MySQL HeatWave

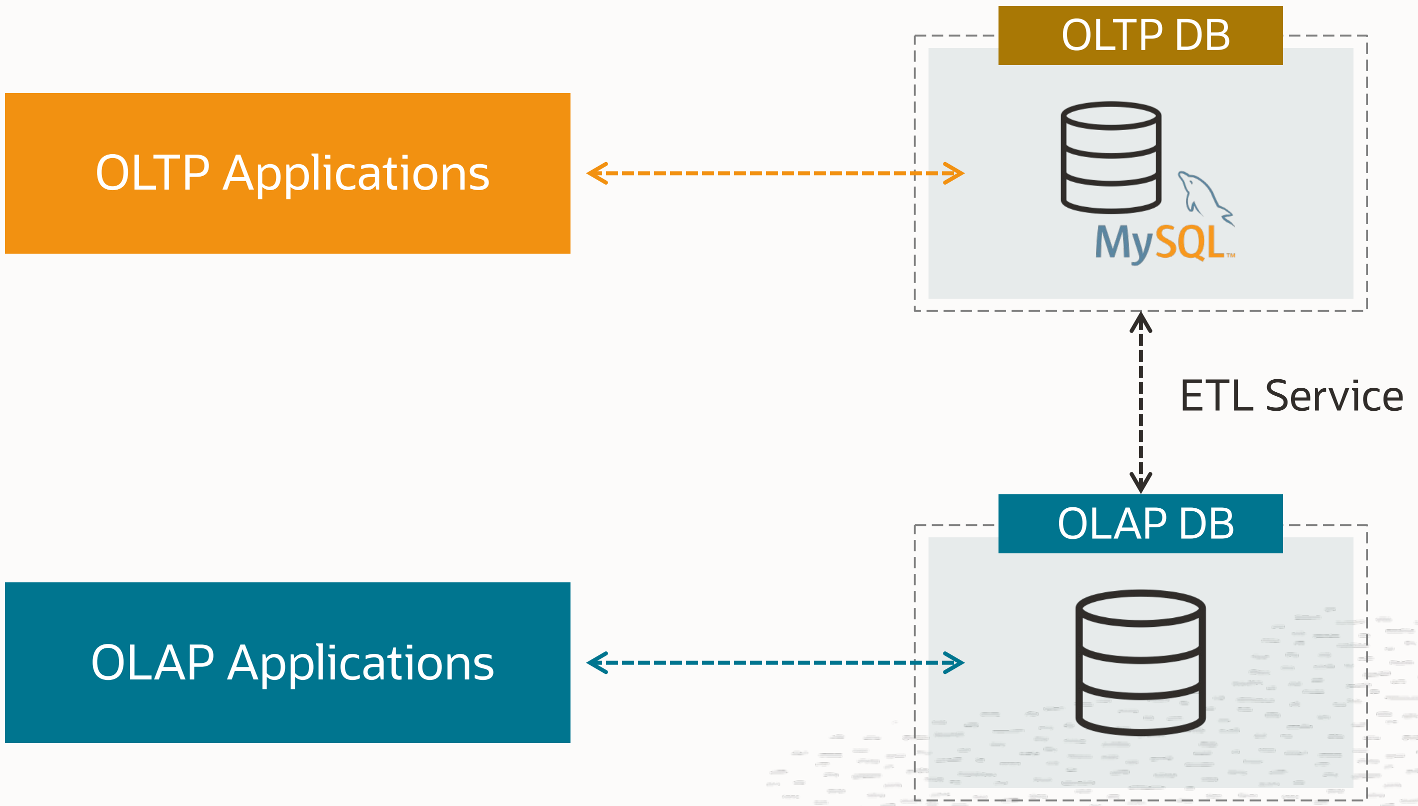
A Deep Dive Into Architecture and Optimizations

Cagri Balkesen, Ph.D.
Architect, MySQL HeatWave
January 31, 2025

Safe harbor statement

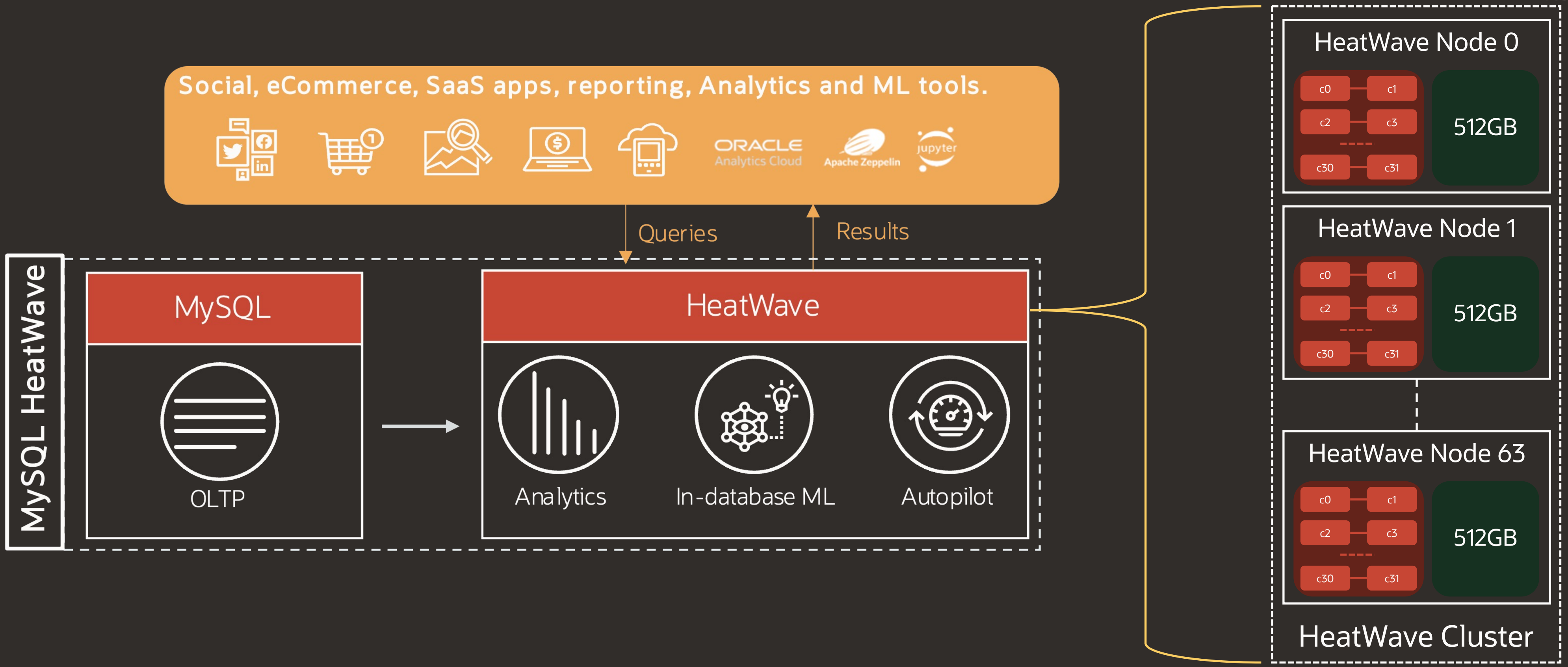
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

MySQL users have needed separate systems for OLTP and OLAP



MySQL HeatWave

OLTP, OLAP/analytics, and ML in one cloud database service – without ETL

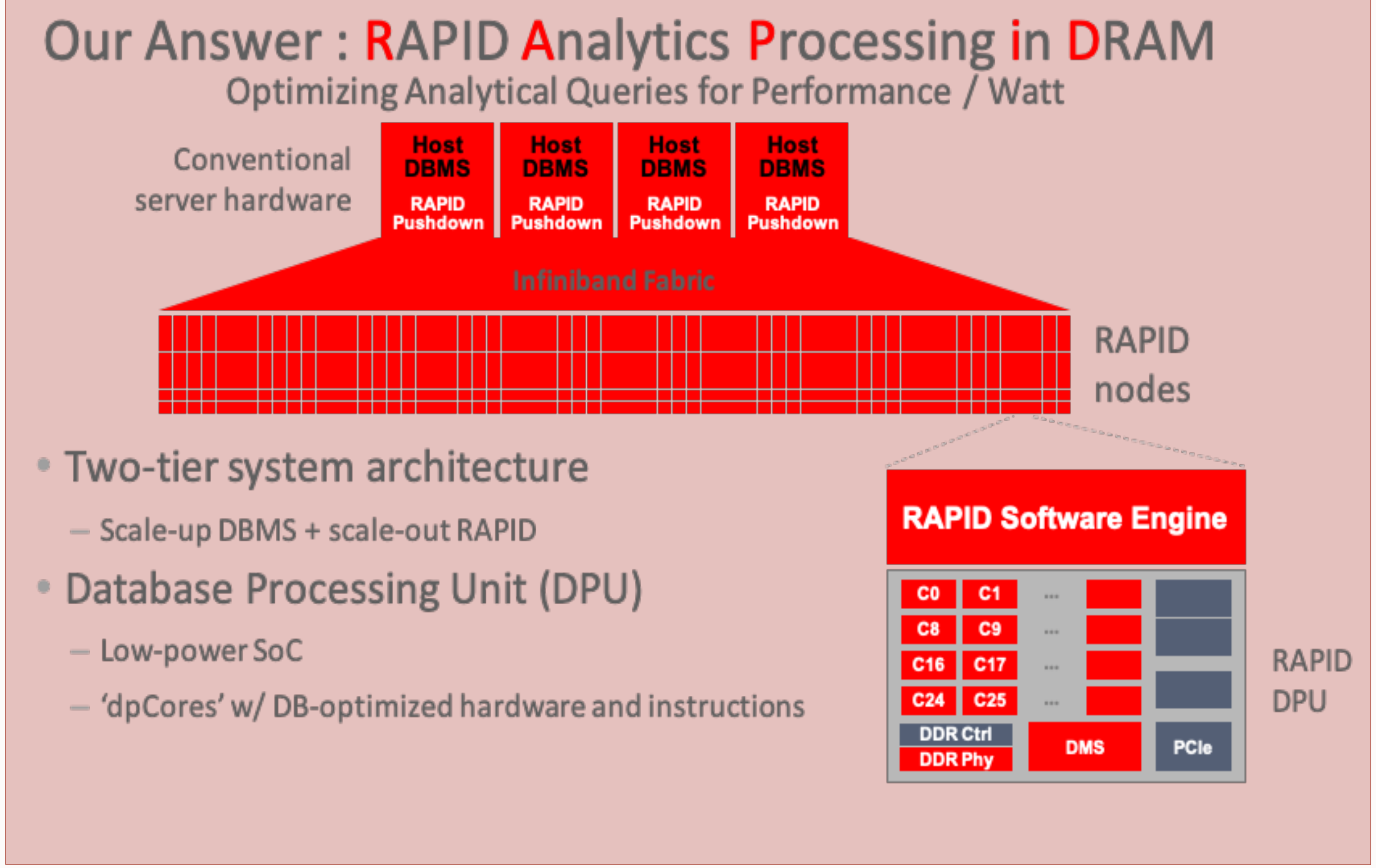


- Single MySQL database for OLTP & analytics applications
- Extreme performance & scalability to hundreds of nodes, thousands of cores

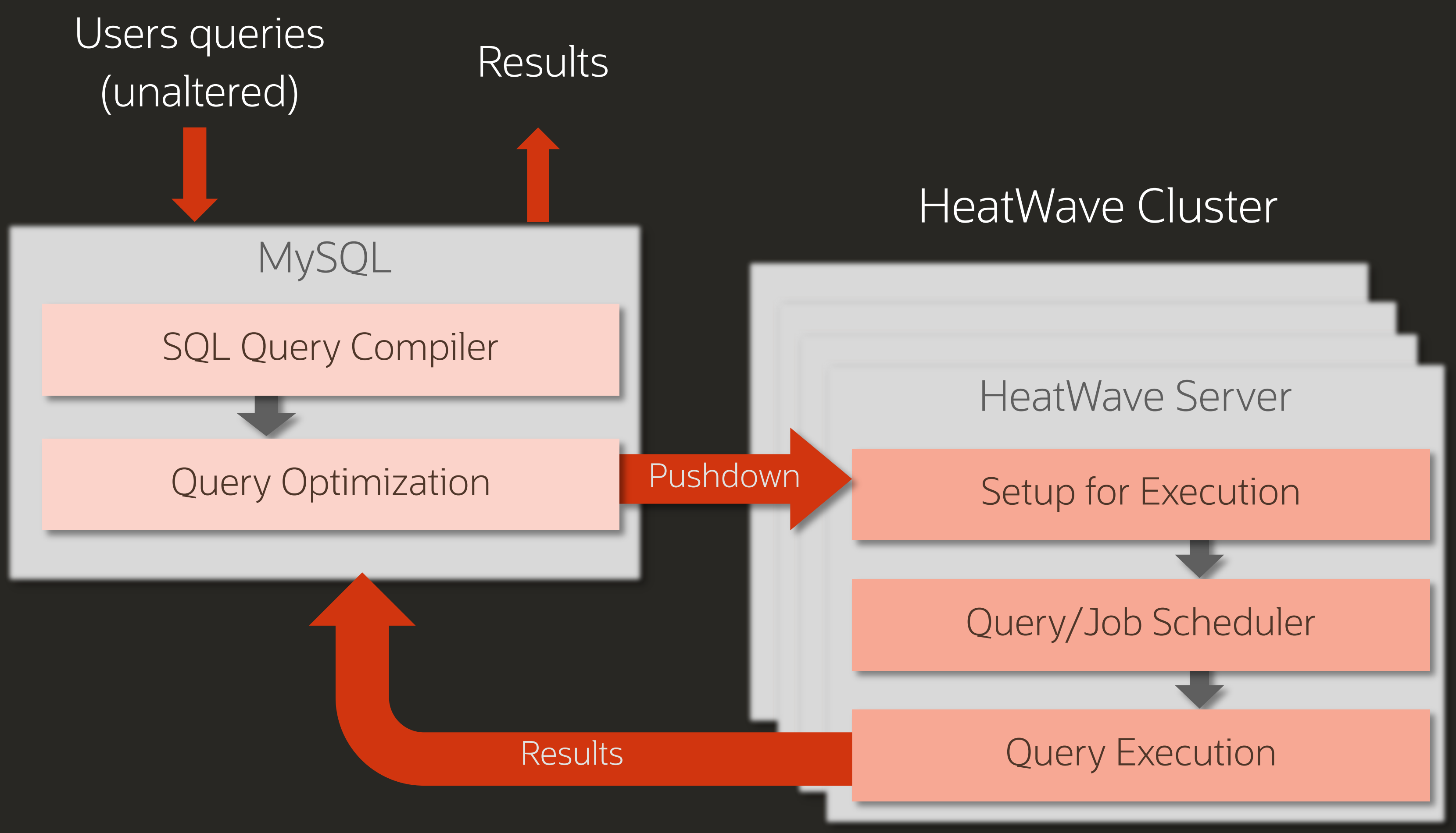
The Research Background on HeatWave

- Multi-year research project out of Oracle Labs with several publications (SIGMOD'18, MICRO'17, BigData'16, ICDE'16) and patents
- Project **RAPID**: Initial research project focused on software-hardware co-design with power/performance efficiency
- Scalable software design and architecture completely tech transferred to HeatWave
- Further SW enhancements and cloud tuning to compensate the lack of specialized hardware

Slide from SIGMOD 2018



Query Processing Architecture



MySQL HeatWave Analytics/OLAP Engine

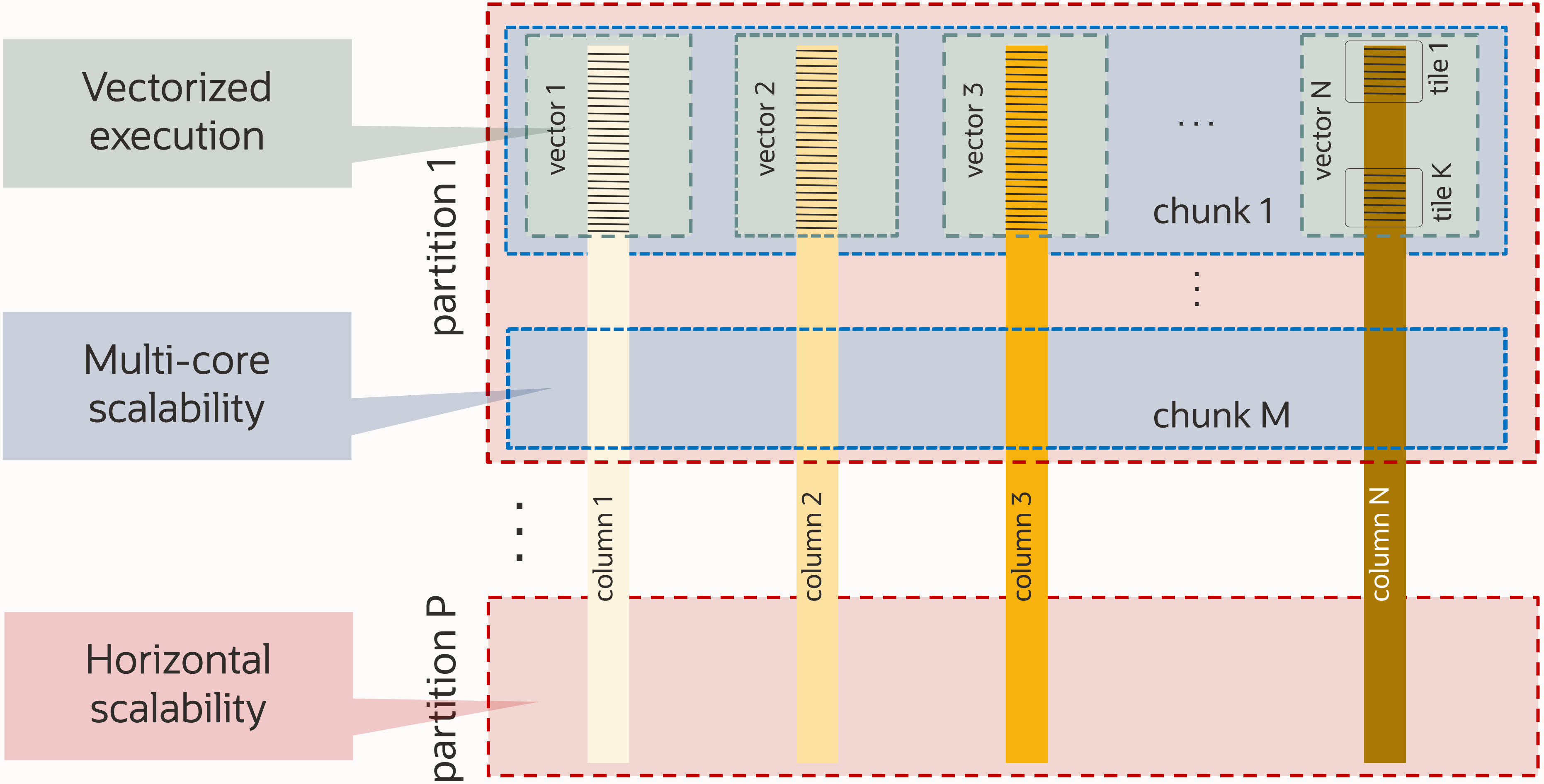
Architected for massive scale & performance

1 In-Memory, hybrid columnar processing

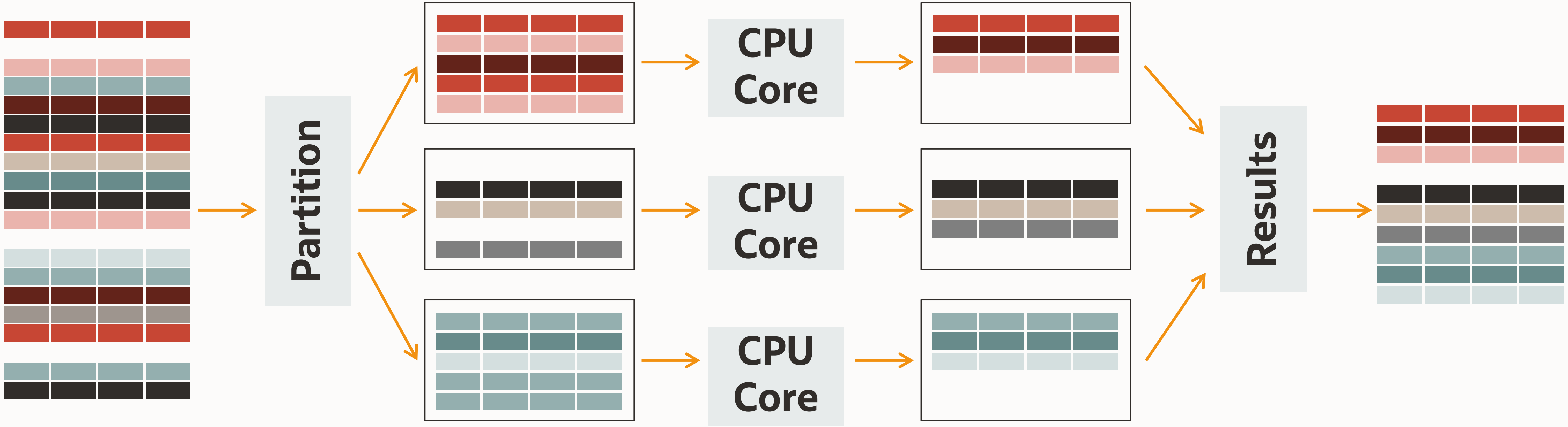
2 Massive inter- and intra-node parallelism optimized for cloud (OCI & AWS)

3 Distributed query processing algorithms (state-of-the art, based on research)

1. In-Memory hybrid columnar processing



2. Massively parallel architecture



- High-fanout workload-aware partitioning
- Machines & CPU cores can further process partitioned data in parallel
- Optimized for cache size and memory hierarchy of underlying hardware

3. Distributed algorithms optimized for cloud (OCI & AWS)

Partition data to fit into cache

- Partition at near memory bandwidth with shape specific optimizations (e.g. hash computation)
- Ensure partitions reside in CPU cache

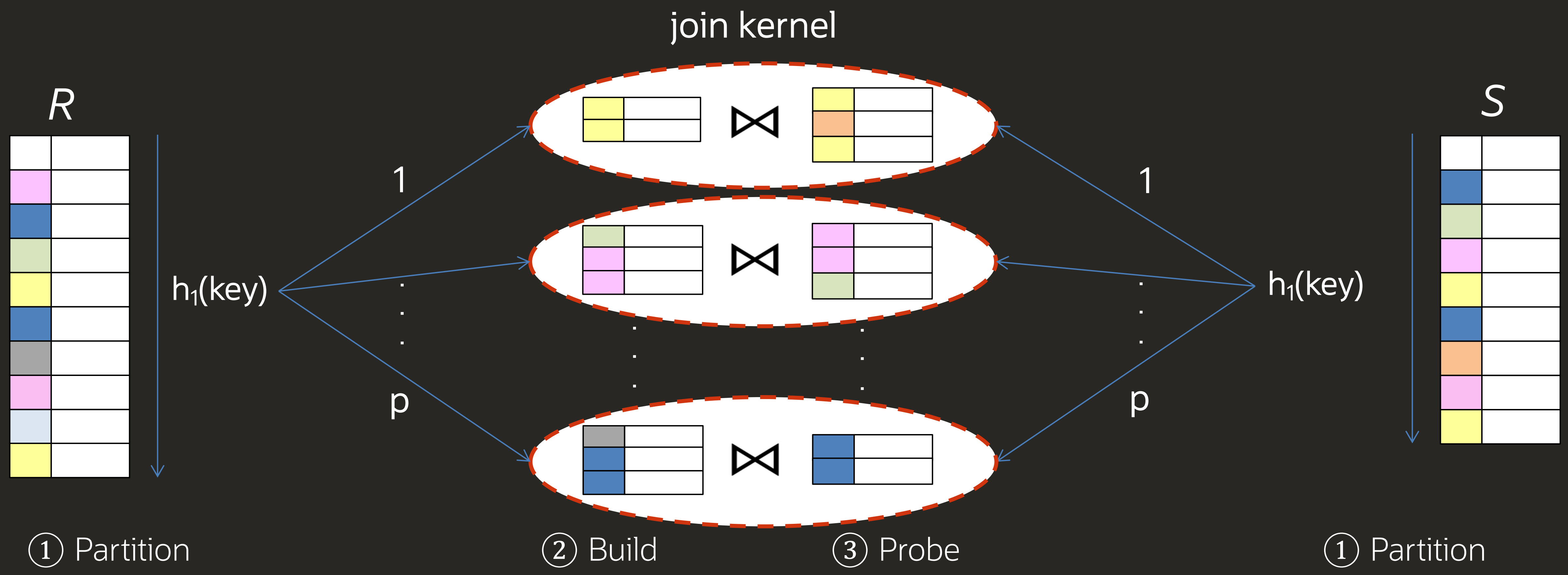
Process partitions as fast as possible

- Highly vectorized build & probe join kernels
- Hardware-conscious, hand-tuned primitives (e.g. using wide SIMD, AVX2 registers)

Overlap compute with communication

- Network optimizations for cloud (OCI) interconnect
- Intelligent scheduling of execute & transfer

State-of-the-Art Vectorized Hash Join

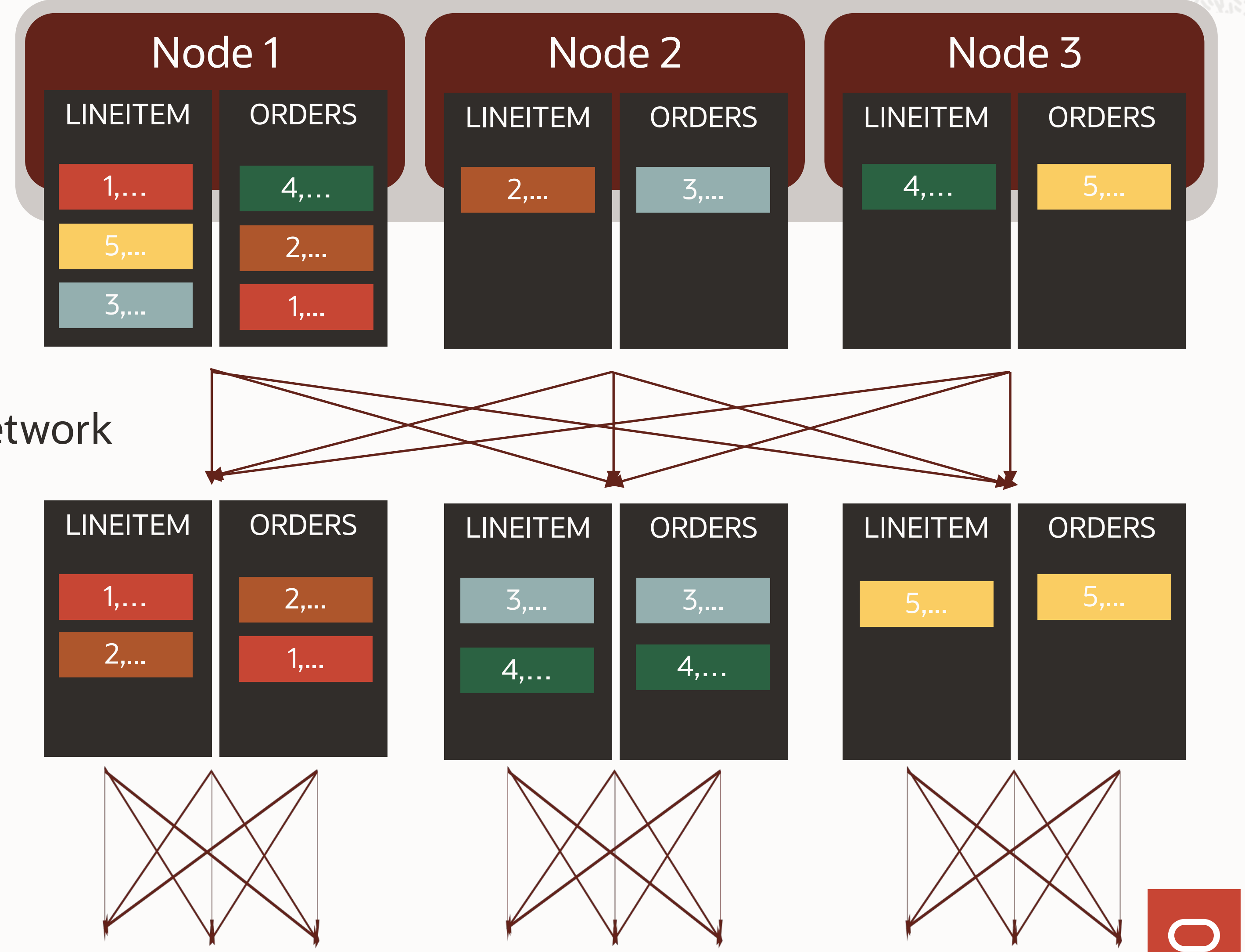


Each CPU core executes a **join kernel** between small R and S partitions
Hash tables are typically compact and fits into lower level CPU caches

C. Balkesen; J. Teubner; G. Alonso; M. T. Özsu, Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware, IEEE ICDE 2013

Distributed Relational Join using Vectorized Hash Join Kernels

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;

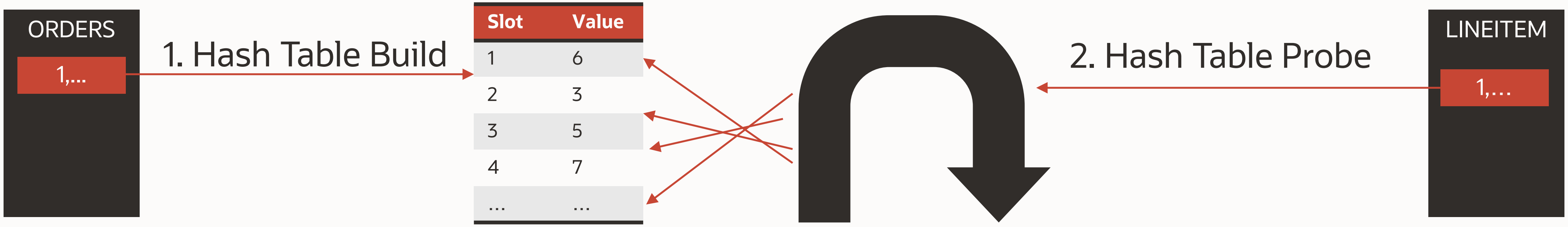


System Partitioning Over Network

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

How much do we need to partition?

Thread-Local Hash Join



Heavy Random Access

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

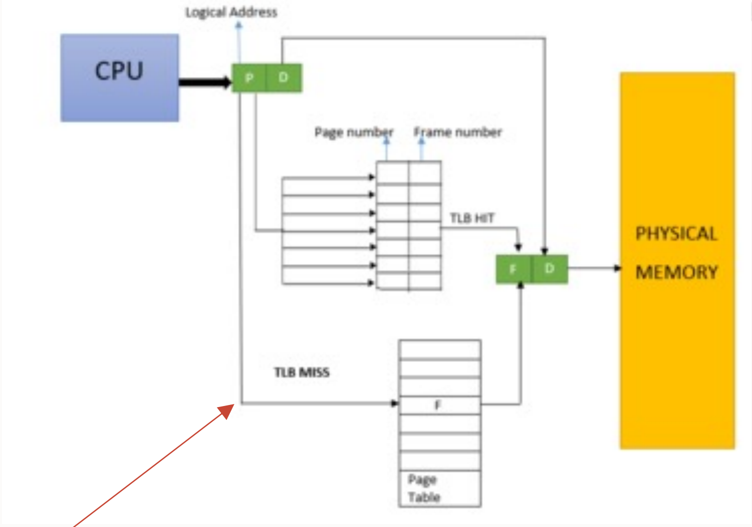
<u>Back of the envelope</u>

- 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.



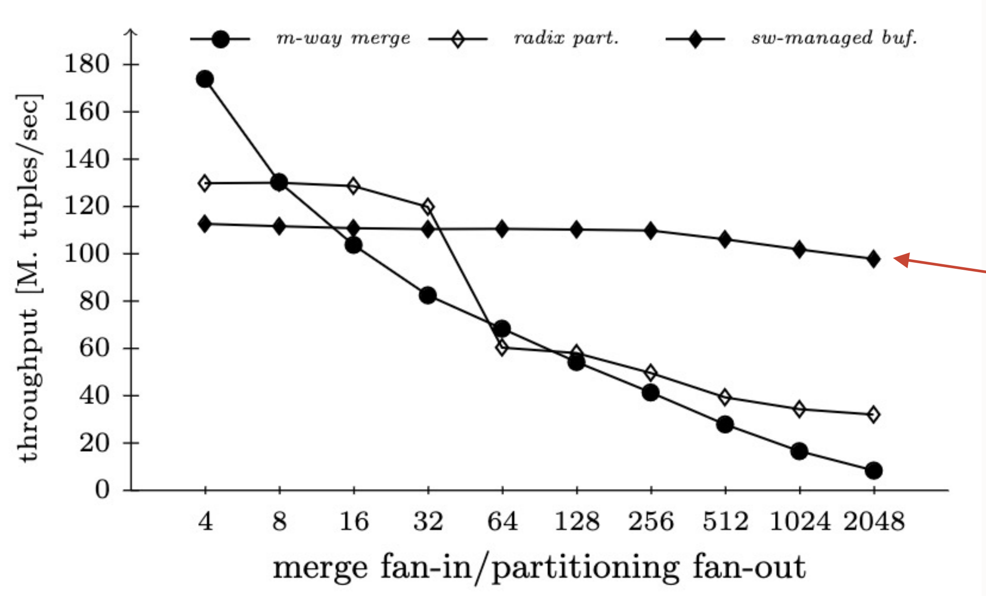
<u>TLB (Translation Lookaside Buffer)</u>

- 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; page walk might perform multiple memory accesses.
- Typical TLB has around 64-512 entries.
- Any partitioning fanout larger than 512 is likely to cause a huge performance impact.

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.



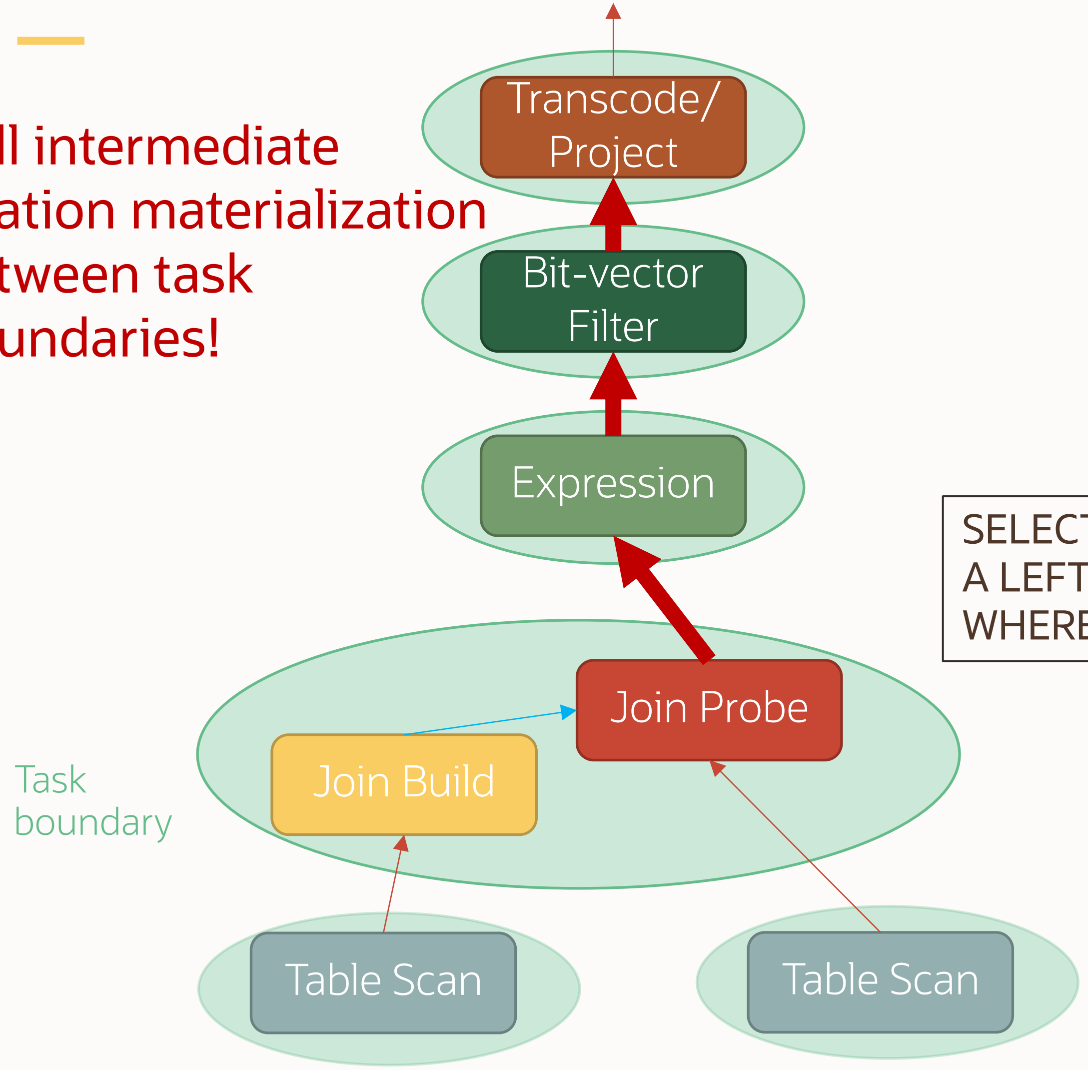
- SW-managed buffers are a good optimization for the TLB bottleneck, yet higher partitioning fanout can still cause a problem.

[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

Memory-Conscious Query Processing

Operator Pipelining and Fusing

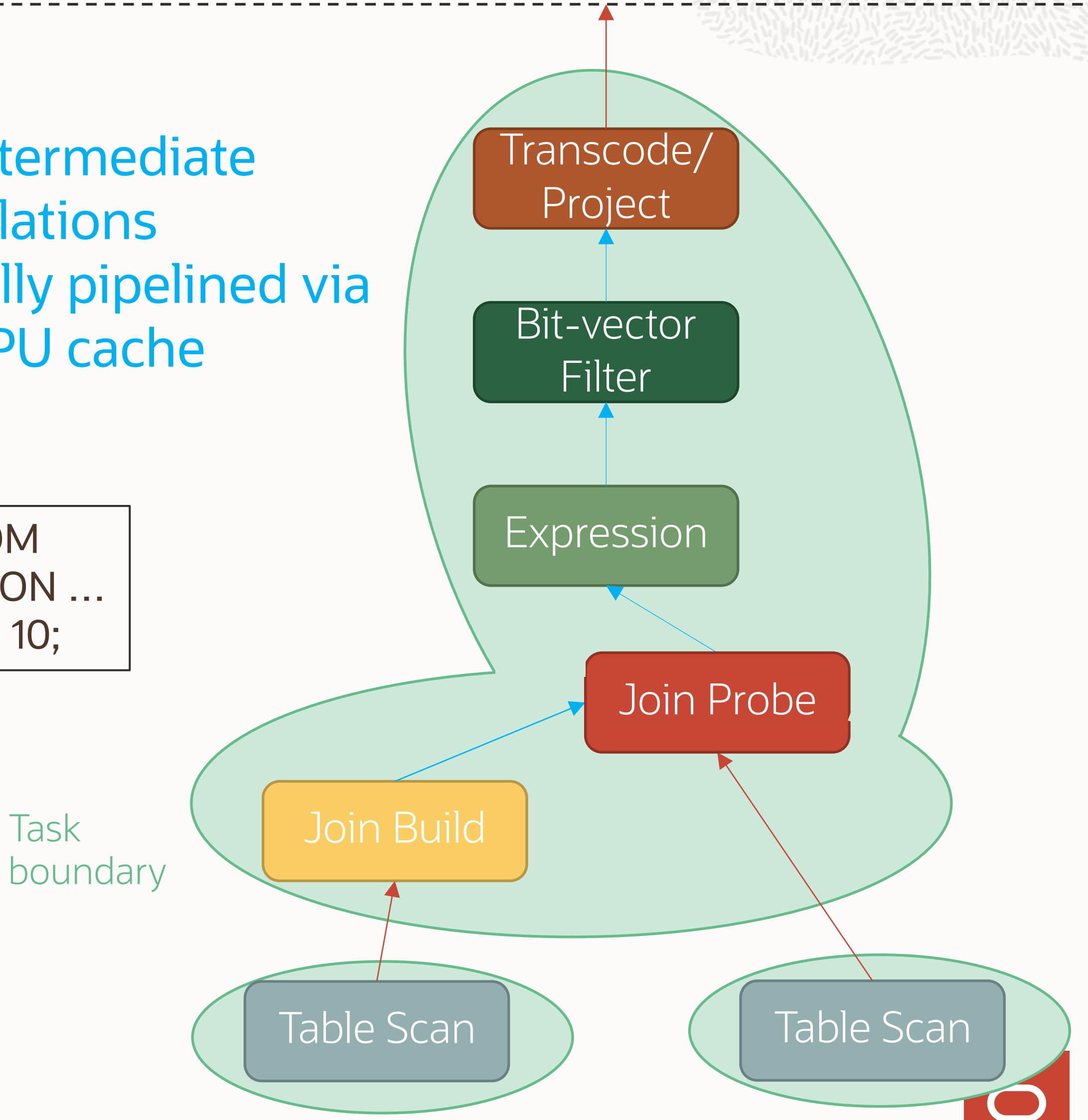
Full intermediate relation materialization between task boundaries!



vs.

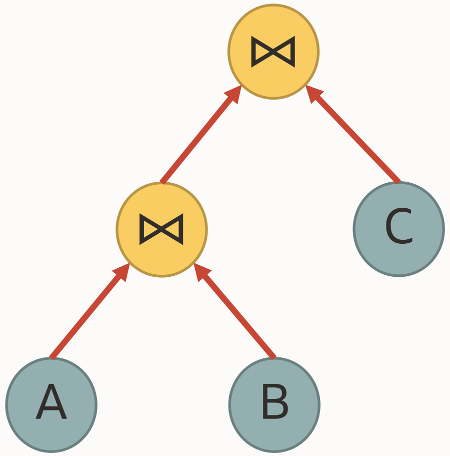
SELECT … FROM
A LEFT JOIN B ON …
WHERE A + B > 10;

Intermediate relations fully pipelined via CPU cache



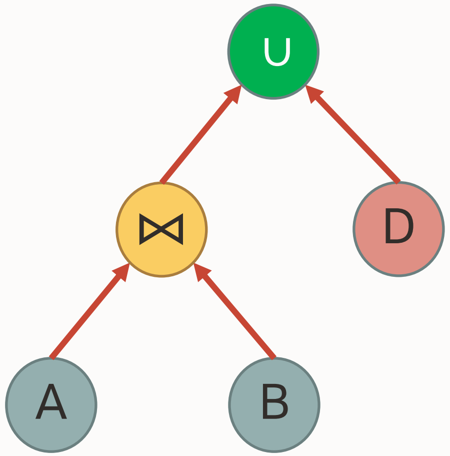
Auto Query Plan Improvement

Optimizer learns and improves query plan based on queries executed earlier



Node	Statistics
A	70
B	150
A ⋈ B	1000
C	…
A ⋈ B ⋈ C	…

Runtime statistics

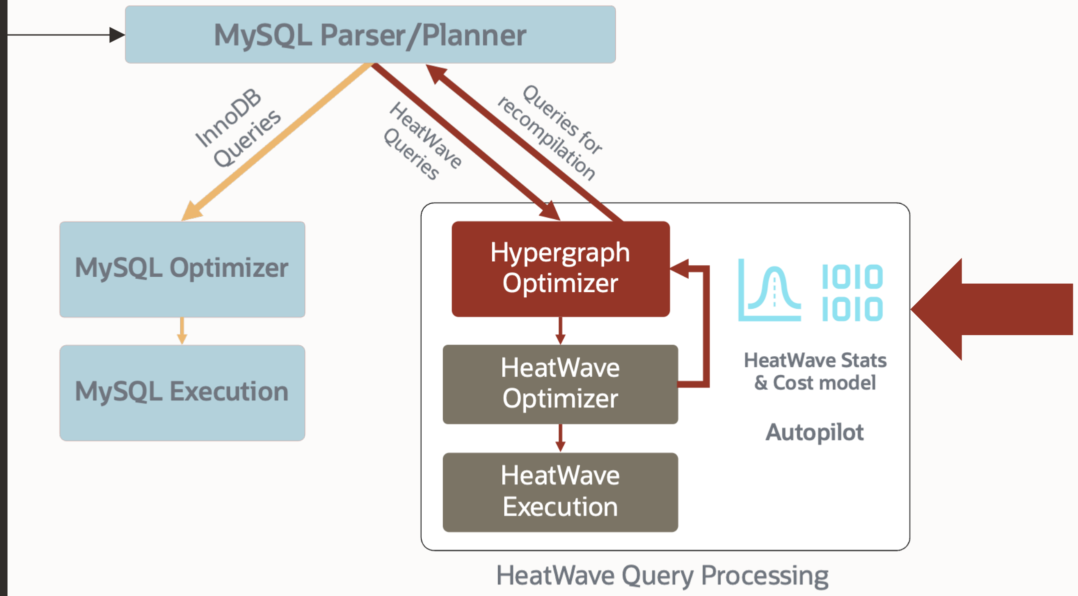


- Traditional caching techniques are not intelligent
- With Autopilot, system gets better as more queries are run
- 24TB TPC-H, TPC-DS performance improved by 40%

Query Optimization: Holistic Optimization

TPC-H Query 21

```sql
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;
```



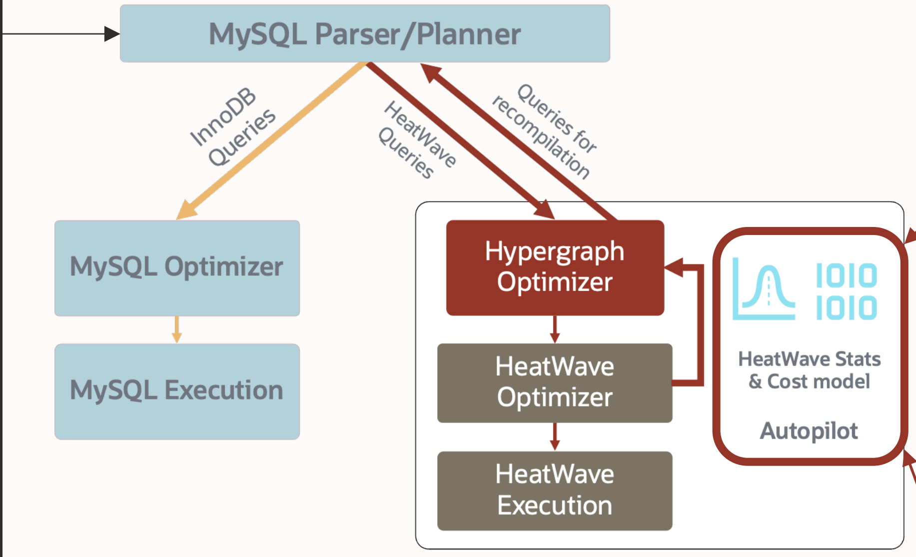
Hypergraph Optimizer gets information from HeatWave Optimizer per subgraph:

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

Query Optimization: A Detailed Overview

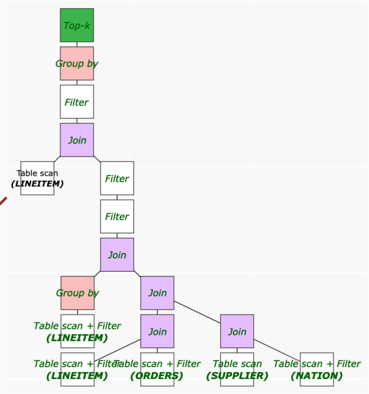
TPC-H Query 21

```
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;
```

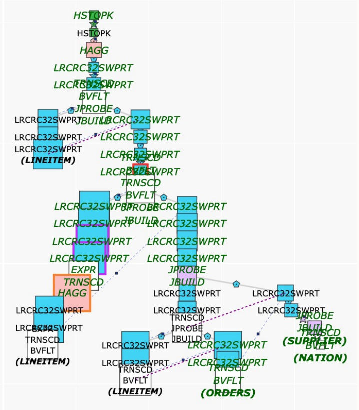


HeatWave Query Processing

Logical Plan



Physical Plan

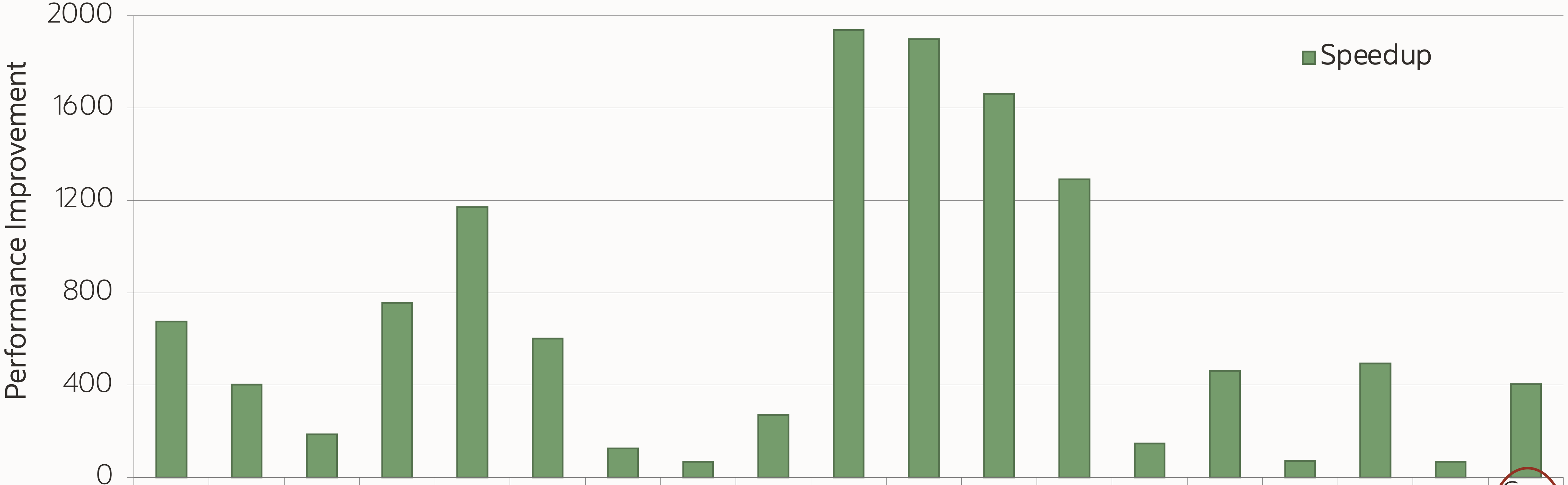


Logical plan decisions:
- Nodes are matched to previous runs to obtain accurate cardinality results (filter selectivity, join cardinality)
- Logical transformations (subquery pushdown)
- …

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

HeatWave dramatically speeds up analytic queries: 400x Faster

Improvement over MySQL 8.0 on TPC-H queries (400G, 64 cores)



	Q1	Q3	Q4	Q5	Q6	Q7	Q8	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q18	Q19	Q21	Q22	Geo Mean
Speedup	675	403	186	756	1171	602	125	68	271	1938	1899	1661	1291	148	462	72	494	69	404

MySQL HeatWave

MySQL database service with a massively-scalable integrated analytics engine

Single MySQL database for OLTP & analytics applications

All existing applications work without changes

Extreme performance: Accelerates MySQL for OLAP queries by orders of magnitude, scales to thousands of cores

Dramatically faster and lower cost

