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

<table>
<thead>
<tr>
<th>Join Algorithm</th>
<th>IO Cost</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Nested Loop Join</td>
<td>$M + (m \times N)$</td>
<td>1.3 hours</td>
</tr>
<tr>
<td>Block Nested Loop Join</td>
<td>$M + (M \times N)$</td>
<td>50 seconds</td>
</tr>
<tr>
<td>Index Nested Loop Join</td>
<td>$M + (M \times C)$</td>
<td>Variable</td>
</tr>
<tr>
<td>Sort-Merge Join</td>
<td>$M + N + (\text{sort cost})$</td>
<td>0.75 seconds</td>
</tr>
<tr>
<td>Hash Join</td>
<td>$3 \times (M + N)$</td>
<td>0.45 seconds</td>
</tr>
</tbody>
</table>
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

- Hashing is faster than Sort-Merge.
- Sort-Merge is faster with wider SIMD.

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

- New optimizations and results for Radix Hash Join.

- Trade-offs between partitioning and non-partitioning Hash-Join.

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

- 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

Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
</table>

Partitions

\[ \text{hash}_{P}(\text{key}) \]

Combined

\[ P_1 \]
\[ P_2 \]
\[ \vdots \]
\[ P_n \]
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).

<table>
<thead>
<tr>
<th>Keys</th>
<th>8 9 1 2 3 0 8 4 6 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix</td>
<td>8 1 2 0 4 6</td>
</tr>
</tbody>
</table>
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

Step #1: Inspect input, create histograms

hash_{p}(key)

| 0 7 |
| 1 8 |
| 1 9 |
| 0 7 |
| 0 3 |
| 1 1 |
| 1 5 |
| 1 0 |

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3
Radix Partitions

Step #2: Compute output offsets

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
Radix Partitions

<table>
<thead>
<tr>
<th>hash_{p}(key)</th>
<th>Partition 0: 2</th>
<th>Partition 1: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step #3: Read input and partition

- Partition 0, CPU 0: 0 7
- Partition 0, CPU 1: 0 7
- Partition 1, CPU 0: 0 3
- Partition 1, CPU 1: 1 8
- Partition 1, CPU 1: 1 9
- Partition 1, CPU 1: 1 1
- Partition 1, CPU 1: 1 5
- Partition 1, CPU 1: 1 0
Radix Partitions

Recursively repeat until target number of partitions have been created
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

hash(key)

A
B
C
D
E
F

hash(A) | A

Buckets
Chained Hashing
Chained Hashing

hash(key)

A
B
C
D
E
F

hash(B) | B
hash(A) | A
hash(C) | C
hash(F) | F

hash(D) | D
hash(E) | E

HyPer

64-bit Bucket Pointers
48-bit Pointer
16-bit Bloom Filter
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 $R$. 
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

hash(key)

hash(B) | B [0]

hash(A) | A [0]

hash(C) | C [1]

hash(D) | D [1]

A[0] == E[0]
C[1] == E[1]
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

\[ \text{hash(key)} \]

\begin{align*}
\text{Neighborhood Size} &= 3 \\
\text{Neighborhood \#1} \\
\text{Neighborhood \#2} \\
\text{Neighborhood \#3} \\
\vdots
\end{align*}
Hopscotch Hashing

hash(key)

A
B
C
D
E
F

hash(A) | A

Neighborhood Size = 3

Neighborhood #3
Hopscotch Hashing

`hash(key)`

```
A
B
C
D
E
F
```

```
hash(B) | B
hash(A) | A
```

Neighborhood Size = 3

Neighborhood #1
Hopscotch Hashing

hash(key)
A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

Neighborhood Size = 3

Neighborhood #3
Hopscotch Hashing

\[
\text{hash}(\text{key})
\]

\[
\begin{align*}
\text{hash}(B) & \mid B \\
\text{hash}(A) & \mid A \\
\text{hash}(C) & \mid C \\
\text{hash}(D) & \mid D \\
\end{align*}
\]

\textit{Neighborhood Size = 3}
Hopscotch Hashing

```plaintext
hash(key)
A
B
C
D
E
F
```

```
hash(B) | B
hash(A) | A
hash(C) | C
hash(D) | D
```

**Neighborhood Size = 3**

**Neighborhood #3**
Hopscotch Hashing

**hash(key)**

A
B
C
D
E
F

**hash(B) | B**

**hash(A) | A**

**hash(C) | C**

**hash(D) | D**

*Neighborhood Size = 3*
Hopscotch Hashing

Diagram showing the concept of Hopscotch Hashing:
- Function: `hash(key)`
- Example:
  - `hash(B)` maps to cell `B` in the right section.
  - `hash(A)` maps to cell `A`, within a neighborhood of size 3.
  - `hash(C)` maps to cell `C`.
  - `hash(E)` maps to cell `E`.
  - `hash(D)` maps to cell `D`.

Legend:
- `hash(key)` function mapping.
- Neighborhood size = 3
- Neighborhood #3
Cuckoo Hashing

- Use **multiple tables** 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

Hash Table #1

<table>
<thead>
<tr>
<th>hash₁(Y)</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Insert X

<table>
<thead>
<tr>
<th>hash₁(X)</th>
<th>hash₂(X)</th>
</tr>
</thead>
</table>

Insert Y

<table>
<thead>
<tr>
<th>hash₁(Y)</th>
<th>hash₂(Y)</th>
</tr>
</thead>
</table>

Insert Z

<table>
<thead>
<tr>
<th>hash₁(Z)</th>
<th>hash₂(Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash₁(Y)</td>
<td>hash₂(X)</td>
</tr>
</tbody>
</table>

Hash Table #2

<table>
<thead>
<tr>
<th>hash₂(Z)</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash₂(X)</th>
<th>X</th>
</tr>
</thead>
</table>
Probe Phase
Probe Phase

- For each tuple in $S$, 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 $R$.
  - If inputs were partitioned, then assign each thread a unique partition.
  - Otherwise, synchronize their access to the cursor on $S$. 

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

<table>
<thead>
<tr>
<th></th>
<th>No-P</th>
<th>Shared-P</th>
<th>Private-P</th>
<th>Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partitioning</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Input scans</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Sync during</strong></td>
<td>–</td>
<td>Spinlock per tuple</td>
<td>Barrier</td>
<td>Barriers</td>
</tr>
<tr>
<td><strong>partitioning</strong></td>
<td>Shared</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
<tr>
<td><strong>Hash table</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Sync during</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>build phase</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sync during</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>probe phase</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

**Intel Xeon CPU X5650 @ 2.66GHz**
*6 Cores with 2 Threads Per Core*

<table>
<thead>
<tr>
<th></th>
<th>Partition</th>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Partitioning</td>
<td>60.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Partitioning</td>
<td>67.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private Partitioning</td>
<td>76.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix</td>
<td>47.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **3.3x cache misses**
- **70x TLB misses**
- **24% faster than No Partitioning**
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

![Graph showing evaluation results for Radix Hash Join with varying number of partitions.](image)

**Intel Xeon CPU X5650 @ 2.66GHz**

**Varying the # of Partitions**

- **Partition**
- **Build**
- **Probe**

**Cycles / Output Tuple**

<table>
<thead>
<tr>
<th>Radix / 1-Pass</th>
<th>Radix / 2-Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>8192</td>
</tr>
<tr>
<td>32768</td>
<td>32768</td>
</tr>
<tr>
<td>131072</td>
<td>131072</td>
</tr>
</tbody>
</table>

- **+24%**
- **-5%**

▲ No Partitioning
Radix Hash Join - Uniform Dataset

![Graph showing evaluation of Radix Hash Join with varying number of partitions.](image)
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