ORACLE

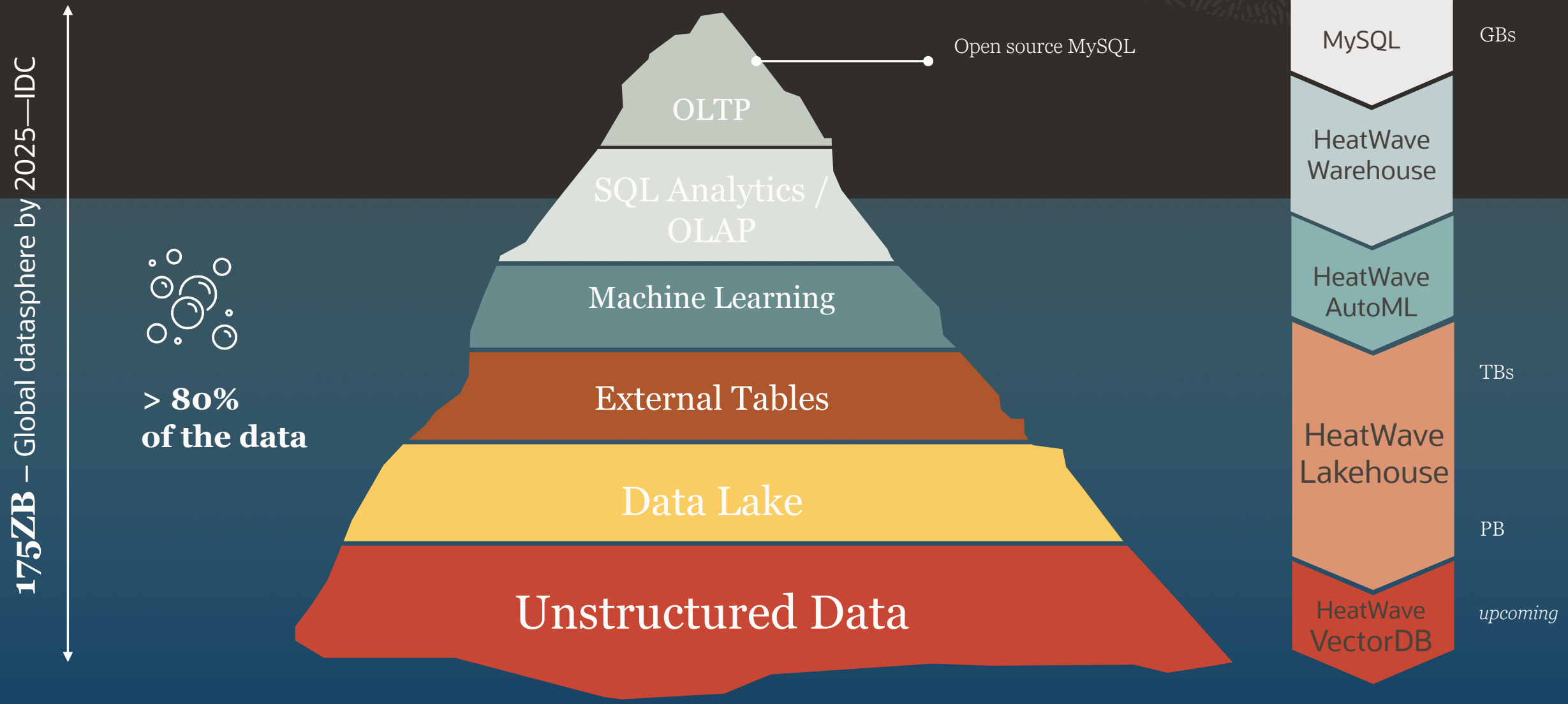
MySQL HeatWave Lakehouse: Scaling Queries to Thousands of Cores

High Performance OLAP on Unstructured Data At Scale

Kaan Kara, PhD

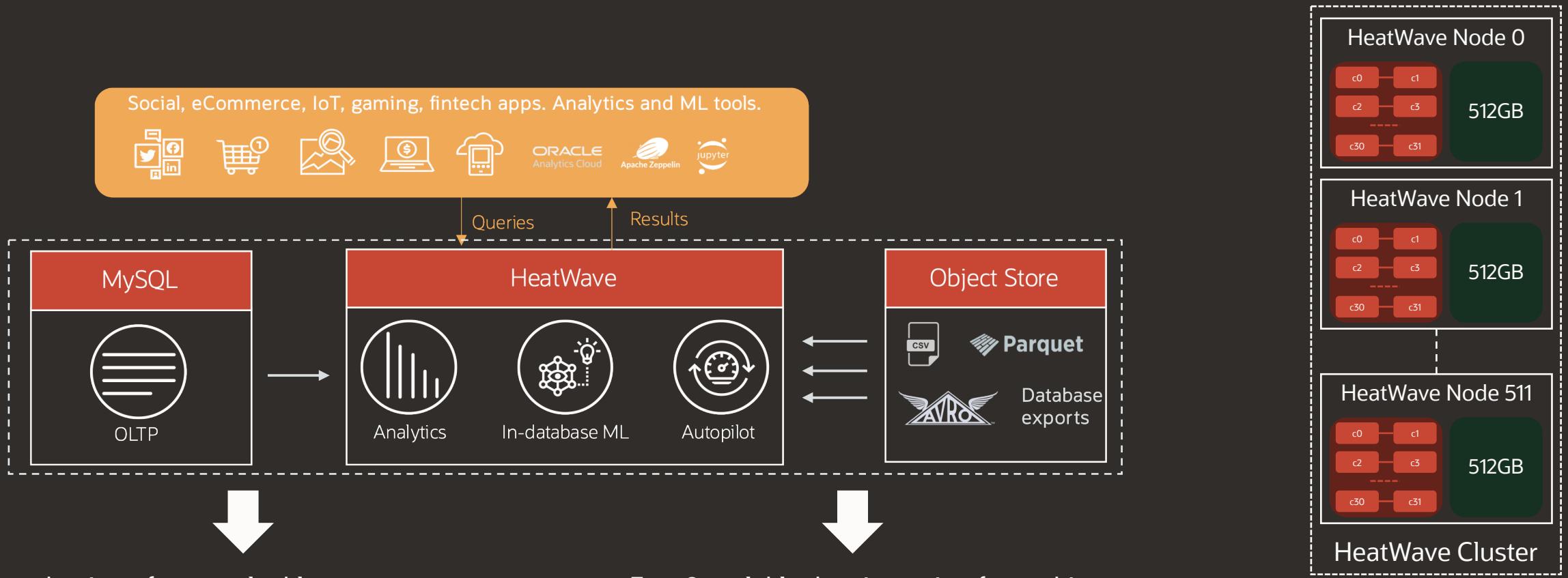
Principal Member of Technical Staff

Unprecedented increase in volume of data



MySQL HeatWave

OLTP, real-time analytics, and ML in one cloud database service – without ETL



262TB memory

- Introduction of external table concept
- HeatWave as the primary engine for data lake
- Automatic schema inference for external tables

- Fast & scalable data ingestion from object store
- Statistics collection at ingestion for query performance
- Scaling query processing to thousands of cores

Lakehouse End-to-End Workflow

1. Create table with primary engine Lakehouse
   - Auto provisioning based on data size and properties
   - Auto schema inference based on data properties

2. Load the table
   - HeatPump reads files from object store
   - It transforms them to HeatWave format, loads to memory

3. Run queries
   - Queries on tables both from InnoDB and Lakehouse

```
CREATE TABLE documents <create_definition> ENGINE=LAKEHOUSE
ENGINE_ATTRIBUTE='{
"location":...,
"file":[...],
"dialect":{"format":csv|avro|parquet,...}
}'
SECONDARY_ENGINE=RAPID;



ALTER TABLE documents SECONDARY_LOAD; -- Lakehouse Table
ALTER TABLE transactions SECONDARY_LOAD; -- InnoDB Table



SELECT * FROM documents, transactions ...
```

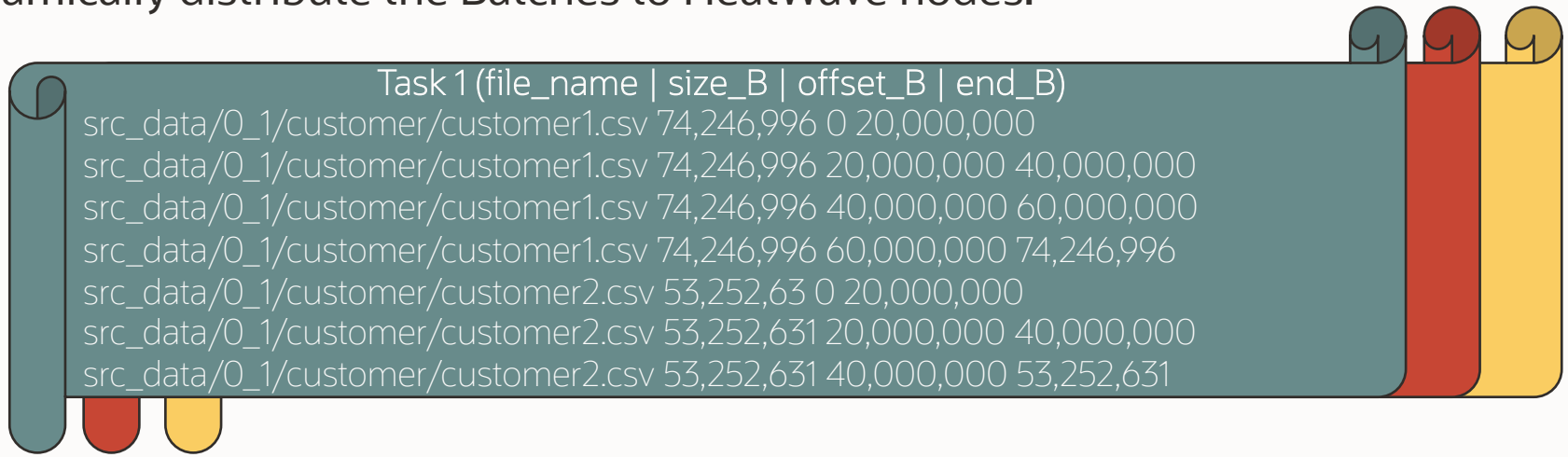
HeatPump: Achieving Scalable & High Performance Ingestion

How should we parallelize ingestion?

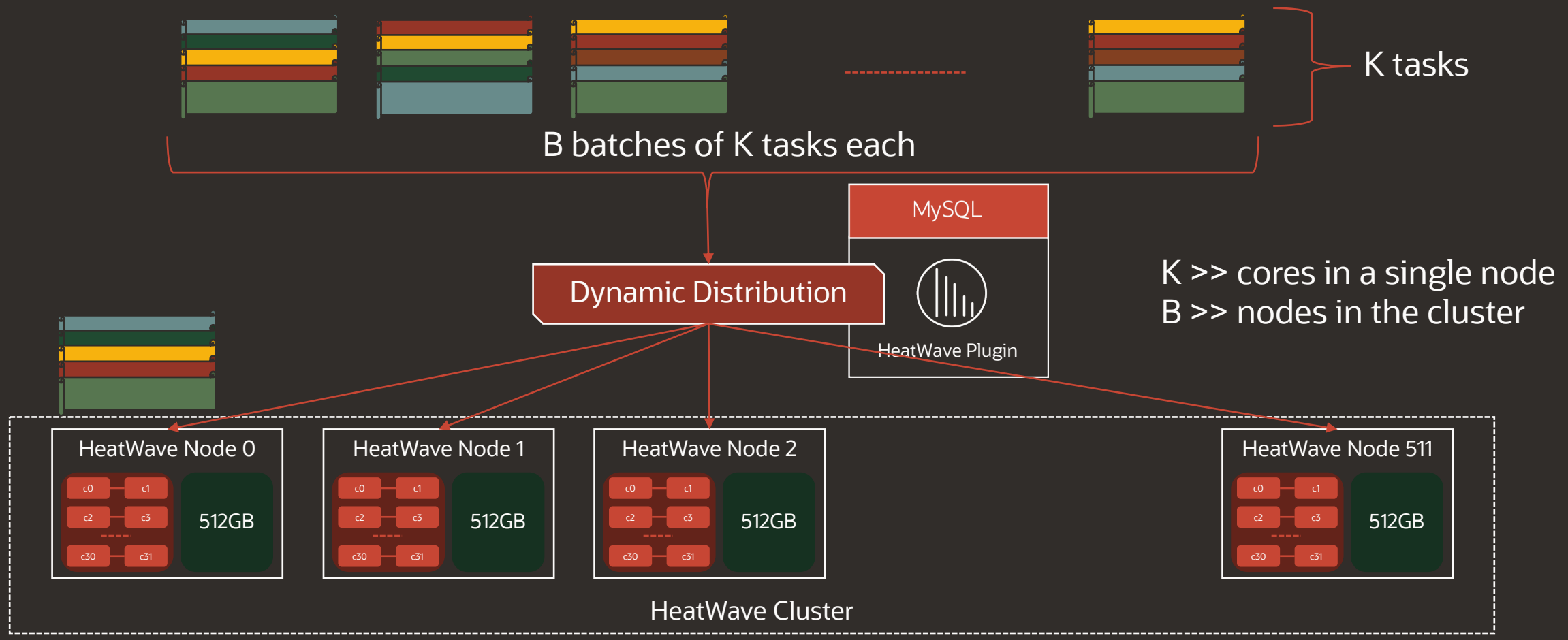
- File-wise distribution across nodes will likely suffer from data skew (file size differences, different data properties etc.)
- We need a custom distribution scheme to ensure robust performance regardless of what is ingested

---

- Create Tasks (~10MB data to process for each thread)
- Create Batches of Tasks (10s of Tasks to process for each node)
- Dynamically distribute the Batches to HeatWave nodes.

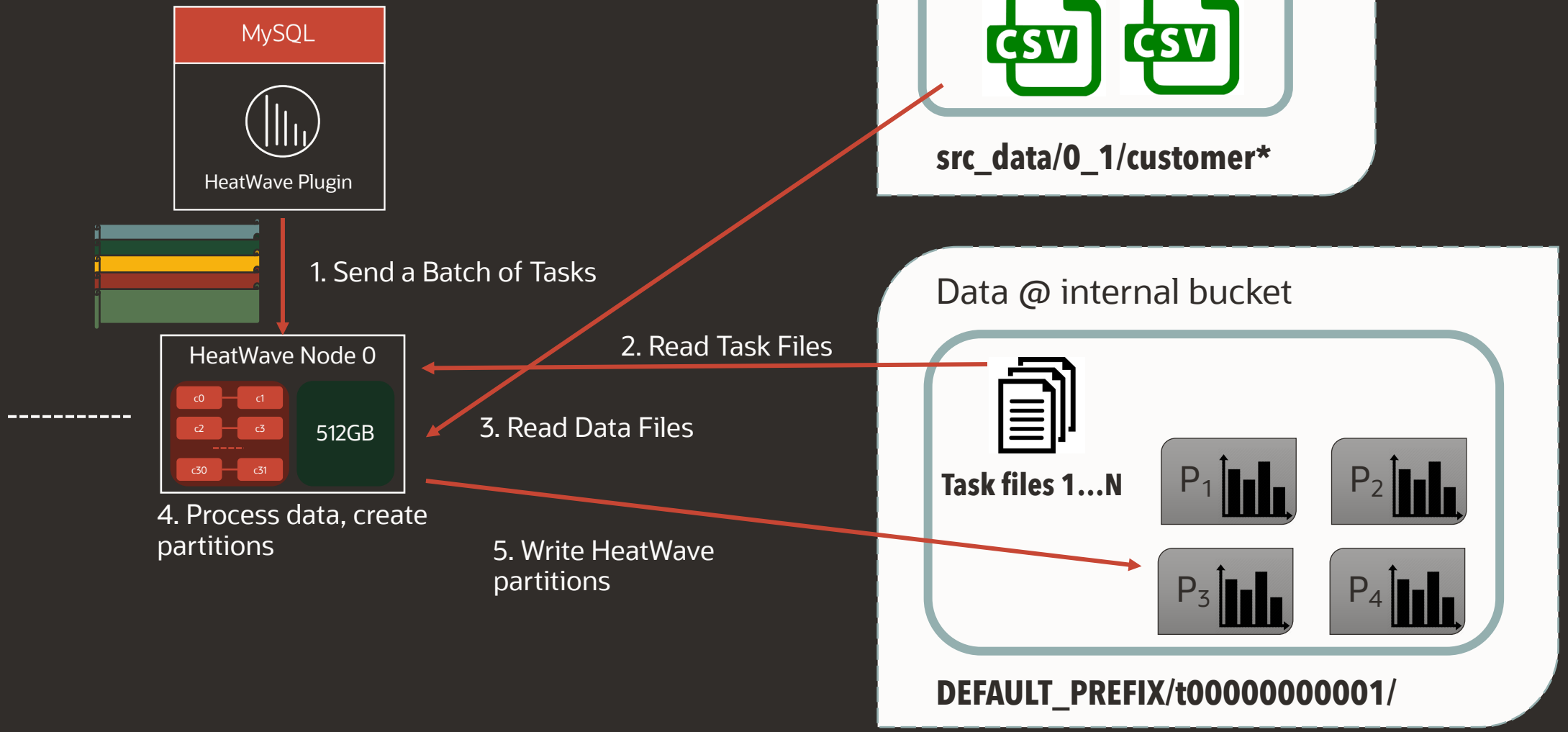


Load Parallelism: Distribute to Nodes Dynamically



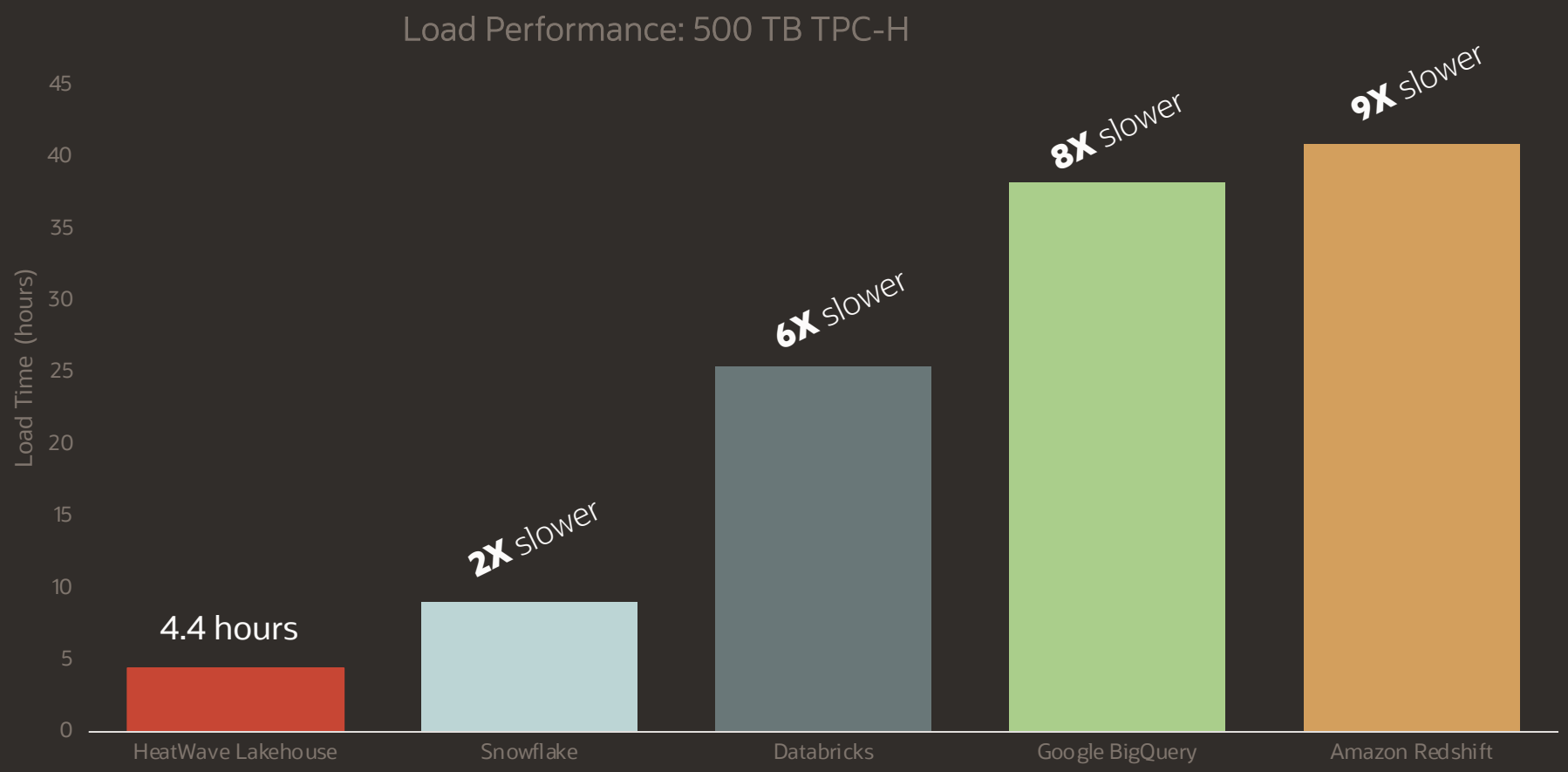
HeatPump Ingestion

Steps in a HeatWave Node



Object Store Ingestion Performance

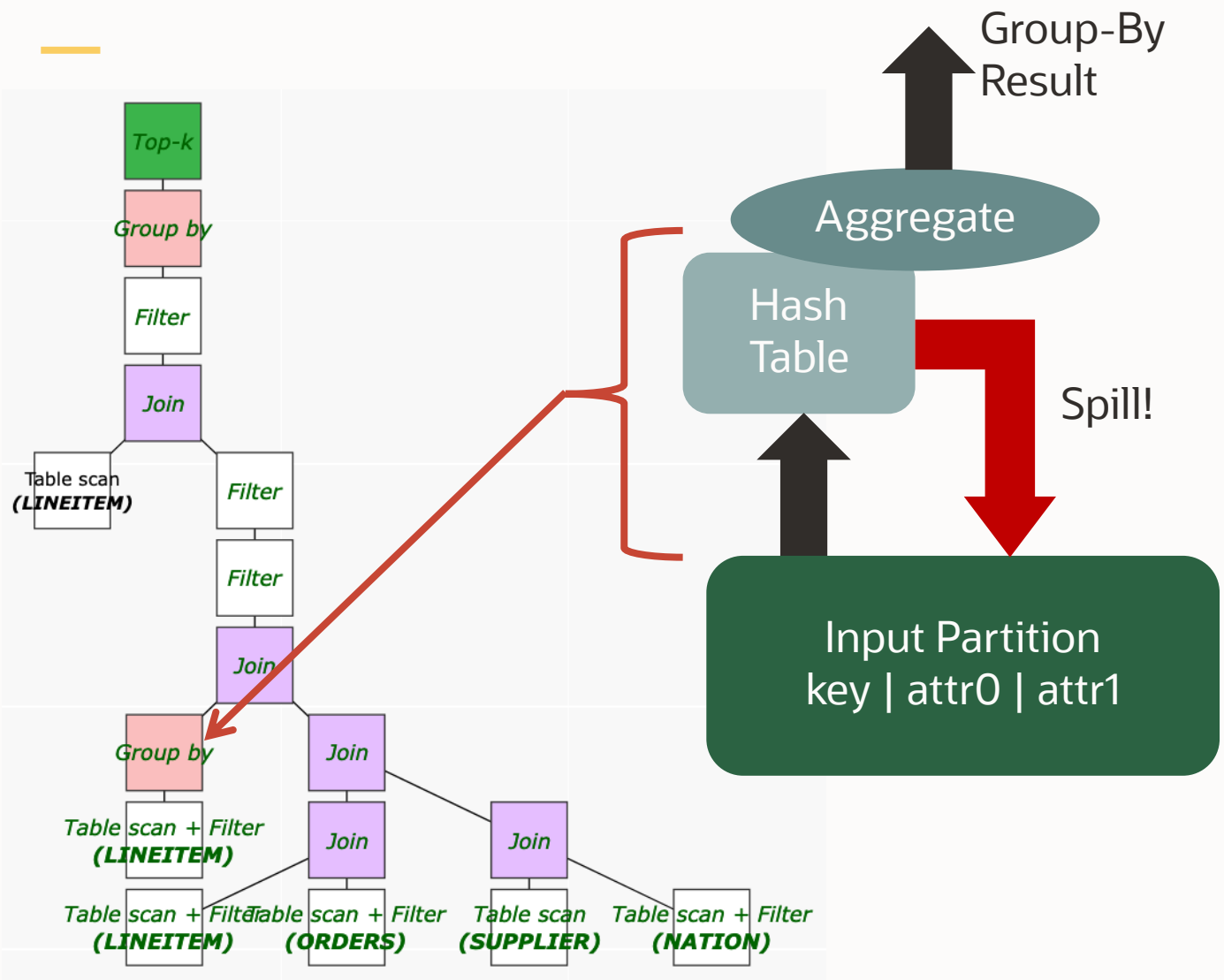
Faster compared to competition



Query Performance: Scaling Relational Processing to Thousands of Cores

Adaptive Execution

Group-By Implementation Details



Main question: **How large should the hash table be?**

- Whenever we have a collision, we spill the data to retry next time.
- In order to avoid spill, HT needs to be sufficiently large.
- If the HT is too large
  - Performance suffers due to bad cache locality
  - Memory consumption becomes problematic

Ideally, the HT should be exactly as large as the number of distinct keys, i.e. number of groups.

Can we get this information at runtime?

Cardinality Estimation At Runtime With HLL

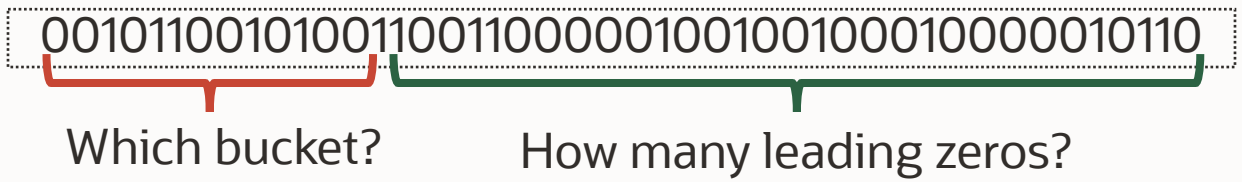
Main intuition: In a set of uniformly hashed values, a sequence of k consecutive zeros in a given hash will occur with a probability of 1/2^k

Example

- **Set1: 0, 1, 0, 1, 0, 1 … => max leading number of zeros is 1: Cardinality estimate 2^1 = 2**
- **Set2: 0, 1, 2, 3…, 128 => max leading number of zeros is 7: Cardinality estimate 2^7 = 128**
- Allows a cardinality estimate with very small memory footprint
- However
  - The estimate is only in powers of 2
  - The variance of the estimate is very high

HyperLogLog

1. Hash each item to obtain a X-bit hash
2. Use b bits to determine a bucket
3. Use the rest (X-b) bits to get the max leading number of zeros
4. Update the current max in the bucket
5. At the end, get a harmonic mean of each bucket's estimation

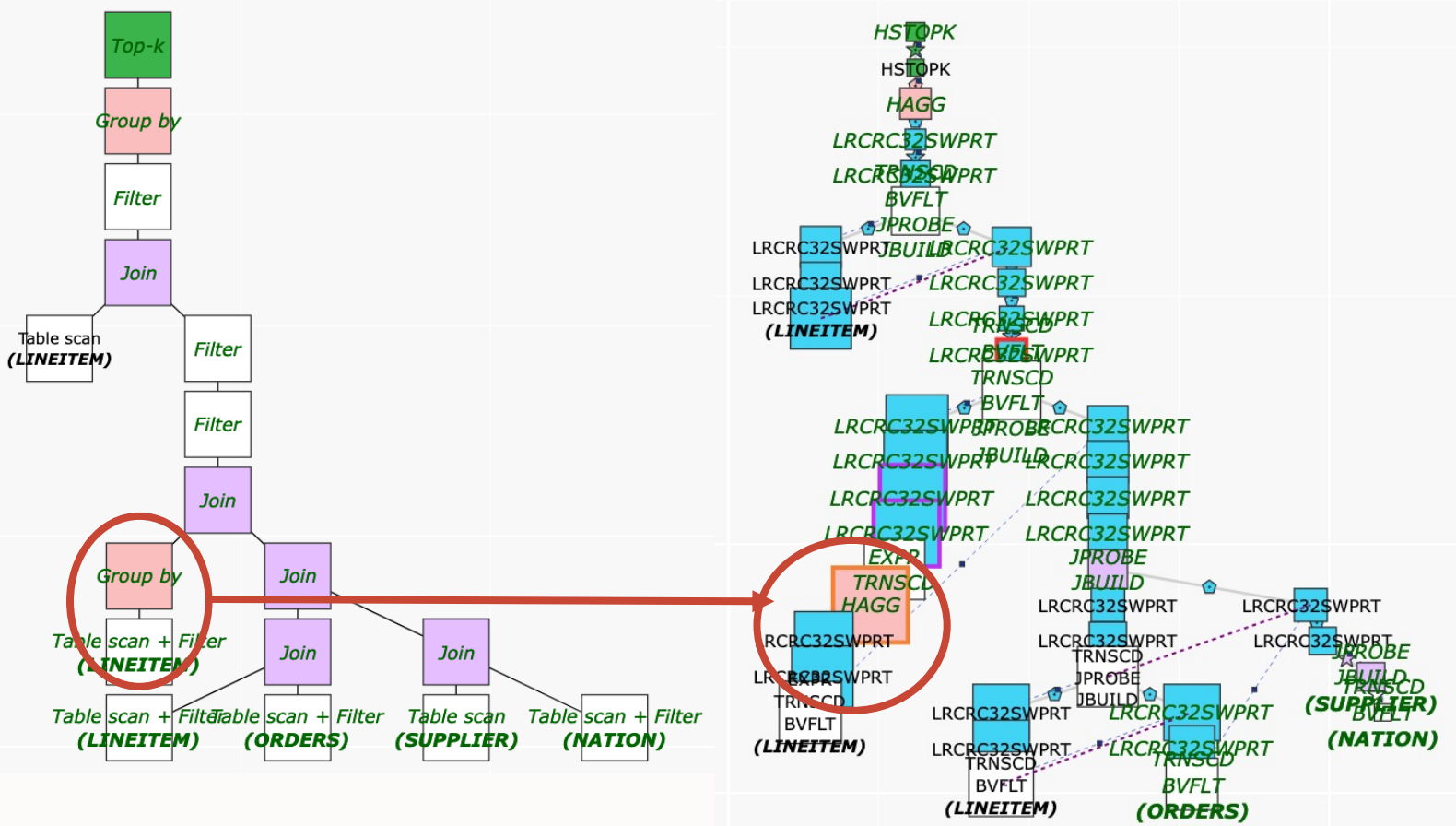


Cardinality Estimation At Runtime

Utilizing HyperLogLog

Logical Plan

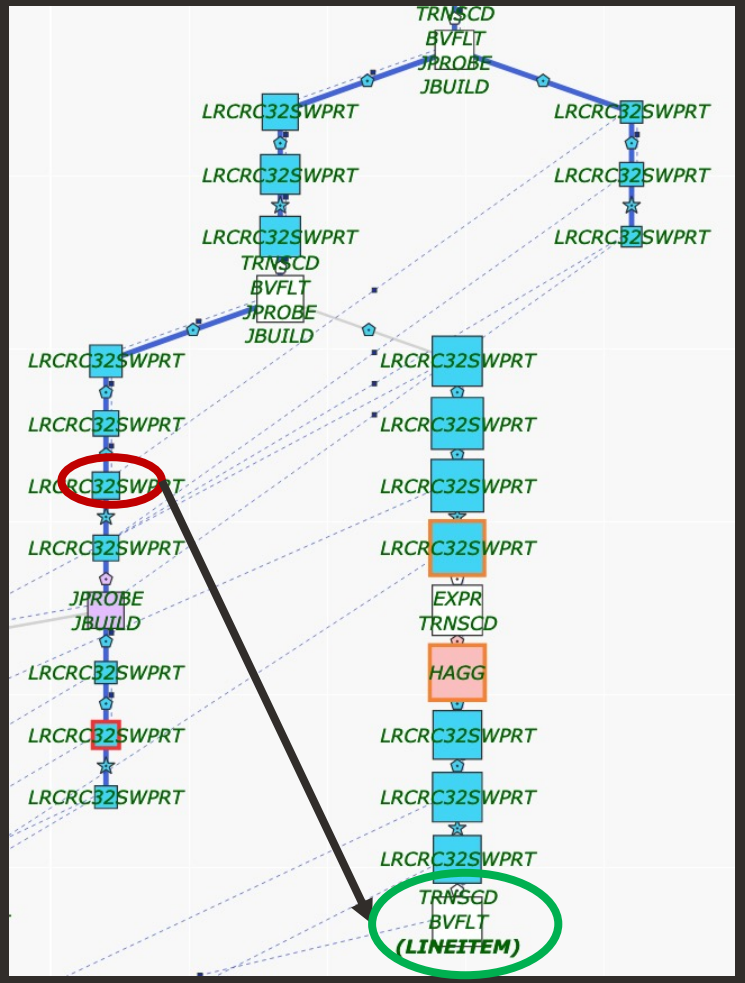
Physical Plan

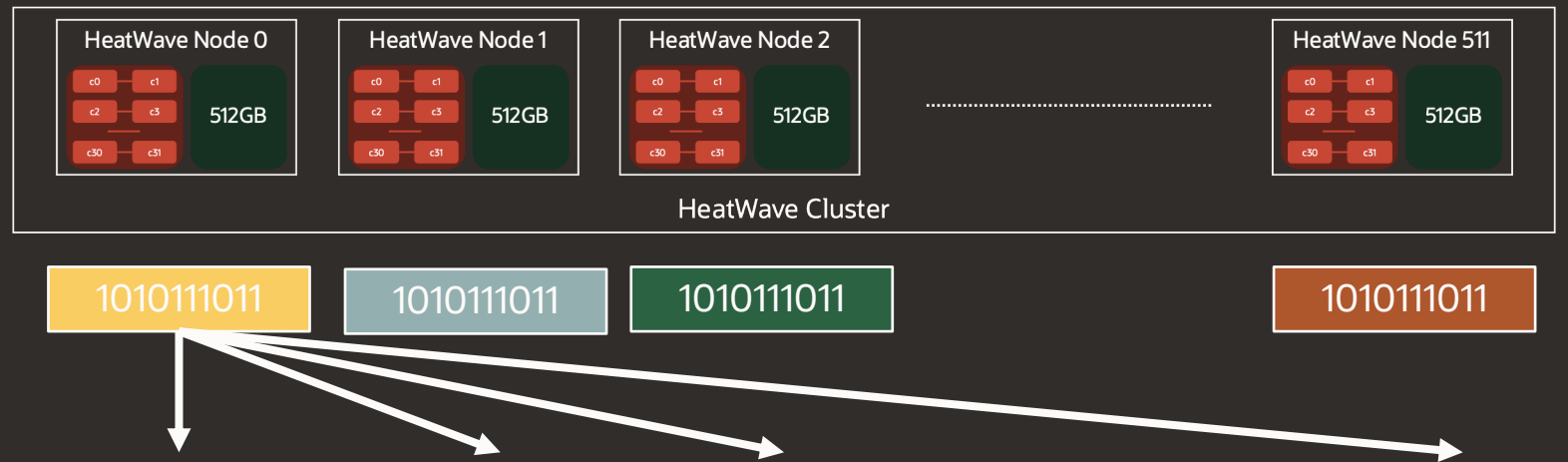


During Physical Plan Compilation

- Inject an HLL estimator at the partitioning operator below group-by.
- HLL will build the cardinality for each partition and the hash table will be sized accordingly.
- HLL estimation (mainly due to hashing) has some overhead => cost based decision to inject (e.g., if we have stats cache).

Bloom Filter Sharing

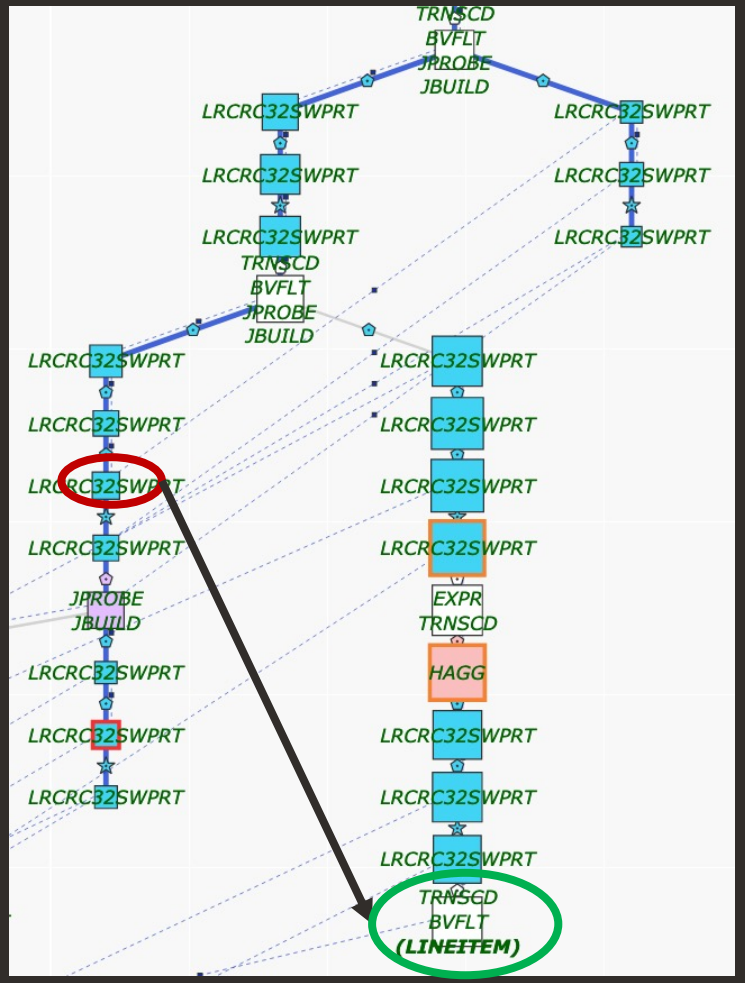


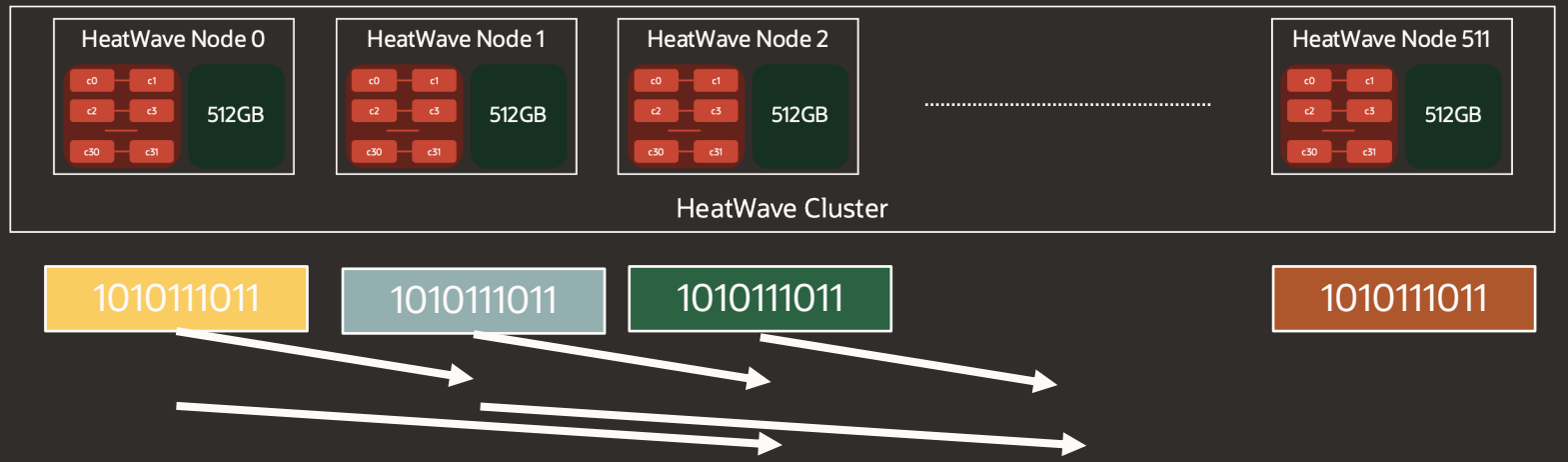


BCAST Cost (time): 2 * BF_size * #nodes / network_bandwidth

→ 2 * 4GB * 512 / 1GB/s = 68 minutes

Bloom Filter Sharing (Binary Merge)





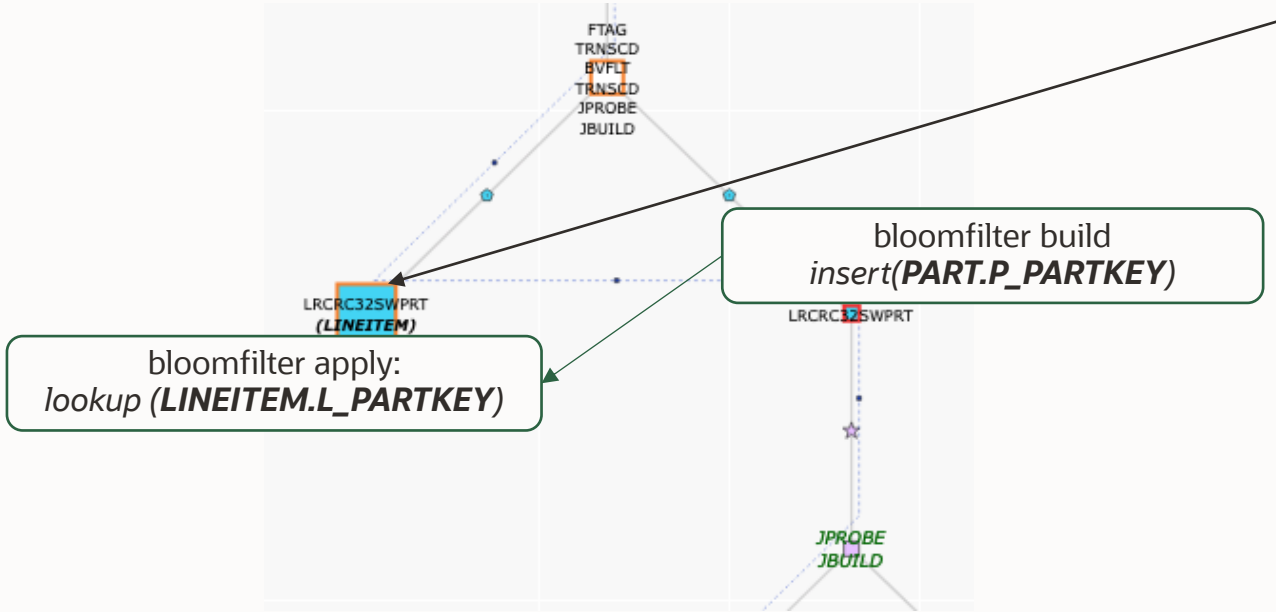
At each round, send to neighbor 2^r % N

- Round 0 (r = 0): 1 % N -th neighbor
- Round 1 (r = 1): 2 % N -th neighbor
- Round 2 (r =2): 4 % N -th neighbor
- …

Cost (time): 2 * BF_size * log2(#nodes) / network_bandwidth

→ 2 * 4GB * 9 / 1GB/s = 72 seconds

Late Decompression
With Bloom Filter



This node projects 3 columns:

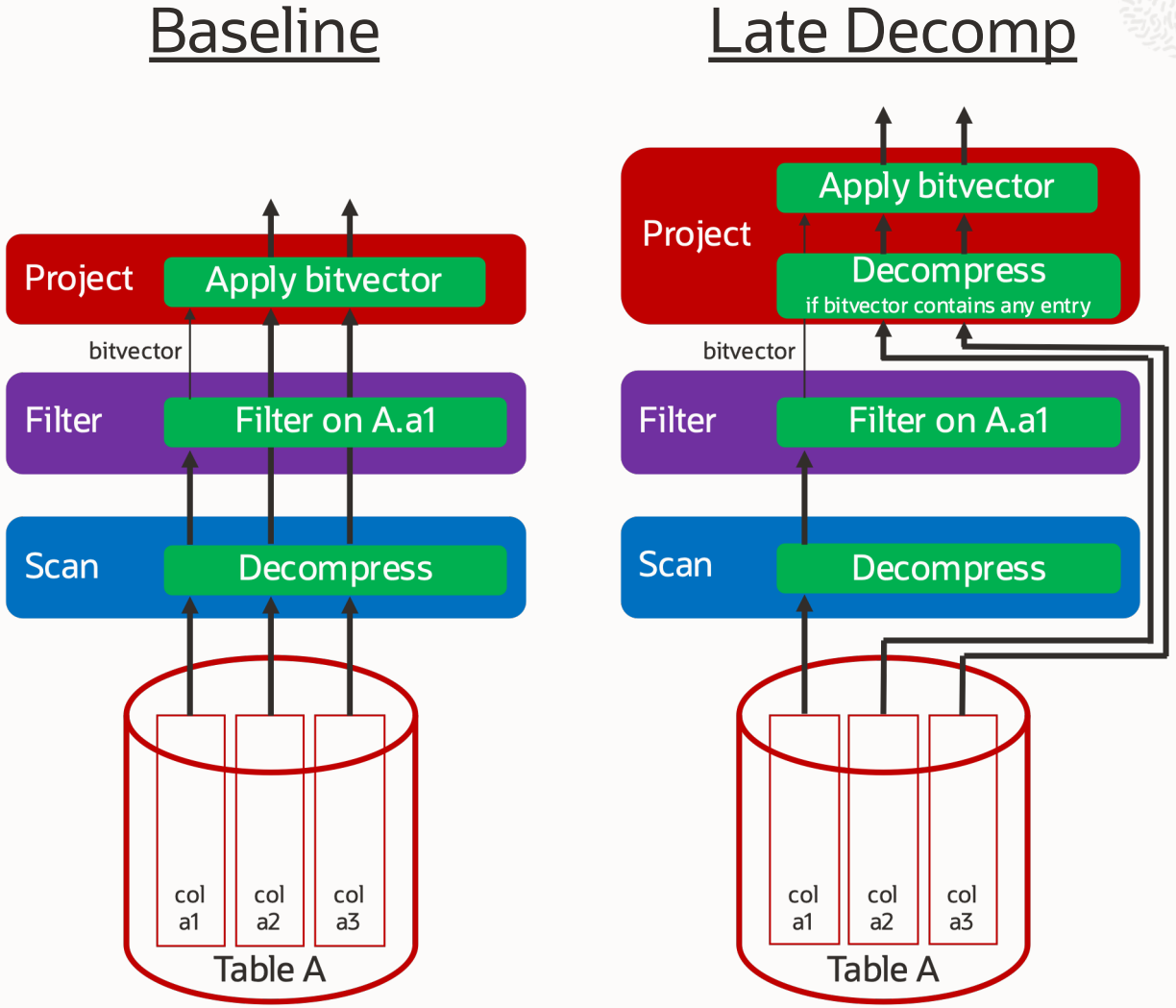
1. L_PARTKEY (4 Bytes)
2. L_EXTENDEDPRICE (8 Bytes)
3. L_QUANTITY (8 Bytes)

- Only L_PARTKEY needs to be decompressed early for the bloomfilter evaluation.
- The other two columns can be late decompressed.
- If the bloomfilter eliminates entire blocks of data, decompression can be skipped for them!

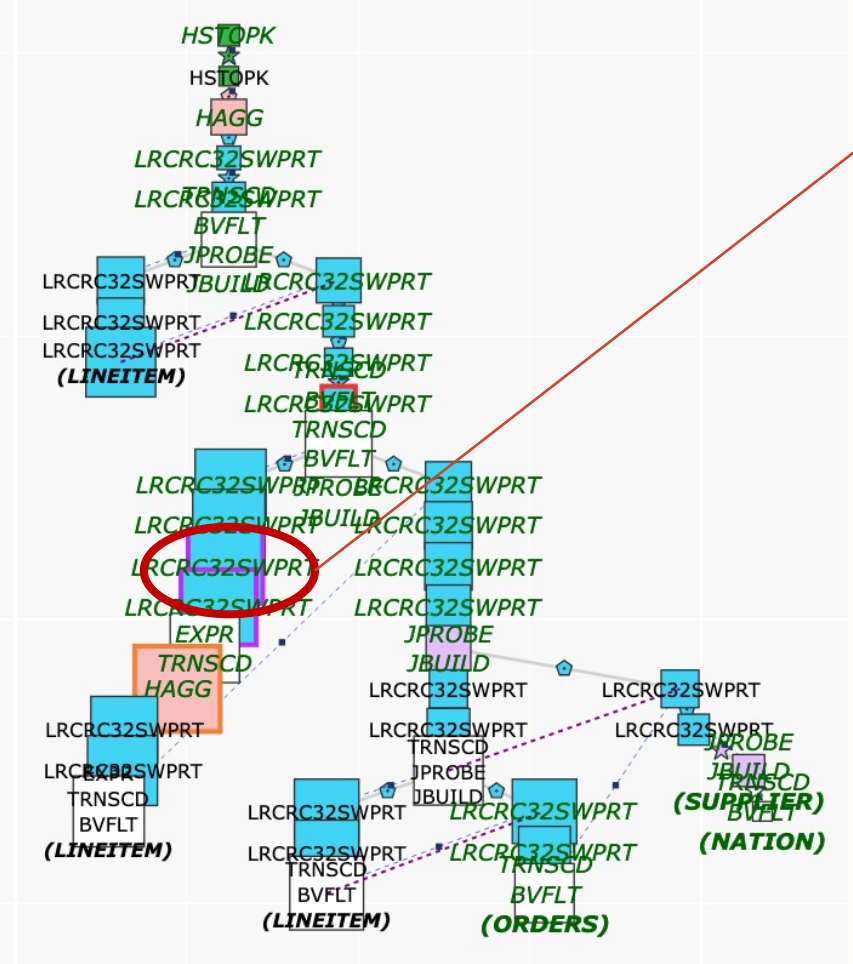
Runtime savings: 3.1s (from 6.9s down to 3.8s)

Late Decompression
Implementation Sketch

```
SELECT a1, a2, a3
FROM A
WHERE a1 < 10;
```



Barrier For Network Tasks



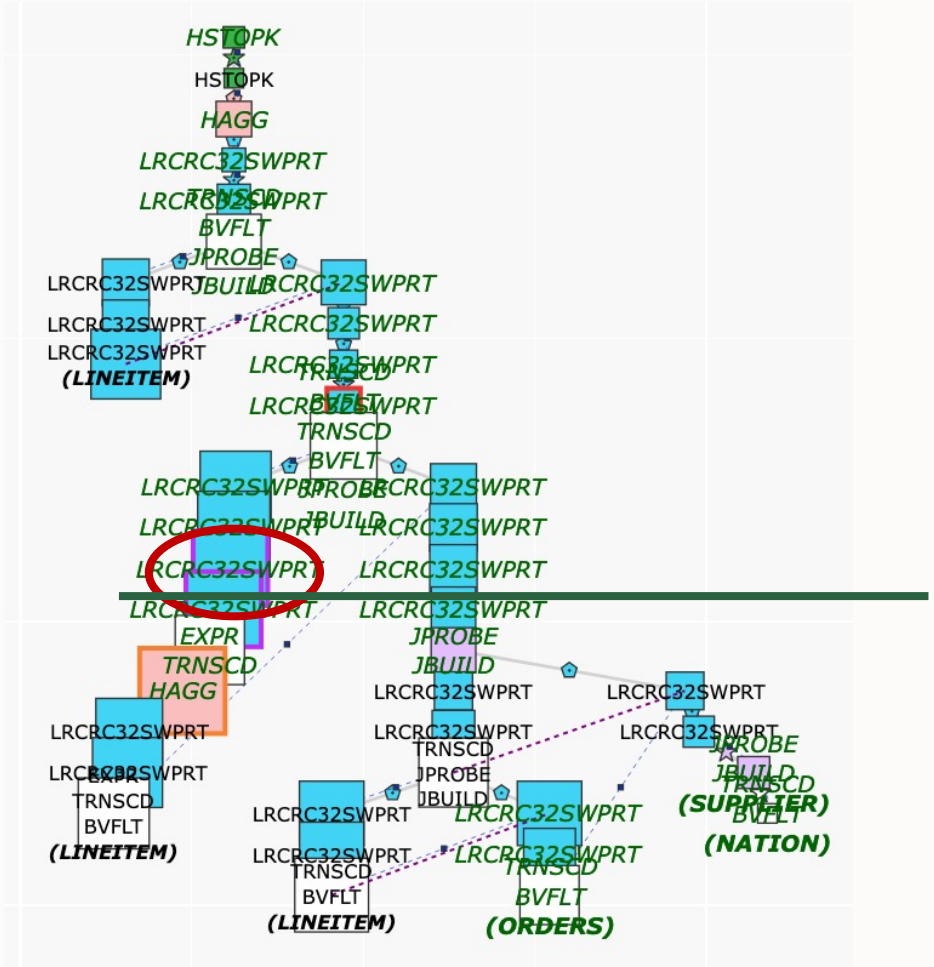
A System Partitioning Task

- Each node partitions its local relation using a partitioning key and sends its local partition
- Straggler issue
  - Slow nodes will not consume their local data fast enough
  - Slow nodes receive data earlier and network buffers start filling up

Worst case: 2x memory (could not send anything + received everything)

- In Lakehouse Scale (up to 512 nodes), more likely to face this issue and OOM.

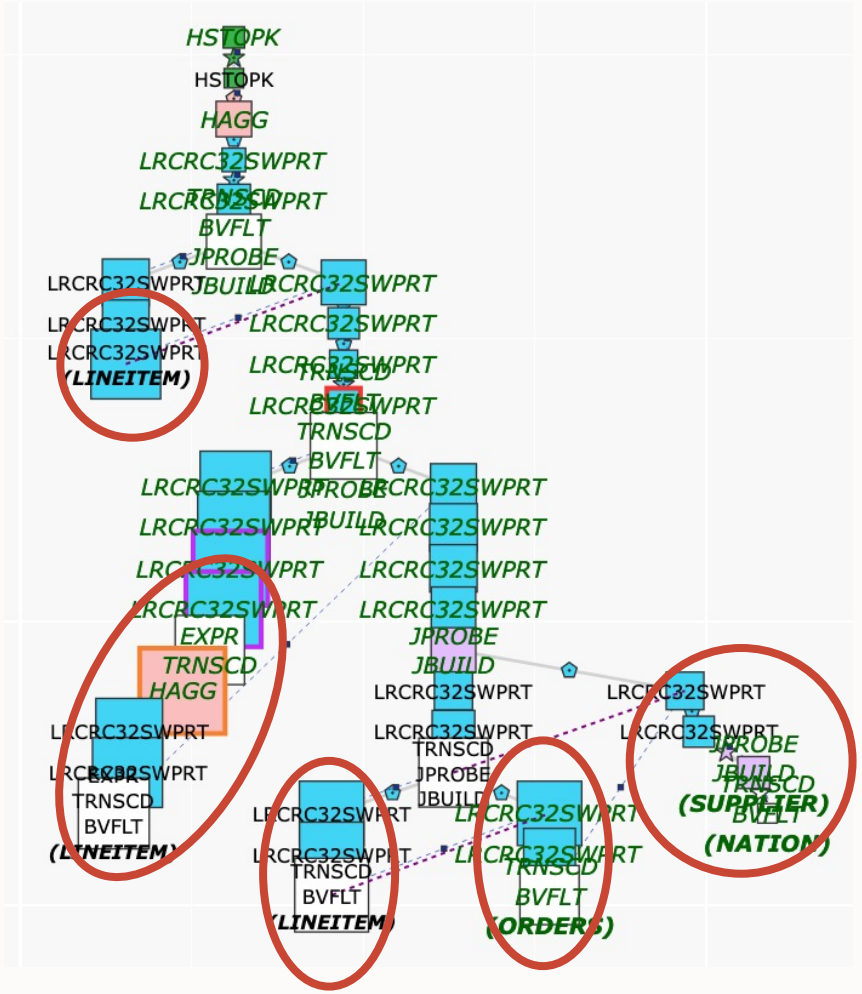
Barrier For Network Tasks



Barrier before any system partitioning task

- All nodes wait until we reach this task
- No performance penalty, since we had to wait for the slowest node anyway

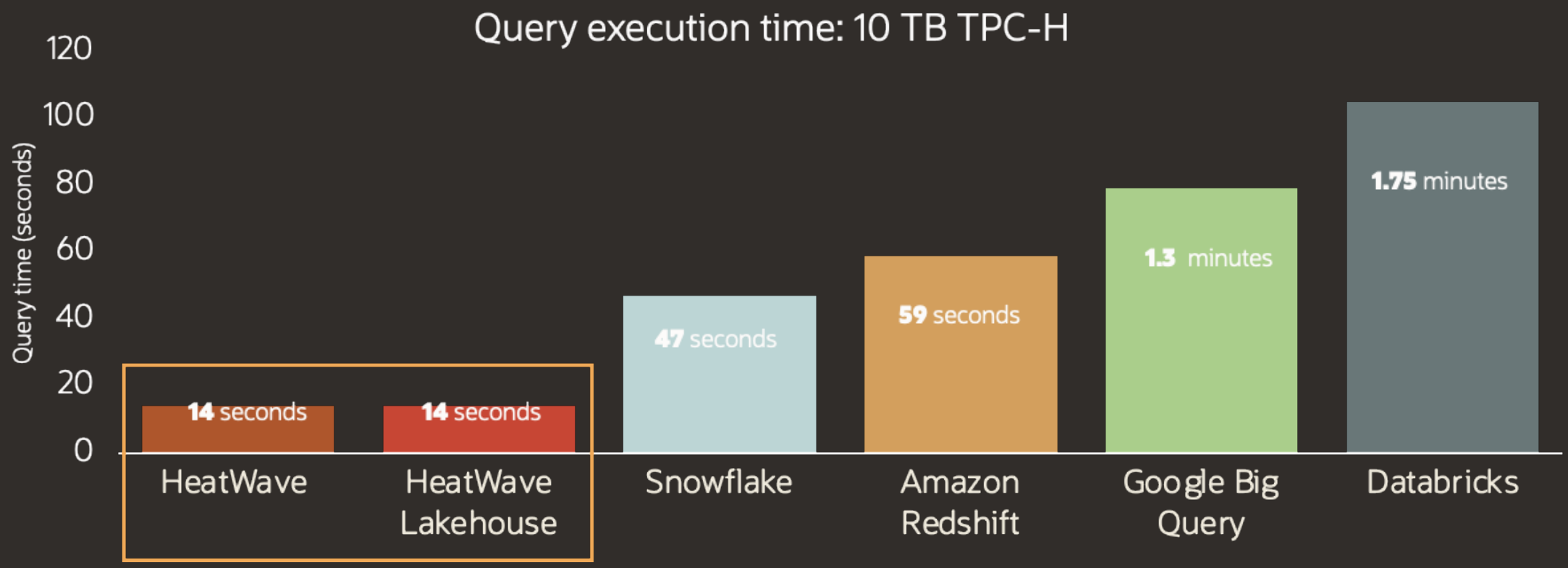
Memory Aware Task Scheduling



HeatWave has a compile-time task scheduling

- Optimized mode: Schedule all network-heavy subtrees first (overlap network with compute as much as possible)
  - High memory consumption…
- Conservative mode
  - Overlap until we reach a peak memory usage threshold.
  - The rest of the tasks prioritize data reduction.

MySQL HeatWave Lakehouse Performance



Same performance whether data comes from InnoDB or Object Store

Summary

MySQL HeatWave Lakehouse: Scaling Queries to Thousands of Cores

- Lakehouse End-to-End Workflow
- High performance Data Ingestion
  - Robust parallelism via intra-file processing

- Relational Processing at Lakehouse Scale
  - Adaptive Execution Capabilities (Runtime Cardinality Estimation)
  - Efficient Bloom Filter Sharing
  - Late Decompression
  - Network Barriers to avoid straggler OOM
  - Memory-aware Task Scheduling

ORACLE