Uncovering Databricks Costs: How We Identified Expensive and Idle Clusters to Optimize Our Bill

Anurag Sharma 6 min read · Jul 14, 2025

As a data engineer at an analytics company, cost optimization is my secret weapon. Completing daily tasks doesn't earn me that extra hike, but finding ways to save the company money? That's where I can shine. At our organization, Databricks is a core part of our data platform, powering everything from ad-hoc queries to production jobs. But with great power comes a hefty bill. I noticed our Databricks costs creeping up, and I wasn't about to let unnecessary expenses go unchecked.

My mission: figure out which clusters and jobs were burning cash, especially those sitting idle, and summarize the costs to drive smarter decisions.

In this blog, I'll share how I tackled Databricks cost issues, dug into the root causes, and built reports to pinpoint expensive and idle resources. This is the first part of our cost optimization journey.

The Problem: A Growing Databricks Bill

Our team relies heavily on Databricks for analytics, ETL pipelines, and machine learning workloads. But as our usage grew, so did our cloud bill. I suspected two culprits:

1. Long-Running or Idle Clusters: Developers spinning up all-purpose clusters for ad-hoc analysis, then forgetting to shut them down.

2. Costly Jobs: Production jobs running on oversized clusters or with inefficient configurations, racking up Databricks Units (DBUs) and cloud VM costs.

Without visibility into which clusters or jobs were driving costs, we were flying blind. I needed to understand Databricks' cost structure and break down expenses by job, cluster, and time of day to identify patterns.

Step 1: Understanding Databricks Costs

Databricks bills based on DBUs (Databricks Units), which measure compute usage, plus cloud provider costs (e.g., Azure VMs or AWS EC2 instances). There are two main compute types:

• Jobs Compute: Dedicated clusters for scheduled jobs, typically cheaper per DBU and easier to attribute costs.

• All-Purpose Compute: Shared clusters for interactive queries, development, and analysis, which are pricier and harder to tie to specific tasks.

The catch? All-purpose clusters often stay active longer than needed, especially when developers leave them running overnight or over weekends. And without granular cost tracking, we couldn't tell which jobs or teams were responsible for the biggest chunks of our bill.

To get answers, I turned to Databricks' system tables and APIs, which provide detailed metadata on jobs, clusters, and usage. My goal was to calculate costs at a job and cluster level, including runtime and idle time, and summarize them by team and time of day.

Step 2: Investigating Costs with System Tables

Databricks' system.lakeflow and system.billing schemas (available since mid-2024) were a goldmine for cost analysis. These tables let me query job runtimes, cluster details, and DBU consumption directly with SQL. Here's how I approached it:

Data Sources

> *• system.lakeflow.job_run_timeline: Tracks job run durations, including start and end times.*
>
> *• system.lakeflow.job_task_run_timeline: Maps job runs to clusters, helping identify which clusters were used.*

> *• system.compute.clusters: Provides cluster metadata, like node type and DBU rate.*
>
> *• system.billing.usage: Records DBU consumption by job, cluster, and compute type.*

Building the Cost Table

I created a Delta table to centralize cost data, with columns for:

> *• Job Name: The name of the job or run.*
>
> *• Owner Team: The team responsible (extracted from job metadata or tags).*
>
> *• Job Runtime: Total runtime in hours.*
>
> *• DBU Cost: DBUs consumed multiplied by our DBU price (from our Azure contract).*
>
> *• Cloud Cost: Estimated VM costs based on node type and runtime.*
>
> *• Total Cost: DBU cost + cloud cost.*

Here's a simplified query to calculate job costs:

```
WITH job_runs AS (
  SELECT
  workspace_id,
  run_id,
  job_id,
  run_name,
  SUM(period_end_time - period_start_time) / 1000.0 / 3600.0 AS run_hours
  FROM system.lakeflow.job_run_timeline
```

```
  WHERE run_type = 'SUBMIT_RUN'
  AND period_start_time > CURRENT_TIMESTAMP() - INTERVAL 30 DAYS
  GROUP BY ALL
),
task_clusters AS (
  SELECT
  workspace_id,
  run_id,
  EXPLODE(compute_ids) AS cluster_id
  FROM system.lakeflow.job_task_run_timeline
  WHERE array_size(compute_ids) > 0
),
clusters AS (
  SELECT
  workspace_id,
  cluster_id,
  node_type_id,
  dbu_rate
  FROM system.compute.clusters
  QUALIFY ROW_NUMBER() OVER (PARTITION BY workspace_id, cluster_id ORDER BY chan
),
usage AS (
  SELECT
  workspace_id,
  usage_metadata.job_id,
  usage_metadata.run_id,
  SUM(usage_quantity) AS total_dbus
  FROM system.billing.usage
  WHERE usage_type = 'JOBS_COMPUTE'
  AND usage_start_time > CURRENT_TIMESTAMP() - INTERVAL 30 DAYS
  GROUP BY ALL
)
SELECT
  jr.run_name AS job_name,
  'Team_' || jr.job_id AS owner_team,  - Replace with actual team mapping
  jr.run_hours,
  u.total_dbus * AS dbu_cost,
  jr.run_hours * AS cloud_cost,  - Replace with VM price per node type
  (u.total_dbus * + jr.run_hours * ) AS total_cost
FROM job_runs jr
LEFT JOIN task_clusters tc ON jr.workspace_id = tc.workspace_id AND jr.run_id =
LEFT JOIN clusters c ON tc.workspace_id = c.workspace_id AND tc.cluster_id = c.c
LEFT JOIN usage u ON jr.workspace_id = u.workspace_id AND jr.run_id = u.run_id
WHERE u.total_dbus IS NOT NULL
INTO my_cost_db.job_costs;
```

• I used a placeholder and (from our Azure pricing) to calculate costs.

• For team ownership, I mapped job IDs to teams using a lookup table (we tagged jobs with team names in Databricks).

• VM costs were estimated by mapping node types (e.g., Standard_DS3_v2) to Azure pricing.

Identifying Idle Clusters

To find idle all-purpose clusters, I queried system.compute.clusters for clusters with no recent job or task activity but still running:

```
SELECT
  c.cluster_id,
  c.node_type_id,
  c.cluster_name,
  c.state,
  c.dbu_rate * * (CURRENT_TIMESTAMP() - c.last_activity_time) / 1000.0 / 3600.0
FROM system.compute.clusters c
LEFT JOIN system.lakeflow.job_task_run_timeline t
  ON c.cluster_id = c.cluster_id
  AND t.start_time > CURRENT_TIMESTAMP() - INTERVAL 24 HOURS
WHERE c.state = 'RUNNING'
  AND t.run_id IS NULL
  AND c.last_activity_time < CURRENT_TIMESTAMP() - INTERVAL 2 HOURS;
```

This revealed clusters costing us a lot daily because they were left active after developers finished their work.

Step 3: Summarizing Costs by Time and Time of Day

To spot patterns, I built reports summarizing costs by date and hour of the day. This helped identify peak usage periods and times when idle clusters were most prevalent.

Reports

Daily Cost Breakdown:

```
SELECT
 date_trunc('day', usage_start_time) AS usage_day,
 owner_team,
 SUM(total_cost) AS daily_cost
 FROM my_cost_db.job_costs
 GROUP_BY usage_day, owner_team;
```

This showed which teams drove costs on specific days, highlighting spikes (e.g., heavy ETL jobs).

Hourly Cost by Hour:

```
SELECT
 hour(date_trunc('hour', usage_start_time) AS usage_hour,
 usage_type,
 SUM(total_dbus * usage_cost ) AS hourly_dbu_cost
 FROM system.billing_usage.usage
```

```
    WHERE usage_start_time > CURRENT_TIMESTAMP() - INTERVAL 30 DAYS
    GROUP_BY usage_hour, usage_type;
```

This revealed that all-purpose compute costs peaked during work hours (9 AM – 6 PM) but stayed high overnight due to idle clusters.

Visualizing Insights

I used Databricks SQL to create dashboards with these queries, showing:

> • *Top 10 Most Expensive Jobs: Sorted by total cost, with team names.*
>
> • *Idle Cluster Costs: Highlighted clusters with no activity, tagged by owner.*
>
> • *Cost by Hour Trends: A line chart showing DBU and cloud costs by hour, exposing nighttime idle usage.*

These reports were shared with team leads and developers, clearly indicating where costs originated.

Step 4: Key Findings

Our investigation uncovered several eye-openers:

> Idle Clusters Were a Major Culprit: All-purpose clusters left running overnight or over weekends cost us ~15% of our monthly bill. Some were forgotten for weeks!

> Oversized Clusters for Jobs running on large clusters (e.g., 16 nodes) when smaller ones (e.g., 4 nodes) would suffice, especially for lightweight ETL tasks.
>
> Team-Level Insights: One team's frequent ad-hoc analysis on all-purpose clusters accounted for 35% of all-purpose compute costs.
>
> Nighttime Spikes: Jobs scheduled at odd hours (e.g., 3 AM) used expensive clusters unnecessarily, as developers didn't adjust configurations for off-peak workloads.

Takeaways

If you're struggling with Databricks costs, start by:

1. Leveraging System tables (system.lakeflow and system.billing) to track job runtimes and DBUs.

2. Building a Cost table to summarize expenses by job, team, and time.

3. Monitoring on Idle Clusters: They're silent budget killers.

4. Creating Dashboards to visualize cost patterns and share with stakeholders.

Cost optimization isn't just about cutting corners – it's about making data-driven decisions work that add value. For me, it's a way to stand out in an analytics company where every dollar saved counts.