ORACLE

MySQL HeatWave: A Deep Dive Into Optimizations

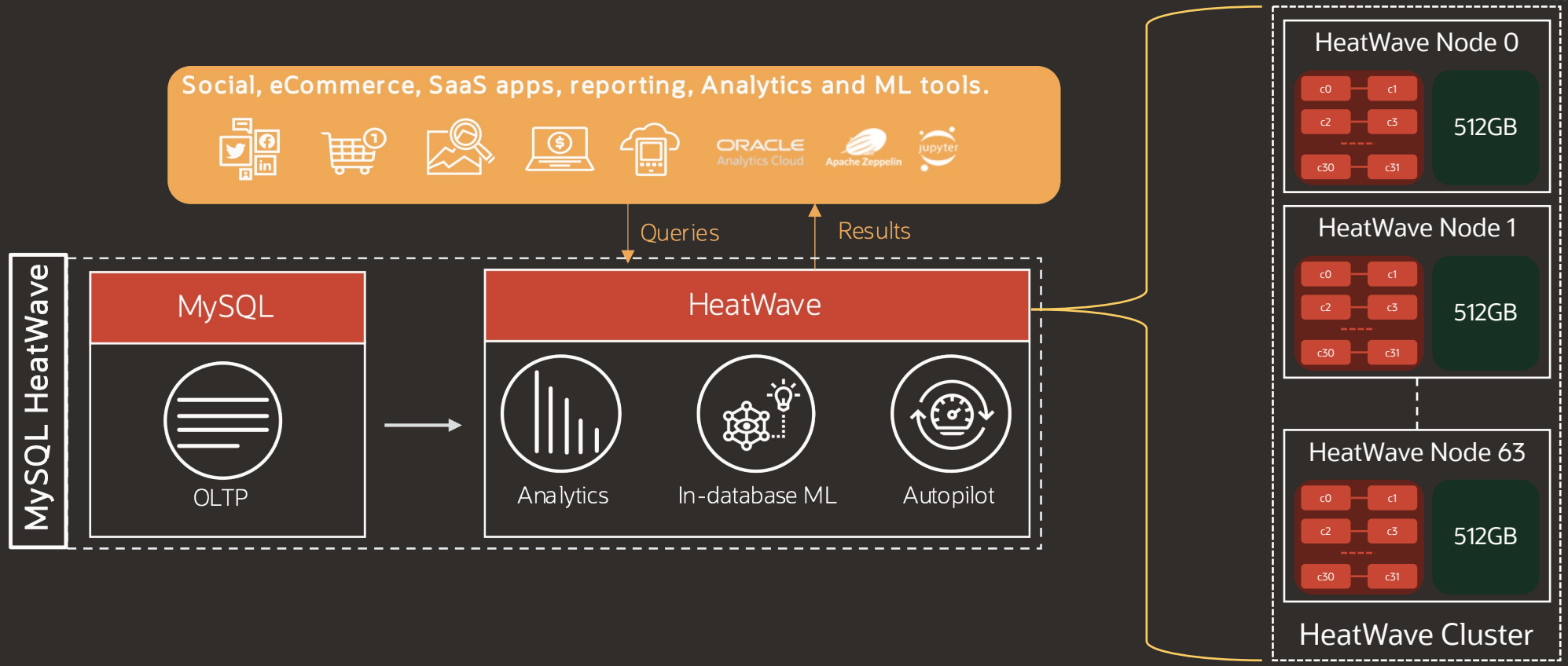
Robust OLAP Query Performance At Scale

Kaan Kara, PhD

Principal Member of Technical Staf

MySQL HeatWave

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



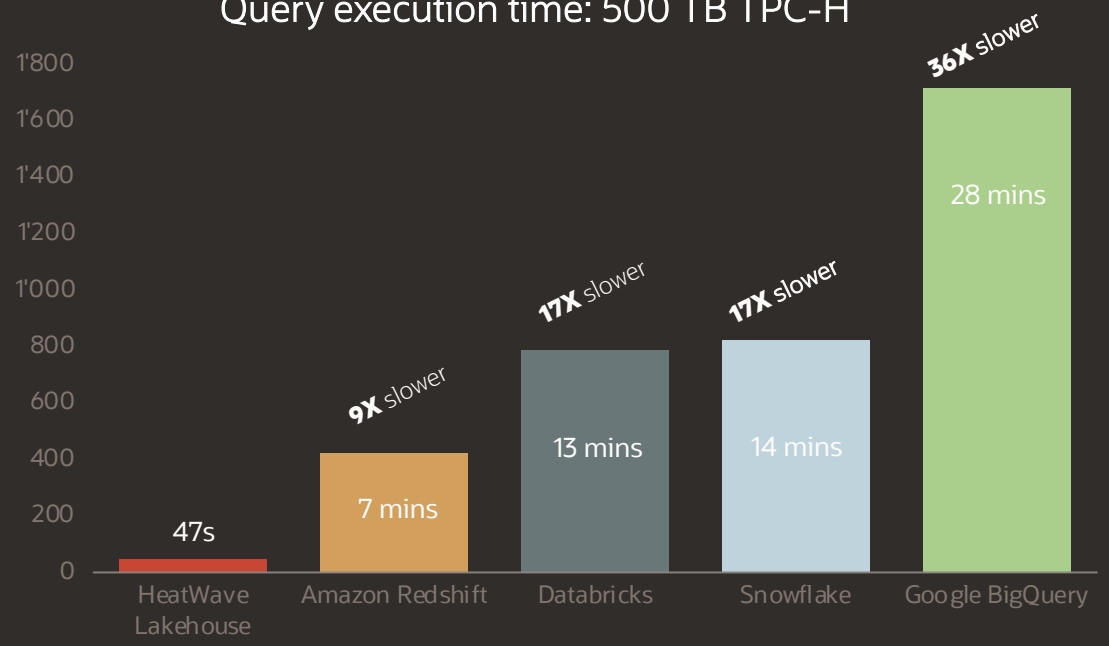
How do we scale-out query processing to hundreds of nodes and thousands of cores efficiently?

1. Main Principal in Execution: Data Partitioning
2. Query Optimization: A Holistic Approach

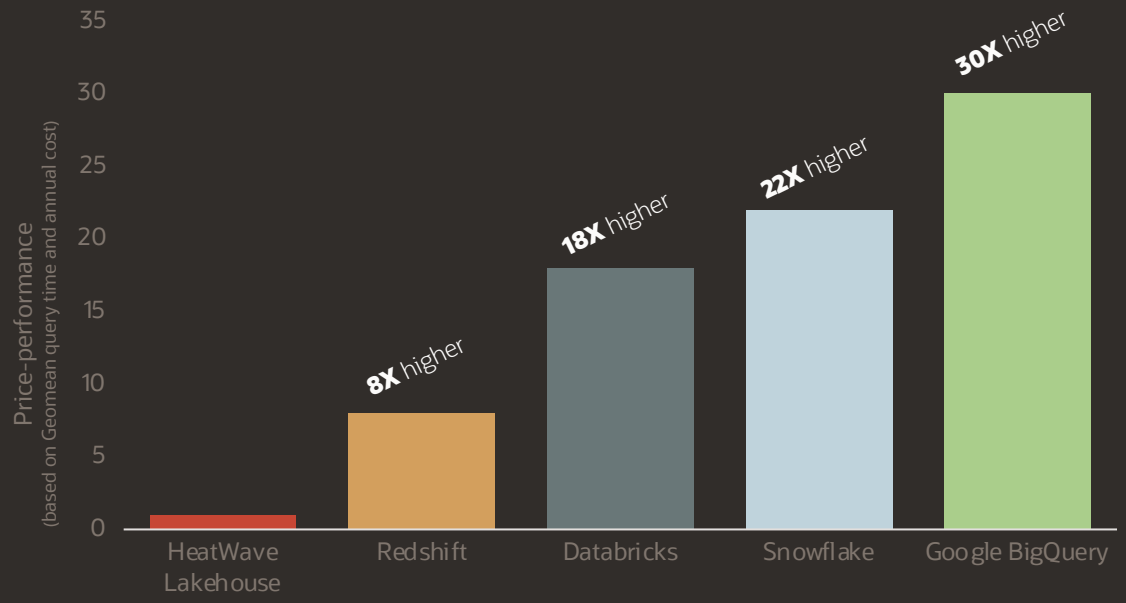
MySQL HeatWave Query Performance

Significant Advantage in Perf and Price/Perf

Query execution time: 500 TB TPC-H



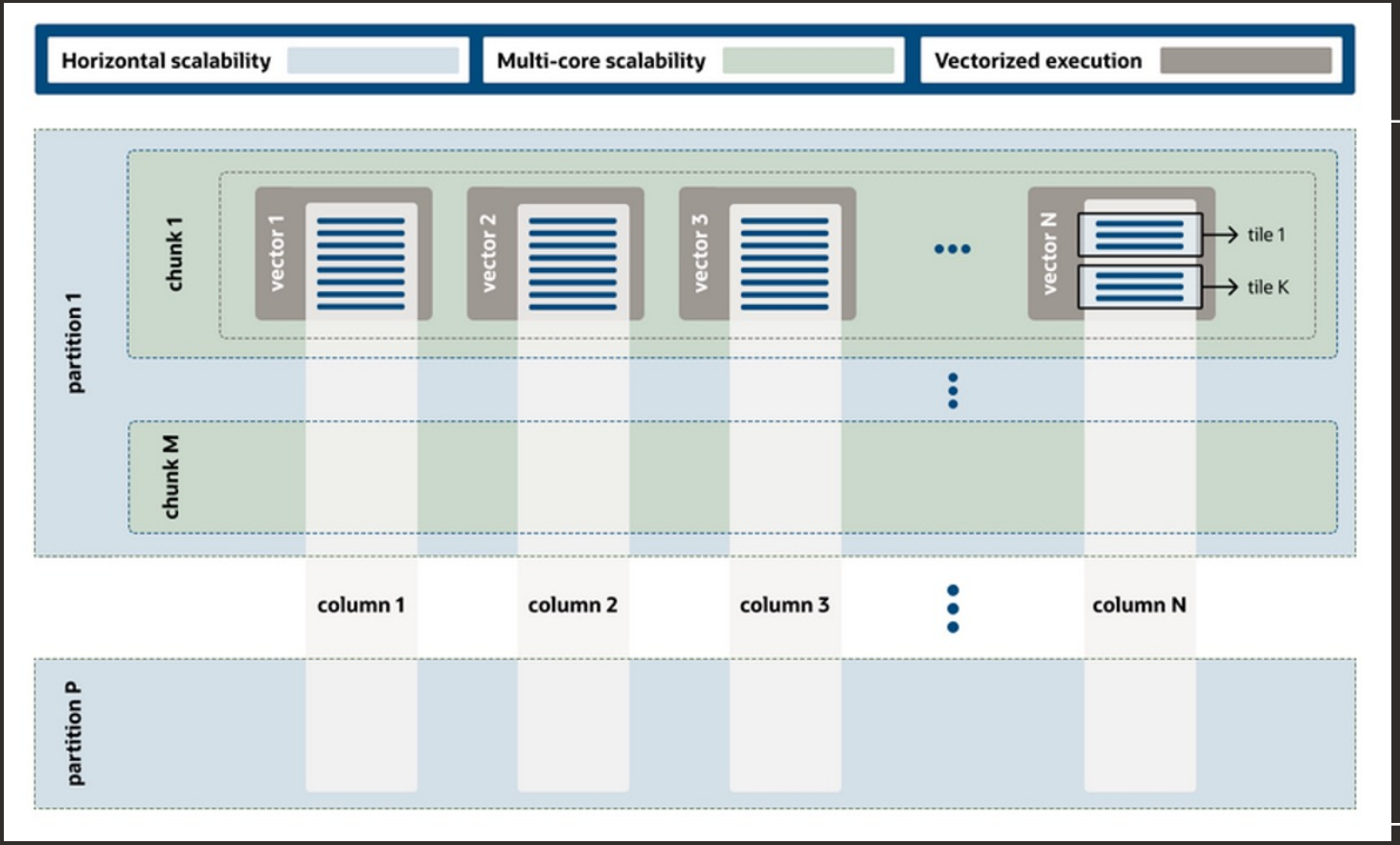
Price-performance: 500 TB TPC-H



https://github.com/oracle/heatwave-tpch

In-Memory Hybrid Columnar Format

OLAP Performance Fundamentals

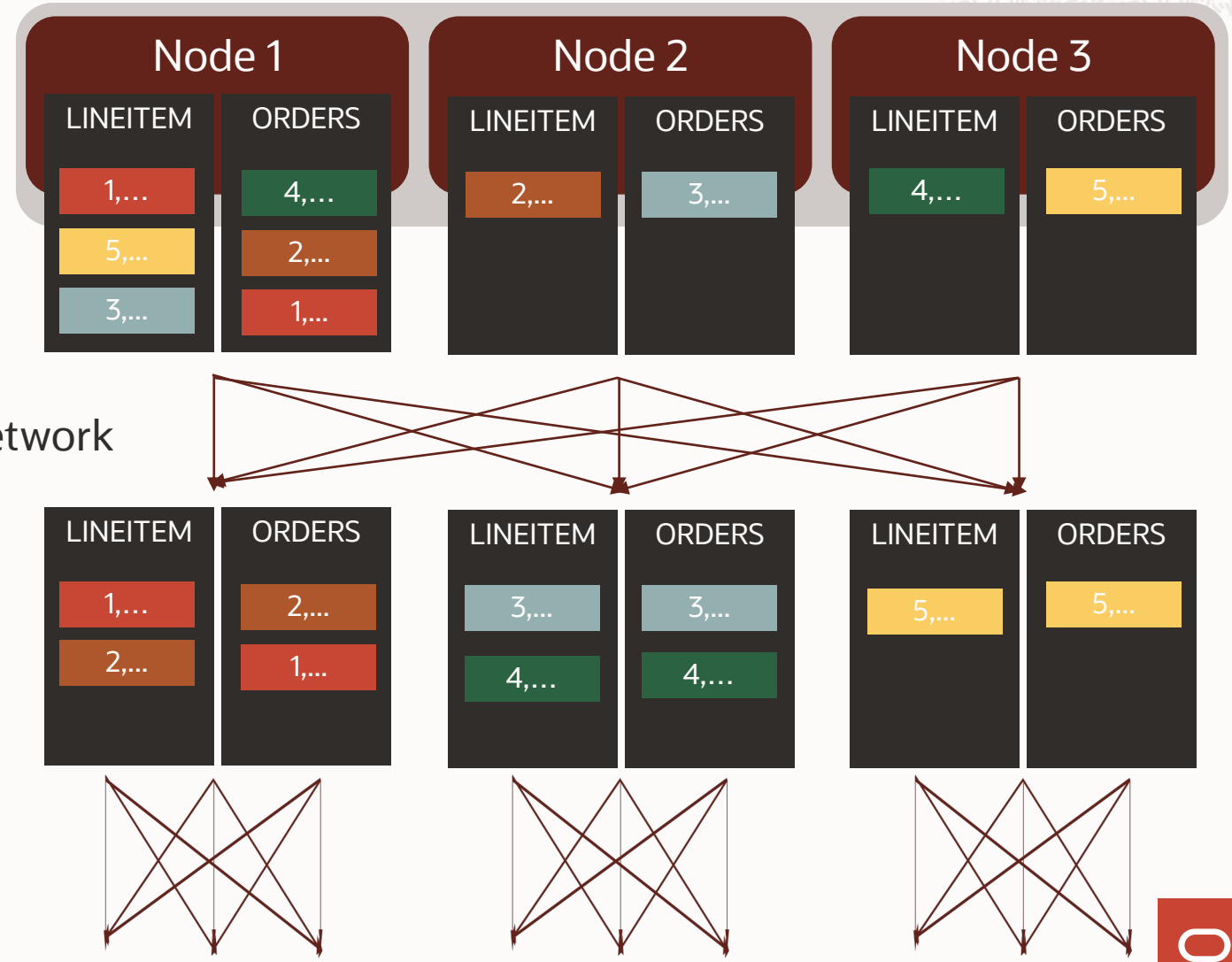


High fanout partitioning is the core aspect of high performance.

Distributed Relational Join Processing

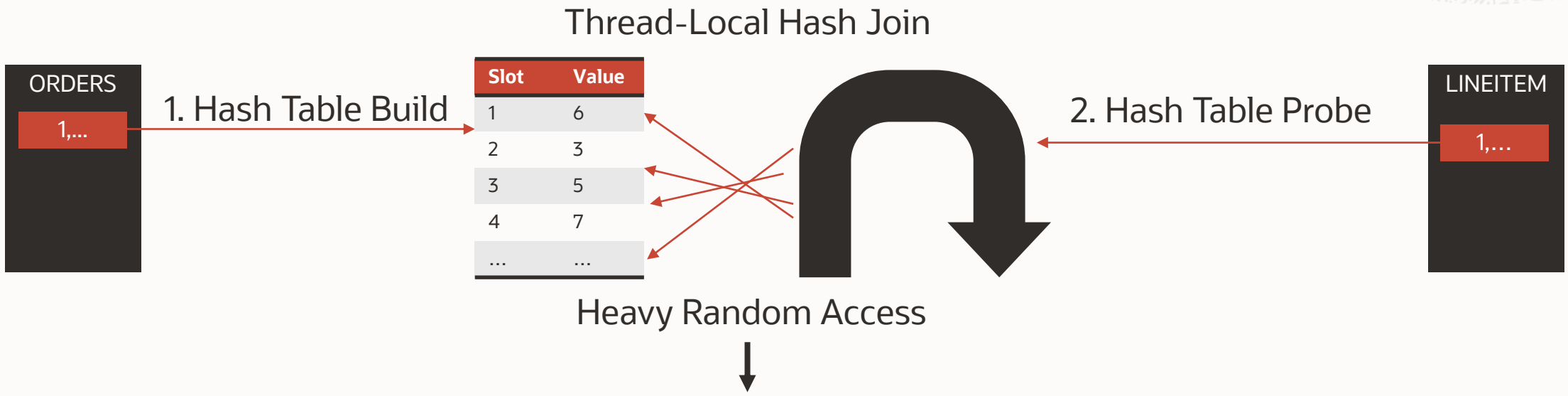
QUERY

```
SELECT L_LINENUMBER, L_DISCOUNT
FROM LINEITEM JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
WHERE O_ORDERDATE = '2022-04-12'
ORDER BY L_LINENUMBER;
```



Local Partitioning. Why?
1. Multicore Parallelism
2. Cache Locality

How much do we need to partition?



- Cache misses are very costly in the hot loop.
- We want the hash table to fit L1 cache (64KB).

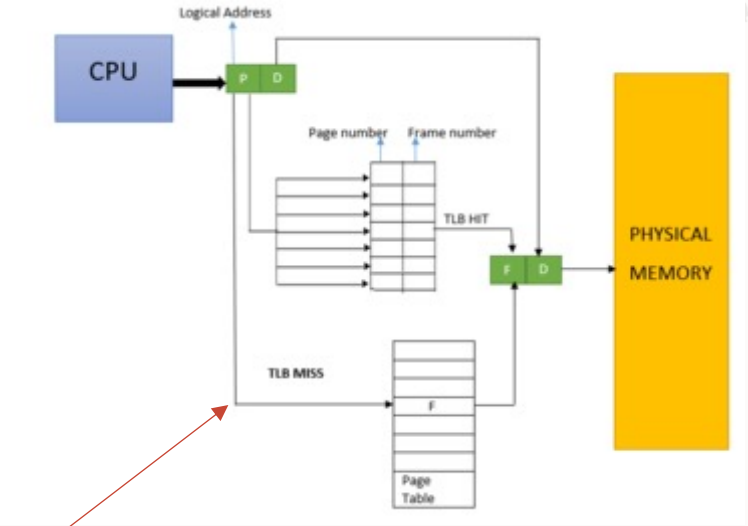
Back of the envelope

- Orders Table: key column is 8B integers, 768 Billion rows: 6.1TB
- 6.1TB / 64KB = ~96 Million Partitions
- Next power of two: **2^27, a partitioning fanout of 134 Million**

Data Partitioning Problem: Where is the bottleneck?

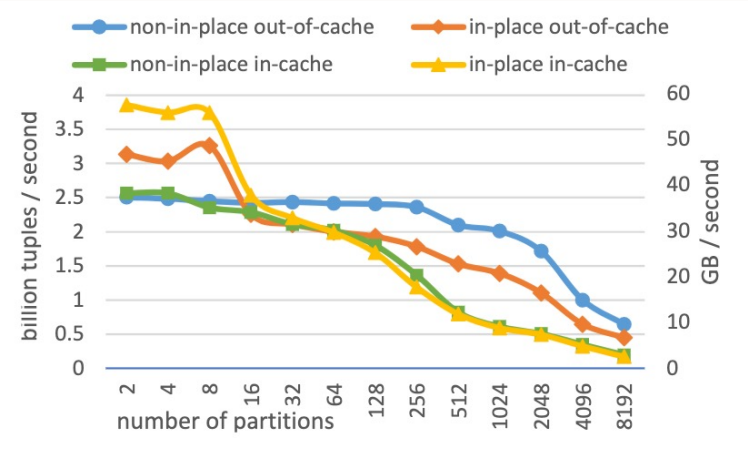
```
foreach tuple t in relation R:
        p = hash(t.key)
        partitions[p][counts[p]++] = t
```

- p is random and will cause heavy random access.
- In case the range of p is high (high fanout), chances of TLB misses are high.



TLB (Translation Lookaside Buffer)

- A specialized cache for virtual-to-physical address translation.
- If a virtual address is not found, expensive *page walk* occurs.
  - Can be even more expensive than simple cache miss, because page walk might perform multiple memory accesses.
- Typical TLB has around 64 entries.
- Any partitioning fanout larger than 64 is likely to use more than 64 pages, causing a huge performance impact.



[1] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In ACM SIGMOD, 2014

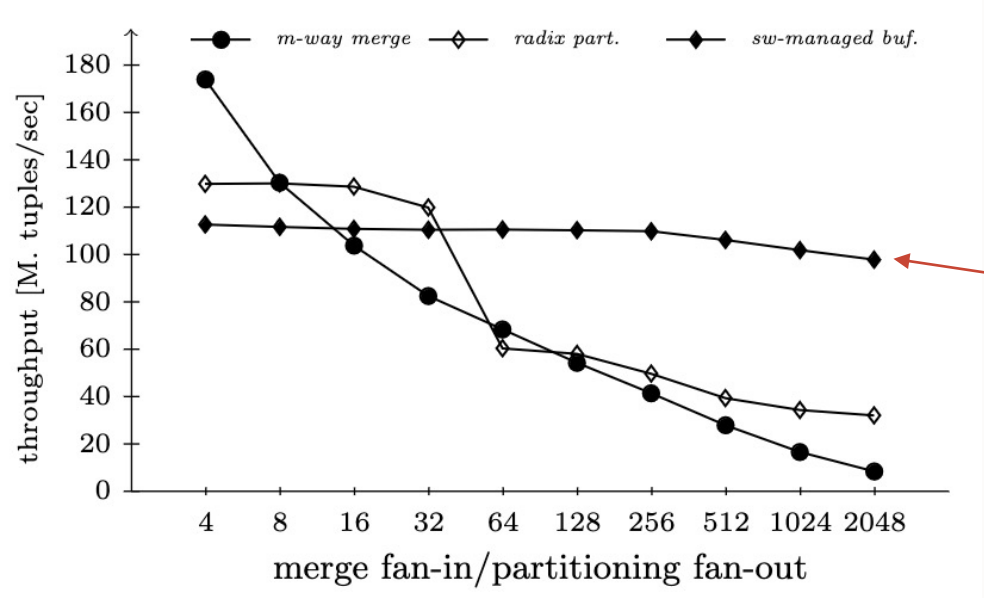
Reducing the TLB bottleneck with SW-managed buffers [2]

```
foreach tuple t in relation R:
        p = hash(t.key)
        buffers [p][ counts [p]++ % N ] = t
        if (counts[p] % N == 0)
                copy buffers[p] to partitions[p]
```

- Maintain smaller (i.e. cacheline size) buffers per partition.
- Most of the writes happen to a small number of pages; avoiding TLB misses.

Back of the envelope

- 4KB page / 64B cacheline = 1024 partitions per page
- Assume 32 threads do 2048 fanout partitioning => this will consume 64 TLB entries.

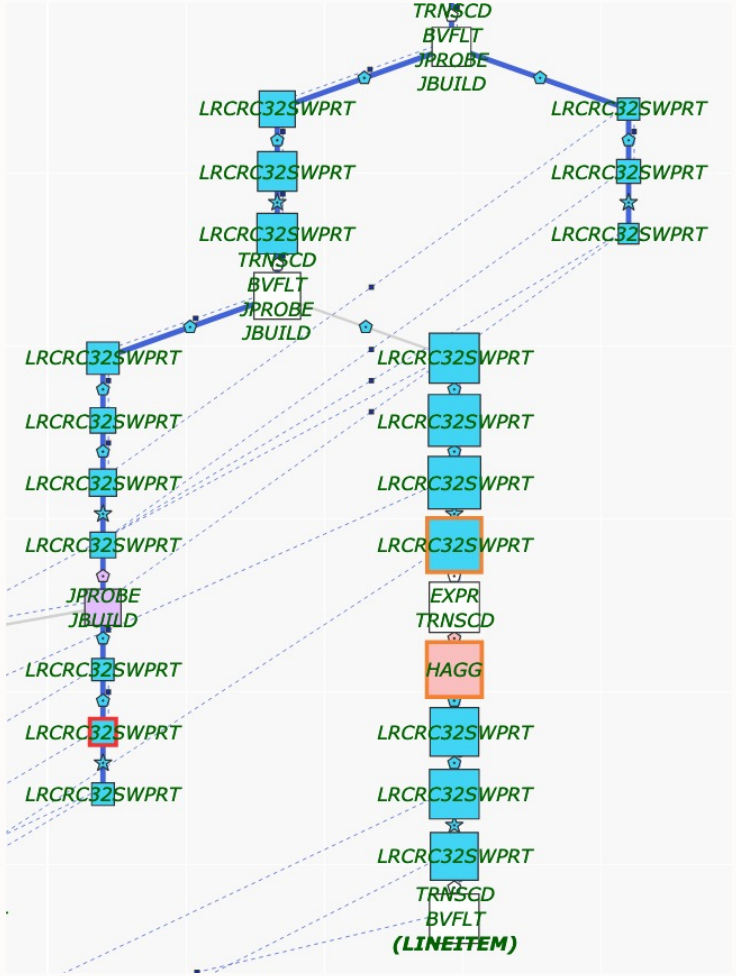


- SW-managed buffers are a good optimization for the TLB bottleneck, yet higher partitioning fanout can still cause a problem.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. PVLDB, 7(1):85–96, 2013

Multi Round Partitioning

An excerpt from TPC-H q21 physical plan

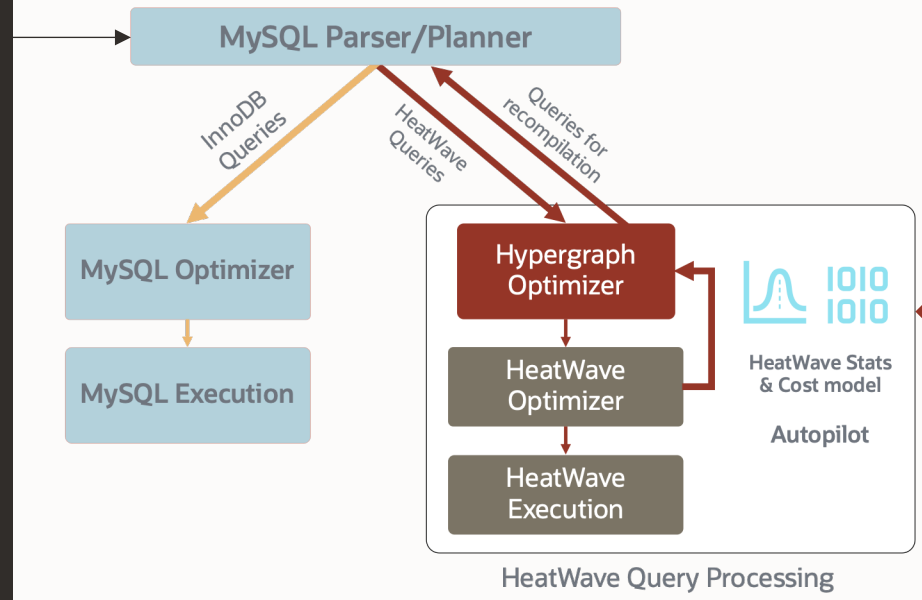


MySQL HeatWave: Query Optimization

Query Optimization: Holistic Optimization

TPC-H Query 21

```
SELECT
s_name,
count(*) as numwait
FROM
supplier,
lineitem l1,
orders,
nation
WHERE
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
SELECT
*
FROM
lineitem l2
WHERE
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
SELECT
*
FROM
lineitem l3
WHERE
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'SAUDI ARABIA'
GROUP BY
s_name
ORDER BY
numwait desc,
s_name
limit
100;
```



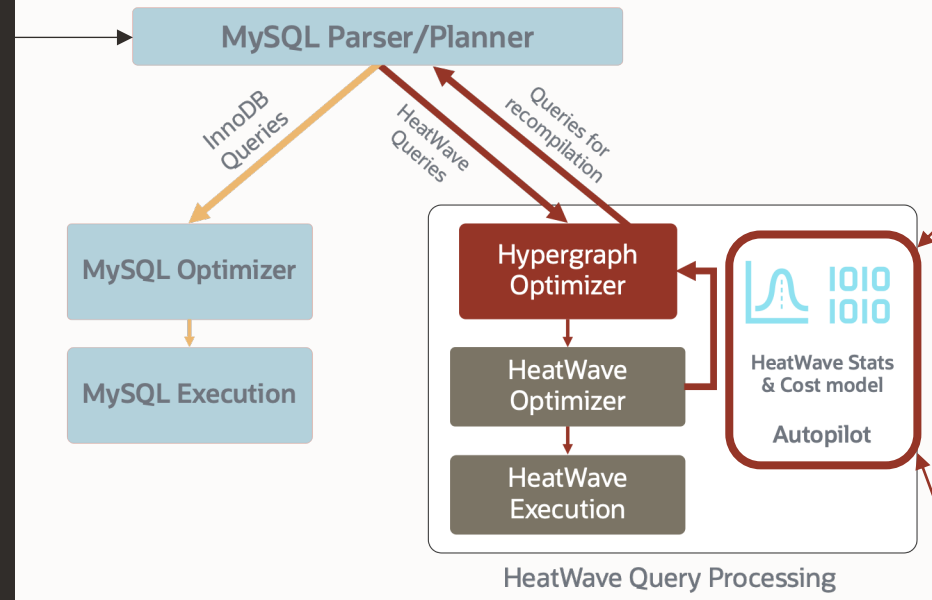
Hypergraph Optimizer gets information from HeatWave Optimizer per subgraph:

- Cardinality stats from previous runs.
- Cost model from HeatWave physical optimizer.

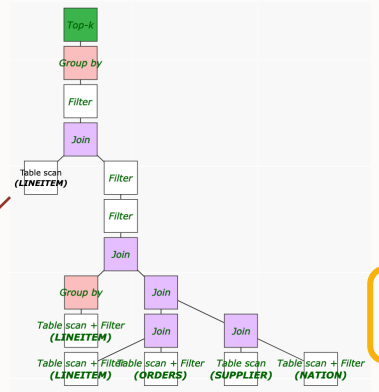
Query Optimization: A Detailed Overview

TPC-H Query 21

```
SELECT
s_name,
count(*) as numwait
FROM
supplier,
lineitem l1,
orders,
nation
WHERE
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
SELECT
*
FROM
lineitem l2
WHERE
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
SELECT
*
FROM
lineitem l3
WHERE
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'SAUDI ARABIA'
GROUP BY
s_name
ORDER BY
numwait desc,
s_name
limit
100;
```



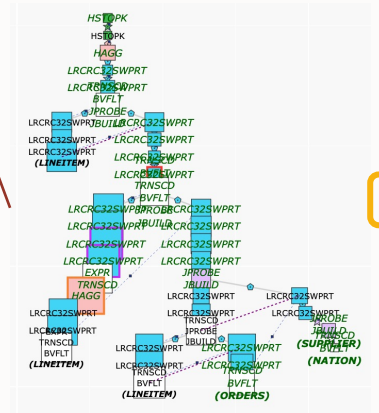
Logical Plan



Logical plan decisions:

- Nodes are matched to previous runs to obtain accurate cardinality results (filter selectivity, join cardinality)
- Logical transformations (subquery pushdown)
- …

Physical Plan

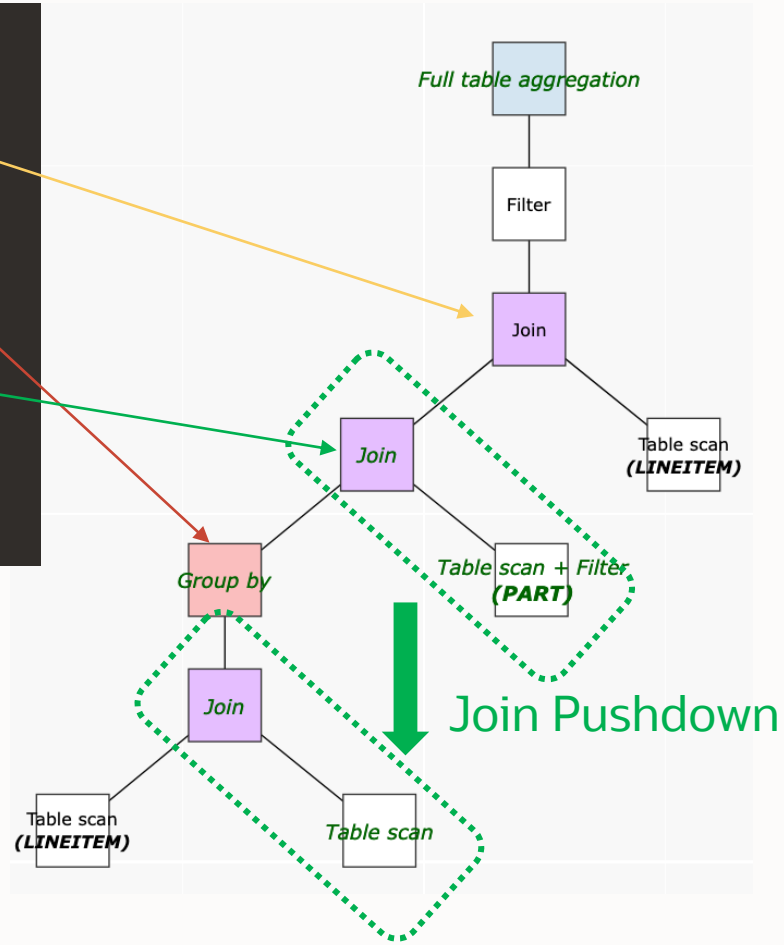


Physical plan decisions:

- Join pattern (broadcast vs. partitioned)
- Partitioning fanout / rounds
- Bloom filter attachment
- Data placement key
- …

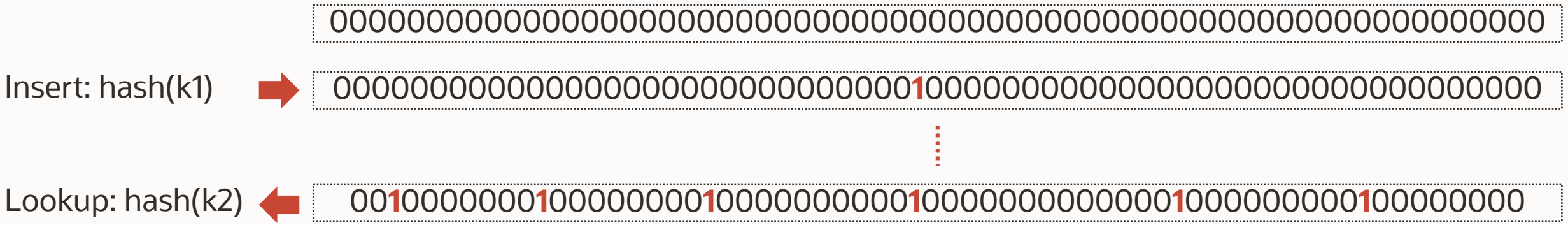
Logical Optimization: Pushdown Transformation

```
SELECT
SUM(L1.L_EXTENDEDPRICE) / 7.0 AS AVG_YEARLY
FROM
LINEITEM, PART,
(SELECT
L_PARTKEY AS TEMP_KEY,
0.2 * AVG(L_QUANTITY) AS TEMP_AVG
FROM
LINEITEM
GROUP BY L_PARTKEY) AS T1
WHERE
P_PARTKEY = L_PARTKEY
AND P_BRAND = 'Brand#23'
AND P_CONTAINER = 'MED BOX'
AND P_PARTKEY = T1.TEMP_KEY
AND L_QUANTITY < T1.TEMP_AVG;
```



- Logical equivalence: We can perform the join between T1.L_PARTKEY with P_PARTKEY before or after the group-by. Which one is cheaper though?
  - Intuition: Selective join, especially with filtered build-side, is usually cheap.
- Without pushdown, group-by has to process 180 Billion rows from LINEITEM.
- After the pushdown, group-by processes 180 Million rows (1000x reduction).

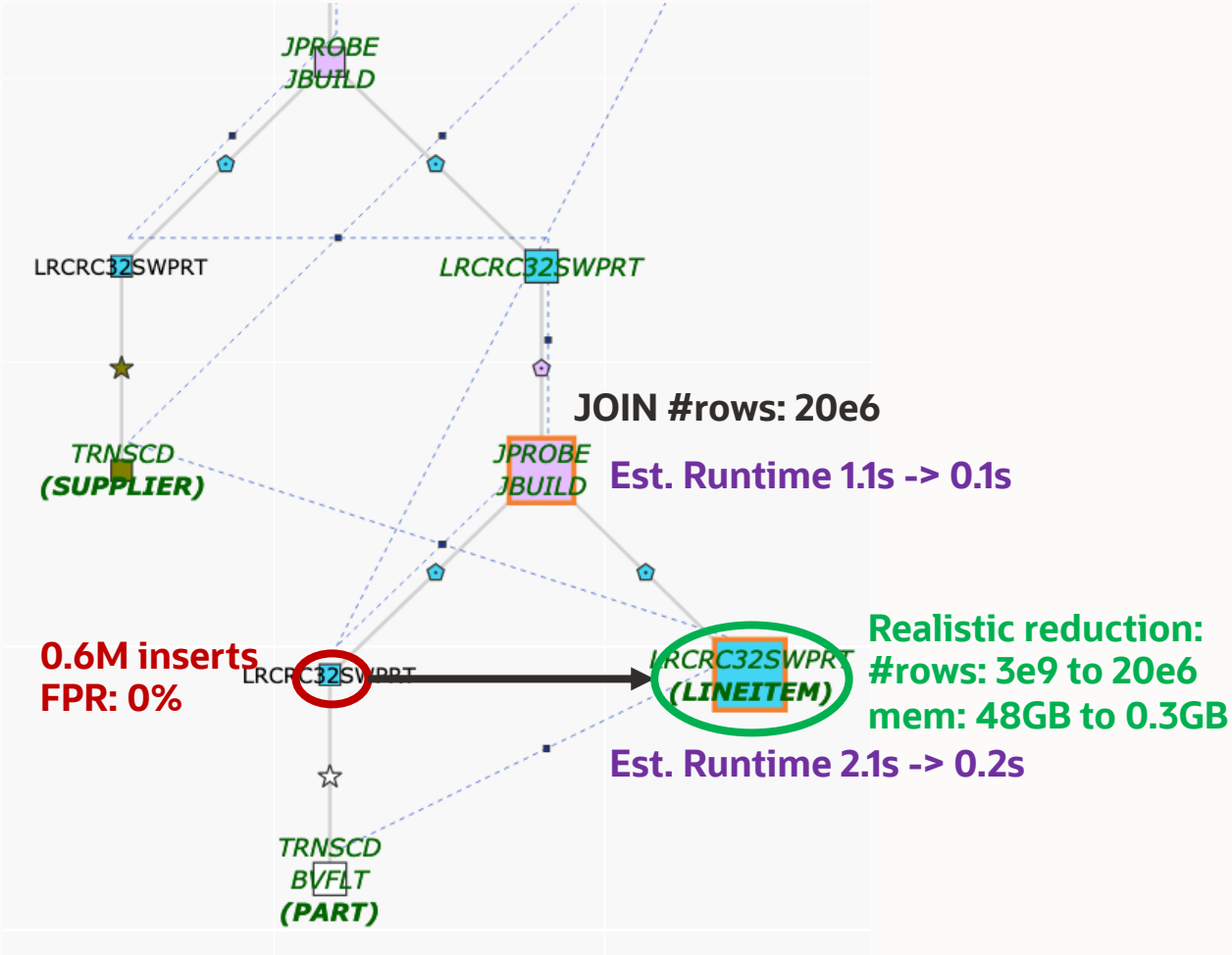
Physical Optimization: Bloom Filter



- False positives are possible (due to collisions at insert)
- False positive rate estimate: *E=(1-exp(-kN/m))^k* **(m: #bits, N: #inserts, k: #hashes)**

  - **For a low false positive rate:**
    - **More bits**
    - **Less inserts**

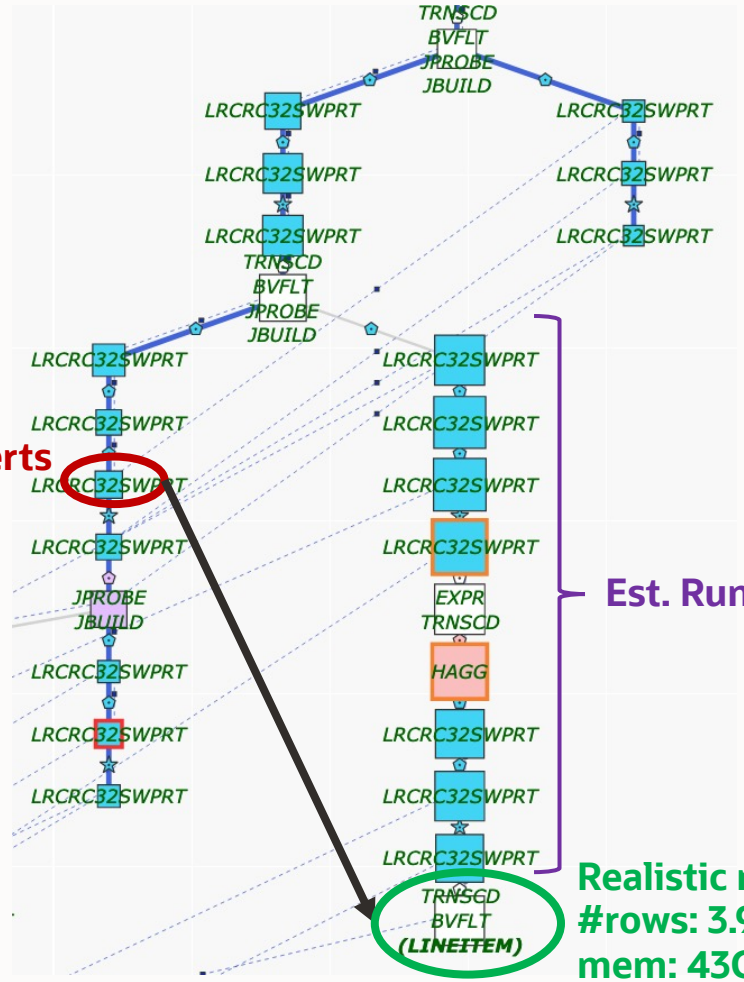
Bloom Filter Application: TPC-H q08



Observations

- **False-Positive-Rate:** More inserts require larger BF. Trade-off between low FPR and cache-locality.
- **Expected filtering** does not only depend on FPR, but also join output cardinality. Advantage only for high selective joins.
- **Runtime improvement:** Depends on the filtering rate, however BF lookup itself turns out to be costly. Apply a BF only for good filtering rate.
- **Join pattern:** System broadcast on the build side. No need to share local BFs across the cluster.

Bloom Filter Application: TPC-H q21



72M inserts
FPR: 0%

Est. Runtime 13s -> 1s

Realistic reduction:
#rows: 3.9e9 to 270e6
mem: 43GB to 3GB

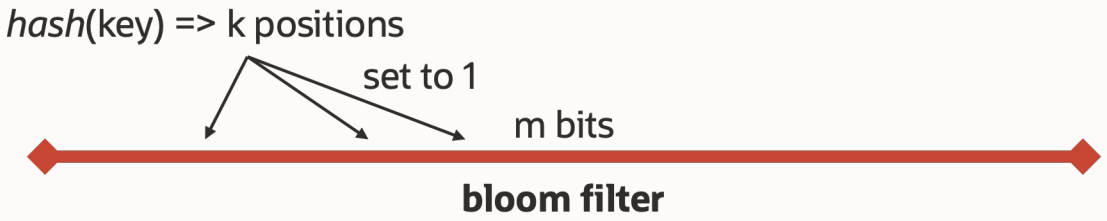
Observations

- **Runtime improvement** is great for this query, since we can push down the BF filtering through many operators.
- **Join pattern:** System partitioning on the build side. Costly, since we need to share local BFs across the network. Cost-based decision for these type of queries.

Bloom Filter Implementation

Standard vs. Blocked

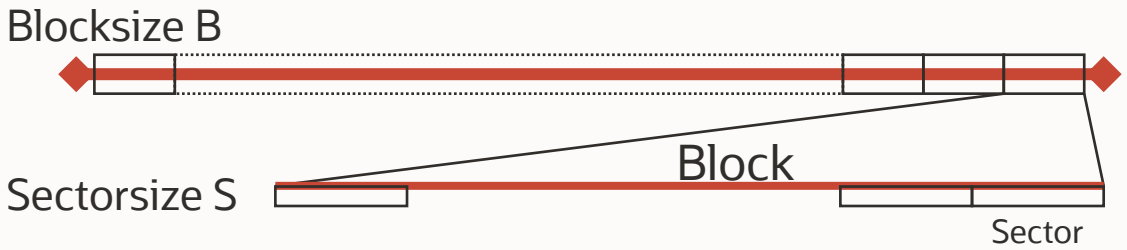
Standard



- Set k bits in the entire bloom filter buffer.
- Each insert/lookup requires accessing k cachelines.

- - k random accesses per single lookup.
- - Hard to parallelize with AVX (row dimension rather than hash dimension).
- + Better false-positive-rate.

Blocked



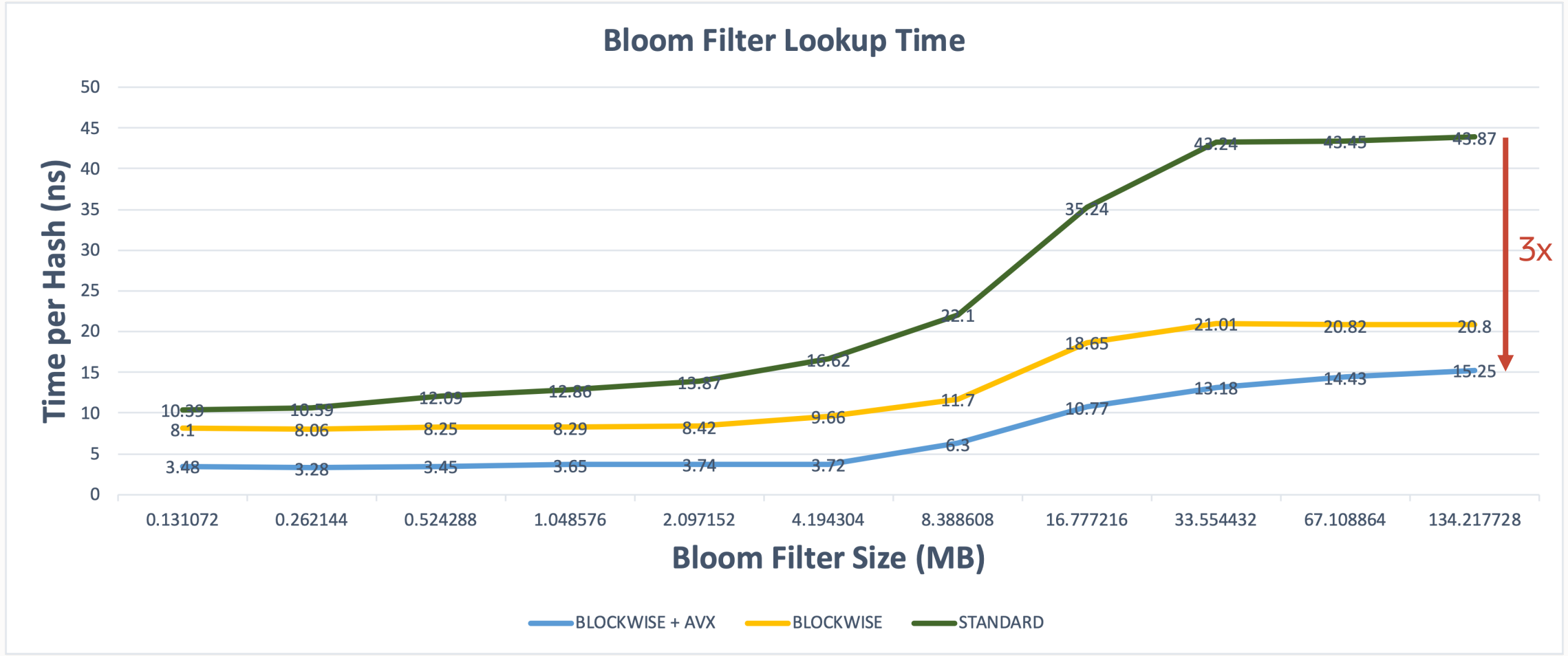
- Set 1 bit in each sector in a block, for each insert. (B/S bits per insert)
- Each insert/lookup requires accessing a single block.

Choose to accommodate AVX2:
B: 256 bits, S: 32 bits => 8 hashes (close to the optimal 7)
A single lookup can happen in few 256-bit AVX instructions.

- + Single access per lookup.
- + Suitable for AVX.
- - Worse false-positive-rate.

Bloom Filter Implementation

Standard vs. Blocked

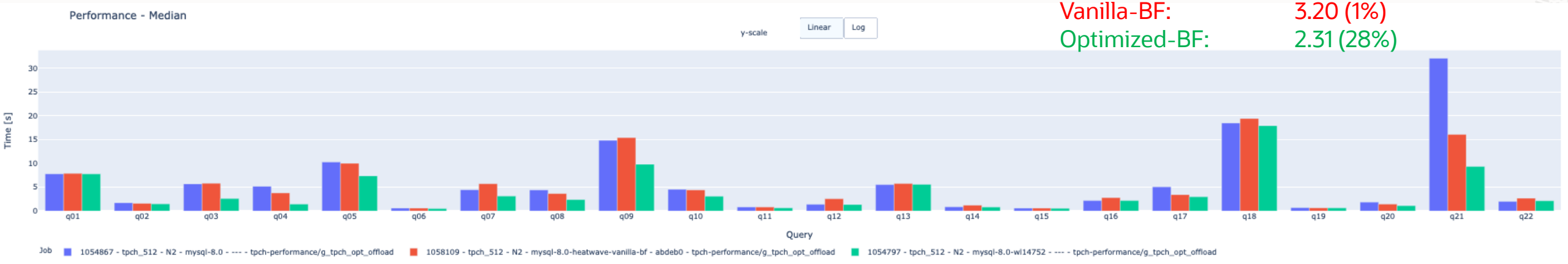


Vanilla-BF vs Optimized-BF

TPC-H 512GB (Baseline, Vanilla-BF, Optimized-BF)

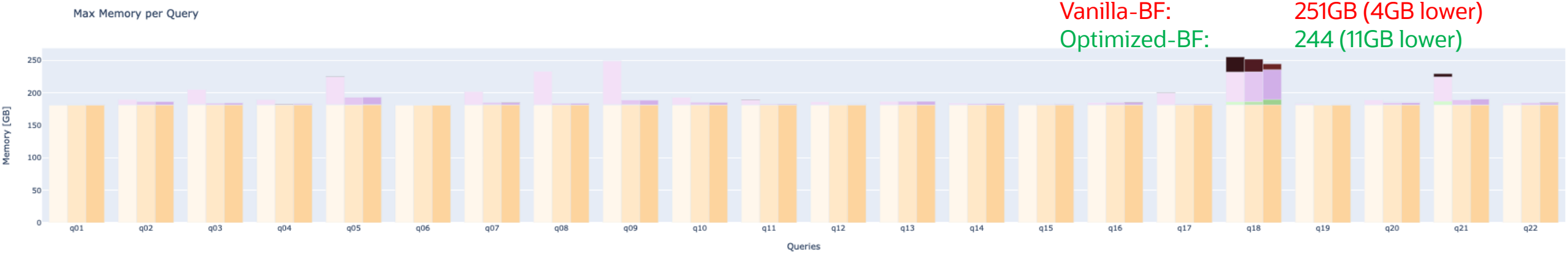
Geomean

Baseline:	3.22
Vanilla-BF:	3.20 (1%)
Optimized-BF:	2.31 (28%)



Max memory

Baseline:	255GB
Vanilla-BF:	251GB (4GB lower)
Optimized-BF:	244 (11GB lower)



Summary

MySQL HeatWave: A Deep Dive into Optimizations

- Data partitioning is a first principle
  - CPU optimizations
- A Holistic Optimization Approach
  - Inform MySQL Hypergraph optimizer with HeatWave query properties (logical and physical).
- Logical optimizations save lots of work
  - Pushdown transformations
  - Cardinality results from stats cache
  - …
- Physical optimizations needed to achieve good scalability
  - Bloom filter applications
  - Join patterns (partitioned vs. broadcast)
  - Group-by patterns (high-NDV, medium-NDB, low-NDV, approx. low-NDV)
  - Data placement
  - Late decompression
  - …

ORACLE