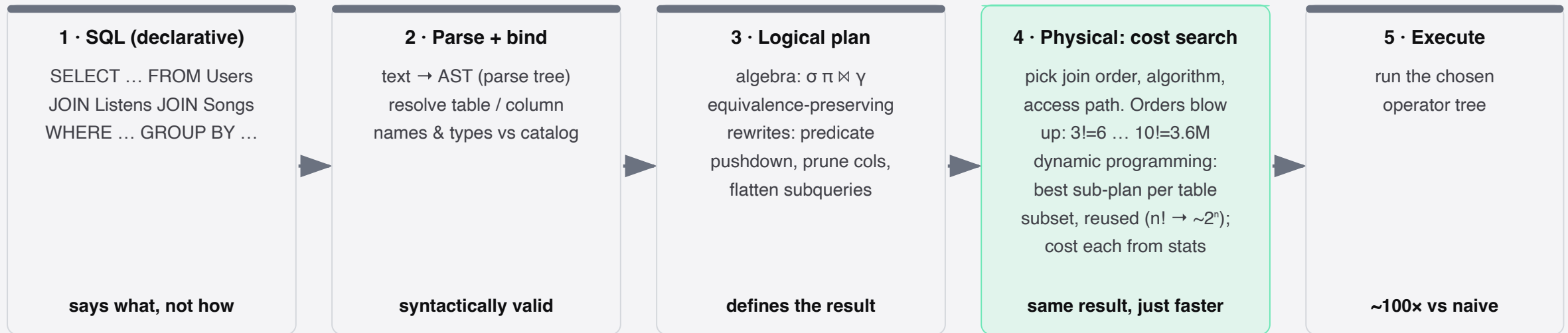


M3

Query Optimizer & Statistics

The optimizer estimates the cost of alternative plans and chooses the cheapest.

Choose a cheaper plan for the same answer



Preserve the query's result; choose how to compute it using estimated costs.

M3

IO Cost Summary: Comparing Query Plans

Add up each plan's I/O costs.

One query: join, group, then sort

```
-- Count recommendation-listen matches for each song.  
SELECT r.song_id, COUNT(r.recommendation_id) AS recommendation_count  
FROM Recommendations AS r  
-- Goal 1: match the same user and song.  
JOIN Listens AS l ON r.user_id = l.user_id AND r.song_id = l.song_id  
-- Goal 2: count matching pairs per song.  
GROUP BY r.song_id  
-- Goal 3: show the largest counts first.  
ORDER BY COUNT(r.recommendation_id) DESC;
```

Which listens match a recommendation?

Recommendations

recommendation_id	user_id	song_id
10	1	497
11	2	497
12	1	88
13	3	75
...	...	...

Listens

listen_id	user_id	song_id
1	1	497
2	1	497
3	2	497
4	3	75
5	2	88
...	...	...

Plan 1: Sort-Merge Join

Goal 0 · Input tables

T1 Recommendations (R)

T2 Listens (L)

T1, T2, ... identify tables. P(T3) is the number of pages in table T3.

Operation	Output table	I/O cost
Goal 1 · Join (T1) and (T2) Match (song_id, user_id)	T3 Joined rows Sorted by (song_id, user_id) One row per matching pair	SMJ(T1, T2)
Goal 2 · Group by song_id Read (T3); count matches per song	T4 Counts per song song_id, recommendation_count One row per matched song	$C_r \times P(T3) + C_w \times P(T4)$
Goal 3 · Sort (T4) by count Largest count first	Final result song_id, recommendation_count Descending recommendation_count	BigSort(T4)
Total cost	$SMJ(T1, T2) + C_r \times P(T3) + C_w \times P(T4) + BigSort(T4)$	

Sort-merge join leaves rows in song order, so grouping needs no repartitioning.

Plan 2: Hash-Partition Join

Goal 0 · Input tables

T1 Recommendations (R)

T2 Listens (L)

T1, T2, ... identify tables. P(T3) is the number of pages in table T3.

Operation	Output table	I/O cost
Goal 1 · Join (T1) and (T2) Match (song_id, user_id)	T3 Joined rows Partitioned by (song_id, user_id) One row per matching pair	HPJ(T1, T2)
Goal 2 · Group by song_id Repartition (T3) by song_id	T3b Repartitioned joined rows All rows for a song in one partition	HP(T3)
Read (T3b); count matches per song	T4 Counts per song song_id, recommendation_count	$C_r \times P(T3b) + C_w \times P(T4)$
Goal 3 · Sort (T4) by count Largest count first	Final result song_id, recommendation_count Descending recommendation_count	BigSort(T4)
Total cost	$HPJ(T1, T2) + HP(T3) + C_r \times P(T3b) + C_w \times P(T4) + BigSort(T4)$	

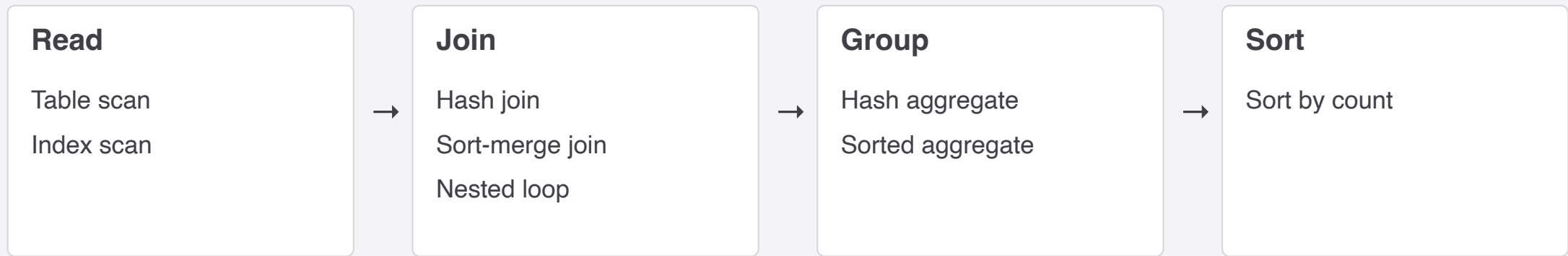
Hashing by (song_id, user_id) requires a repartition by song_id in this plan.

Choose the plan with the smaller total cost

- **Plan 1:** $\text{SMJ}(T1, T2) + C_r \times P(T3) + C_w \times P(T4) + \text{BigSort}(T4)$
- **Plan 2:** $\text{HPJ}(T1, T2) + \text{HP}(T3) + C_r \times P(T3b) + C_w \times P(T4) + \text{BigSort}(T4)$

Do the math: calculate both total costs and choose the lower-cost plan.

Combine operators into a plan



Choose the lowest estimated total cost.

For the recommendation query, choose access methods, a join, an aggregation method, and a final sort by count. Partition or sort only where an operator needs it. Compare the complete plans using estimated rows, pages, memory, and existing order.

M3

Reading PostgreSQL EXPLAIN

EXPLAIN shows PostgreSQL's chosen plan and estimated costs.

What PostgreSQL actually chose

Same recommendation query

Sort (cost=2.38..2.39 rows=3 width=12)

Sort Key: (count(r.recommendation_id)) DESC

-> HashAggregate (cost=2.32..2.35 rows=3 width=12)

Group Key: r.song_id

-> Hash Join (cost=1.10..2.31 rows=3 width=8)

Hash Cond: ((l.user_id = r.user_id) AND (l.song_id = r.song_id))

-> Seq Scan on listens l (cost=0.00..1.09 rows=9 width=8)

-> Hash (cost=1.04..1.04 rows=4 width=12)

-> Seq Scan on recommendations r (cost=0.00..1.04 rows=4 width=12)

Read bottom-up.

rows = estimated output · cost = planner units

Groups fit in RAM: no repartition.

Actual PostgreSQL 17.9 output, using the Query Plans Colab's schema and nine listens plus four added sample recommendations. The original Colab leaves Recommendations empty. After ANALYZE, PostgreSQL chose Hash Join, HashAggregate, and Sort. The groups fit in RAM, so this plan avoids the extra repartition in the hand-worked Plan 2. These tiny-data costs are not predictions for large tables.

Statistics estimate the work

1. **Sample:** inspect a sample of the table (example: 1% of its rows).
2. **Summarize:** keep row and page counts, distinct-value counts, common-value frequencies, and histograms of value ranges.
3. **Store and use:** save these compact summaries in the database's system catalog, organized by table and column. The optimizer uses them to estimate output rows and compare plan costs.

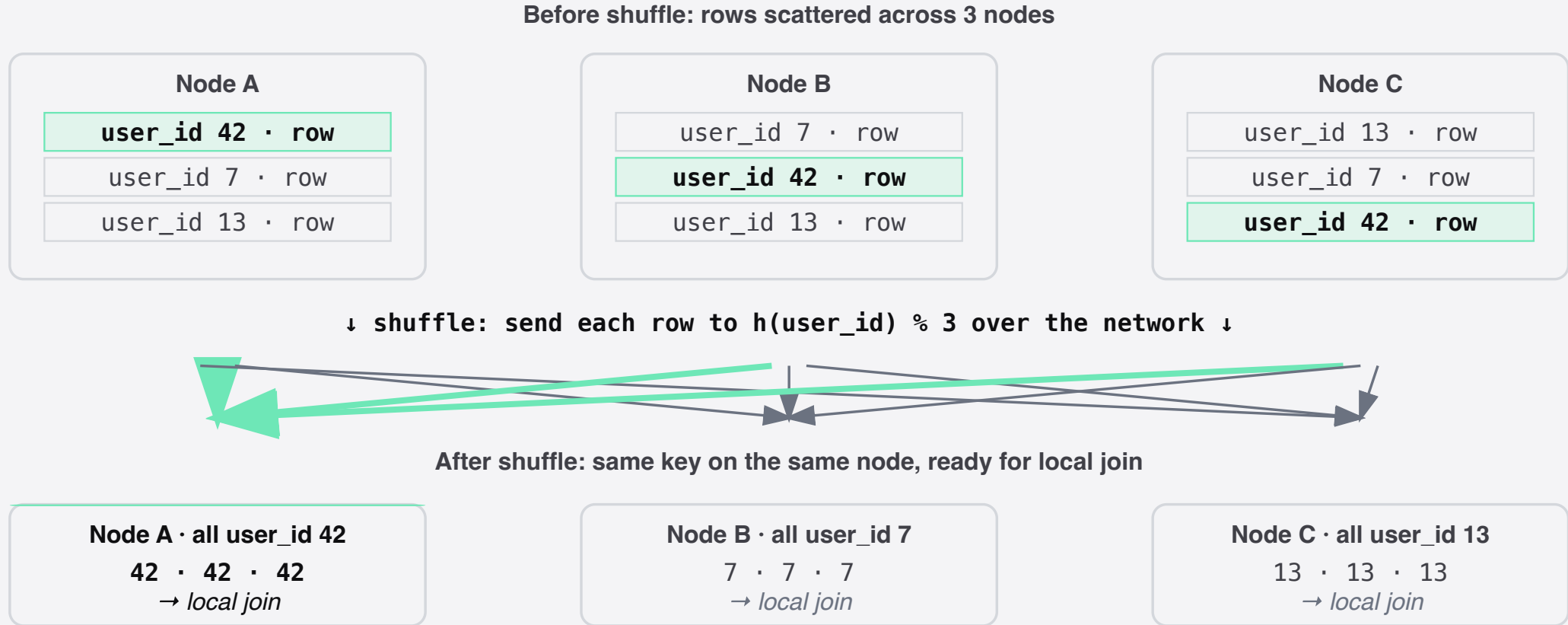
Example: for `WHERE user_id BETWEEN 1 AND 100`, suppose the statistics estimate that 1% of rows match. That is the **selectivity**. With 1 million input rows, the estimated output **cardinality** is $1,000,000 \times 1\% = \mathbf{10,000 \text{ rows}}$.

M3

Distributed Query Planning

Partition the data, run local operators, and shuffle rows
when matching keys must meet.

Send matching keys to the same machine



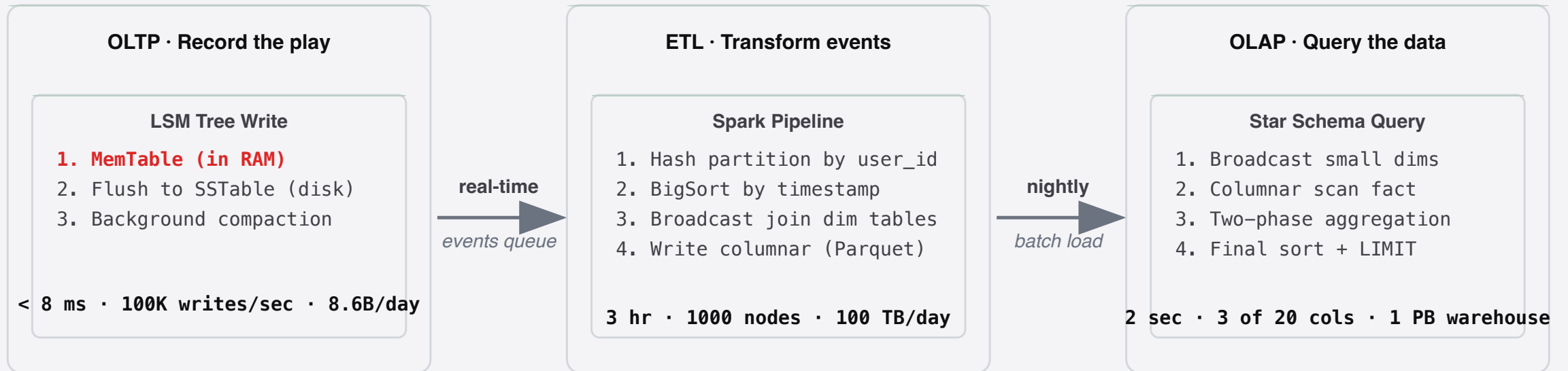
Rows begin scattered across multiple nodes. Hashing each row on the join key picks its target machine, so every row sharing a key travels over the network to the same node. Once the keys sit together, each node runs an ordinary local join on the rows it now owns. This is exactly hash partitioning with the network as the destination, so the cost is the same $2N$ IO plus the network transfer.

M3

Spotify End-to-End: How Everything Connects

Follow a play event through writes, batch processing, and analytics.

One click connects OLTP, ETL, and OLAP



Fast writes → batch processing → efficient scans

LSM writes feed a batch pipeline that produces columnar data for analytics.

M3

Module 3 Capstone: Picking the Right Algorithm

Choose the data structure and algorithm that fit each workload.

Which join algorithm should I use?

1. **Non-equi joins:** use **BNLJ**.
2. **Equi-joins:** use the [equation sheet](#) to calculate **BNLJ**, **HPJ**, and **SMJ** costs. Choose the cheapest.

Do the math.

Choose the key, then the index

Original value

Key to index

Word or user ID



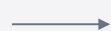
Use the original key

Embedding (e.g., 1 KB)



Full key: exact matching
Or LSH code: similarity candidates

Location coordinates



Spatial code (e.g., geohash)
Nearby-region candidates

Hash index

Exact key or bucket lookup

B+ tree

Exact key lookup or ordered key ranges

For LSH or spatial codes, check candidates' actual similarity or distance.

Ordering an embedding's bytes does not preserve vector proximity.

Smaller keys fit more entries per page.

Update pages or batch writes?

B+ tree / hash index	LSM tree
Update the relevant pages	Buffer updates in memory
Split pages or buckets; reorganize as needed	Flush sorted SSTable files
Scattered page writes can become expensive	Batch writes, then merge files through compaction
Pay for page updates as writes arrive	Reads may check multiple files; compaction adds work

Update pages now, or batch writes and compact later.

Goal 1: Artist-Facing UI for Royalty Stats

Problem: Artists need to track geographical trends and specific date ranges, like post-release impacts. How do you handle fast range queries on trillions of listens?

▼ Show Solution

- **Logic:** Deploy a SQL engine for counting and mapping zip codes.
- **Indexing:** Implement a **B+ Tree Index** on `Listens(song_id, listen_time)`.
- **The "Why":** For a fixed song, `listen_time` is ordered, so a B+ Tree seeks to the first key in the range and then scans forward across adjacent leaves.

Goal 2: Fan Preference Clustering

Problem: Clustering similar users or songs for marketing requires searching through high-dimensional feature vectors. What's the strategy?

▼ Show Solution

- **Concept:** Use Embeddings with Locality Sensitive Hashing (LSH).
- **Design:** Generate embeddings (lyrics, tempo, etc.) and use an **LSH Index** to group similar vectors.
- **The "Why":** High dimensions break the ordering a B+ Tree depends on. LSH maps similar items to the same hash bucket with high probability.

Goal 3: Local Fan Club Meetups

Problem: Identifying hyper-local communities when zip codes fall short. How do you pinpoint zones with over 100 fans for "listen parties"?

▼ Show Solution

- **Concept:** Geo-Hashing (H3).
- **Design:** Map user addresses to **H3 cells**. Execute: `GROUP BY H3(address) HAVING count(user) > 100`.
- **The "Why":** H3 maps latitude and longitude onto a hierarchical grid of cells, so the same data aggregates at whatever zoom the question needs.

Goal 4: "Live" Taylor Swift Drops

Problem: When 50,000 users tune into a live song drop, random page updates can become expensive. How do you manage these updates?

▼ **Show Solution**

- **Concept:** LSM Trees.
- **Design:** Use an **LSM Tree** to handle the write surge, keeping partial counts in a MemTable and flushing to SSTables.
- **The "Why":** LSM Trees convert random writes into sequential flushes, ideal for high-bandwidth telemetry data.

Goal 5: Handling Viral Surges

Problem: Distributing table traffic across a cluster during a viral song surge. What's the approach?

▼ **Show Solution**

- **Concept:** Hash Partitioning.
- **Design:** Partition data across nodes using a **Hash function** on `user_id`.
- **The "Why":** Hashing user IDs spreads different users' events across nodes. Hashing only the viral song ID would put its events on one node; uneven user traffic can still create hot spots.