TODAY’S AGENDA

Background
Parallel Hash Join
Hash Functions
Hash Table Implementations
Evaluation
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 don’t implement hash join.

But a index nested-loop join with a small number of target tuples is more or less equivalent to a hash join.
HASHING VS. SORTING

1970s – Sorting
1980s – Hashing
1990s – Equivalent
2000s – Hashing
2010s – ???
PARALLEL JOIN ALGORITHMS

→ Hashing is faster than Sort-Merge.
→ Sort-Merge will be faster with wider SIMD.

→ Sort-Merge is already faster, even without SIMD.
→ New optimizations and results for Radix Hash Join.

Source: Cagri Balkesen
JOIN ALGORITHM DESIGN GOALS

Goal #1: Minimize Synchronization
→ Avoid taking latches during execution.

Goal #2: Minimize CPU Cache Misses
→ Ensure that data is always local to worker thread.
IMPROVING CACHE BEHAVIOR

Factors that affect cache misses in a DBMS:
→ Cache + TLB capacity.
→ Locality (temporal and spatial).

Non-Random Access (Scan):
→ Clustering to a cache line.
→ Execute more operations per cache line.

Random Access (Lookups):
→ Partition data to fit in cache + TLB.

Source: Johannes Gehrke
PARALLEL HASH JOINS

Hash join is the most important operator in a DBMS for OLAP workloads.

It’s important that we speed it up by taking advantage of multiple cores.
→ We want to keep all of the cores busy, without becoming memory bound
CLOUDERA IMPALA

% of Total CPU Time Spent in Query Operators
Workload: TPC-H Benchmark

- HASH JOIN: 49.6%
- SEQ SCAN: 25.0%
- UNION: 3.1%
- AGGREGATE: 19.9%
- OTHER: 2.4%
**HASH JOIN** \((R \Join S)\)

**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.
PARTITION PHASE

Split the input relations into partitioned buffers by hashing the tuples’ join key(s).
→ The hash function used for this phase should be different than the one used in the build phase.
→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.

Contents of buffers depends on storage model:
→ **NSM**: Either the entire tuple or a subset of attributes.
→ **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.
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.
→ Have to use a latch to synchronize threads.

**Approach #2: Private Partitions**
→ Each thread has its own set of partitions.
→ Have to consolidate them after all threads finish.
SHARE PARTITIONS

Data Table
# SHARED PARTITIONS

**Data Table**

\[\text{Data Table} \quad \text{hash}_p(\text{key})\]

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

\[\#_p \quad \#_p \quad \#_p\]
SHARED PARTITIONS

Data Table

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

Partitions

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

\[ P_1 \]
\[ P_2 \]
\[ \ldots \]
\[ P_n \]
**SHARED PARTITIONS**

Data Table

- **A**
- **B**
- **C**

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

Partitions

- **P_1**
- **P_2**
- **\ldots**
- **P_n**
PRIVATE PARTITIONS

\( \text{hash}_p(key) \)
PRIVATE PARTITIONS

Data Table

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

Partitions

\[\text{hash}_p(key)\]
PRIVATE PARTITIONS

Data Table

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

Partitions

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

Combined

\[ P_1 \]

\[ P_2 \]

\[ \vdots \]

\[ P_n \]
RADIX PARTITIONING

Scan the input relation multiple times to generate the partitions.

Multi-step pass over the relation:
→ **Step #1**: Scan \( R \) and compute a histogram of the # 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.
The radix is the value of an integer at a particular position (using its base).

**Input**

| 89 | 12 | 23 | 08 | 41 | 64 |
The radix is the value of an integer at a particular position (using its base).
The radix is the value of an integer at a particular position (using its base).
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.

**Input**

| 1 | 2 | 3 | 4 | 5 | 6 |

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

<table>
<thead>
<tr>
<th>#p</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
</table>

Source: Spyros Blanas
**RADIX PARTITIONS**

*Step #1: Inspect input, create histograms*

<table>
<thead>
<tr>
<th>#p</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>18</td>
</tr>
<tr>
<td>#p</td>
<td>19</td>
</tr>
<tr>
<td>#p</td>
<td>07</td>
</tr>
<tr>
<td>#p</td>
<td>03</td>
</tr>
<tr>
<td>#p</td>
<td>11</td>
</tr>
<tr>
<td>#p</td>
<td>15</td>
</tr>
<tr>
<td>#p</td>
<td>10</td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
**RADIX PARTITIONS**

Step #1: Inspect input, create histograms

```
hash_p(key)

<table>
<thead>
<tr>
<th>#p</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>18</td>
</tr>
<tr>
<td>#p</td>
<td>19</td>
</tr>
<tr>
<td>#p</td>
<td>07</td>
</tr>
<tr>
<td>#p</td>
<td>03</td>
</tr>
<tr>
<td>#p</td>
<td>11</td>
</tr>
<tr>
<td>#p</td>
<td>15</td>
</tr>
<tr>
<td>#p</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Source: Spyros Blanas

CMU 15-721 (Spring 2017)
Step #1: Inspect input, create histograms

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3

Source: Spyros Blanas
RADIX PARTITIONS

Step #2: Compute output offsets

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>#p</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 0: 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 1: 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 0: 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 1: 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1

Source: Spyros Blanas
RADIX PARTITIONS

Step #3: Read input and partition

07
03
07
18
19
07
03
11
15
10

Partition 0: 2
Partition 1: 2

07
03

Partition 0, CPU 0
Partition 0, CPU 1

Partition 0, CPU 0
Partition 1, CPU 0
Partition 1, CPU 1

Partition 0, CPU 1
Partition 1, CPU 1

Source: Spyros Blanas
**RADIX PARTITIONS**

**Step #3: Read input and partition**

<table>
<thead>
<tr>
<th>#p</th>
<th>0</th>
<th>7</th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td></td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>7</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Partition 0: 2
- Partition 1: 2

- Partition 0, CPU 0
- Partition 0, CPU 1

<table>
<thead>
<tr>
<th>#p</th>
<th>0</th>
<th>7</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

- Partition 1: 3
- Partition 1, CPU 0
- Partition 1, CPU 1

Source: Spyros Blanas
RADIX PARTITIONS

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 0: 3

hash_p(key)

Source: Spyros Blanas
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

<table>
<thead>
<tr>
<th>p</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 0: 2
Partition 1: 2

Partition 0
Partition 1

Source: Spyros Blanas
### RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>#p</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>18</th>
<th>19</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 0:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 1:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 0:</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition 1:</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
#RADIX PARTITIONS

Recursively repeat until target number of partitions have been created.

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>Partition 0: 2</th>
<th>Partition 1: 2</th>
<th>Partition 0: 1</th>
<th>Partition 1: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>07</td>
<td>18</td>
<td>07</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>7</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>03</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>15</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>11</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>15</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>10</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
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.
→ The hash function must be different than the one that was used in the partition phase.
We don’t want to use a cryptographic hash function for our join algorithm.

We want something that is fast and will have a low collision rate.
HASH FUNCTIONS

**MurmurHash** (2008)
→ Designed to a fast, general purpose hash function.

**Google CityHash** (2011)
→ Based on ideas from MurmurHash2
→ Designed to be faster for short keys (<64 bytes).

**Google FarmHash** (2014)
→ Newer version of CityHash with better collision rates.

**CLHash** (2016)
→ Fast hashing function based on carry-less multiplication.
**HASH FUNCTION BENCHMARKS**

*Intel Xeon CPU E5-2630v4 @ 2.20GHz*

![Graph showing performance of different hash functions vs key size.](image)

- std::hash
- MurmurHash3
- CityHash
- FarmHash
- CLHash

Source: Fredrik Widlund

CMU 15-721 (Spring 2017)
HASH FUNCTION BENCHMARKS

Intel Xeon CPU E5-2630v4 @ 2.20GHz

Source: Fredrik Widlund

Throughput (MB/sec)

Key Size (bytes)

std::hash  MurmurHash3  CityHash  FarmHash  CLHash

0 1 6000 12000 18000
1 32 51 151 201 251

32 64 128 192
HASH TABLE IMPLEMENTATIONS

Approach #1: Chained Hash Table

Approach #2: Open-Addressing Hash Table

Approach #3: Cuckoo Hash Table
CHAINED HASH TABLE

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 HASH TABLE

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

[Diagram showing a chained hash table with nodes and arrows connecting them.]
OPEN-ADDRESSING HASH TABLE

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.
→ Have to store the key in the table to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.
OPEN-ADDRESSING HASH TABLE

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

\[ X \]
\[ Y \]
\[ Z \]
\[ \vdots \]
\[ \text{hash}_B(X) \]
\[ X \]
\[ \vdots \]
**Open-Addressing Hash Table**

$$\text{hash}_B(key)$$

- $X$
- $Y$
- $Z$

- $\text{hash}_B(Y) | Y$
- $\text{hash}_B(X) | X$
**OPEN-ADDRESSING HASH TABLE**

The hash function $\text{hash}_B(key)$ is used to map keys to indices in the hash table. For example, $\text{hash}_B(X)$ maps to index $X$, and $\text{hash}_B(Y)$ maps to index $Y$. The diagram illustrates the mapping of keys $X$, $Y$, and $Z$ to their respective indices in the hash table.
OPEN-ADDRESSING HASH TABLE

$\text{hash}_B(key)$

$X \rightarrow \text{hash}_B(X) \rightarrow X$

$Y \rightarrow \text{hash}_B(Y) \rightarrow Y$

$Z \rightarrow \text{hash}_B(Z) \rightarrow Z$

$\ldots$
OBSERVATION

To reduce the # of wasteful comparisons during the join, it is important to avoid collisions of hashed keys.

This requires a chained hash table with ~2x the number of slots as the # of elements in R.
CUCKOO HASH TABLE

Use multiple hash 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 and deletions are always $O(1)$ because only one location per hash table is checked.
Cuckoo Hash Table

Hash Table #1

Insert X

\[ hash_{B_1}(X) \]

hash_{B_2}(X)

Hash Table #2
**CUCKOO HASH TABLE**

**Hash Table #1**

- Insert $X$
- $\text{hash}_{B_1}(X)$
- $\text{hash}_{B_2}(X)$

**Hash Table #2**
Cuckoo Hash Table

Hash Table #1

Insert X
\[ \text{hash}_{B1}(X) \text{ hash}_{B2}(X) \]

Insert Y
\[ \text{hash}_{B1}(Y) \text{ hash}_{B2}(Y) \]

X

Hash Table #2
**Cuckoo Hash Table**

**Hash Table #1**

- Insert $X$
  - $\text{hash}_{B_1}(X)$
  - $\text{hash}_{B_2}(X)$

- Insert $Y$
  - $\text{hash}_{B_1}(Y)$
  - $\text{hash}_{B_2}(Y)$

**Hash Table #2**

- $\text{Insert } Y$
  - $\text{hash}_{B_1}(Y)$
  - $\text{hash}_{B_2}(Y)$

- $X$

- $Y$
Cuckoo Hash Table

Hash Table #1

Insert X
hash_{B1}(X) \quad hash_{B2}(X)

Insert Y
hash_{B1}(Y) \quad hash_{B2}(Y)

Insert Z
hash_{B1}(Z) \quad hash_{B2}(Z)

Hash Table #2

X

Y

Z
Cuckoo Hash Table

Hash Table #1

Insert X
\[ \text{hash}_{B_1}(X) \quad \text{hash}_{B_2}(X) \]

Insert Y
\[ \text{hash}_{B_1}(Y) \quad \text{hash}_{B_2}(Y) \]

Insert Z
\[ \text{hash}_{B_1}(Z) \quad \text{hash}_{B_2}(Z) \]

Hash Table #2

Z
Cuckoo Hash Table

Hash Table #1

Insert X
hash_{B1}(X) \quad hash_{B2}(X)

Insert Y
hash_{B1}(Y) \quad hash_{B2}(Y)

Insert Z
hash_{B1}(Z) \quad hash_{B2}(Z)

hash_{B1}(Y)

Hash Table #2

Z
**Cuckoo Hash Table**

Hash Table #1

**Insert X**

\[ hash_{B1}(X) \quad hash_{B2}(X) \]

**Insert Y**

\[ hash_{B1}(Y) \quad hash_{B2}(Y) \]

**Insert Z**

\[ hash_{B1}(Z) \quad hash_{B2}(Z) \]

\[ hash_{B1}(Y) \]

\[ hash_{B2}(X) \]

Hash Table #2

**Z**

**X**
We have to make sure that we don’t get stuck in an infinite loop when moving keys.

If we find a cycle, then we can rebuild the entire hash tables with new hash functions.

→ With **two** hash functions, we (probably) won’t need to rebuild the table until it is at about 50% full.

→ With **three** hash functions, we (probably) won’t need to rebuild the table until it is at about 90% full.
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 $S$. 
HASH JOIN VARIANTS

No Partitioning + Shared Hash Table
Non-Blocking Partitioning + Shared Buffers
Non-Blocking Partitioning + Private Buffers
Blocking (Radix) Partitioning
# 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partitioning</td>
<td>_</td>
<td>Spinlock</td>
<td>Barrier,</td>
<td>Barrier,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>per tuple</td>
<td>once at end</td>
<td>4 * #passes</td>
</tr>
<tr>
<td><strong>Hash table</strong></td>
<td>Shared</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
<tr>
<td><strong>Sync during</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>build phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sync during</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>probe phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BENCHMARKS

Primary key – foreign key join
→ Outer Relation (Build): 16M tuples, 16 bytes each
→ Inner Relation (Probe): 256M tuples, 16 bytes each

Uniform and highly skewed (Zipf; s=1.25)

No output materialization
** HASH JOIN – UNIFORM DATA SET **

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Cycles / Output Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Partitioning</td>
<td>60.2</td>
</tr>
<tr>
<td>Shared Partitioning</td>
<td>67.6</td>
</tr>
<tr>
<td>Private Partitioning</td>
<td>76.8</td>
</tr>
<tr>
<td>Radix</td>
<td>47.3</td>
</tr>
</tbody>
</table>

Source: [Spyros Blanas](#)
**HASH JOIN – UNIFORM DATA SET**

*Intel Xeon CPU X5650 @ 2.66GHz*

*6 Cores with 2 Threads Per Core*

<table>
<thead>
<tr>
<th>Method</th>
<th>Partition</th>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Partitioning</td>
<td>60.2</td>
<td>63.2</td>
<td>76.8</td>
</tr>
<tr>
<td>Shared Partitioning</td>
<td>67.6</td>
<td>63.2</td>
<td>73.4</td>
</tr>
<tr>
<td>Private Partitioning</td>
<td>76.8</td>
<td>63.2</td>
<td>73.4</td>
</tr>
<tr>
<td>Radix</td>
<td>47.3</td>
<td>63.2</td>
<td>73.4</td>
</tr>
</tbody>
</table>

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

Source: Spyros Blanas
HASH JOIN – SKEWED DATA SET

Source: Spyros Blanas
We have ignored a lot of important parameters for all of 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 DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions

Source: Spyros Blanas
RADIX HASH JOIN – UNIFORM DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions

<table>
<thead>
<tr>
<th>Radix / 1-Pass</th>
<th>Radix / 2-Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>Probe</td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

No Partitioning  →  Radix  →  Ideal

Source: Spyros Blanas
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

Multi-threading hides cache & TLB miss latency.

Source: Spyros Blanas
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

No Partitioning  Radix  Ideal

Multi-threading hides cache & TLB miss latency.
Radix join has fewer cache & TLB misses but this has marginal benefit.

Source: Spyros Blanas
EFFECTS OF HYPER-THREADING

_Design and Analysis of Database Systems_ (DAS), CMU 15-721 (Spring 2017)

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

Multi-threading hides cache & TLB miss latency.

Radix join has fewer cache & TLB misses but this has marginal benefit.

Non-partitioned join relies on multi-threading for high performance.

Source: Spyros Blanas

![Graph showing the speedup of different join methods with and without hyper-threading.][1]

[1]: https://www.example.com/graph.png
PARTING THOUGHTS

On modern CPUs, a simple hash join algorithm that does not partition inputs is competitive.

There are additional vectorization execution optimizations that are possible in hash joins that we didn’t talk about. But these don’t really help...
NEXT CLASS

Parallel Sort-Merge Joins