Parallel Hash Join

Recap

Scheduling

- For each query plan, the DBMS must decide where, when, and how to execute it.
  - How many tasks should it use?
  - How many CPU cores should it use?
  - What CPU core should the tasks execute on?
  - Where should a task store its output?
- The DBMS always knows more than the OS.

Join Algorithms: Summary

Join Algorithm	IO Cost	Example
Simple Nested Loop Join	M + (m x N)	1.3 hours
Block Nested Loop Join	M + (M x N)	50 seconds
Index Nested Loop Join	M + (M x C)	Variable
Sort-Merge Join	M + N + (sort cost)	0.75 seconds
Hash Join	3 x (M + N)	0.45 seconds

Today's Agenda

- Background
- Partition Phase
- Build Phase
- Probe Phase
- Evaluation

Background

Parallel Join Algorithms

- Perform a join between two relations on multiple threads simultaneously to speed up operation.
- Two main approaches:
  - **Hash Join**
  - **Sort-Merge Join**
- We won't discuss nested-loop joins.

Observation

- Many OLTP DBMSs do **not** implement hash join.
- But an **index nested-loop join** with a small number of target tuples is at a high-level equivalent to a hash join.

Hashing vs. Sorting

- 1970s – Sorting (External Merge-Sort)
- 1980s – Hashing (Database Machines)
- 1990s – Equivalent
- 2000s – Hashing (For Unsorted Data)
- 2010s – Hashing (Partitioned vs. Non-Partitioned)
- 2020s – ???

Parallel Join Algorithms

SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUS
VLDB 2009
ORACLE intel

→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

DESIGN AND EVALUATION OF MAIN MEMORY HASH JOIN ALGORITHMS FOR MULTI-CORE CPUS
SIGMOD 2011
WISCONSIN UNIVERSITY OF WISCONSIN-MADISON

→ Trade-offs between partitioning & non-partitioning Hash-Join.

MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012
HyPer

→ Sort-Merge is already faster than Hashing, even without SIMD.

MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013
HyPer

→ Ignore what we said last year.
→ You really want to use Hashing!

MAIN-MEMORY HASH JOINS ON MULTI-CORE CPUS: TUNING TO THE UNDERLYING HARDWARE
ICDE 2013
Systems@ETH zürich

→ New optimizations and results for Radix Hash Join.

AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016
UNIVERSITÄT DES SAARLANDES

→ Hold up everyone! Let's look at everything more carefully!

Design Goals

- **Goal 1: Minimize Synchronization**
  - Avoid taking latches during execution.
- **Goal 2: Minimize Memory Access Cost**
  - Ensure that data is always local to worker thread.
  - Reuse data while it exists in CPU cache.

Improving Cache Behavior

- Factors that affect cache misses in a DBMS:
  - Cache + TLB capacity.
  - Locality (temporal and spatial).
- **Sequential Access (Scan):**
  - Clustering data to a cache line.
  - Execute more operations per cache line.
- **Random Access (Lookups):**
  - Partition data to fit in cache + TLB.

Parallel Hash Join

- Hash join is the most important operator in a DBMS for OLAP workloads.
- It is important that we speed up our DBMS's join algorithm by taking advantage of multiple cores.
- We will focus on in-memory DBMSs.
    - We want to keep all cores busy, without becoming memory bound.

Hash Join

- **Phase 1: Partition (optional)**
  - Divide the tuples of **R** and **S** into sets using a hash on the join key.
- **Phase 2: Build**
  - Scan relation **R** and create a hash table on join key.
- **Phase 3: Probe**
  - For each tuple in **S**, look up its join key in hash table for **R**. If a match is found, output combined tuple.
- Reference

Partition Phase

Partition Phase

- Split the input relations into partitioned buffers by hashing the tuples' join key(s).
  - Ideally the cost of partitioning is **less** than the cost of cache misses during build phase.
  - *a.k.a.*, **hybrid hash join** / **radix hash join**.
- Contents of buffers depends on storage model:
  - **NSM:** Usually the entire tuple.
  - **DSM:** Only the columns needed for the join + offset.

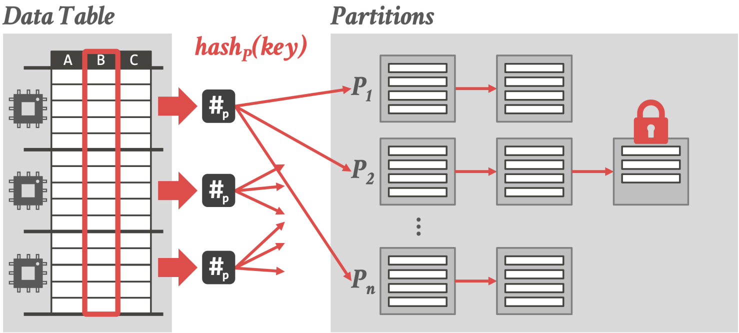
Partition Phase

- **Approach 1: Non-Blocking Partitioning**
  - Only scan the input relation once.
  - Produce output incrementally.
- **Approach 2: Blocking Partitioning (Radix)**
  - Scan the input relation multiple times.
  - Only materialize results all at once.
  - *a.k.a.*, radix hash join.

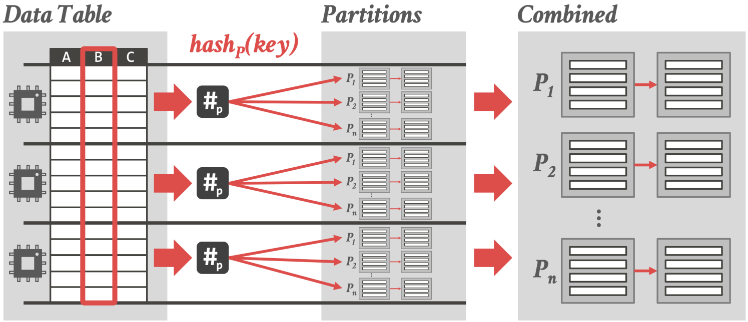
Non-Blocking Partitioning

- Scan the input relation only once and generate the output on-the-fly.
- **Approach 1: Shared Partitions**
  - Single global set of partitions that all threads update.
  - Must use a latch to synchronize threads.
- **Approach 2: Private Partitions**
  - Each thread has its own set of partitions.
  - Must consolidate them after all threads finish.

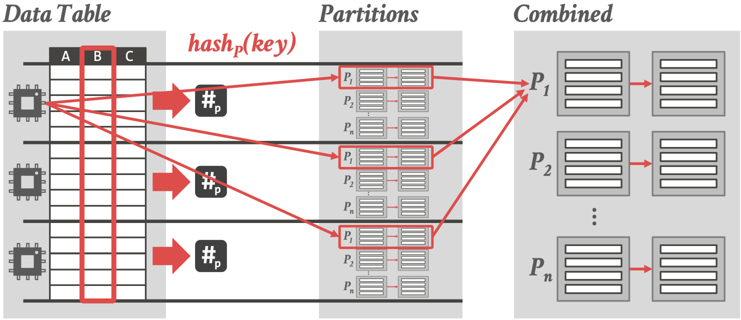
Shared Partitions



Private Partitions



Private Partitions

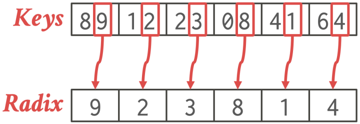


Blocking Partitioning (Radix Partitioning)

- Scan the input relation multiple times to generate the partitions.
- No need to synchronize.
- Multi-step pass over the relation:
  - Step 1: Scan **R** and compute a histogram of the number of tuples per hash key for the radix at some offset.
  - Step 2: Use this histogram to determine output offsets by computing the **prefix sum**.
  - Step 3: Scan **R** again and partition them according to the hash key.

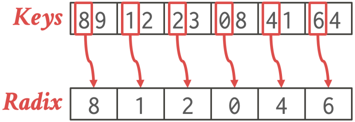
Radix

- The radix of a key is the value of an integer at a position (using its base).



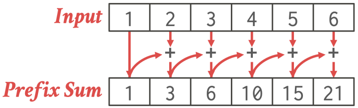
Radix

- The radix of a key is the value of an integer at a position (using its base).

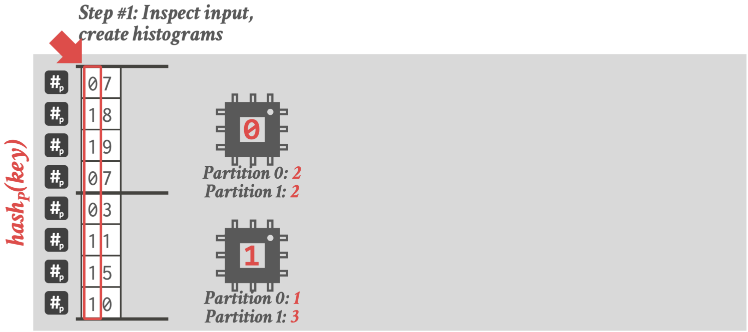


Prefix Sum

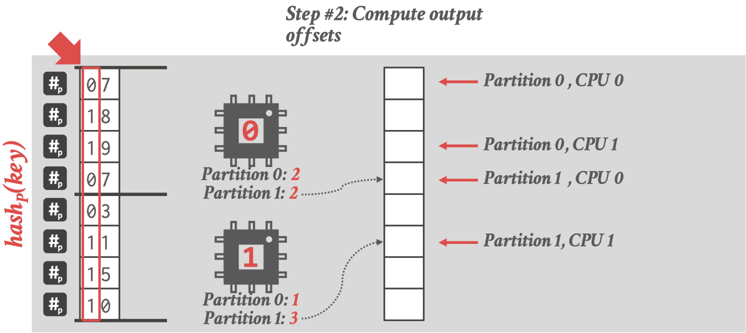
- The **prefix sum** of a sequence of numbers $(x_0, x_1, \ldots, x_n)$ is a second sequence of numbers $(y_0, y_1, \ldots, y_n)$ that is a running total of the input sequence.



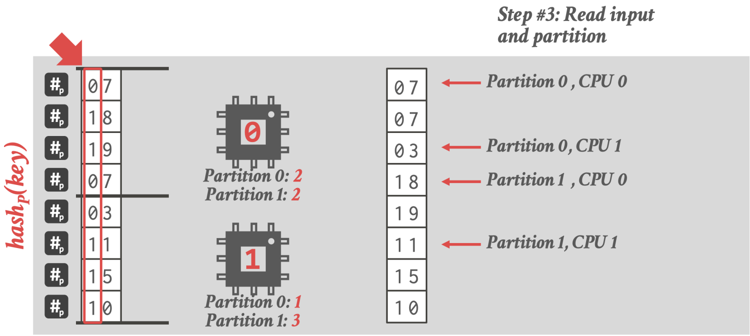
Radix Partitions



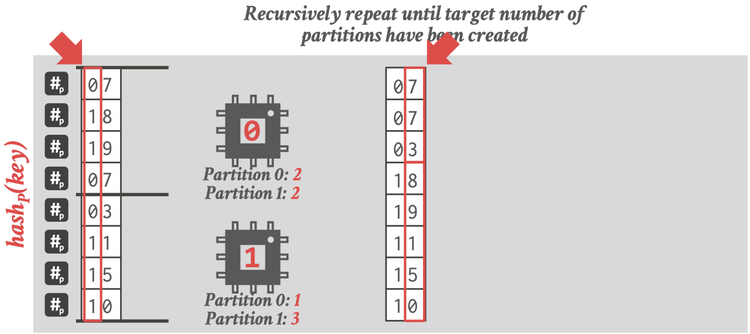
Radix Partitions



Radix Partitions



Radix Partitions



Build Phase

Build Phase

- The threads are then to scan either the tuples (or partitions) of **R**.
- For each tuple, hash the join key attribute for that tuple and add it to the appropriate bucket in the hash table.
  - The buckets should only be a few cache lines in size.

Hash Table

- **Design Decision 1: Hash Function**
  - How to map a large key space into a smaller domain.
  - Trade-off between being fast vs. collision rate.
- **Design Decision 2: Hashing Scheme**
  - How to handle key collisions after hashing.
  - Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.

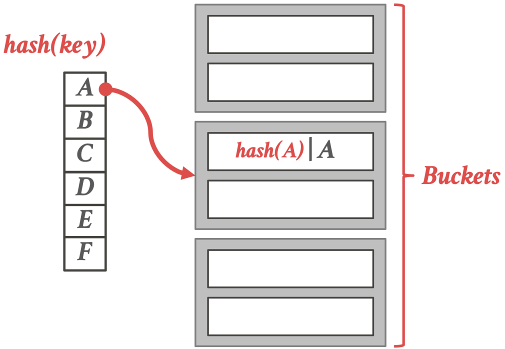
Hashing Schemes

- Approach 1: Chained Hashing (Dynamic)
- Approach 2: Linear Probe Hashing (Static)
- Approach 3: Robin Hood Hashing (Static)
- Approach 4: Hopscotch Hashing (Static)
- Approach 5: Cuckoo Hashing (Static)

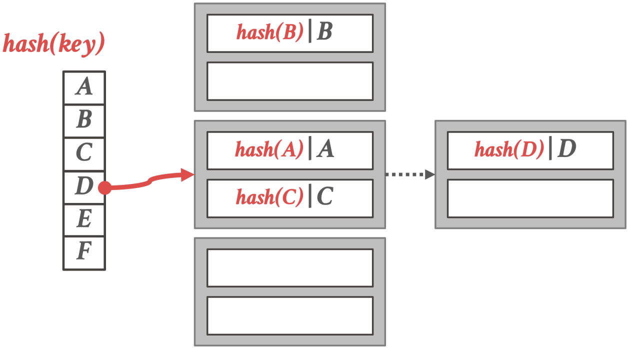
Chained Hashing

- Maintain a linked list of **buckets** for each slot in the hash table.
- Resolve collisions by placing all elements with the same hash key into the same bucket.
  - To determine whether an element is present, hash to its bucket and scan for it.
  - Insertions and deletions are generalizations of lookups.

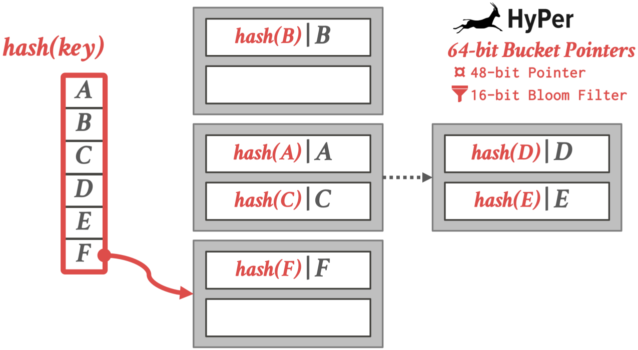
Chained Hashing



Chained Hashing



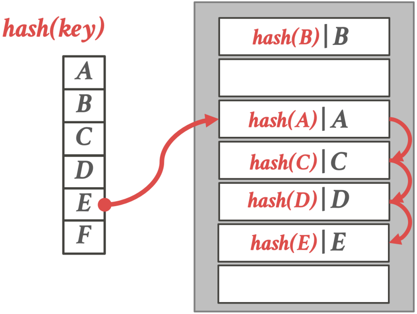
Chained Hashing



Linear Probe Hashing

- Single giant table of slots.
- Resolve collisions by linearly searching for the next free slot in the table.
  - To determine whether an element is present, hash to a location in the table and scan for it.
  - Must store the key in the table to know when to stop scanning.
  - Insertions and deletions are generalizations of lookups.

Linear Probe Hashing



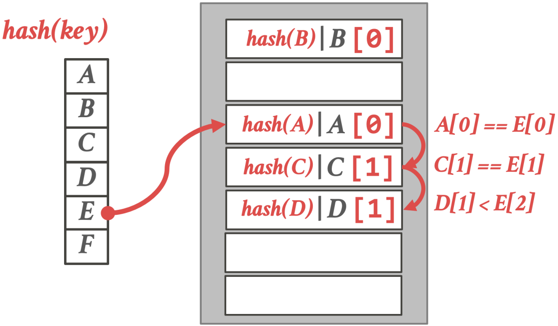
Observation

- To reduce the number of wasteful comparisons during the join, it is important to avoid collisions of hashed keys.
- This requires a chained hash table with $2\times$ the number of slots as the number of elements in $\underline{\mathbf{R}}$.

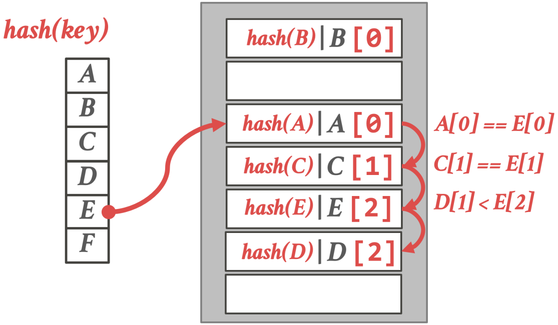
Robin Hood Hashing

- Variant of linear probe hashing that steals slots from **rich** keys and give them to **poor** keys.
  - Each key tracks the number of positions they are from where its optimal position in the table.
  - On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

Robin Hood Hashing



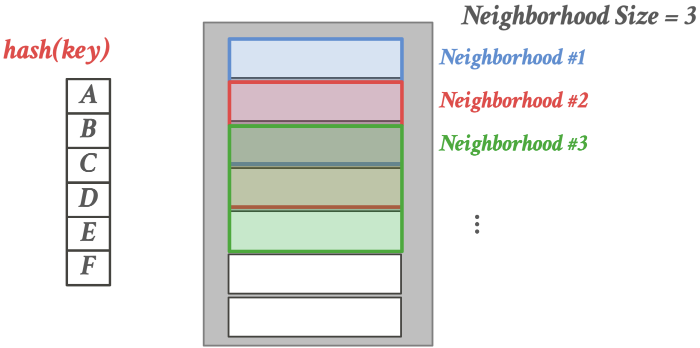
Robin Hood Hashing



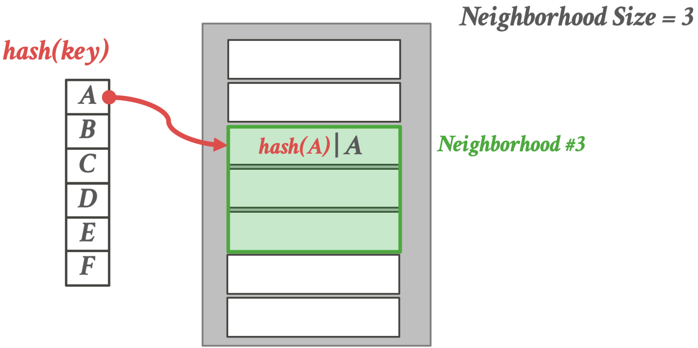
Hopscotch Hashing

- Variant of linear probe hashing where keys can move between positions in a **neighborhood**.
  - A neighborhood is contiguous range of slots in the table.
  - The size of a neighborhood is a configurable constant.
- A key is guaranteed to be in its neighborhood or not exist in the table.

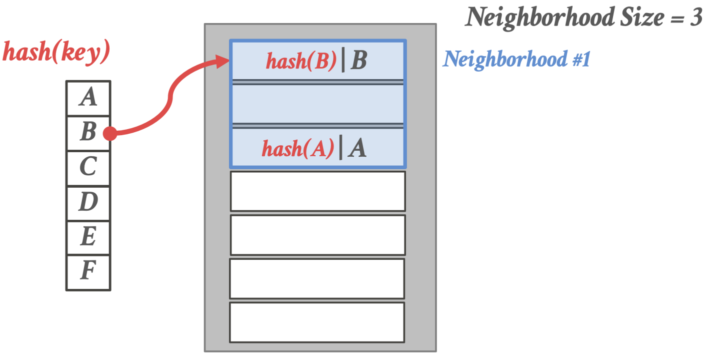
Hopscotch Hashing



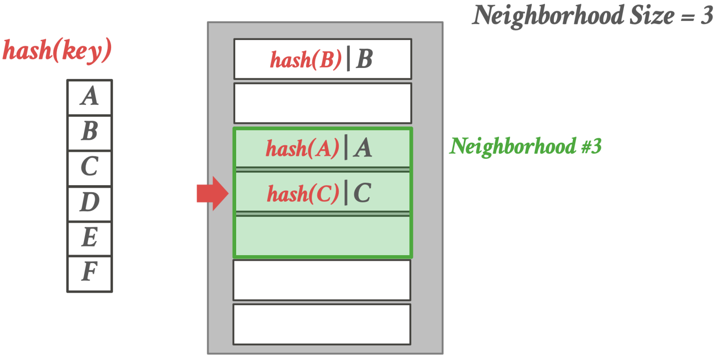
Hopscotch Hashing



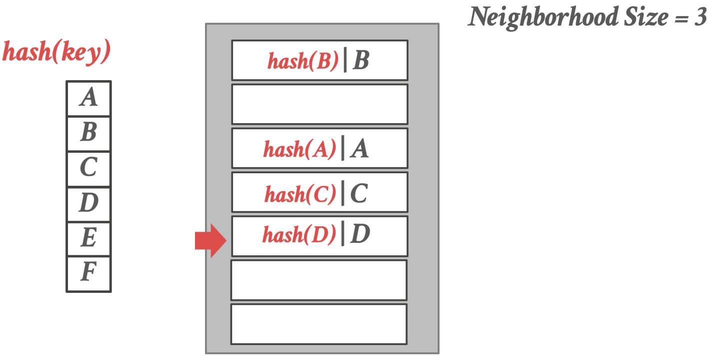
Hopscotch Hashing



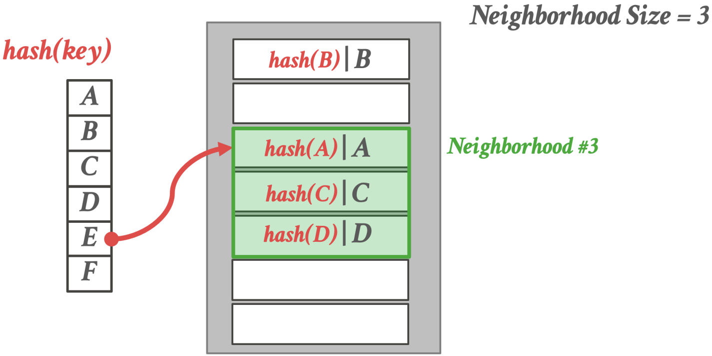
Hopscotch Hashing



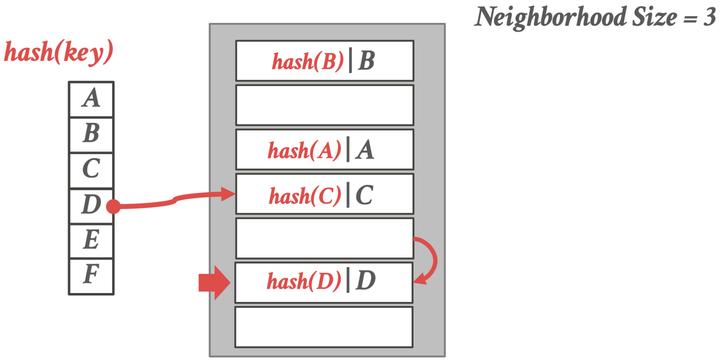
Hopscotch Hashing



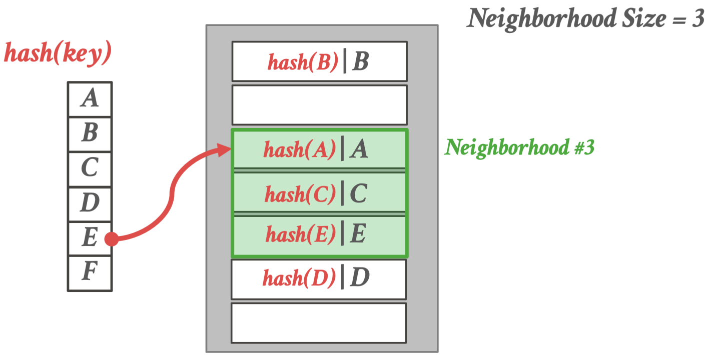
Hopscotch Hashing



Hopscotch Hashing



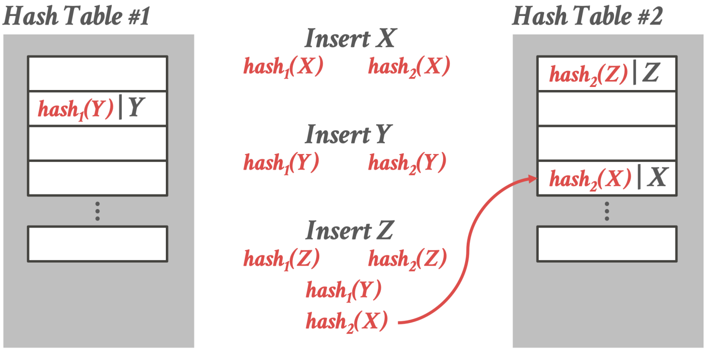
Hopscotch Hashing



Cuckoo Hashing

- Use <u>**multiple tables**</u> with different hash functions.
  - On insert, check every table and pick anyone that has a free slot.
  - If no table has a free slot, evict the element from one of them and then re-hash it find a new location.
- Look-ups are always O(1) because only one location per hash table is checked.

Cuckoo Hashing



Probe Phase

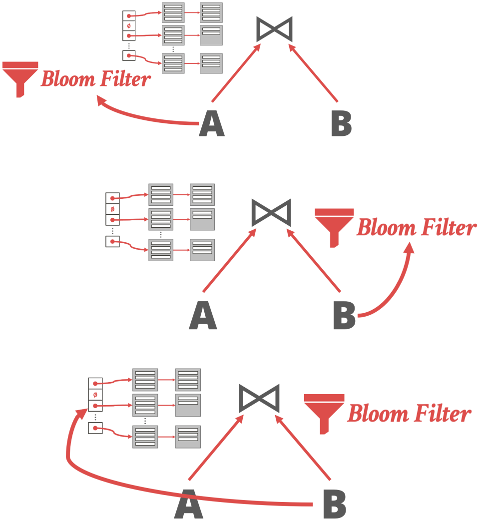
Probe Phase

- For each tuple in <u>**S**</u>, hash its join key and check to see whether there is a match for each tuple in corresponding bucket in the hash table constructed for <u>**R**</u>.
  - If inputs were partitioned, then assign each thread a unique partition.
  - Otherwise, synchronize their access to the cursor on <u>**S**</u>.

Probe Phase – Bloom Filter

- Create a Bloom Filter during the build phase when the key is likely to not **exist** in the hash table.
  - Threads check the filter before probing the hash table.
  - This will be faster since the filter will fit in CPU caches.
  - *a.k.a.*, called sideways information passing.

Probe Phase – Bloom Filter



Evaluation

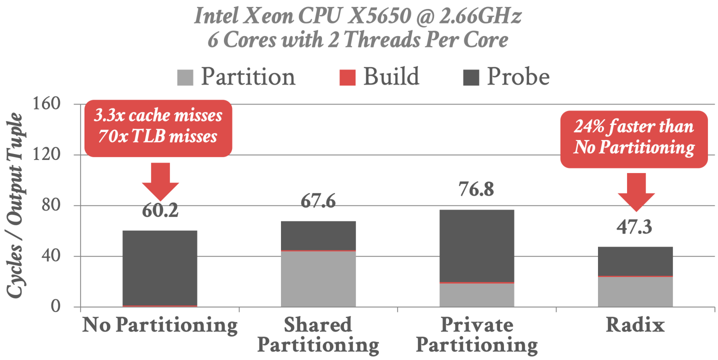
Hash Join Variants

	No-P	Shared-P	Private-P	Radix
Partitioning	No	Yes	Yes	Yes
Input scans	0	1	1	2
Sync during partitioning	–	Spinlock per tuple	Barrier	Barriers
Hash table	Shared	Private	Private	Private
Sync during build phase	Yes	No	No	No
Sync during probe phase	No	No	No	No

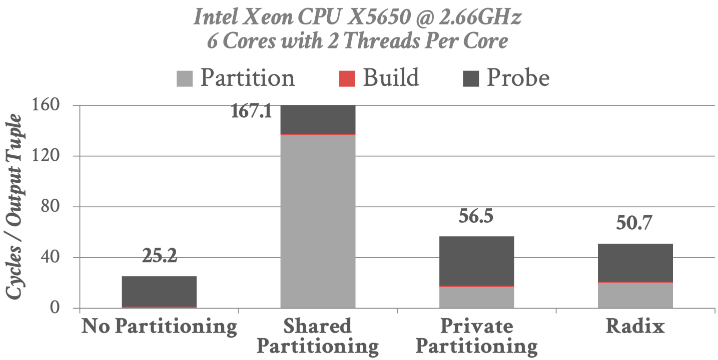
Benchmarks

- Primary key – foreign key join
  - Outer Relation (Build): 16 M tuples, 16 bytes each
  - Inner Relation (Probe): 256 M tuples, 16 bytes each
- Uniform and highly skewed (Zipf; s=1.25)
- No output materialization
- Reference

Hash Join - Uniform Dataset



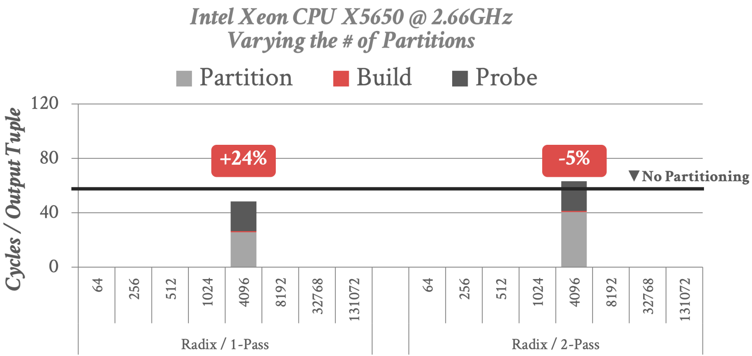
Hash Join - Skewed Dataset



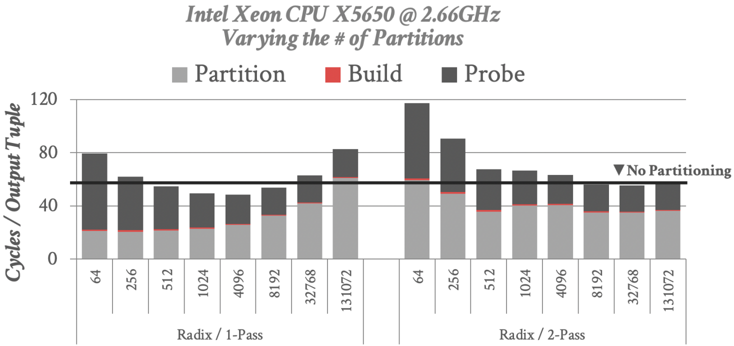
Observation

- We have ignored a lot of important parameters for all these algorithms so far.
  - Whether to use partitioning or not?
  - How many partitions to use?
  - How many passes to take in partitioning phase?
- In a real DBMS, the optimizer will select what it thinks are good values based on what it knows about the data (and maybe hardware).

Radix Hash Join - Uniform Dataset



Radix Hash Join - Uniform Dataset



Conclusion

Conclusion

- Partitioned-based joins outperform no-partitioning algorithms in some settings, but it is non-trivial to tune it correctly.
- AFAIK, every DBMS vendor picks one hash join implementation and does not try to be adaptive.
- Next Class
  - Parallel Sort-Merge Join Algorithms