maintaining efficiency even under a continuous data stream. Schema evolution is implemented by adding nullable fields and using a side-by-side strategy when removing columns, eliminating downtime even at hundreds of thousands of RPS.

For analytical data marts and machine learning on base tables, CDC based on the Storage Write API is implemented. The change log, with exactly-once semantics, is either materialized in BigQuery or sent to Pub/Sub, ensuring that consumers receive only deltas. Query optimization is achieved using the ROW_NUMBER() OVER and TIMELINE_GAP_FILL operators, employing LAG()/LEAD() or ARRAY_AGG instead of heavy joins, and BigQuery DataFrames v2 with partial ordering, which eliminates unnecessary shuffling and preserves lazy evaluation. Cost control involves a two-phase profiling approach (EXPLAIN plus INFORMATION_SCHEMA.JOBS metrics), enabling empirical verification of the impact of changes on total bytes processed.

A set of ten rules—from mandatory partition filtering and ingest lag monitoring to judicious column selection for clustering and replacing string identifiers with surrogate keys—creates a stable balance between performance and cost. Eliminating anti-patterns (absence of partition filter, unsystematic clustering, legacy API usage, JavaScript UDFs, and cross-region queries) restores performance and cost-effectiveness. Ultimately, the methods described ensure SLA levels for latency and budget when handling data streams of billions of events per day.

REFERENCES

1. "Internet of Things and Data Placement," Dell Technologies. https://infohub.delltechnologies.com/en-US/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/ (accessed May 01, 2025).
2. M. Faraz, "What is Google BigQuery Streaming Insert and Its Working?" Hevo Data, Oct. 25, 2024. https://hevodata.com/learn/bigquery-streaming-insert/ (accessed May 02, 2025).
3. "Estimate and control costs," Google Cloud. https://cloud.google.com/bigquery/docs/best-practices-costs (accessed May 03, 2025).
4. S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time Series Management Systems: A Survey," IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 11, pp. 2581–2600, Nov. 2017, doi: https://doi.org/10.1109/tkde.2017.2740932.
5. "Streaming pipelines," Google Cloud. https://cloud.google.com/dataflow/docs/concepts/streaming-pipelines (accessed May 04, 2025).
6. "Upgrade a streaming pipeline," Google Cloud. https://cloud.google.com/dataflow/docs/guides/upgrade-guide (accessed May 05, 2025).
7. "Stream table updates with change data capture," Google Cloud. https://cloud.google.com/bigquery/docs/change-data-capture (accessed May 07, 2025).
8. "BigQuery subscriptions," Google Cloud. https://cloud.google.com/pubsub/docs/bigquery (accessed May 07, 2025).
9. "Introduction to partitioned tables," Google Cloud. https://cloud.google.com/bigquery/docs/partitioned-tables (accessed May 08, 2025).
10. "Introduction to clustered tables," Google Cloud. https://cloud.google.com/bigquery/docs/clustered-tables (accessed May 09, 2025).
11. "Optimize query computation," Google Cloud. https://cloud.google.com/bigquery/docs/best-practices-performance-compute (accessed May 15, 2025).
12. "BigQuery Pricing," Google Cloud. https://cloud.google.com/bigquery/pricing (accessed May 17, 2025).
13. "Use BigQuery DataFrames," Google Cloud. https://cloud.google.com/bigquery/docs/use-bigquery-dataframes#version-2 (accessed May 19, 2025).

14. "JOBS view," Google Cloud. https://cloud.google.com/bigquery/docs/information-schema-jobs (accessed May 20, 2025).
15. "Query partitioned tables," Google Cloud. https://cloud.google.com/bigquery/docs/querying-partitioned-tables (accessed Jun. 20, 2025).
16. "Manage partition and cluster recommendations," Google Cloud—https://cloud.google.com/bigquery/docs/manage-partition-cluster-recommendations (accessed May 22, 2025).