Excursion: Radix Partitioned Hash Join

Query Processing – Joins
What is a Join?

Combination of tuples from different tables

General categories

- *Inner Join* – Combine the tuples of two tables by combining each tuple of the first input table with each tuple of the second table to apply a join-predicate, which joins two tuples only if they match
- *Outer Join* – If the join predicate matches, tuples from both join relations are used, if not, the non-matching tuple is filled with NULL values

Further Specializations

- *Equi-Join* – most often used join-type, e.g., `tbl_a.attr_1=tbl_b.attr_1`
- *Inequality-Join* – often used in time-series analyses, e.g., joining over range predicates

Radix Join

Query Processing – Joins
Join Execution in Main Memory

The three basic Join Algorithms

1. Nested-Loop Join
2. Sort-Merge Join
   - Principle of merging sorted lists
3. Hash-Based Join
   - Build hash map for smaller join relation
   - Sequential scan over larger join relation and probe into hash map

Radix Join

Query Processing – Joins
Nested-Loop Join

Despite its runtime complexity of *O(m*n)* one of the most used joins

Reasons

- Low memory consumption
- Runtime complexity often neglectable for small inputs (e.g., small input sizes due to restriction upfront filtering)
- Often the default join operator when an join attributes is indexed (e.g., joining foreign key partners)

Radix Join

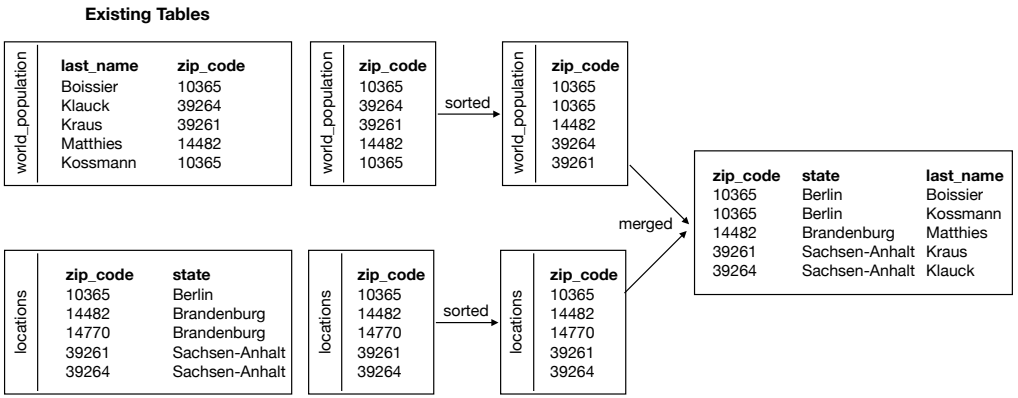
Query Processing – Joins
Sort-Merge Join

Algorithm

- Sort both join keys
- Merge both sorted lists to join

- Fully sequential operation
- With modern CPU-optimized sorting techniques (e.g., bitonic sort) sort-merge joins can outperform hash joins
- Fast for already sorted tables

Sort-Merge-Join



Radix Join

Query Processing – Joins
Which Join Algorithms to choose?

Hasso Plattner Institut

When to choose which algorithm?

- *Nested-Loop Join* – very small data set; both other algorithms would require too much time for additional data structures and sorting; indexed join column
- *Sort-Merge Join* – join column(s) already sorted, inequality joins, default fallback
- *Hash Join* – one relation significantly smaller; equi-join

Complexity comparison

- *Nested-Loop Join – O(N*M)*
- *Sort-Merge Join – O(N*log(N) + M*log(M))*
- *Hash Join – O(N+M)*

But

- *Sort-merge Join* on already sorted input has a complexity of *O(N+M)* since the data does not need to be sorted
- *Hash Join* performs best if the hash map is expected to be very small (thus probably cache-resident)

Radix Join

Query Processing – Radix-Partitioned Hash Join
Joining Large Data Sets

Hyrise[2] is optimized for mixed workloads including complex analytical joins over large data sets. Joining is the *most expensive operation* in mixed workloads.

Text book implementations do not scale well on large multi-core systems:

1. *Hash Join:* random accesses to large non-cache-resident hash maps are not prefetchable and exhibit bad caching utilization.
2. *Sort-based Join:* overly expensive sorting costs in the sort-phase for very large data sets.

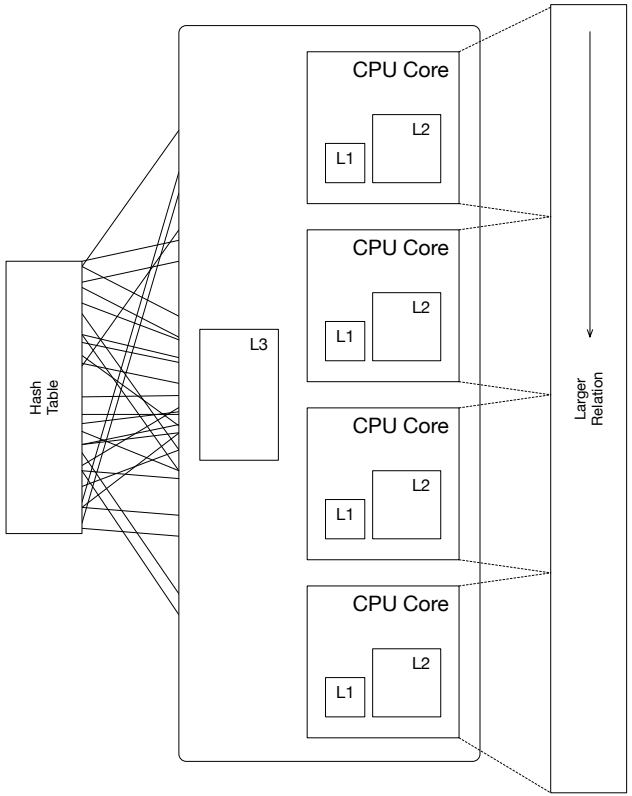
Radix Join

Query Processing – Radix-Partitioned Hash Join

- Larger relation is sequentially scanned in parallel

But …

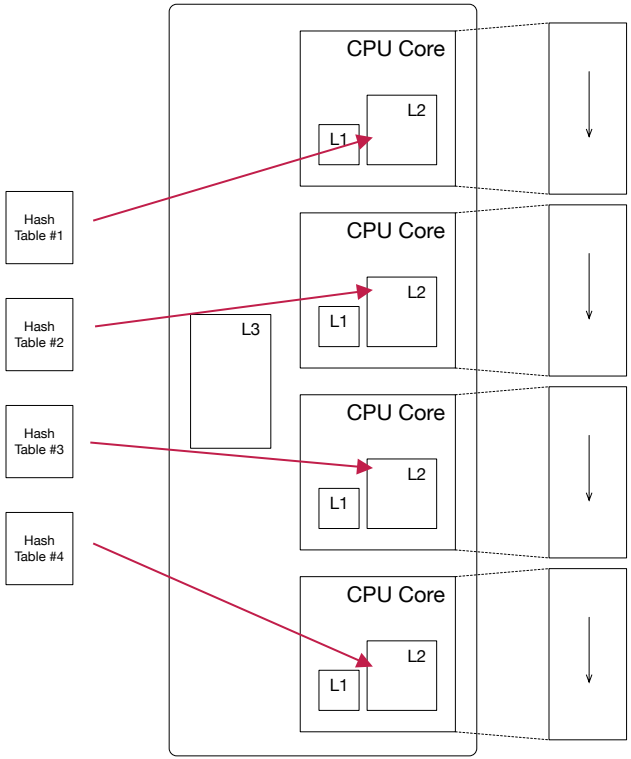
- Hash table of smaller relation is larger than cache sizes
- CPU cores are randomly accessing the hash table which potentially leads to an L3 cache miss for every access



Radix Join

Query Processing – Radix-Partitioned Hash Join

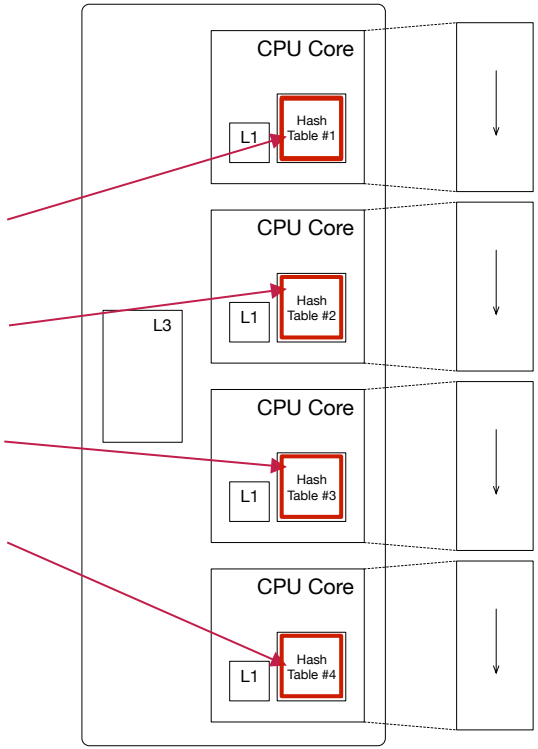
- Key idea: partition both relation in smaller partitions
- We aim to chose a partition size such that the hash tables can be cache-resident



Radix Join

Query Processing – Radix-Partitioned Hash Join

- CPU cores still process larger relation sequentially
- Random accesses during probing phase are now fully cache-resident in L2 or L3 cache
- Usually, the added work to partition the data is offset by faster joining phase



Radix Join

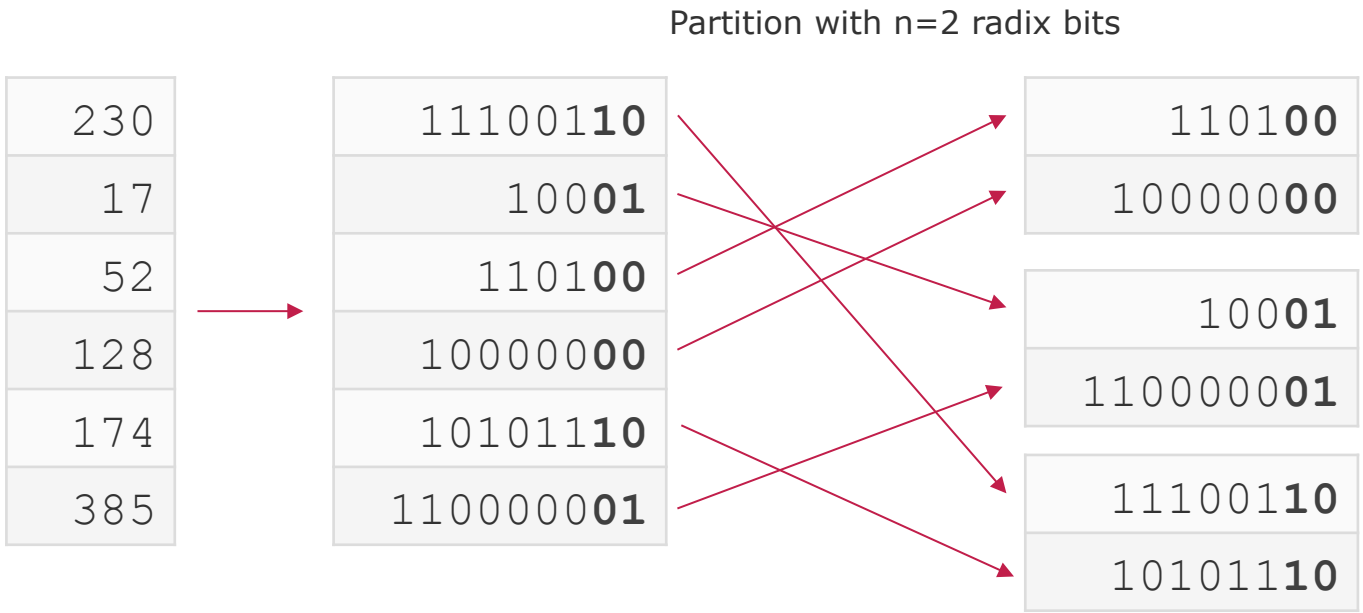
Query Processing – Radix-Partitioned Hash Join
Joining Large Data Sets

- Parallelization should be used when load permits it
- *Idea:* add additional work to partition input relations into smaller partitions that can be joined locally

Radix Partitioning

- Partitions a data set by their *n* least significant bits
- Introduces (potentially multiple) additional scans of the data to join
- Apparently expensive, but surprisingly efficient on columnar IMDBs

Query Processing – Radix-Partitioned Hash Join
Radix Partitioning



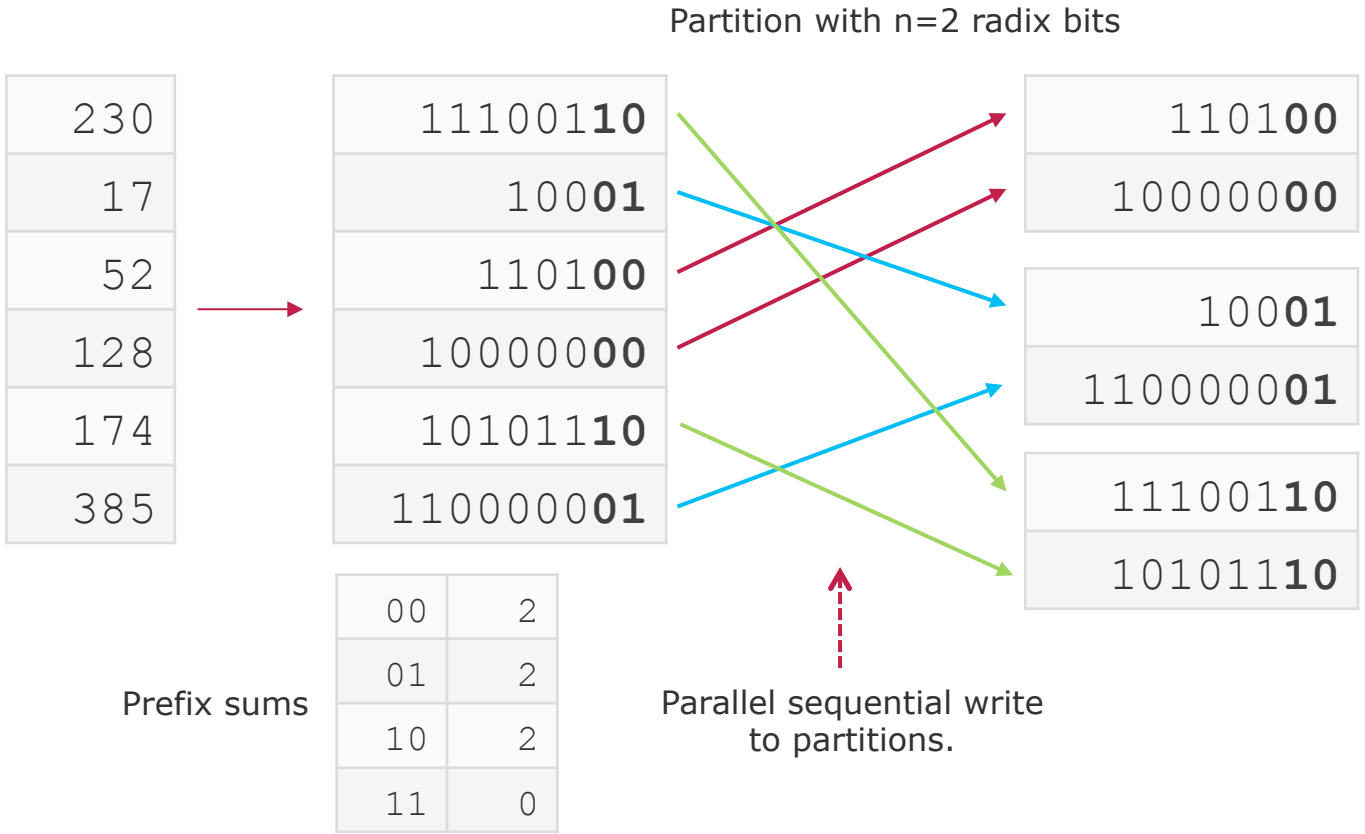
Radix Join

Query Processing – Radix-Partitioned Hash Join
Radix Partitioning

Process

- Determine number of radix bits *n,* prepare 2^n (atomic) counters
- First pass: scan entire join columns, determine radix bits for each value and increase the counter accordingly
- Allocate a pre-sized output vector using the calculated prefix sums of the counters
- Second pass: using the prefix sums, directly write values in the corresponding slot of the output vector (sequential writes)
- Usually straightforward to parallelize (in Hyrise^2 parallelized on chunk level)

Query Processing – Radix-Partitioned Hash Join
Radix Partitioning



Radix Join

Query Processing – Radix-Partitioned Hash Join
Radix Partitioning – Why?

- *Radix partitioning* is the fastest way to *range partition* data without expensive comparisons
- Using least significant bits avoids most data skews
- For equi-join, only radix clusters with the same radix bit pattern have to be joined
  - For a large enough number of radix bits, the hash tables will thus probably be cache-resident
  - Thus, random lookups but fully cached hash table

Query Processing – Radix-Partitioned Hash Join Summary

We learned that the join is one of the most expensive database operations

- As such, joins are one of the most tuned pieces of code in databases

We discussed the radix-partitioned hash join

- Radix-partitioning allows for improved cache locality of hash maps and
- Allows for improved parallelization

We learned that the optimal join-type choice depends on many aspects