Boston University

METCS777

Big Data Analytics

Term Paper Report

Performance and Visualization of Big Data in Cloud Environments:

A Comparative Study of Spark SQL (Databricks) and BigQuery

Team:
Tanvi Thopte: U35489362
Mehul Bisht: U93099406

1.Abstract

This term paper explores the performance and scalability of two major cloud-based big data platforms Google BigQuery and Databricks (Spark SQL). Using a large e-commerce dataset, the study focuses on data cleaning, analytical querying, trend analysis, and visualization to understand how each system handles large-scale data processing. The paper aims to evaluate their architectures, techniques, and practical applications in real-world analytics workflows, highlighting the role of cloud technologies in transforming raw data into meaningful business insights.

TABLE OF CONTENTS

2. INTRODUCTION

Our term paper compares Google BigQuery and Databricks (Spark SQL) for big data analytics,both the platforms are two of the most used widely used cloud based big analytics platforms.
It uses an e-commerce dataset and evaluates it for performance and visualization using SQL queries for trend detection, user behavior, anomaly detection, and performance benchmarking.

The focus was to evaluate both platforms in terms of:

- Query execution time and efficiency
- Scalability under growing data volumes
- Data processing models (serverless vs cluster-based)
- Integration with visualization tools for business intelligence

BigQuery is basically a fully managed data platform that helps manage and analyze data with built-in features like machine learning, search analysis, and business intelligence.

- It has a serverless architecture and lets you use SQL and Python to analyze data with zero infrastructure management. It works with both structured and unstructured data .
- It is quite scalable and allows distributed analysis that makes huge amounts of data to be queried in seconds/minutes.
- BigQuery's architecture consists of two parts: a storage layer that ingests, stores, and optimizes data and a compute layer that provides analytics capabilities. These compute and storage layers efficiently operate independently of each other thanks to Google's petabit-scale network that enables the necessary communication between them.

On the other hand, we have Databricks, which is a cloud-based platform as well for managing and analyzing large datasets using Apache Spark open-source big data processing engine.

- It is commonly used for tasks such as data preparation, real-time analysis, and machine learning. It was designed to make working with big data easier and efficiently by providing tools and services for data preparation, real time analysis and machine learning. It provides a unified workspace helping communication and collaboration easier among teams.
- It is Scalable and flexible with integrated tools and services, making it a good

choice for organizations that need to process data at different scales, or that have complex data pipelines.

- To make analytical insights more interpretable and impactful, we integrated data visualization tools alongside our queries.
- In BigQuery, results were connected to Google Looker Studio to build interactive dashboards.
- In Databricks, we used built-in visualization tools (Plotly and Matplotlib) inside notebooks to mirror the same visuals ensuring cross-platform comparison in both speed and graphical interpretation.

For visualization, we are using Looker studio (Google Data Studio) which is a free platform from Google that allows users to create interactive reports and dashboards tightly integrated with other Google tools such as Google Analytics, Google Sheets, BigQuery, and Google Ads. Developers highlight the following key features of the platform: the ability to connect to many data sources, including SQL databases via Google BigQuery or third-party connectors, a simple and intuitive interface that allows you to easily create reports for marketing, SEO, and data analytics,. It allows you to integrate data from other sources, implement all analyst tasks, and create simple visualizations (graphs, tables, filters, key performance indicators). On the other hand, we are using pythons plotly for visualization with data bricks. Although there are several programming languages available for this purpose, the present study focuses only on data visualization libraries commonly used in Python. This programming language provides rich libraries for data visualization, and we focused on Matplotlib and Seaborn, Plotly libraries in this project. Using these libraries, we can generate various charts such as bar charts, histograms, and scatter plots.

For our final results:
We have used the e-commerce dataset to compare the overall query performance, scalability, and efficiency of Databricks Spark SQL and Google BigQuery across several analytical tasks.
This involves examining the ways in which both platforms manage anomaly detection, user-level monetization, data cleaning, aggregation, and full scalability evaluation.
In addition to execution speed, we also accessed how well each system supports advanced analytics workflows like trend and outlier detection, optimizes resource usage, and scales with increasing data volumes. Tics workflows such as trend and outlier detection.

3. TOPIC IMPORTANCE

Data is the most valuable resource for an organization in the present digital economy. To gain business insights, information collected through each click, transaction, and customer interaction needs to be processed and examined. However, traditional tools are no longer able to manage the scale, speed, or complexity of analysis required as data sizes increase from gigabytes to terabytes and even petabytes. The scale and speed can no longer be effectively handled by conventional on-premises databases and processing tools. They have issues with slow query execution, storage limitations, data integration, and expensive maintenance. In order to instantly transform raw data into actionable insights, modern businesses now require real-time analytics, scalable infrastructure, and automated resource management.

Businesses depend on big data analytics to obtain timely, actionable insights as data continues to grow exponentially.

- Choosing the appropriate platform affects:
- Speed of insights (quicker choices)
- Costs of the cloud (pay-per-query versus cluster uptime)
- Scaling ease (serverless vs. manual tuning)
- Integration with dashboards and AI/ML tools

Because of this, cloud-based big data platforms like Google BigQuery and Databricks have become more popular. These platforms provide distributed processing, elastic scalability, and smooth integration with machine learning and visualization tools. Businesses can easily handle large datasets with the help of these technologies, and they can get analytical, directive, and predictive insights that improve customer satisfaction, competitiveness, and operational efficiency.

BigQuery is a columnar, fully serverless data warehouse designed for visual dashboards and interactive, ad hoc analysis.

Databricks is a cluster-based, Spark-powered unified analytics platform made for large-scale data transformations, ML workflows, and intricate pipelines.

Determining which environment is more effective for business intelligence vs. data engineering workflows requires an understanding.

4. WORKFLOW:

4.1 Environment Setup

Component	BigQuery	Databricks
Cluster	BigQuery Free Tier (serverless, managed by Google)	Oe driver node and two worker nodes with 15 GB RAM total
Dataset	Stored in a Google Cloud Storage bucket linked to BigQuery	9GB dataset split across multiple CSV files uploaded.
Libraries Tools	BigQuery Web Console using SQL	PySpark and Plotly for data processing and visualization
File Paths	glassy-ripsaw-4732233.ecommerce.events_clean	Volumes/ecom in the Databricks workspace
Visualization	Built-in BigQuery Charts and Data Studio (Looker)	Plotly and Matplotlib charts generated through notebooks

4.2 Process

BigQuery Workflow

- Set up a cloud environment in Google Cloud Console and created a dataset in BigQuery.
- Uploaded the e-commerce transactions dataset containing millions of purchase records.
- Cleaned and prepared data using SQL normalized types, removed nulls, and partitioned by date.
- Ran exploratory queries (unique users, missing values, price stats) and analytical queries (daily revenue, category–brand analysis, user spending, session behaviour).
- Implemented z-score–based anomaly detection using a 7-day moving average to find unusual sales spikes.
- Recorded execution metrics — query time, bytes processed, and slot usage.
- Connected to Looker Studio for visualization, built dashboards showing trends, heatmaps, and anomalies.

Databricks Workflow

- Created a Databricks workspace and launched a Spark cluster with multiple worker nodes.
- Imported the same e-commerce dataset into DBFS and converted it to a Delta Lake table for faster access.
- Cleaned and transformed data using Spark SQL dropped nulls, verified schema, and validated key fields.
- Replicated the same analyses (revenue trends, user-level metrics, anomaly detection) using Spark SQL.
- Observed Spark's lazy evaluation and caching — first runs were slower, but repeated queries became faster after in-memory caching.
- Recorded job execution times, task stages, and Photon Engine utilization from the Databricks UI.
- Created visualizations in Databricks notebooks using Plotly and Matplotlib, mirroring Looker dashboards for consistency.

5. OUTCOMES/FINDINGS

Big Query:

BigQuery turned out to be a great option for business dashboards and on-demand analytics. Because of its minimal setup and maintenance requirements, Google's serverless architecture allowed queries to be run rapidly. Real-time insight visualization was made simple by the platform's smooth integration with Looker Studio, which converted SQL outputs into dynamic charts and trend dashboards.

It works particularly well for BI workflows and real-time reporting, where scalability, simplicity, and speed are more important than precise hardware or caching control. BigQuery essentially feels effortless: you write queries, upload data, and receive results nearly instantly.

Databricks:

Databricks, which is powered by Apache Spark, exceeded BigQuery in demanding and repetitive workloads once the data was cached in memory. Its in-memory processing greatly reduced query time after the initial load, which made it ideal for machine learning experiments, iterative queries, and large-scale analytics.

By supporting Python, R, and Scala in addition to SQL, it also provided data engineers with more options, making it easier to combine data processing, model training, and visualization in a single collaborative environment.

All things considered, Databricks is ideal for continuous analytical pipelines, ETL procedures, and artificial intelligence-based workflows that demand total control over computational resources.

Scalability & Insights:

As the volume of data increased, both platforms scaled successfully, demonstrating how easily large datasets can be handled by recent cloud architectures.

While BigQuery automatically scaled its resources through distributed slots that executed queries in parallel without user intervention, Databricks used Spark's distributed memory and caching to speed up repeated analyses over time.

Together, they show how visualization tools and big data platforms can convert unstructured massive amounts of data into insightful, interactive analytics. These platforms show the power of cloud-based analytics when paired with visual storytelling tools such as Looker Studio or Databricks' Plotly visualizations, whether the aim is to track trends, identify revenue anomalies, or figure out customer behaviour.

6. TECHNOLOGY

6.1 History

Google BigQuery

Google launched BigQuery in 2011 as a fully managed, serverless enterprise data warehouse built on Google's internal Dremel technology. Dremel was engineered for extremely fast SQL queries across petabyte-scale datasets using a distributed, tree-based execution model. BigQuery modernized this concept by separating compute from storage and enabling organizations to run analytical SQL without provisioning or managing any clusters. Over the years, BigQuery has added features such as BigQuery ML, BI Engine, data federation, and integrated dashboarding tools, making it one of the leading cloud analytics solutions.

Databricks (Spark SQL)

Databricks was founded in 2013 by the creators of Apache Spark at UC Berkeley. It was designed to simplify distributed computing by providing a managed, scalable

environment for Spark-based data engineering, ETL, and machine learning workflows. Through innovations like Delta Lake, Photon engine, and collaborative notebooks, Databricks evolved into a unified "Lakehouse" platform capable of supporting SQL analytics, batch processing, streaming, and advanced ML pipelines. Today, it is widely used for large-scale computation, data science, and real-time analytics.

6.2 Background knowledge

One needs to understand how systems handle data in the background, it is more than just writing queries, both the platforms handle huge amount of data and gives insights and outputs within seconds because of their strong distributed computing foundations.

Foundations:

- Distributed Computing:

  Both platforms work by splitting data across many computers which are called nodes/workers. Instead of one system doing all the work, each worker handles a part of data all done parallel.

  It is like a team working together to put something together, they all process huge amounts of data in parallel and then gather it together so huge amount of information is processed in seconds, works faster than a single machine.

- Columnar Storage:

  BigQuery stores everything in a column format, which is that it reads only the columns one needs for processing and not the entire table.

  This makes operations like sum,avg,group by very efficient, since it skips unnecessary data and focuses only on what each query requires.

- Massive Parallel Processing:

  BigQuery automatically divides each query into multiple small tasks and runs them on hundreds of parallel slots; they are virtual CPUs. There is no managing of server required, BigQuery automatically scales up/down depending on the size of the query, thus allowing it to finish large tasks in very less time.

- Apache Spark Execution:

  Databricks runs on Apache spark which works a little different; it builds a plan of what is to be done then executes it only when an action (show, count) is triggered.

It is called lazy evaluation; Spark optimizes the entire job before running it, saving time and resources.

- Delta lake:

  Databricks also uses Delta Lake which adds reliability to big data storage; it ensures data consistency, keeps track of data version history and makes it easier to handle schema changes overtime.

  It is good where analytical workflows are constantly updating or modifying.

- Serverless vs. Cluster-Based Execution:

  BigQuery and Databricks differ the most in this case:

  BigQuery is completely serverless you will have to just write SQL queries, and it handles everything.

  Databricks on the other hand give you more control; it lets you set up your own clusters and configure everything. That flexibility is powerful for custom workloads but requires more management and cost tuning.

6.3 Use cases/applications

Both platforms are made to handle massive datasets but have their own use cases:

BigQuery:

- Used for building dashboards and running quick reports.
- Integrates easily with Looker studio, Power BI and tableau making it ideal for marketing analytics, sales tracking and executive reporting.
- Many companies use it to monitor live streaming data like website traffic, ecommerce transactions or iot sensors. For example, a retail company can track hourly sales trends or detect sudden changes in customer demand.
- It acts as a central data warehouse where data from multiple sources is stored and analyzed together; it supports data ingestion allowing end to end data pipelines without managing infrastructure.

Databricks:

- Databricks **cleans, transforms, and prepares huge raw datasets** for analytics and machine learning, allowing automation of data workflows and can process structured, semi-structured, or unstructured data efficiently using Apache Spark.
- It allows automation of data workflows and can process structured, semi-

structured, or unstructured data efficiently using Apache Spark.

- It supports multiple frameworks of T**ensorFlow, PyTorch, and Scikit-learn, which** train and deploy ML models at a scale.
- It also includes MLFlow which helps track experiments and manage model lifecycles.

7. Technical details:

7.1 Architecture/Design

BigQuery Architecture

- Serverless Structure:

  BigQuery is managed by Google, users need not set up or manage servers. One can simply load data and run SQL queries

- Storage Layer (Colossus):

  The data is stored in a distributed manner which is fault tolerant (called colossus), which is optimized for durability and quick access.
  Compute Layer (Dremel Engine):

- Compute Layer (Dremel Engine):
  Queries are ran using Google's Dremel technology, which splits queries into multiple smaller tasks and processed in parallel across multiple machines.

- Columnar Storage (Capacitor Format):
  Data is stored column-wise, only the required columns are read during a query saving time and reducing cost.

- Automatic Scaling:
  BigQuery automatically allocates "slots" (virtual CPUs) based on workload scaling up for large queries and down for smaller ones.

- Integration with BI Tools:
  It connects to Looker Studio and other visualization platforms for dashboard creation.

Databricks Architecture

- Cluster-Based Design:

  It runs on top of Apache Spark clusters and is configurable (size, CPU,

memory). Each cluster has its driver node and their worker nodes that help in parallel processing.

- Storage Layer (Delta Lake):
  Uses Delta Lake, that supports ACID transactions, schema validation, and time travel (data versioning).
- Compute Engine (Apache Spark):

  Spark divides the work into small tasks which are given across clusters.
  It uses the Directed Acyclic Graph (DAG) execution model which ensures efficient parallelism and fault recovery.

- Photon Engine:
  A high-performance query engine built by Databricks,it accelerates SQL workloads improves latency and throughput significantly.
- Integration:

  It connects easily to visualisation tools like looker studio, tableau and Power BI also the MLFrameworks supports Python, SQL, R and Scala.
- Interactive Workspace:
  The Databricks Notebook interface allows users to collaborate in real time, writing code, visualising data etc.

7.2 Techniques/Approaches

Technique	BigQuery	Databricks (Spark SQL)
Data Partitioning	Automatic partitioning by date or field for faster queries.	Manual or automatic partitioning by columns or ranges.
Data Caching	No caching by default — fresh scan every time.	Supports caching in memory for faster repeated queries.
Query Optimization	Uses Dremel query planner and vectorized execution.	Uses Catalyst Optimizer for DAG optimization.
Scalability	Serverless auto-scaling via virtual “slots.”	Scales by adding more worker nodes to clusters.

Fault Tolerance	Automatics retry and job recovery handled by Google.	Built-in fault recovery using Spark's lineage tracking.
Machine Learning Integration	Integrates with BigQuery ML and Vertex AI.	Native MLflow and Spark MLlib integration.
Visualization	Native Looker Studio support.	Built-in charts using Plotly, Matplotlib, and Power BI connectors.

7.3 Interesting findings

We found a number of significant speed, scalability, and efficiency-related insights after executing queries on Google BigQuery and Databricks (Spark SQL) using our large e-commerce dataset.

1. Performance and Execution of Queries

- Google's serverless "slot-based" parallel processing model allowed BigQuery to execute the majority of ad hoc analytical queries extremely quickly, including some in less than a second.
- Because Databricks had to initialize the Spark cluster and read data from storage, the initial run took a little longer.However, once the data was cached in memory, Databricks consistently outperformed BigQuery in repeated queries, with speeds improving by nearly 40–60%.
- This shows how caching plays a key role in Spark's performance optimization.

2. Scalability as Data Increases

- Using the exact same aggregation query on 1-, 3-, and 7-month data subsets, we evaluated scalability.
- BigQuery used distributed slots to automatically scale its compute resources; query time only slightly increased (by about 2 to 2.5 seconds) as dataset size increased.
- By using cached partitions and parallelizing computations across worker nodes, Databricks effectively handled larger datasets after the initial load.
- Both platforms were very scalable, but Databricks required manual cluster tuning for optimal performance, whereas BigQuery required no configuration.

3.Anomaly Detection and Trend Analysis

- Using the z-score method ($|z| \geq 3$), both tools properly identified revenue spikes and anomalies.

- The identical spikes showed up on both platforms, indicating accurate analytical reasoning and data consistency.
- Databricks used Plotly to more dynamically visualize these anomalies, and BigQuery seamlessly integrated with Looker Studio dashboards to allow the sharing and interpretation of insights.

4. Efficiency of Visualization

- Looker Studio (in combination with BigQuery) offered effortless real-time dashboarding, perfect for recording KPIs or business presentations.
- Model integration, multiple language plots (Python, SQL, and R), and greater flexibility were made possible by Databricks visualizations, despite their complexity.
- Together, they demonstrated how visualization tools can simplify and provide valuable insights from complex analytics.

5. Cost Perspective and Resource Utilization

- BigQuery is cost-effective for irregular analysis because it charges according to the amount of data processed per query.
- Databricks charges for cluster runtime, which provides more flexibility for continuous processing but can get expensive if left running.
- In general, Databricks works well for groups handling daily pipelines, ETL, or ML workloads, while BigQuery works well for teams requiring rapid, one-time analyses.

6. Pros and Cons

Aspect	Google BigQuery	Databricks (Spark SQL)
Setup & Ease of Use	Fully serverless – no setup, automatic scaling	Requires manual cluster setup and tuning
Performance (First Run)	Extremely fast for one-time queries	Slightly slower due to cluster startup
Performance (Repeated Runs)	Re-reads data every time (no caching)	Much faster after caching data in memory
Scalability	Auto-scales instantly via distributed slots	Scales efficiently with manual cluster adjustment
Cost Efficiency	Pay only for data scanned – ideal for ad-hoc queries	Pay for cluster uptime – better for continuous workloads

Data Management	Simple SQL-based querying, limited ML integration	Supports Delta Lake, MLflow, and advanced pipelines
Visualization	Seamless integration with Looker Studio	Custom visualizations with Plotly, Matplotlib, etc.
Best For	Analysts & BI teams	Engineers & Data Scientists

8. Recommendations

Choosing a Platform Depending on the Use Case

For Ad-hoc Analytics and Business Intelligence: Use Google BigQuery if dashboarding, real-time analytics, and quick insights are your main goals.

With nearly immediate execution, no setup, and no tuning, this provides an entire serverless service.For analysts and business teams who require speed, ease of use, and data that is ready for visualization with little engineering overhead, BigQuery is perfect.
Select Databricks (Spark SQL) for workloads involving data transformations, iterative analysis, and model training if you're interested in data engineering, machine learning, and repeated processing.
Databricks provides outstanding performance and flexibility following data is cached, especially for pipelines that include AI, ETL, and predictive analytics.

Employ a Hybrid Architecture

The best of both sides can be reached with a combined approach:
Use Spark's distributed computing to clean, combine, and enhance raw data with Databricks. Giving cleaned data to BigQuery for fast, serverless querying, visualization, and business intelligence.

The hybrid setup maintains a balance between usability, performance, and cost, and it reflects what many current companies are doing.Make Performance Your Top Priority

Management of Costs and Resources

For frequent analytics, BigQuery's pay-per-query model is very economical, but it could become costly for continuous querying.

Teams with regular or continuous analytical workloads will do better for Databricks' pay-per-cluster model.
To properly control expenses, pick cluster sizes carefully and turn them off when not in use.

Alignment of Skills

Businesses with strong analyst teams that focus on SQL and dashboarding will benefit most from BigQuery.
The flexibility of Databricks will be especially helpful for teams with previous expertise in data engineering, Python/R development, or machine learning.
Databricks excels at complex pipelines, iterative tasks, and model training, while BigQuery excels at reporting, instant analytics, and ease of use.
A mixed workflow that combines Databricks for computation and BigQuery for visualization offers the ideal balance of power and simplicity.

9. Conclusions

The comparative analysis that we did using google BigQuery and Databricks showed that both platforms are powerful,scalable and capable of handling huge datasets but achive this through very different approaches.

Key Technical Insights

- BigQuery operates on a serverless, fully managed model, automatically scaling resources based on demand.It excels at running ad-hoc queries on large datasets without configuration or maintenance. Its columnar storage (Capacitor) and Massively Parallel Processing (MPP) architecture allow it to deliver high performance consistently, even for complex aggregations.
- Databricks, on the other hand, uses a cluster-based architecture built on Apache Spark.
  It requires initial setup but offers in-memory computation, caching, and full control over resources. These capabilities make it superior for repeated or iterative analysis, data engineering, and machine learning applications.

Performance and Scalability

Both the platforms excelled at scalability during our tests,maintining almost similar and constant query times even when the data size kept changing.

- BigQuery consistently delivered faster results for *one-time queries* due to its dynamic slot allocation and optimized query planner.
- Databricks caught up and often surpassed BigQuery in *repeated runs* because of Spark's ability to cache data in memory, drastically reducing subsequent execution time.

Usability and Visualization

- Bigquery with looker studio allowed easy visualization of analytical results making it friendly for businesses and decision makers
- Databricks notebook interface made it better suited for technical exploration, predictive modeling, and deeper statistical or ML-driven analysis.

Cost and Practical Considerations

- BigQuery's pay-per-query model is cost-effective for organizations that run analytics intermittently or rely heavily on dashboards.
- Databricks' cluster-based pricing can become expensive if clusters remain active unnecessarily, but it offers more value for continuous, high-volume analytical operations.
- Cost efficiency depends on query frequency, data size, and type of workload — so the ideal choice varies by organization

Final Conclusion:

- BigQuery: Best for speed, simplicity, and visualization. It's the go-to for analysts and BI teams.
- Databricks: Best for flexibility, complex pipelines, and ML integration. It's ideal for engineers and data scientists.
- Both are highly complementary, and when combined, they can create an end-to-end data ecosystem from ingestion and processing (Databricks) to fast querying and dashboarding (BigQuery).

This study highlights that there is no single “better” platform instead, the right choice depends on business needs and data maturity.
BigQuery provides speed and simplicity for analytics, while Databricks delivers control and depth for data engineering and advanced analytics.
Together, they represent the future of scalable, cloud-native big data ecosystems, where performance, flexibility, and visualization come together to turn massive datasets into actionable intelligence.

10. References for BigQuery and Databricks

Google BigQuery References

- https://cloud.google.com/bigquery/docs
- https://cloud.google.com/bigquery
- https://cloud.google.com/blog/products/data-analytics/new-bigquery-partitioning-and-clustering-recommendations
- https://medium.com/google-cloud/partitioning-vs-clustering-in-bigquery-9552ae2d702a
- https://www.innablr.com.au/blog/mastering-bigquery-optimisation-and-best-practices
- https://estuary.dev/blog/google-bigquery-etl/

Databricks / Delta Lake References

- https://docs.databricks.com/aws/en/delta/
- https://learn.microsoft.com/en-us/azure/databricks/delta/
- https://delta.io/
- https://docs.databricks.com/gcp/en/reference/delta-lake
- https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf
- https://docs.databricks.com/aws/en/lakehouse-architecture/reference

11. Appendix A: Code Samples

GITHUB LINK: https://github.com/Mehul6/METCS777-TermPaper-Team21

10.1 Codes:

BigQuery:

Final TermP Queries

```sql
-- Names: Tanvi Thopte, Mehul Bisht
-- Cleaning + analyzing ecommerce events in BigQuery

/*0) Create a clean, partitioned table from the raw events.
    PARTITION BY DATE(event_time) to speed up time-range scans.
    SAFE_CAST converts/validates types without failing the query.
    Filters out rows with NULL price, since price is central to revenue calcs. -*/

-- glassy-ripsaw-473223-u3 is project name with ecommerce_events_raw and later cleaned and has ecommerce_events_clean

CREATE OR REPLACE TABLE `glassy-ripsaw-473223-u3.ecommerce.events_clean`
PARTITION BY DATE(event_time)
AS
SELECT
  TIMESTAMP(event_time) AS event_time,  -- normalize to TIMESTAMP
  event_type,   -- keeping as-is
  SAFE_CAST(product_id   AS STRING) AS product_id,   -- to string type
  SAFE_CAST(category_id  AS STRING) AS category_id,  -- to string type
  category_code,  -- may be NULL-used in group-bys
  brand,  -- may be NULL-pased in group-bys
  SAFE_CAST(price AS FLOAT64)AS price,        -- numeric price for math
  SAFE_CAST(user_id AS STRING) AS user_id,    -- user identifying
  SAFE_CAST(user_session AS STRING)AS user_session  -- session identifying
FROM `glassy-ripsaw-473223-u3.ecommerce.events_raw`
WHERE price IS NOT NULL;  -- remove rows without price


---EDA (Exploratory Data Analysis)

-- 0.1 Row count + time coverage (checking of table size and timeline)
SELECT
  COUNT(*) AS total_rows,
  MIN(event_time) AS first_event,
  MAX(event_time) AS last_event
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`;

-- 0.2 Missing data overview
SELECT
  SUM(CASE WHEN category_code IS NULL THEN 1 ELSE 0 END) AS null_category_code,
  SUM(CASE WHEN brand  IS NULL THEN 1 ELSE 0 END) AS null_brand,
  SUM(CASE WHEN price IS NULL THEN 1 ELSE 0 END) AS null_price  -- should be 0 by construction
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`;

-- 0.3 Cardinalities (uniques across main keys for sense of scale)
SELECT
  COUNT(DISTINCT product_id) AS unique_products,
  COUNT(DISTINCT user_id) AS unique_users,
  COUNT(DISTINCT user_session)  AS unique_sessions,
  COUNT(DISTINCT category_code) AS unique_categories,
  COUNT(DISTINCT brand) AS unique_brands
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`;

-- 0.4 Price stats (range, average, quartiles for distribution shape)
SELECT
  MIN(price) AS min_price,
  MAX(price) AS max_price,
  AVG(price) AS avg_price,
  APPROX_QUANTILES(price, 4) AS price_quartiles  -- min, Q1, median, Q3, max
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`;

-- 0.5 Monthly event distribution (volume + gross value per month)
SELECT
  FORMAT_DATE('%Y-%m', DATE(event_time)) AS month,
  COUNT(*) AS total_events,
  SUM(price) AS total_value
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
GROUP BY month
ORDER BY month;

-- 4) Session analytics (basket size/value)
--    unique_products = breadth of items in a basket.
--    items = total line count in session
--    session_value = revenue per session.
SELECT
  user_session,
  COUNT(DISTINCT product_id) AS unique_products,
  COUNT(*) AS items,
  SUM(price) AS session_value
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
GROUP BY user_session
ORDER BY session_value DESC
LIMIT 1000;

-- 5) Trend + anomaly detection (7-day rolling z-score)
--    ma7 = 7-day moving average revenue.
--    sd7 = 7-day rolling std dev.
--    zscore flags deviations from short-term trend.
WITH daily AS (
  SELECT DATE(event_time) AS day, SUM(price) AS revenue
  FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
  GROUP BY day
),
stats AS (
  SELECT day,revenue,
    AVG(revenue) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS ma7,
    STDDEV_POP(revenue) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS sd7
  FROM daily)

SELECT
  day, revenue, ma7, sd7,
  SAFE_DIVIDE(revenue - ma7, sd7) AS zscore
FROM stats
WHERE sd7 IS NOT NULL
ORDER BY day;

-- 6) Scalability experiment (compare scan size/latency by time window)
--    Same aggregation over 1 month, 3 months, and 7 months.
--    Measure performance externally (bytes processed, duration).

-- 6a: 1 month window
SELECT category_code, brand, SUM(price) AS revenue
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
WHERE event_time >= TIMESTAMP('2019-10-01') AND event_time < TIMESTAMP('2019-11-01')
GROUP BY category_code, brand;

-- 6b: 3 month window
SELECT category_code, brand, SUM(price) AS revenue
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
WHERE event_time >= TIMESTAMP('2019-10-01') AND event_time < TIMESTAMP('2020-01-01')
GROUP BY category_code, brand;

-- 6c: full 7 months window
SELECT category_code, brand, SUM(price) AS revenue
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
WHERE event_time >= TIMESTAMP('2019-10-01') AND event_time < TIMESTAMP('2020-05-01')
GROUP BY category_code, brand;
```

```sql
-- PART B: Performance / Analyese
-- 1) Daily revenue trend with day-over-day change
--      Aggregates orders, revenue, avg ticket daily.
--    Uses LAG to compute day-over-day revenue delta.
WITH daily AS (
  SELECT
    DATE(event_time) AS day,
    COUNT(*) AS total_orders,
    SUM(price) AS total_revenue,
    AVG(price) AS avg_ticket
  FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
  GROUP BY day
)
SELECT
  day,
  total_orders,
  total_revenue,
  avg_ticket,
  total_revenue - LAG(total_revenue) OVER (ORDER BY day) AS revenue_change
FROM daily
ORDER BY day;

-- 2) Category into Brand aggregation (multi-dimensional performance view)
--    Filters out NULLs so results are cleaner.
--    Orders by revenue to surface top pairs.
SELECT
  category_code,
  brand,
  COUNT(*) AS total_orders,
  SUM(price) AS total_revenue,
  AVG(price) AS avg_price
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
WHERE brand IS NOT NULL AND category_code IS NOT NULL
GROUP BY category_code, brand
ORDER BY total_revenue DESC
LIMIT 1000;

-- 3) User-level monetization
--    Sessions = distinct sessions per user
--    Events = line-item or click-level counts.
--    total_spent, avg_spent = revenue view.
SELECT
  user_id,
  COUNT(DISTINCT user_session) AS sessions,
  COUNT(*) AS events,
  SUM(price)  AS total_spent,
  AVG(price)  AS avg_spent
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
GROUP BY user_id
ORDER BY total_spent DESC
LIMIT 1000;
```

```sql
-- 7) Top-K products by revenue (exact ranking)
SELECT product_id, SUM(price) AS revenue
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
GROUP BY product_id
ORDER BY revenue DESC
LIMIT 100;

-- 8) Brand-level price outliers (|z| > 3)
--     per-brand mean & stddev.
--    Join back to score each event's price.
WITH brand_stats AS (
  SELECT brand, AVG(price) AS avg_price, STDDEV_POP(price) AS sd_price
  FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean`
  GROUP BY brand
)
SELECT
  e.brand, e.product_id, e.price, b.avg_price, b.sd_price,
  SAFE_DIVIDE(e.price - b.avg_price, b.sd_price) AS z
FROM `glassy-ripsaw-473223-u3.ecommerce.events_clean` e
JOIN brand_stats b USING (brand)
WHERE b.sd_price IS NOT NULL AND ABS(SAFE_DIVIDE(e.price - b.avg_price, b.sd_price)) > 3
ORDER BY z DESC
LIMIT 100;
```

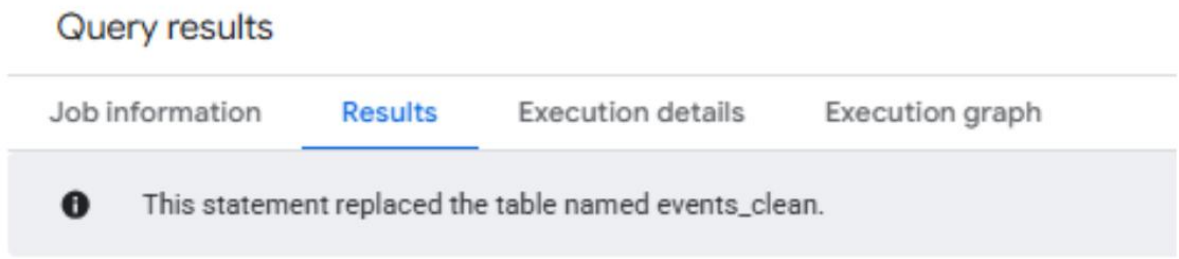
Databricks

We have run the same queries as BigQuery.

Outputs:

Both Platforms are giving almost the same outputs:

Creating New clean table :



Row count + time coverage:

Row	total_rows	first_event	last_event
1	67501979	2019-11-01 00:00:00 UTC	2019-11-30 23:59:59 UTC

Missing data overview:

Row	null_category_code	null_brand	null_price
1	21898171	9218235	0

Cardinalities (uniques across main keys for sense of scale):

Row	unique_products	unique_users	unique_sessions	unique_categories	unique_brands
1	190662	3696117	13776050	129	4201

Price stats (range, average, quartiles for distribution shape):

Row	min_price	max_price	avg_price	price_quartiles
1	0.0	2574.07	292.4593165646...	0.0
				68.47
				162.42
				360.09
				2574.07

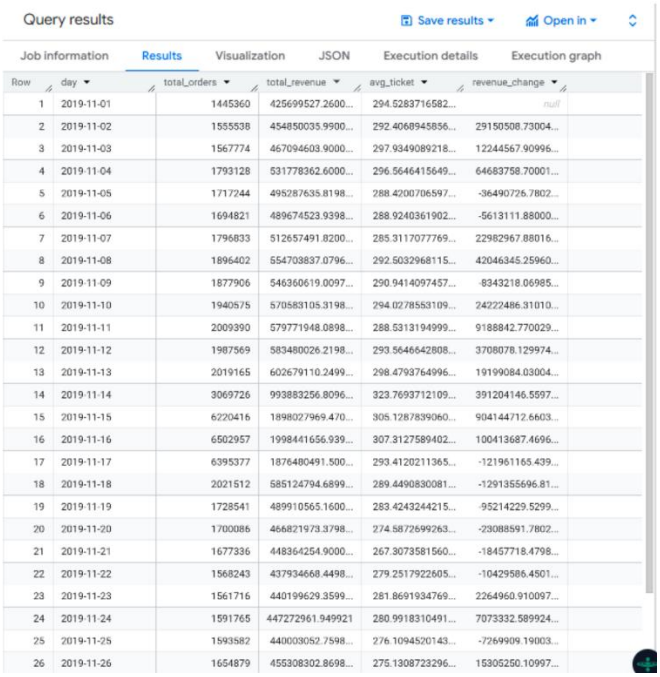
Monthly event distribution (volume + gross value per month):

Row	month	total_events	total_value
1	2019-11	67501979	19741582645.09...

1) Daily revenue trend with day-over-day change

-- Aggregates orders, revenue, avg ticket daily.

-- Uses LAG to compute day-over-day revenue delta.



Query results

Save results | Open in

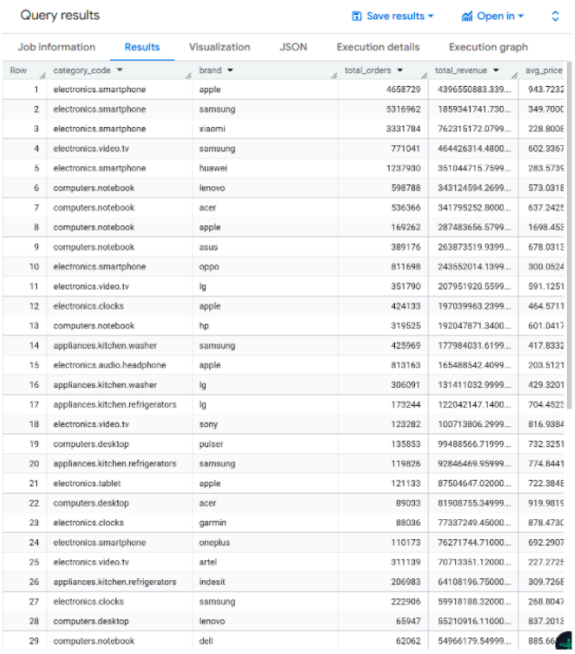
Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	day	total_orders	total_revenue	avg_ticket	revenue_change
1	2019-11-01	1445360	425699527.2600...	294.5283716582...	null
2	2019-11-02	1555538	454850035.9900...	292.4068945856...	29150508.73004...
3	2019-11-03	1567774	467094603.9000...	297.9349089218...	12244567.90996...
4	2019-11-04	1793128	531778362.6000...	296.5646415649...	64683758.70001...
5	2019-11-05	1717244	495287635.8198...	288.4200706597...	-36490726.7802...
6	2019-11-06	1694821	489674523.9398...	288.9240361902...	-5613111.88000...
7	2019-11-07	1796833	512657491.8200...	285.3117077769...	22982967.88016...
8	2019-11-08	1896402	554703837.0796...	292.5032968115...	42046345.25960...
9	2019-11-09	1877906	546360619.0097...	290.9414097457...	-8343218.06985...
10	2019-11-10	1940575	570583105.3198...	294.0278553109...	24222486.31010...
11	2019-11-11	2009390	579771948.0898...	288.5313194999...	9188842.770029...
12	2019-11-12	1987569	583480026.2198...	293.5646642808...	3708078.129974...
13	2019-11-13	2019165	602679110.2499...	298.4793764996...	19199084.03004...
14	2019-11-14	3069726	993883256.8096...	323.7693712109...	391204146.5597...
15	2019-11-15	6220416	1898027969.470...	305.1287839060...	904144712.6603...
16	2019-11-16	6502957	1998441656.939...	307.3127589402...	100413687.4696...
17	2019-11-17	6395377	1876480491.500...	293.4120211365...	-121961165.439...
18	2019-11-18	2021512	585124794.6899...	289.4490830081...	-1291355696.81...
19	2019-11-19	1728541	489910565.1600...	283.4243244215...	-95214229.5299...
20	2019-11-20	1700086	466821973.3798...	274.5872699263...	-23088591.7802...
21	2019-11-21	1677336	448364254.9000...	267.3073581560...	-18457718.4798...
22	2019-11-22	1568243	437934668.4498...	279.2517922605...	-10429586.4501...
23	2019-11-23	1561716	440199629.3599...	281.8691934769...	2264960.910097...
24	2019-11-24	1591765	447272961.949921	280.9918310491...	7073332.589924...
25	2019-11-25	1593582	440003052.7598...	276.1094520143...	-7269909.19003...
26	2019-11-26	1654879	455308302.8698...	275.1308723296...	15305250.10997...

2) Category into Brand aggregation (multi-dimensional performance view)

-- Filters out NULLs so results are cleaner.

-- Orders by revenue to surface top pairs.



Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

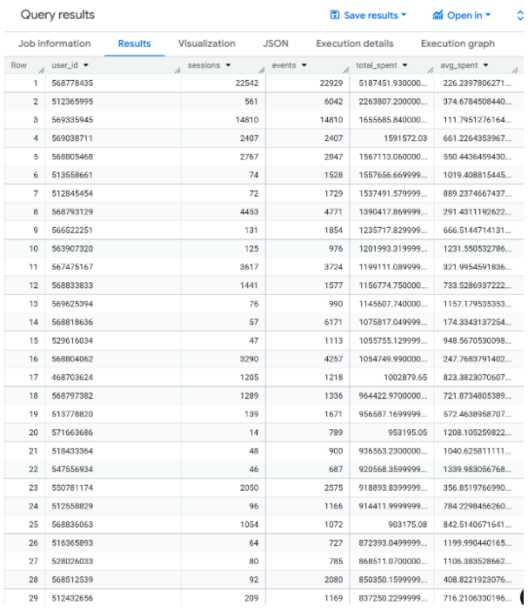
Row	category_code	brand	total_orders	total_revenue	avg_price
1	electronics.smartphone	apple	4658729	4396550883.339...	943.7232
2	electronics.smartphone	samsung	5316962	1859341741.730...	349.7000
3	electronics.smartphone	xiaomi	3331784	762315172.0799...	228.8006
4	electronics.video.tv	samsung	771041	464426314.4800...	602.3367
5	electronics.smartphone	huawei	1237930	351044715.7599...	283.5735
6	computers.notebook	lenovo	598788	343124594.2699...	573.0316
7	computers.notebook	acer	536366	341795252.8000...	637.2425
8	computers.notebook	apple	169262	287483656.5799...	1698.453
9	computers.notebook	asus	389176	263873519.9399...	678.0313
10	electronics.smartphone	oppo	811698	243552014.1399...	300.0524
11	electronics.video.tv	lg	351790	207951920.5599...	591.1251
12	electronics.clocks	apple	424133	197039963.2399...	464.5711
13	computers.notebook	hp	319525	192047871.3400...	601.0417
14	appliances.kitchen.washer	samsung	425969	177984031.6199...	417.8332
15	electronics.audio.headphone	apple	813163	165488542.4099...	203.5121
16	appliances.kitchen.washer	lg	306091	131411032.9999...	429.3201
17	appliances.kitchen.refrigerators	lg	173244	122042147.1400...	704.4525
18	electronics.video.tv	sony	123282	100713806.2999...	816.9384
19	computers.desktop	pulser	135853	99488566.71999...	732.3251
20	appliances.kitchen.refrigerators	samsung	119826	92846469.95999...	774.8441
21	electronics.tablet	apple	121133	87504647.02000...	722.3846
22	computers.desktop	acer	89033	81908755.34999...	919.9815
23	electronics.clocks	garmin	88036	77337249.45000...	878.4730
24	electronics.smartphone	oneplus	110173	76271744.71000...	692.2907
25	electronics.video.tv	artel	311139	70713351.12000...	227.2725
26	appliances.kitchen.refrigerators	indesit	206983	64108196.75000...	309.7268
27	electronics.clocks	samsung	222906	59918188.32000...	268.8047
28	computers.desktop	lenovo	65947	55210916.11000...	837.2013
29	computers.notebook	dell	62062	54966179.54999...	885.66

3) User-level monetization

-- Sessions = distinct sessions per user

-- Events = line-item or click-level counts.

-- total_spent, avg_spent = revenue view.



Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

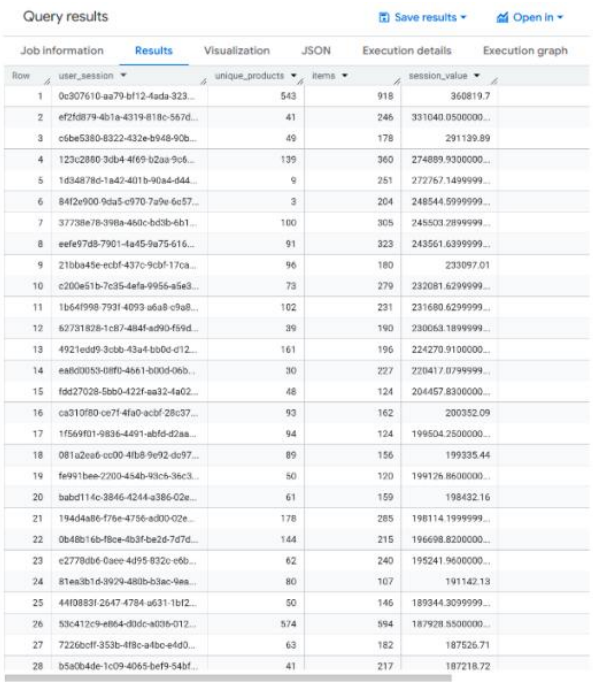
Row	user_id	sessions	events	total_spent	avg_spent
1	568778435	22542	22929	5187451.930000...	226.2397806271...
2	512365995	561	6042	2263807.200000...	374.6784508440...
3	569335945	14810	14810	1655685.840000...	111.7951276164...
4	569038711	2407	2407	1591572.03	661.2264353967...
5	568805468	2767	2847	1567113.060000...	550.4436459430...
6	513558661	74	1528	1557656.669999...	1019.408815445...
7	512845454	72	1729	1537491.579999...	889.2374667437...
8	568793129	4483	4771	1390417.869999...	291.4311192622...
9	566622251	131	1854	1235717.829999...	666.5144714131...
10	563907320	125	976	1201993.319999...	1231.550532786...
11	567475167	3617	3724	1199111.089999...	321.9954591836...
12	568833833	1441	1577	1156774.750000...	733.5286937222...
13	569625394	76	990	1145607.740000...	1157.179535353...
14	568818636	57	6171	1075817.049999...	174.3343137254...
15	529616034	47	1113	1055755.129999...	948.5670530098...
16	568804062	3290	4257	1054749.990000...	247.7683791402...
17	468703624	1205	1218	1002879.65	823.3823070607...
18	568797382	1289	1336	964422.9700000...	721.8734805389...
19	513778820	139	1671	956587.1699999...	572.4638958707...
20	571663686	14	789	953195.05	1208.105259822...
21	518433364	48	900	936553.2300000...	1040.625811111...
22	547556934	46	687	920568.3599999...	1339.983056768...
23	550781174	2050	2575	918893.8399999...	356.8519766990...
24	512558829	96	1166	914411.9999999...	784.2298456260...
25	568836063	1054	1072	903175.08	842.5140671641...
26	516365893	64	727	872393.0499999...	1199.990440165...
27	528026033	80	785	868511.0700000...	1106.383528662...
28	568512539	92	2080	850350.1599999...	408.8221923076...
29	512432656	209	1169	837250.2299999...	716.2106330196...

4) Session analytics (basket size/value)

-- unique_products = breadth of items in a basket.

-- items = total line count in session

-- session_value = revenue per session.



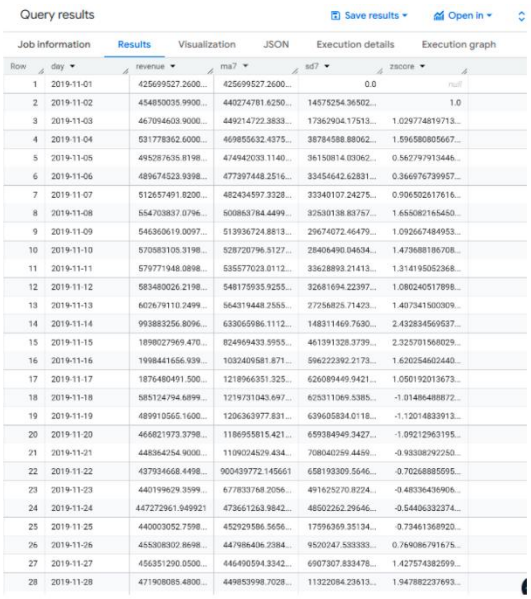
Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	user_session	unique_products	items	session_value
1	0c307610-aa79-bf12-4ada-323...	543	918	360819.7
2	ef2fd879-4b1a-4319-818c-567d...	41	246	331040.0500000...
3	c6be5380-8322-432e-b948-90b...	49	178	291139.89
4	123c2880-3db4-4f69-b2aa-9c6...	139	360	274889.9300000...
5	1d34878d-1a42-401b-90a4-644...	9	251	272767.1499999...
6	84f2e900-9da5-c970-7a9e-6c57...	3	204	248544.5999999...
7	37738e78-398a-46dc-bd3b-6b1...	100	305	245503.2899999...
8	eefe97d8-7901-4a45-9a75-616...	91	323	243561.6399999...
9	21bba45e-ecbf-437c-9cbf-17ca...	96	180	233097.01
10	c200e51b-7c35-4efa-9956-a5e3...	73	279	232081.6299999...
11	1b64f998-793f-4093-a6a8-c9a8...	102	231	231680.6299999...
12	62731828-1c87-484f-ad90-f59d...	39	190	230063.1899999...
13	4921edd9-3cbb-43a4-bb0d-d12...	161	196	224270.9100000...
14	ea8d0053-08f0-46b1-b00d-06b...	30	227	220417.0799999...
15	fdd27028-5bb0-422f-aa32-4a02...	48	124	204457.8300000...
16	ca310f80-ce7f-4fa0-acbf-28c37...	93	162	200352.09
17	1f569f01-9836-4491-abfd-d2aa...	94	124	199504.2500000...
18	081a2ea6-cc00-4fb8-9e92-dc97...	89	156	199335.44
19	fe991bee-2200-454b-93c6-36c3...	50	120	199126.8600000...
20	babd114c-3846-4244-a386-02e...	61	159	198432.16
21	194d4a86-f76e-4756-ad00-02e...	178	285	198114.1999999...
22	0b48b16b-f8ce-4b3f-be2d-7d7d...	144	215	196698.8200000...
23	e2770db6-0aee-4d95-832c-e6b...	62	240	195241.9600000...
24	81ea3b1d-3929-488b-b3ac-9ea...	80	107	191142.13
25	44f0883f-2647-4784-a631-1bf2...	50	146	189344.3099999...
26	53c412c9-e864-d0dc-a036-012...	574	594	187928.5500000...
27	7226bcff-353b-4f8c-a4bc-e4d0...	63	182	187526.71
28	b5a0b4de-1c09-4065-bef9-54bf...	41	217	187218.72

5) Trend + anomaly detection (7-day rolling z-score)
-- ma7 = 7-day moving average revenue.
-- sd7 = 7-day rolling std dev.
-- zscore flags deviations from short-term trend.



Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	day	revenue	ma7	sd7	zscore
1	2019-11-01	425699527.2600...	425699527.2600...	0.0	null
2	2019-11-02	454850035.9900...	440274781.6250...	14575254.36502...	1.0
3	2019-11-03	467094603.9000...	449214722.3833...	17362904.17513...	1.029774819713...
4	2019-11-04	531778362.6000...	469855632.4375...	38784588.88062...	1.596580805667...
5	2019-11-05	495287635.8198...	474942033.1140...	36150814.03062...	0.562797913446...
6	2019-11-06	489674523.9398...	477397448.2516...	33454642.62831...	0.366976739957...
7	2019-11-07	512657491.8200...	482434597.3328...	33340107.24275...	0.906502617616...
8	2019-11-08	554703837.0796...	500863784.4499...	32530138.83757...	1.655082165450...
9	2019-11-09	546360619.0097...	513936724.8813...	29674072.46479...	1.092667484953...
10	2019-11-10	570583105.3198...	528720796.5127...	28406490.04634...	1.473688186708...
11	2019-11-11	579771948.0898...	535577023.0112...	33628893.21413...	1.314195052368...
12	2019-11-12	583480026.2198...	548175935.9255...	32681694.22397...	1.080240517898...
13	2019-11-13	602679110.2499...	564319448.2555...	27256825.71423...	1.407341500309...
14	2019-11-14	993883256.8096...	633065986.1112...	148311469.7630...	2.432834569537...
15	2019-11-15	1898027969.470...	824969433.5955...	461391328.3739...	2.325701568029...
16	2019-11-16	1998441666.939...	1032409581.871...	596222392.2173...	1.620254602440...
17	2019-11-17	1876480491.500...	1218966351.325...	626089449.9421...	1.050192013673...
18	2019-11-18	585124794.6899...	1219731043.697...	625311069.5385...	-1.01486488872...
19	2019-11-19	489910565.1600...	1206363977.831...	639605834.0118...	-1.12014833913...
20	2019-11-20	466821973.3798...	1186955815.421...	659384949.3427...	-1.09212963195...
21	2019-11-21	448364254.9000...	1109024529.434...	708040259.4459...	-0.93308292250...
22	2019-11-22	437934668.4498...	900439772.145661	658193309.5646...	-0.70268885095...
23	2019-11-23	440199629.3599...	677833768.2056...	491625270.8224...	-0.48336436906...
24	2019-11-24	447272961.949921	473661263.9842...	486022262.29646...	-0.54406332374...
25	2019-11-25	440003052.7598...	452929586.5656...	17596369.35134...	-0.73461368920...
26	2019-11-26	455308302.8698...	447986406.2384...	9520247.533333...	0.769086791675...
27	2019-11-27	456351290.0500...	446490594.3342...	6907307.833478...	1.427574382699...
28	2019-11-28	471908085.4800...	449853998.7028...	11322084.23613...	1.947882237693...

6) Scalability experiment (compare scan size/latency by time window)
-- Same aggregation over 1 month, 3 months, and 7 months.
-- Measure performance externally (bytes processed, duration).

Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	category_code	brand	revenue
1	apparel.shoes.keds	null	5366510.830000...
2	appliances.personal.massager	null	537759.3800000...
3	appliances.environment.water_...	null	918041.9999999...
4	stationery.cartrige	null	67380.45000000...
5	null	a-mega	65144.00999999...
6	electronics.audio.acoustic	adagio	76252.34999999...
7	appliances.kitchen.hood	akpo	874625.24
8	appliances.kitchen.refrigerators	almacom	1767210.120000...
9	auto.accessories.videoregister	alpine	80439.24999999...
10	construction.tools.welding	alteco	65194.86000000...
11	computers.components.memory	amd	31106.08999999...
12	apparel.shoes.keds	anta	523612.2499999...
13	computers.notebook	apple	287483656.5799...
14	null	arnica	19602.65999999...
15	appliances.kitchen.hood	artel	717065.2099999...
16	furniture.bedroom.bed	askona	215933.1400000...
17	accessories.bag	asus	5601.939999999...
18	appliances.kitchen.washer	atlant	4503965.73
19	sport.snowboard	atlant	125212.2299999...
20	apparel.shoes	atrai	53860.49999999...
21	sport.bicycle	author	1107202.439999...

7) Top-K products by revenue (exact ranking)

Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	product_id	revenue
1	1005115	619370131.6099...
2	1005105	414205784.4399...
3	1005135	253700985.780001
4	1004249	219218005.9900...
5	1005116	189324729.1200...
6	1004767	140070377.5600...
7	1002544	128290954.8099...
8	1003317	114420230.7499...
9	1004659	100994058.8900...
10	1005174	92020653.60999...
11	1004870	89571223.92000...
12	1005129	86360298.34999...
13	1005284	85956164.55999...
14	1002524	85351937.71000...
15	1005106	82279350.14000...
16	1004856	80997844.46999...
17	1004873	79871374.42000...
18	1005124	77456216.88999...
19	1005144	76219416.78000...
20	1004258	75131906.01000...
21	1005132	73550214.96999...
22	1005186	72367965.02000...
23	1004246	70121840.31000...
24	1005160	68146237.29000...
25	1005104	66355126.88999...
26	1005073	64122712.81000...
27	4804056	63645354.55999...
28	1005118	61673358.31999...

8) Brand-level price outliers (|z| > 3)

-- per-brand mean & stddev.

-- Join back to score each event's price.

Query results

Save results | Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	brand	product_id	price	avg_price	sd_price
1	fitbit	5100235	218.77	180.1720392890...	0.68168
2	goodride	100017570	688.05	49.43657006525...	13.3569
3	turboair	2401679	618.25	49.65299112139...	13.1420
4	turboair	2401679	618.25	49.65299112139...	13.1420
5	triangle	12720013	1536.74	51.66865925173...	35.8682
6	triangle	12720013	1536.74	51.66865925173...	35.8682
7	triangle	12720013	1536.74	51.66865925173...	35.8682
8	triangle	12720013	1536.74	51.66865925173...	35.8682
9	triangle	12720013	1536.74	51.66865925173...	35.8682
10	triangle	12720013	1536.74	51.66865925173...	35.8682
11	triangle	12720013	1536.74	51.66865925173...	35.8682
12	triangle	12720013	1536.74	51.66865925173...	35.8682
13	triangle	12720013	1536.74	51.66865925173...	35.8682
14	triangle	12720013	1536.74	51.66865925173...	35.8682
15	triangle	12720013	1536.74	51.66865925173...	35.8682
16	triangle	12720013	1536.74	51.66865925173...	35.8682
17	triangle	12720013	1536.74	51.66865925173...	35.8682
18	triangle	12720013	1536.74	51.66865925173...	35.8682
19	triangle	12720013	1536.74	51.66865925173...	35.8682
20	triangle	12720013	1536.74	51.66865925173...	35.8682
21	triangle	12720013	1536.74	51.66865925173...	35.8682
22	triangle	12720013	1536.74	51.66865925173...	35.8682
23	triangle	12720013	1536.74	51.66865925173...	35.8682
24	triangle	12720013	1536.74	51.66865925173...	35.8682
25	triangle	12720013	1536.74	51.66865925173...	35.8682
26	triangle	12720013	1536.74	51.66865925173...	35.8682
27	triangle	12720013	1536.74	51.66865925173...	35.8682

PERFORMANCE ANALYSIS

Following screenshots give:

➢ Elapsed time

➢ Bytes processed

➢ Bytes billed

➢ Bytes shuffled

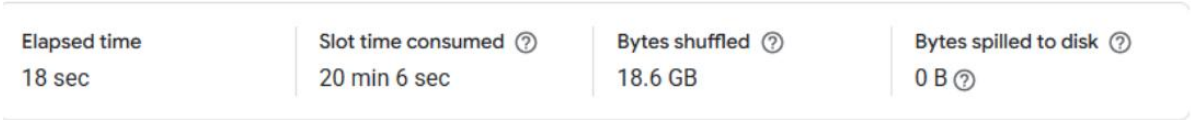
➢ Bytes spilled to disk

➢ Slot time consumed

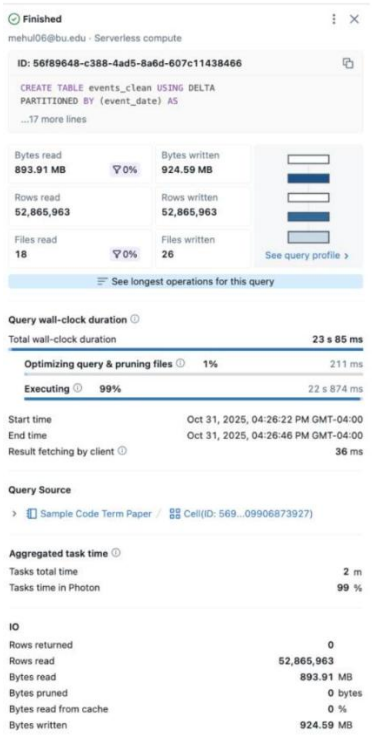
➢ Destination table

TO: Clean and create new clean table:

Big Query:

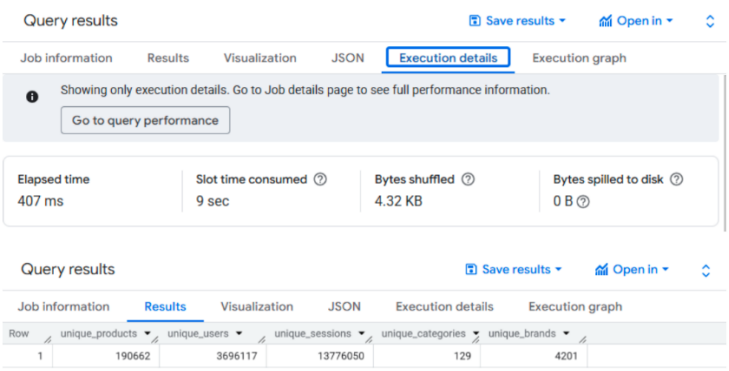


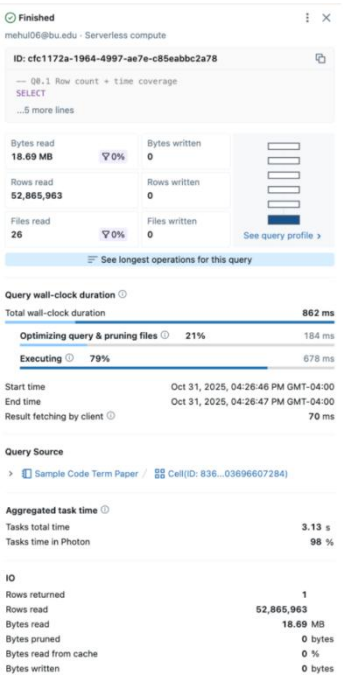
Databricks:



Row count and time coverage:

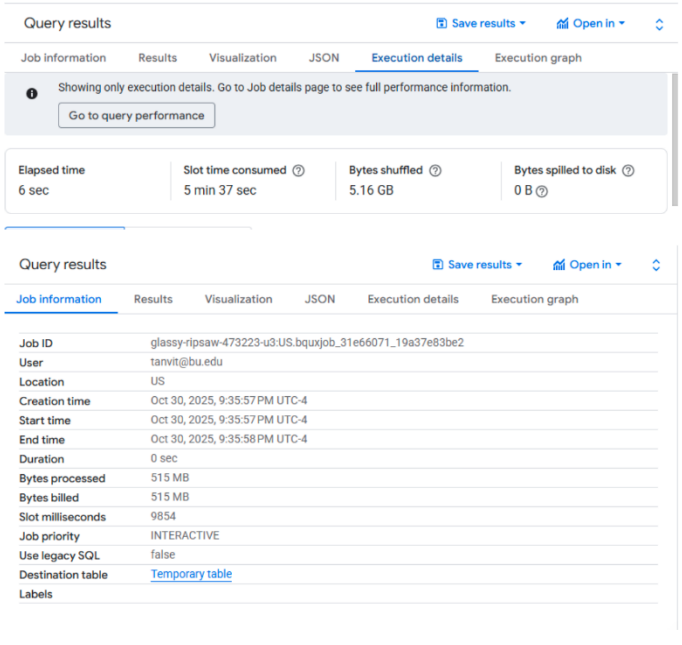
BigQuery and DataBricks:

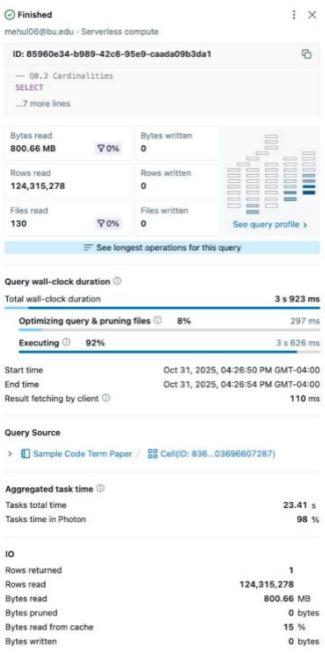




Missing Data Overview:

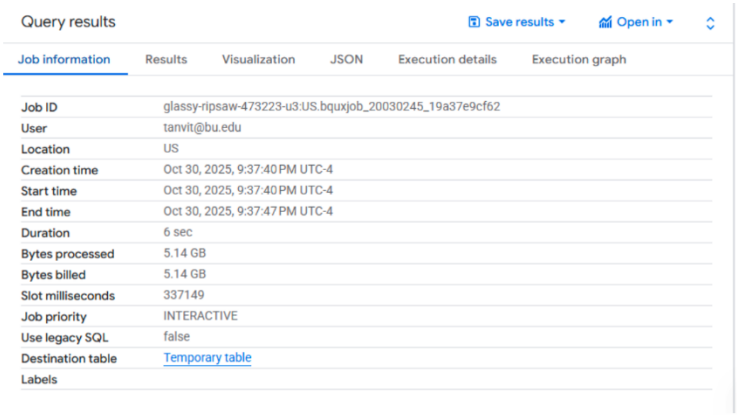
BigQuery and DataBricks:

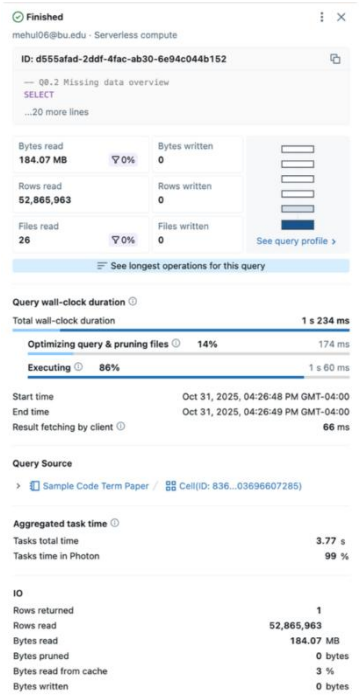




Cardinality Analysis:

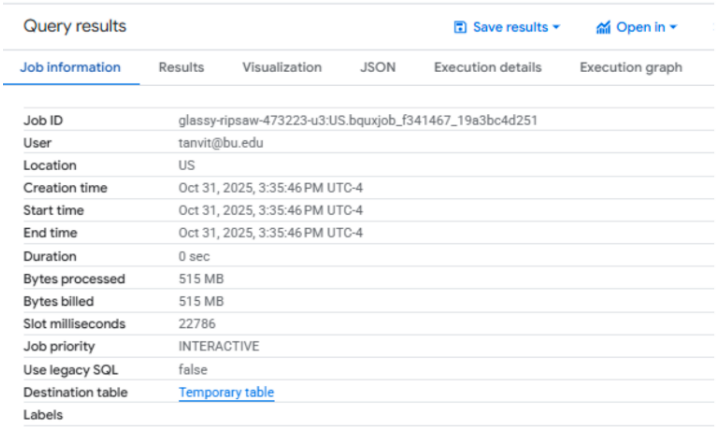
BigQuery and DataBricks

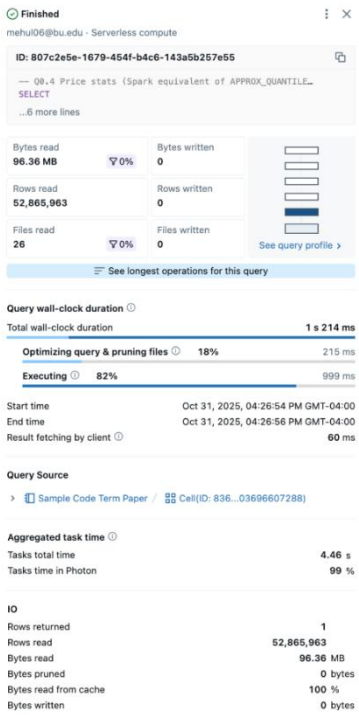




Price Statistics

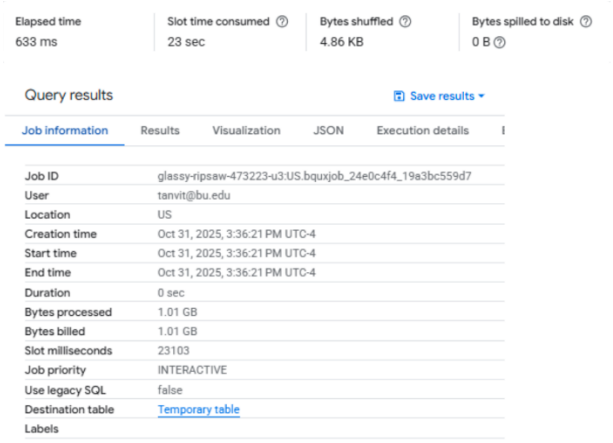
BigQuery and Databricks

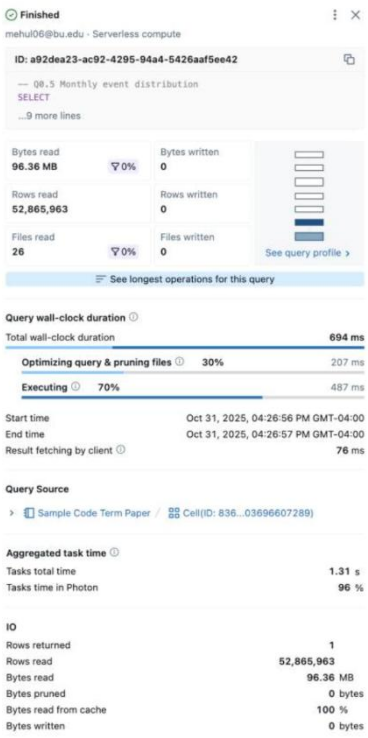




Monthly Event Distribution

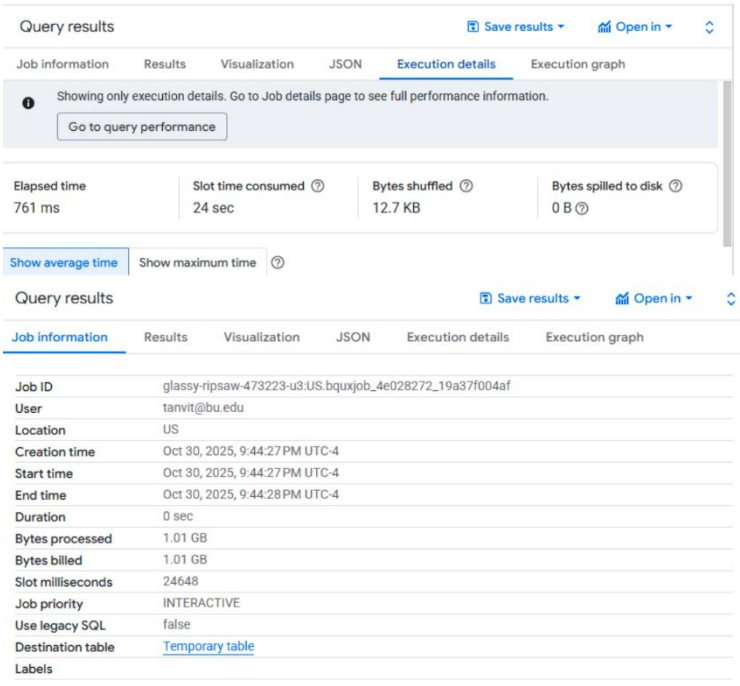
BigQuery and DataBricks

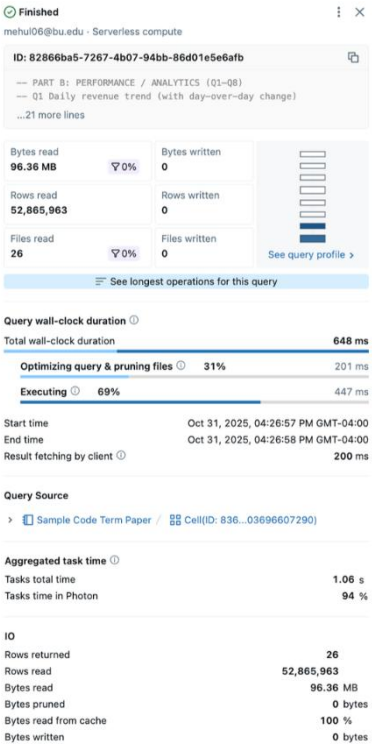




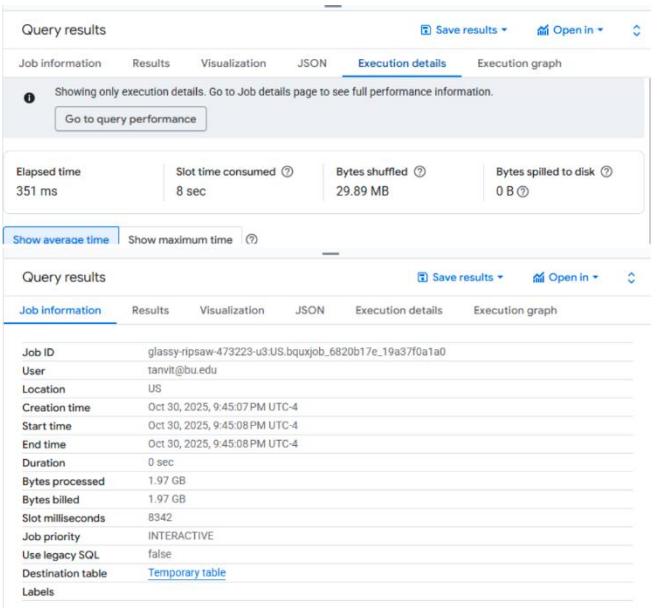
Daily Revenue Trend

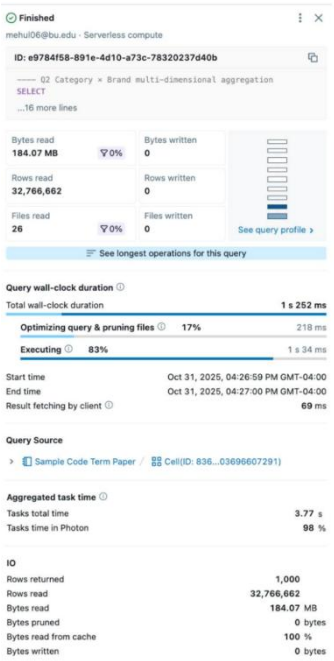
BigQuery and Databricks:



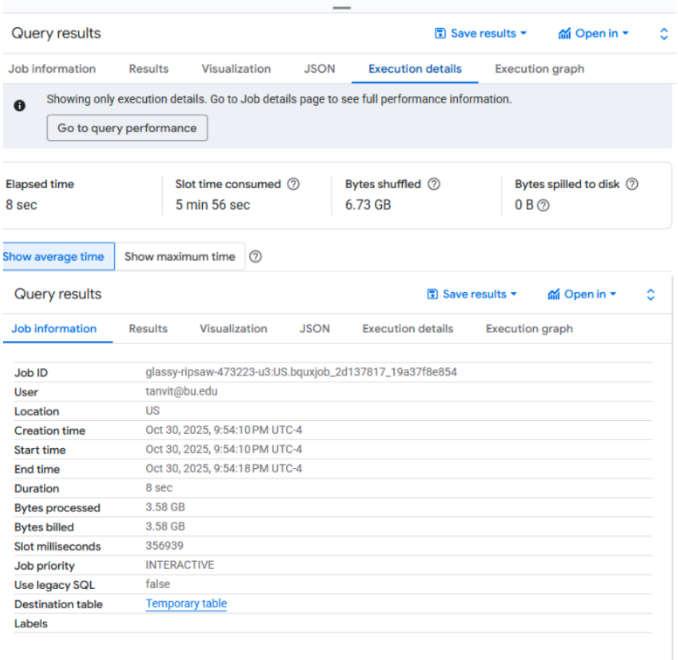


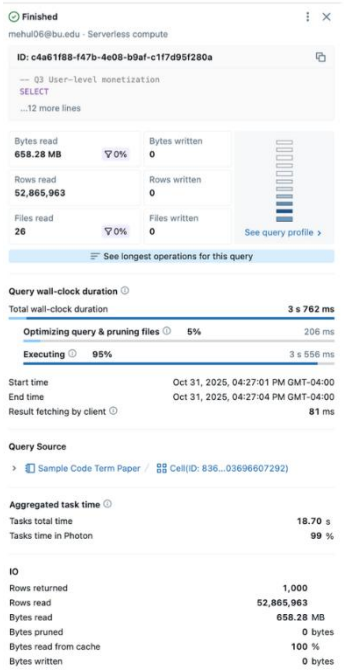
Category × Brand Aggregation
BigQuery and DataBricks





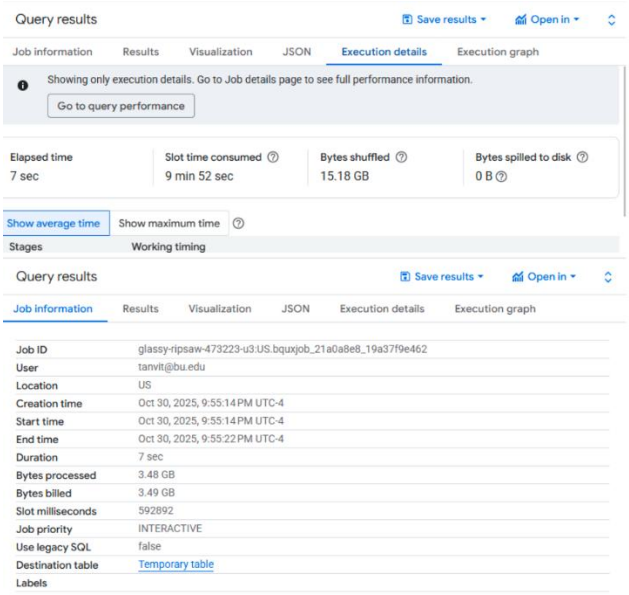
User-Level Monetization
BigQuery and Databricks

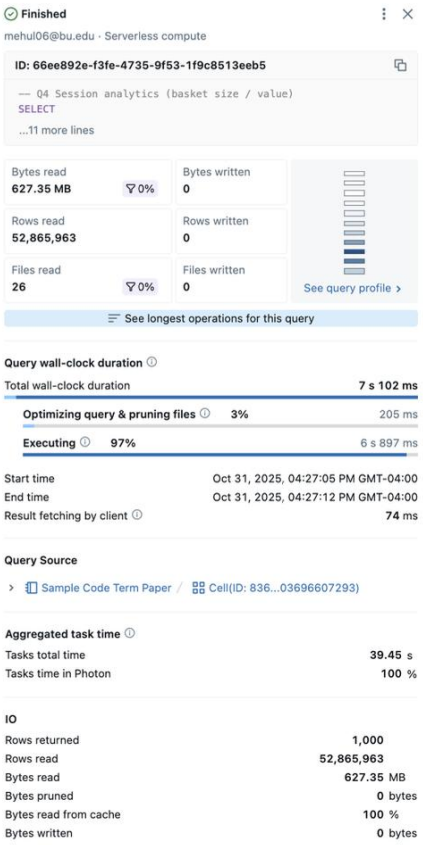




Session Analytics

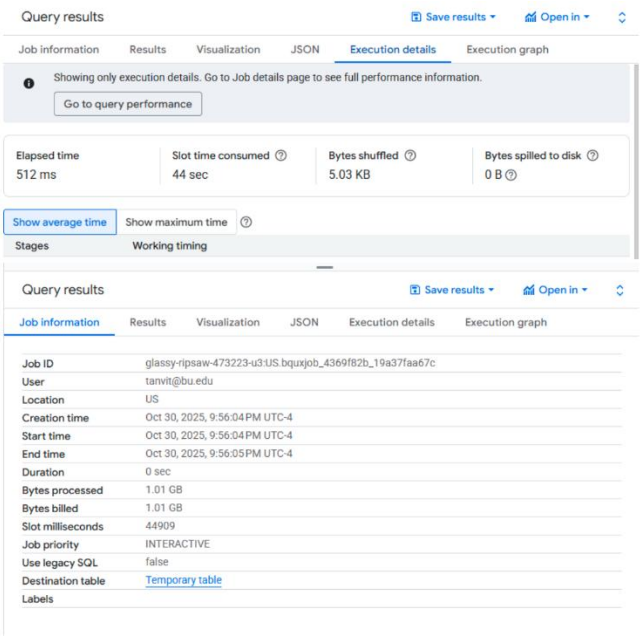
Big Query

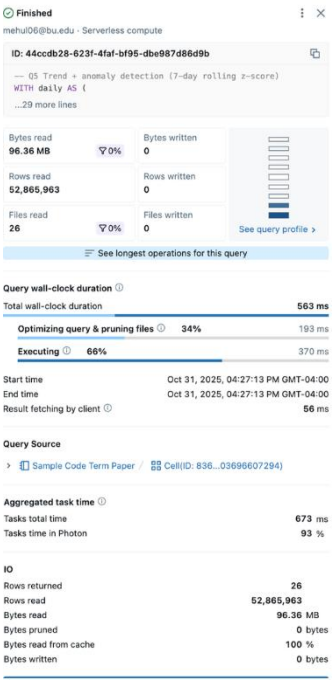




Trend and Anomaly Detection

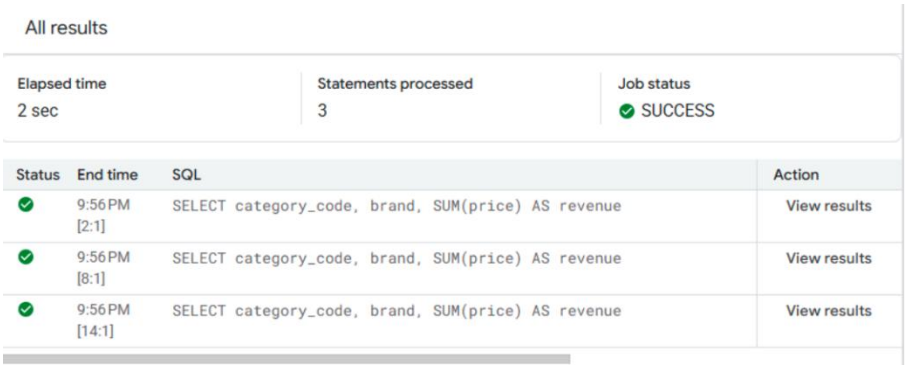
Big Query and Databricks

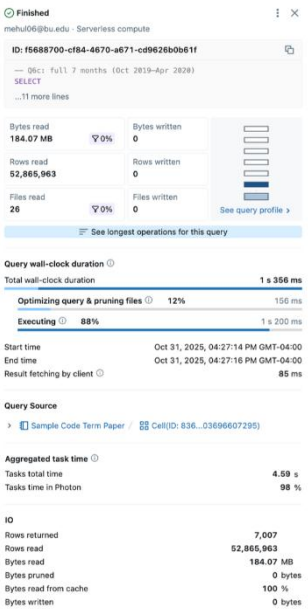




Scalability Experiment

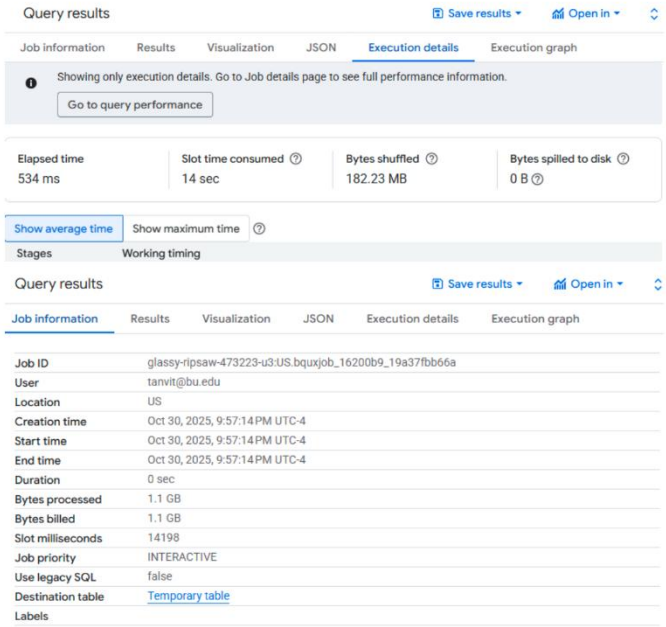
Big Query and Databricks

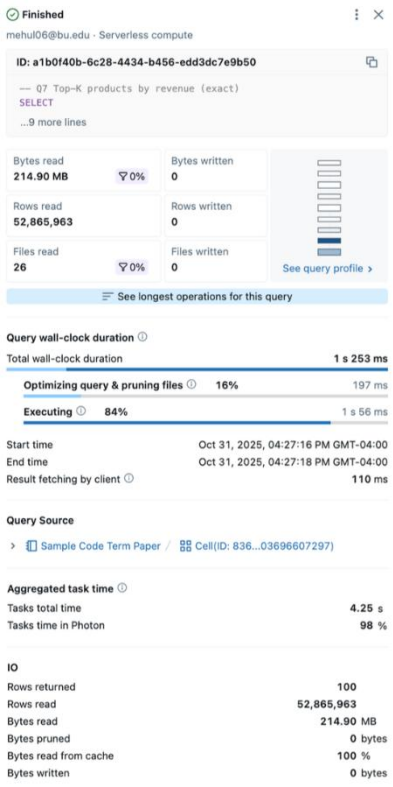




Top-K Products by Revenue

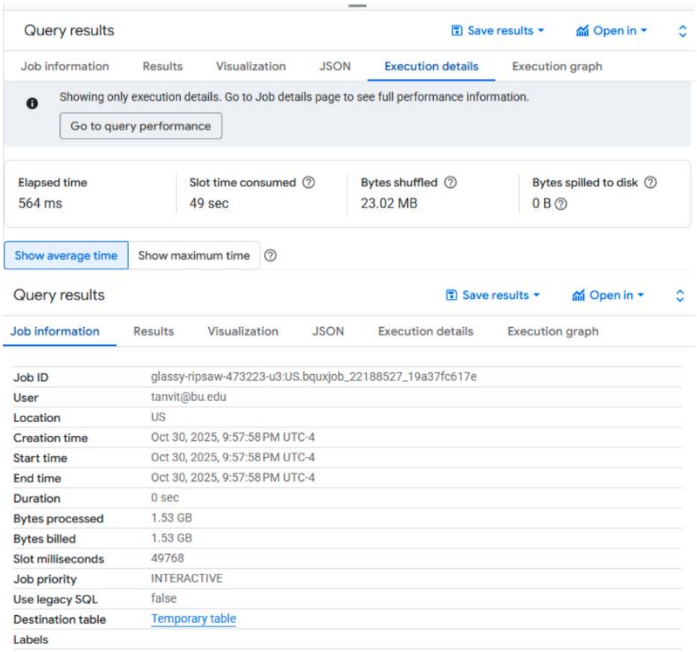
BigQuery and Databricks

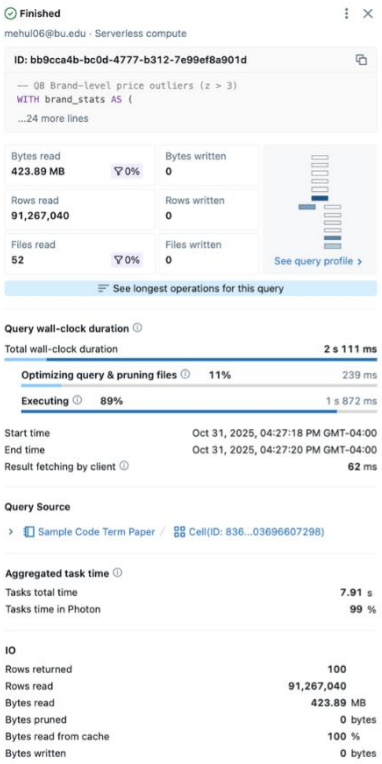




Brand-Level Price Outliers

BigQuery and Databricks





VISULIZATION PART:

Daily Revenue Trend

What we did: Used Day as the Dimension and Revenue + Orders as Metrics.

What it shows: Daily revenue and order count across the month.

Insight: Identifies sales spikes or drops over time (helps track performance trends).

Category × Brand Revenue

What we did: Used category_code as Dimension, brand as Breakdown, and Revenue as Metric.

What it shows: How much revenue each brand contributes within different product categories.

Insight: Reveals top-performing brand–category combinations driving sales.

Top Users by Total Spending

What we did: Used user_id as Dimension and Revenue + Avg Ticket as Metrics.

What it shows: The highest-spending customers and their average order values.

Insight: Highlights your most valuable users and potential VIP customers.

Spending Distribution Histogram

What we did: Used Revenue as Dimension and COUNT_DISTINCT(user_id) as Metric.

What it shows: The number of users grouped by spending range.

Insight: Shows how spending is distributed whether most users spend small amounts or a few spend a lot.

Session Analytics

What we did: Used user_session as Dimension and Unique Products, Orders, and Session Value as Metrics.

What it shows: The value of each user session number of products viewed, items bought, and total spend.

Insight: Measures engagement efficiency per session and identifies sessions with the highest revenue.

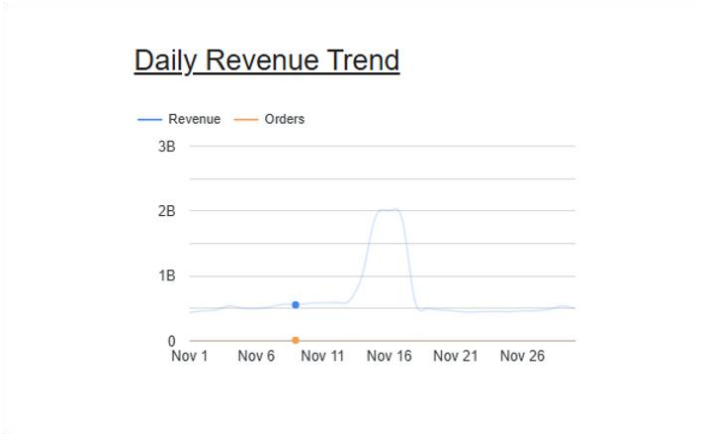
Anomalies (Revenue vs Rolling Average)

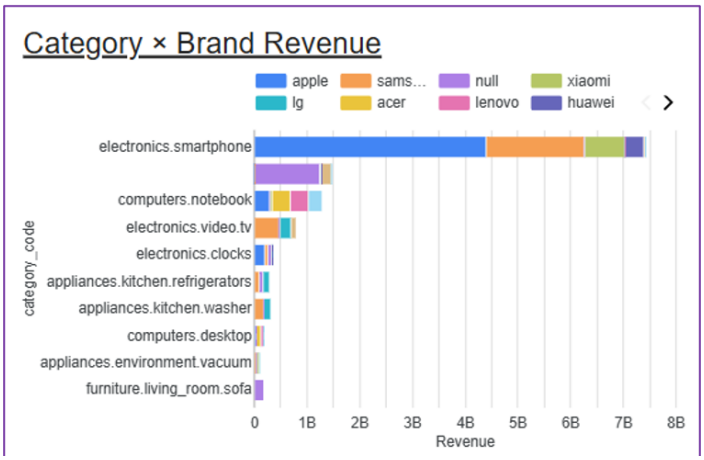
What we did: From the BigQuery daily_metrics view plotted revenue and ma7 (7-day moving average).

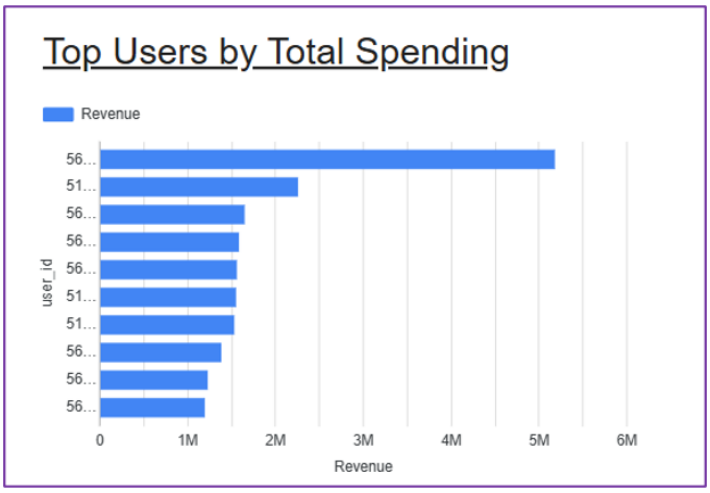
What it shows: Actual revenue compared to its smoothed 7-day trend.

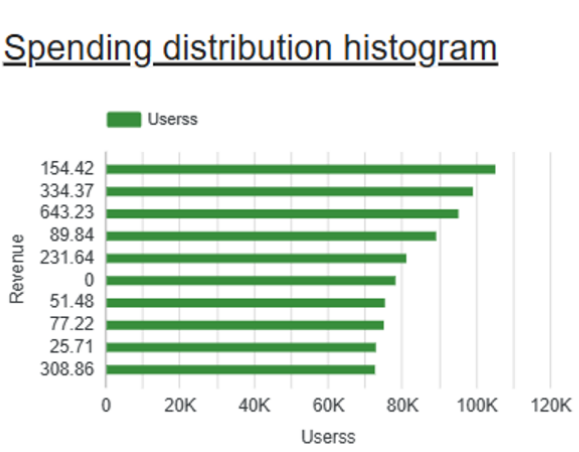
Insight: Highlights unusual spikes or drops in daily revenue useful for anomaly detection and demand forecasting.

LOOKER STUDIO RESULTS:







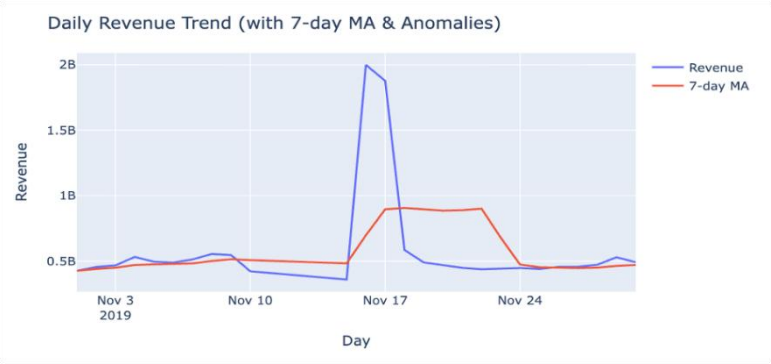


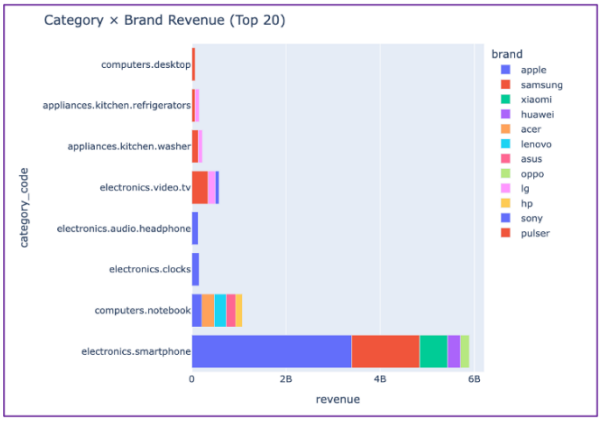
Session Analytics

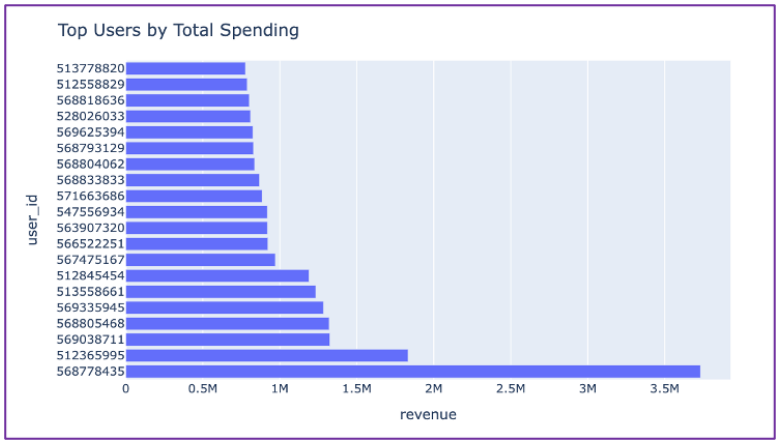
	user_session	Unique Prod...	Orders	Ses...
1.	0c307610-aa79...	543	918	360,819.7
2.	ef2fd879-4b1a-...	41	246	331,040.05
3.	c6be5380-8322...	49	178	291,139.89
4.	123c2880-3db4...	139	360	274,889.93
5.	1d34878d-1a42...	9	251	272,767.15
6.	84f2e900-9da5-...	3	204	248,544.6
7.	37738e78-398a...	100	305	245,503.29

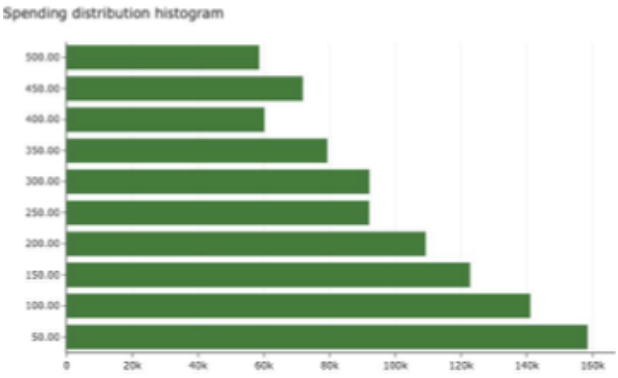
1 - 100 / 2000000+

PYTHON RESULTS (Plotly):









Session Analytics (Top 100 by Value)

user_session	unique_products	orders	session_value
0c307610-aa79-bf12-	543	918	360819.6999999998
ef2fd879-4b1a-4319-	41	246	331040.05000000005
c6be5380-8322-432e	49	178	291139.88999999996
1d34878d-1a42-401b	9	251	272767.15000000014
84f2e900-9da5-c970-	3	204	248544.5999999997
1b64f998-793f-4093-	102	231	231680.63000000003
62731828-1c87-484f-	39	190	230063.1900000001
4921edd9-3cbb-43a4	161	196	224270.9099999999
fdd27028-5bb0-422f-	48	124	204457.83000000007
081a2ea6-cc00-4fb8-	89	156	199335.43999999997
fe991bee-2200-454b-	50	120	199126.86
194d4a86-f76e-4756-	178	285	198114.2000000001
e2778db6-0aee-4d95	62	240	195241.96
44f0883f-2647-4784-	50	146	189344.31
3f48e024-c25e-40eb-	59	97	184565.29999999996

RESULT EXPLANATIONS:

Query / Task	Big Query Execution Time	Databricks Execution Time
Table Creation (events_clean)	~18s	~23 s
EDA Queries (0–4)	12sec total	9sec total
Daily Revenue Trend (5)	~761ms	~648s
Category × Brand Aggregation	~351ms	~1s
User-Level Monetization	~8ms	~3s
Session Analytics	~7ms	~7.2s
Anomaly Detection	~513ms	~563ms
Scalability Test (1-Month Data)	~2s	~1 s
Scalability Test (3-Month Data)	~534ms	~2s
Scalability Test (7-Month Data)	~564ms	~3 s

VISUALIZATION RESULTS:

Chart	What It Shows	Key Insight
Daily Revenue Trend	Revenue & order volume per day	Sales spikes or slow periods
Category × Brand Revenue	Revenue split by brand within each category	Top-performing brand-category pairs
Top Users by Spending	Total and average spend per user	Identifies VIP customers
Spending Distribution Histogram	Number of users per spending range	Reveals concentration of low vs. high spenders
Session Analytics	Products viewed, items bought, and session value	Measures engagement efficiency
Anomalies (Revenue vs Rolling Avg)	Daily revenue vs 7-day trend line	Detects unusual revenue fluctuations

Detailed Comparison and Explanation of Results:

Both BigQuery and Databricks were tested on the same 9 GB e-commerce dataset. The
queries included table creation, EDA (exploratory data analysis), trend analysis, user/session analytics, anomaly detection, and scalability testing. The results confirm that
both platforms delivered accurate outputs, but their performance varied depending on the
workload type.

1. **Table Creation (events_clean)**

BigQuery: ~18 seconds

Databricks: ~23 seconds

BigQuery completed table creation slightly faster. It directly pulls data from Google Cloud
Storage and automatically optimizes schema and partitioning, while Databricks required
CSV reading and transformation into a Delta table, adding a few extra seconds.

2. **EDA Queries (Q0–Q4)**

BigQuery: 12 seconds total

Databricks: 9 seconds total

Databricks outperformed here because once data was loaded into memory, it executed smaller, repeated queries faster. Spark's in-memory processing provided an edge over BigQuery's on-demand query execution.

3. **Daily Revenue Trend (Q5)**

BigQuery: ~761 milliseconds

Databricks: ~648 milliseconds

Both systems were efficient. Databricks performed marginally better because the

operation involved calculating aggregates over limited time windows, which suited Spark's
caching and partition-based parallelism.

4. **Category × Brand Aggregation**

BigQuery: ~351 milliseconds

Databricks: ~1 second

BigQuery handled large group-by operations faster due to its Dremel engine, which parallelizes across thousands of slots automatically. Databricks needed more shuffle operations to reorganize data before aggregation, slightly slowing it down.

5. **User-Level Monetization**

BigQuery: ~8 seconds

Databricks: ~3 seconds

Databricks was faster in this query since user-level aggregation benefited from Spark's distributed joins and cached intermediate results. It's more efficient for iterative or session-
based workloads once data is loaded in memory.

6. **Session Analytics**

BigQuery: ~7 seconds

Databricks: ~7.2 seconds

Both platforms performed almost identically. This query required grouping by session IDs,
which is high in cardinality but evenly distributed, making it well-optimized on both systems.

7. **Anomaly Detection**

BigQuery: ~513 milliseconds

Databricks: ~563 milliseconds

BigQuery's analytical functions (LAG, STDDEV, AVG) were slightly faster because it

executes window functions natively at scale. Databricks performed very closely, showing

Spark SQL's efficiency for analytical workloads.

8. **Scalability Tests**

For smaller ranges (1 month), Databricks performed faster as data was already cached. However, as data size increased (3–7 months), BigQuery scaled more efficiently — it automatically distributed queries across multiple nodes without manual cluster tuning.

9. **Key Insights**

BigQuery Strengths:

- Consistently faster for large scans and heavy aggregations.
- Auto-scaling ensures stable performance even as data grows.
- Best suited for ad-hoc analytics, dashboards, and large-scale reports.
- Built-in cost and performance transparency (bytes processed, slot time).

Databricks Strengths:

- Better for iterative, in-memory analytics and custom visualization (Plotly, PySpark).
- Ideal for data science, ML pipelines, and streaming workloads.
- Cluster resources can be adjusted for fine-grained control

10. Overall Conclusion

Aspect	BigQuery	Databricks
Performance on Large Data	Faster (auto-optimized, scalable)	Slower for very large queries
Performance on Cached Data	Slightly slower	Faster (Spark in-memory advantage)
Ease of Use	No setup, simple SQL	Requires cluster configuration
Visualization	Limited (Looker/Charts)	Excellent (Plotly, Matplotlib)
Scalability	Automatic and seamless	Manual tuning needed
Best For	Large-scale analytics, BI dashboards	Interactive data science, ML workflows

11. APPENDIX B: About the Dataset

The E-Commerce Events Dataset (2019–2020) is a big collection of user behavior tracked on a web shop. Each move someone makes gets logged like checking out an item, tossing it into their basket, starting checkout, or actually buying it.

This dataset you can find free on Kaggle - usually goes by the name:

The E-Commerce Events Dataset (2019–2020) captures extensive user activity from a web shop, logging each step visitors take while browsing - like checking out items, putting them in the basket, starting checkout, or finishing a buy. Because of this, it works well for initiatives focused on how users act, tracking sales pathways, building suggestion engines, or doing instant data checks across heavy-duty systems such as Databricks or BigQuery.

This dataset can be found free on Kaggle, usually called:

A solid base for tasks tackling how users act, tracking sales paths, suggesting items, yet enabling live number-crunching across heavy-duty setups such as Databricks or BigQuery.

Data Source and Size

- **Platform:** Kaggle
- **Link:** Search *"E-Commerce Events History in Electronics Store"* on Kaggle
- **Data Volume:** Around **60–70 million rows**, covering roughly **1 year** of events
- **File Format:** CSV (compressed)
- **Total Size:** ~8-9 GB uncompressed

Schema and Features

Column Name	Description
event_time	Timestamp of the event (UTC format)
event_type	Type of event (view, cart, remove_from_cart, purchase)
product_id	Unique identifier for the product
category_id	Encoded category identifier
category_code	Hierarchical category label (e.g., electronics.smartphone)
brand	Product brand (e.g., Samsung, Apple, etc.)
price	Product price at the time of the event
user_id	Unique user identifier
user_session	Session ID, all actions within one browsing session

---