Taming Data Skew in Production ML Pipelines

Practical Apache Spark Optimization Techniques

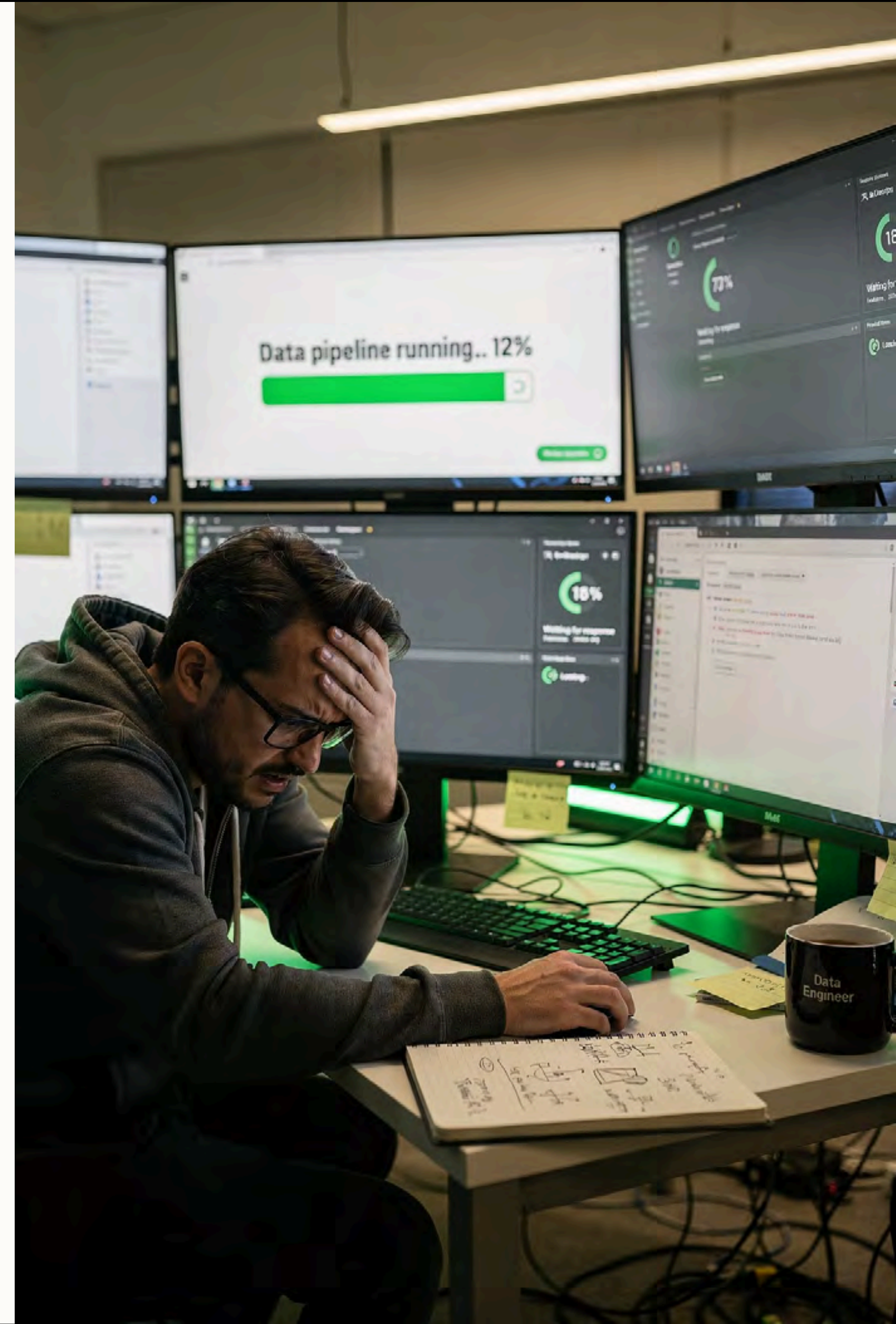
Srihari Babu Godleti
Roku Inc., USA

The Problem: Why Your ML Models Train Slowly

ML pipelines are plagued by a hidden performance killer that turns minutes into hours. Training jobs extend indefinitely, model deployments get delayed, and cloud costs burn through budgets.

The culprit? Data Skew.

This isn't about bad code—it's about uneven data distribution across partitions silently sabotaging your pipeline efficiency.



What is Data Skew?

THE UNBALANCED WORKLOAD PROBLEM

Ideal Distribution

- Work evenly spread across all executors
- All workers processing similar data volumes
- Parallel processing at full efficiency
- Predictable completion times

Reality with Skew

- Some executors process gigabytes while others handle megabytes
- Entire job waits for the slowest executor
- "Straggler" problem kills performance
- Resources sit idle while one worker struggles

Impact on ML Systems

REAL-WORLD CONSEQUENCES

Pipeline Performance

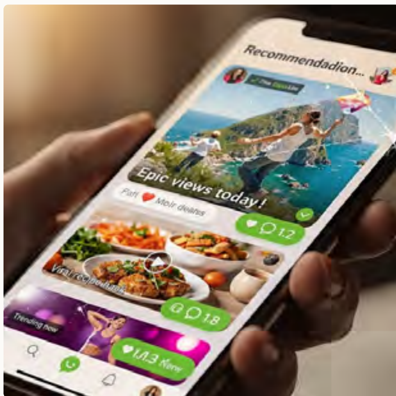
- Feature engineering bottlenecks
- Extended model training times
- Delayed experimentation cycles
- Reduced model freshness

Business Impact

- Higher cloud infrastructure costs
- Slower time-to-market for models
- Reduced data science productivity
- Compromised SLA compliance

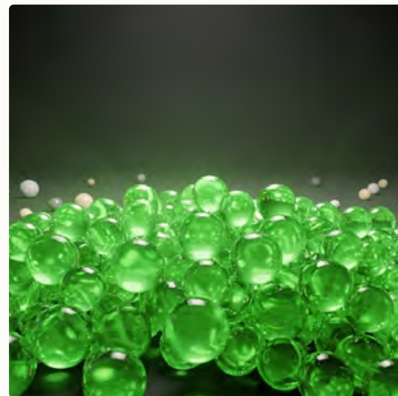
Where Skew Hides in ML Pipelines

⚠ COMMON SCENARIOS



Recommendation Systems

Power users generate the majority of interactions. Popular items dominate training data, and viral content creates processing hotspots that overwhelm single partitions.



Classification Models

Imbalanced datasets with majority classes create natural skew. Certain categories contain disproportionate examples, causing uneven executor workloads.

Computer Vision

Uneven distribution across image classes combined with variable processing time for different image sizes creates both data and computational skew.

Root Cause #1: Natural Imbalances

REAL-WORLD DISTRIBUTION PATTERNS

Training data naturally reflects real-world patterns where user behavior follows power-law distributions, customers concentrate geographically, and seasonal patterns create temporal imbalances.

1

Skewed Source Data

Product popularity, user activity

2

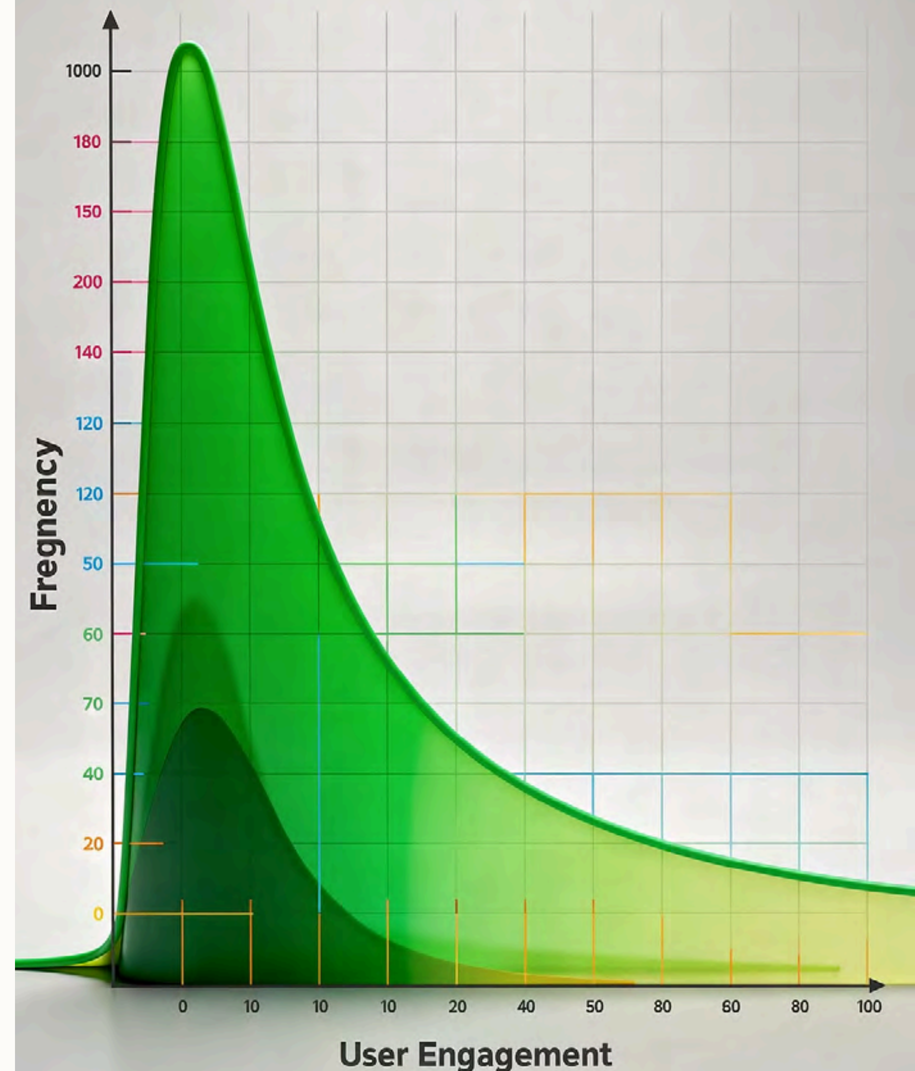
Partition Imbalance

Hot keys dominate partitions

3

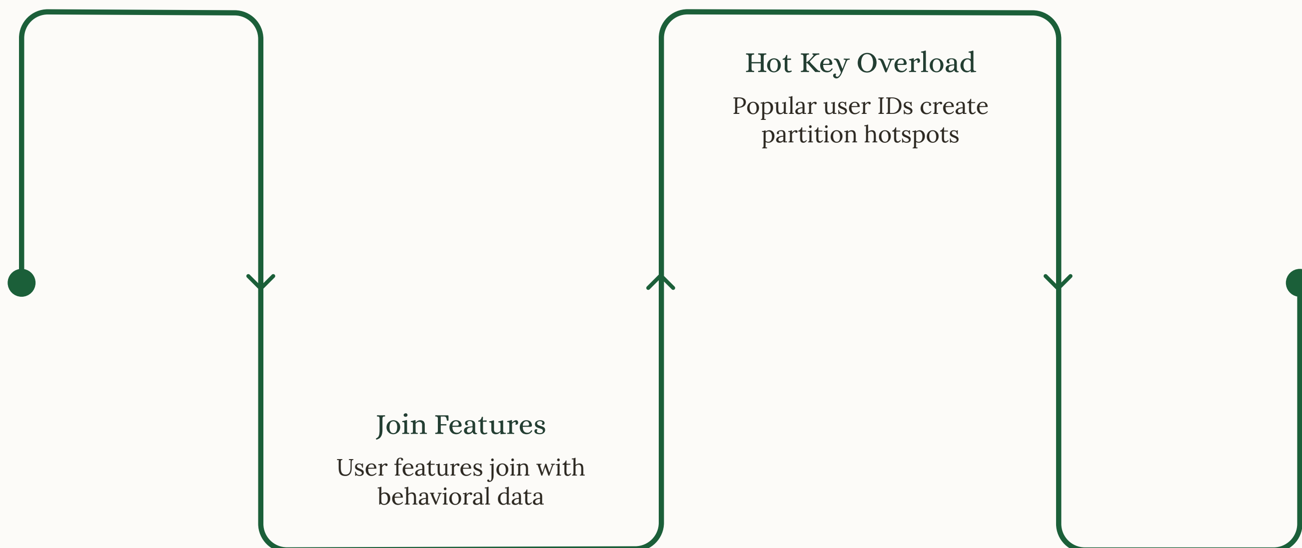
Feature Store Propagation

Problems inherit downstream



Root Cause #2: Join & Aggregation Skew

FEATURE ENGINEERING BOTTLENECKS



Common Scenarios

Join Key Skew: User features combined with behavioral data create concentrated workloads when certain user IDs dominate the dataset.

Aggregation Key Skew: Operations like groupByKey and reduceByKey suffer when some keys have massive value counts, especially in temporal aggregations where daily or weekly summaries vary dramatically.

Root Cause #3: Computational Skew

PROCESSING COMPLEXITY VARIATION

Even perfectly balanced data distribution creates unbalanced workload when transformation complexity varies significantly across records.

Text Vectorization

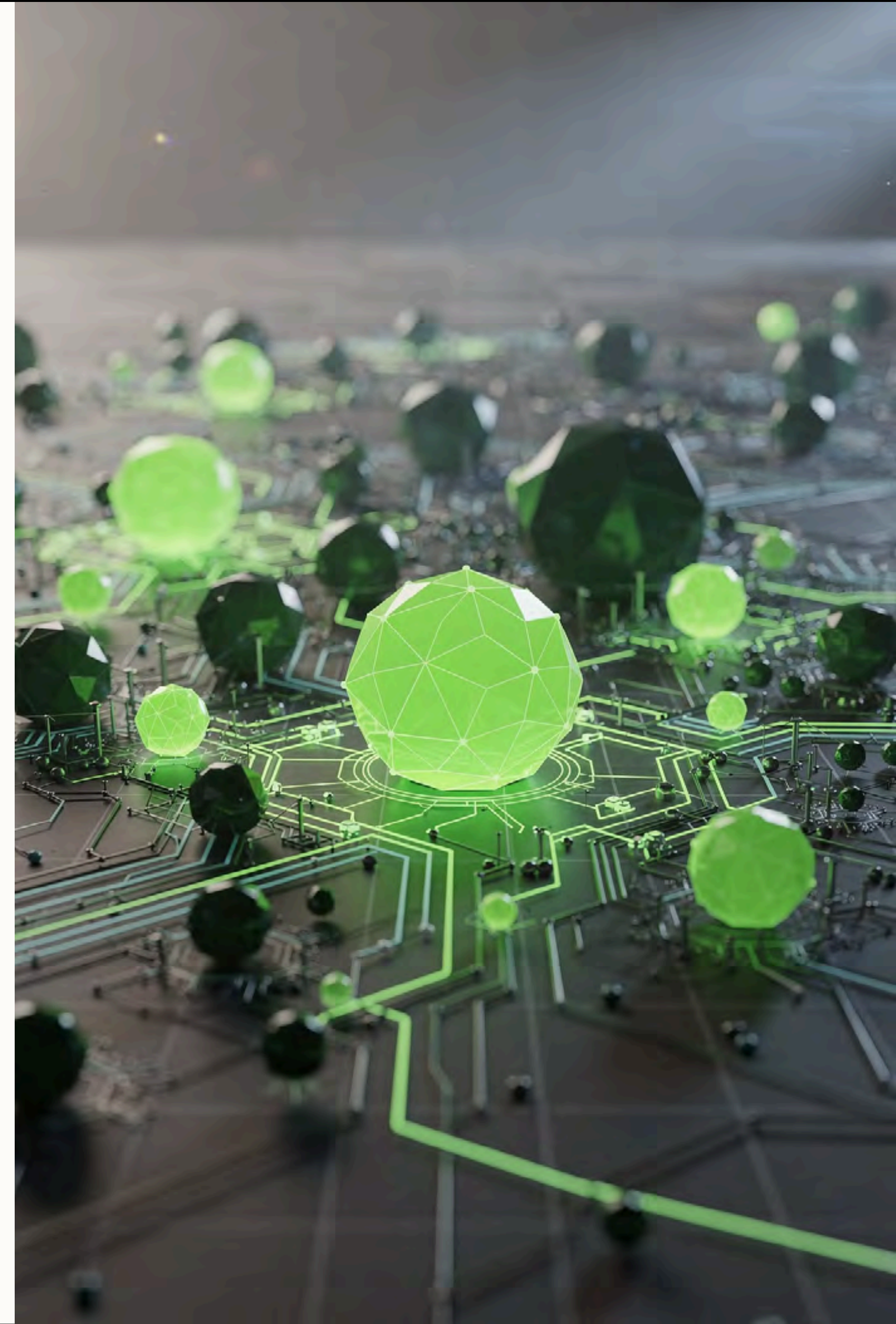
NLP models process variable-length sequences requiring vastly different computation time per document

Feature Engineering

Complex transformations on categorical data with high cardinality create uneven processing loads

Embedding Generation

Deep learning embeddings vary in computation cost based on input characteristics and model complexity



Solution #1: Repartitioning

The Concept

Explicitly redistribute data across the cluster by specifying partition count and column. This forces Spark to rebalance the workload through hash partitioning.

Best For

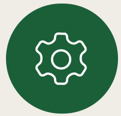
- Moderately skewed datasets
- Multi-stage ML pipelines
- Feature engineering workflows
- Training data preprocessing

Implementation

```
# Repartition by user_id for balanced distribution
df_repartitioned = df.repartition(100, "user_id")

# Process features with better parallelism
df_features = df_repartitioned \
 .groupBy("user_id") \
 .agg(collect_list("interaction"))
```

Repartitioning: When and How to Apply



Optimal Settings

- Partition count: 2-3× total core count
- Partition size: 100-200MB sweet spot
- Balance parallelism with overhead

Limitations

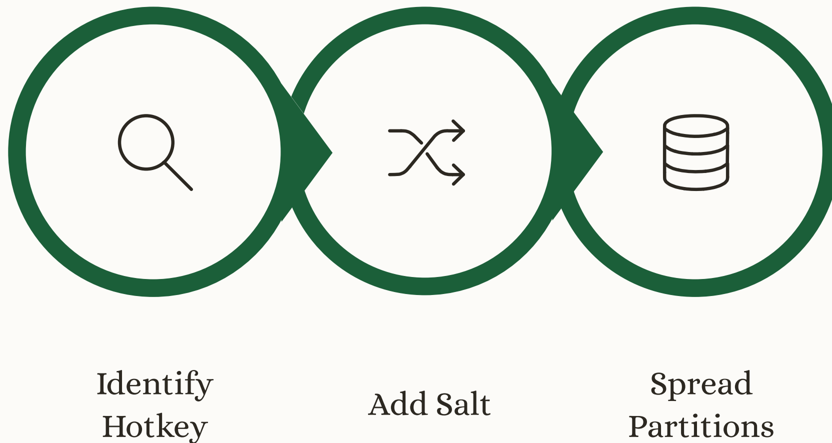
- Introduces shuffle operation cost
- Hash partitioning doesn't change key frequency
- Ineffective for extreme skew scenarios

Performance Gains

- Moderate improvements for typical cases
- Most effective in feature pipeline stages
- Best when amortized across operations

Solution #2: Key Salting

BREAKING UP HOTSPOTS



Implementation

```
# Add salt to distribute power users
df_salted = df.withColumn(
  "user_id_salted",
  concat(
    col("user_id"),
    lit("_"),
    (rand() * 10).cast("int")
  )
)
```

Best For

- Severe skew with identifiable hotspots
- Recommendation systems with power users
- Imbalanced classification datasets

Key Salting: Choosing the Right Salt Factor

Low Salt (2-4)

For moderate skew scenarios with few dominant users or items. Simple implementation with minimal overhead.

Medium Salt (8-16)

Handles severe skew with clear hotspots like viral content or popular items while maintaining balanced partition sizes.

High Salt (32+)

Reserved for extreme skew with single dominant values. Requires complex aggregation logic and risks creating excessive small partitions.

Adaptive Salting

Dynamic approach for changing skew patterns in online learning systems. Requires monitoring framework to adjust salt factor based on observed distribution.

Solution #3: Broadcast Joins

OPTIMIZING ASYMMETRIC FEATURE ENGINEERING

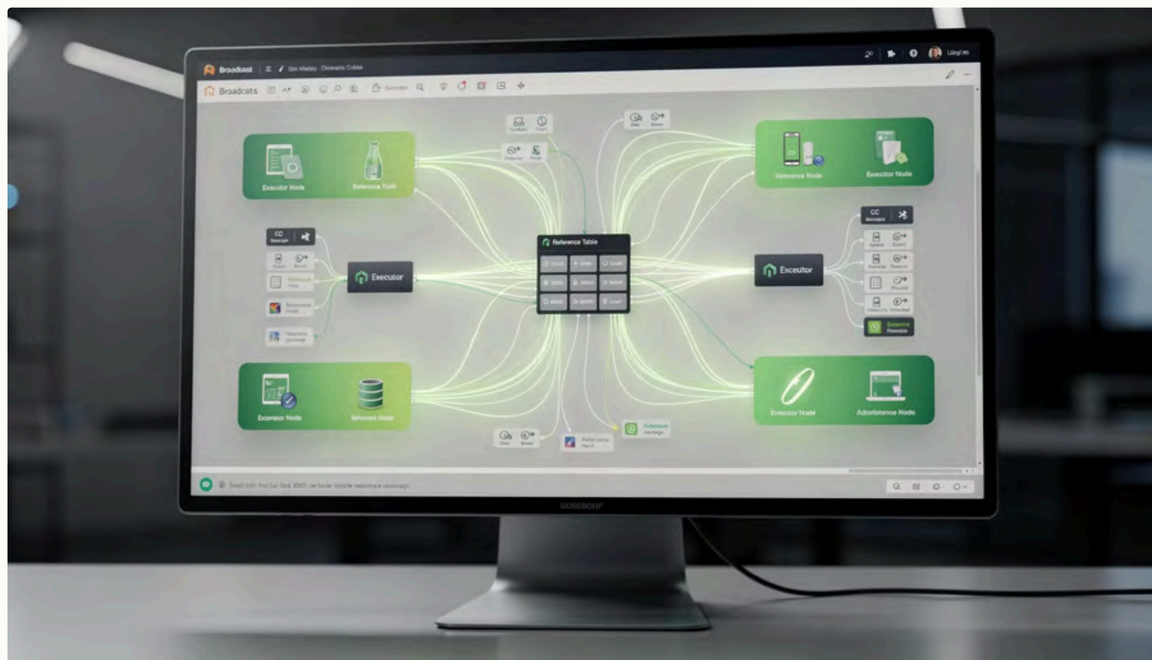
The Concept

Send the small table to all executors so joins are performed locally on each node. This eliminates the shuffle of your large dataset entirely.

Implementation

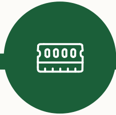
```
# Broadcast small dimension table
from pyspark.sql.functions import
broadcast

df_joined = large_behavioral_data.join(
  broadcast(small_user_features),
  "user_id"
)
```



Broadcast Joins: Best Practices

MEMORY MANAGEMENT & PERFORMANCE

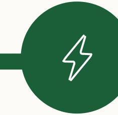


Size Guidelines

Keep broadcast tables under 10GB (typical limit). Use maximum 20-25% of executor memory to avoid GC pressure during training.

When to Use

Significant size disparity between tables, multiple executors need same small dataset, or feature engineering with dimension tables.



Performance Impact

Eliminates network shuffle bottleneck. Scales with table size difference. Critical for maintaining model freshness in production systems.

Best for joining large behavioral data with small dimension tables in star schema feature stores, enriching user demographics with interaction history, or adding item metadata.

Key Takeaways & Action Plan

IMPLEMENTING IN YOUR ML PIPELINES

01

Match Technique to Skew Pattern

Repartitioning for moderate skew in multi-stage pipelines, Key Salting for severe skew with identifiable hotspots, Broadcast Joins for asymmetric feature engineering.

02

Measure and Monitor

Monitor partition metrics in existing pipelines, identify skew patterns in training data, and establish baseline performance benchmarks.

03

Apply Selectively

Implement appropriate techniques targeting specific bottlenecks rather than applying everywhere. Measure impact on training velocity before and after.

04

Reap the Benefits

Expect faster model training and iteration, reduced cloud infrastructure costs, improved resource utilization, and better model freshness with faster deployment velocity.

Thank You