TODAY’S AGENDA

Background
Parallel Hash Join
Hash Functions
Hashing Schemes
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 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
1980s – Hashing
1990s – Equivalent
2000s – Hashing
2010s – Hashing (Partitioned vs. Non-Partitioned)
2020s – ???
PARALLEL JOIN ALGORITHMS

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

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

→ New optimizations and results for Radix Hash Join.

→ Trade-offs between partitioning & 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!
JOIN ALGORITHM 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).

Non-Random Access (Scan):
→ Clustering data 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 is important that we speed up our DBMS's join algorithm by taking advantage of multiple cores. → We want to keep all cores busy, without becoming memory bound.
**HASH JOIN (R⨝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).
→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.
→ Sometimes called 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.
→ Sometimes called *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.
### 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>
</tbody>
</table>

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

**Data Table**

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

<table>
<thead>
<tr>
<th>hash$_p$(key)</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>...</th>
<th>$P_n$</th>
</tr>
</thead>
</table>

**Partitions**

The hash function $hash_p(key)$ maps keys to partitions $P_1, P_2, ..., P_n$. Each partition contains a subset of the data from the table.
PRIVATE PARTITIONS

Data Table

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

Partitions

$\text{hash}_p(key)$

$\#_p$

$\#_p$

$\#_p$

$\#_p$
PRIVATE PARTITIONS

Data Table

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

Partitions

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

\[ P_1 \]
\[ P_2 \]
\[ P_n \]
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

\[ P_1 \]
\[ P_2 \]
\[ \vdots \]
\[ P_n \]
### PRIVATE PARTITIONS

#### Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>hash_p(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_n</td>
</tr>
</tbody>
</table>

#### Partitions

- P_1
- P_2
- ... (n)

#### Combined

- P_1
- P_2
- ... (n)
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 of a key is the value of an integer at a position (using its base).

**Keys**

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

hash\_p(key)
**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>
<tbody>
<tr>
<td>hash(key)</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
### RADIX PARTITIONS

**Step #1: Inspect input, create histograms**

<table>
<thead>
<tr>
<th>hash_p(key)</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:</td>
<td>2</td>
<td></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>
</tr>
</tbody>
</table>

- **Partition 0:** 2
- **Partition 1:** 2

- **Partition 0:** 1
- **Partition 1:** 3
**RADIX PARTITIONS**

Step #2: Compute output offsets

- **Partition 0**: 2
- **Partition 1**: 2

- **Partition 0, CPU 0**
- **Partition 0, CPU 1**
- **Partition 1, CPU 0**
- **Partition 1, CPU 1**
Step #2: Compute output offsets

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3

Partition 0, CPU 0
Partition 1, CPU 0
Partition 0, CPU 1
Partition 1, CPU 1
**RADIX PARTITIONS**

Step #3: Read input and partition

- **Partition 0, CPU 0**: 07
- **Partition 0, CPU 1**: 03
- **Partition 1, CPU 0**: 07
- **Partition 1, CPU 1**: 03

<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>hash_p(key)</td>
<td>Partition 0: 2</td>
<td>Partition 1: 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Partition 0: 1</td>
<td>Partition 1: 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RADIX PARTITIONS

Step #3: Read input and partition

**Partition 0**: 2
- CPU 0
- CPU 1

**Partition 1**: 2
- CPU 0
- CPU 1

**Partition 0**: 1
- CPU 0

**Partition 1**: 3
- CPU 1

**Partition 0**: 0
- CPU 1

**Partition 1**: 0
- CPU 0
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

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

Partition 0: 2
Partition 0: 1
Partition 1: 3
Partition 1: 2
Partition 1: 3
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3
**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>hash(key)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Partition 0**: 2
- **Partition 1**: 2

- **Partition 0**: 1
- **Partition 1**: 3
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.
\[ \rightarrow \text{The buckets should only be a few cache lines in size.} \]
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.
HASH FUNCTIONS

We do not want to use a cryptographic hash function for our join algorithm.

We want something that is fast and will have a low collision rate.
→ **Best Speed:** Always return '1'
→ **Best Collision Rate:** Perfect hashing

See [SMHasher](https://smhasher.net) for a comprehensive hash function benchmark suite.
HASH FUNCTIONS

CRC-64 (1975)
→ Used in networking for error detection.

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

Google CityHash (2011)
→ Designed to be faster for short keys (<64 bytes).

Facebook XXHash (2012)
→ From the creator of zstd compression.

Google FarmHash (2014)
→ Newer version of CityHash with better collision rates.
 HASH FUNCTION BENCHMARK

*Intel Core i7-8700K @ 3.70GHz*

<table>
<thead>
<tr>
<th>Key Size (bytes)</th>
<th>Throughput (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>crc64</td>
</tr>
<tr>
<td>64</td>
<td>std::hash</td>
</tr>
<tr>
<td>128</td>
<td>MurmurHash3</td>
</tr>
<tr>
<td>192</td>
<td>CityHash</td>
</tr>
<tr>
<td>192</td>
<td>FarmHash</td>
</tr>
<tr>
<td>256</td>
<td>XXHash3</td>
</tr>
</tbody>
</table>

Source: Fredrik Widlund

15-721 (Spring 2020)
HASHING SCHEMES

Approach #1: Chained Hashing
Approach #2: Linear Probe Hashing
Approach #3: Robin Hood Hashing
Approach #4: Hopscotch Hashing
Approach #5: Cuckoo Hashing
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

$hash(key)$
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

Buckets
CHAINED HASHING

hash(key)

<table>
<thead>
<tr>
<th>A</th>
<th>hash(A)</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>hash(B)</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>hash(C)</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

hash(D) | D
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C

hash(D) | D

hash(E) | E
CHAINED HASHING

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

\begin{align*}
    & A \\
    & B \\
    & C \\
    & D \\
    & E \\
    & F
\end{align*}

\begin{align*}
    & \text{hash}(B) \mid B \\
    & \text{hash}(A) \mid A \\
    & \text{hash}(C) \mid C \\
    & \text{hash}(F) \mid F \\
    & \text{hash}(D) \mid D \\
    & \text{hash}(E) \mid E
\end{align*}
CHAINED HASHING

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

**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

hash(key)

A
B
C
D
E
F
LINEAR PROBE HASHING

hash(key)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(B)</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15-721 (Spring 2020)
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B
hash(A) | A
hash(C) | C
hash(D) | D
LINEAR PROBE HASHING

$\text{hash(key)}$

$\begin{align*}
\text{hash(A)} & | A \\
\text{hash(B)} & | B \\
\text{hash(C)} & | C \\
\text{hash(D)} & | D \\
\end{align*}$
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B
hash(A) | A
hash(C) | C
hash(D) | D
hash(E) | E
LINEAR PROBE HASHING

hash(key)

\[
\begin{array}{|c|}
\hline
A \\
\hline
B \\
\hline
C \\
\hline
D \\
\hline
E \\
\hline
F \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{hash}(B) | B \\
\hline
\text{hash}(A) | A \\
\hline
\text{hash}(C) | C \\
\hline
\text{hash}(D) | D \\
\hline
\text{hash}(E) | E \\
\hline
\text{hash}(F) | F \\
\hline
\end{array}
\]
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 \( \sim 2 \times \) the number of slots as the # 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(A) | A [∅]

# of "Jumps" From First Position
ROBIN HOOD HASHING

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

A
B
C
D
E
F

\[ \text{hash(A)} | A [\emptyset] \]

\[ \text{hash(B)} | B [\emptyset] \]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [\emptyset]

hash(A) | A [\emptyset]

A[0] == C[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [Ø]
hash(A) | A [Ø]
hash(C) | C [1]

A[0] == C[0]
ROBIN HOOD HASHING

\[
\text{hash(key)} \quad \begin{array}{c}
A \\
B \\
C \\
D \\
E \\
F \\
\end{array}
\]

\[
\begin{array}{c}
\text{hash(B)} | B [\emptyset] \\
\text{hash(A)} | A [\emptyset] \\
\text{hash(C)} | C [1] \\
\end{array}
\]

\[ C[1] > D[0] \]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [0]
hash(A) | A [0]
hash(C) | C [1]
hash(D) | D [1]

C[1] > D[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [\emptyset]

hash(A) | A [\emptyset]

hash(C) | C [1]

hash(D) | D [1]

A[0] == E[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

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

Hashing (key)

A
B
C
D
E
F

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

hash(key)

A  B  C  D  E  F

hash(A) | A [0]
hash(B) | B [0]
hash(C) | C [1]
hash(D) | D [2]
hash(E) | E [2]

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

hash(key)

hash(B) | B [0]
hash(A) | A [0]
hash(C) | C [1]
hash(E) | E [2]
hash(D) | D [2]
hash(F) | F [1]

D[2] > F[0]
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

hash(key)

A
B
C
D
E
F

Neighborhood Size = 3
**HOPSCOTCH HASHING**

**hash(key)**

| A | B | C | D | E | F |

**Neighborhood Size = 3**

**Neighborhood #1**
HOPSCOTCH HASHING

$\text{Neighborhood Size} = 3$

$\text{Neighborhood #1}$

$\text{Neighborhood #2}$

$\text{Neighborhood #3}$

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

A
B
C
D
E
F
HOPSCOTCH HASHING

hash(key)

Neighborhood Size = 3

Neighborhood #3
HOPSCOTCH HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

Neighborhood Size = 3

Neighborhood #3
Hopscotch Hashing

Neighborhood Size = 3

Neighborhood #1

hash(key)

| A | B | C | D | E | F |

hash(A) | A
**HOPSCOTCH HASHING**

**Neighborhood Size = 3**

- Neighborhood #1
  - hash(key)
    - hash(A) | A
    - hash(B) | B

- Elements: A, B, C, D, E, F
Hopscotch Hashing

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

Neighborhood #3
Hopscotch Hashing

Neighborhood Size = 3

Neighborhood #3

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

Neighborhood #3
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C
HO\ P S C O T C H  H A S H I N G

\textit{hash} (key)

\begin{itemize}
  \item \textit{hash(A)} | A
  \item \textit{hash(B)} | B
  \item \textit{hash(C)} | C
\end{itemize}

\textit{Neighborhood Size} = 3
Hopscotch Hashing

Neighborhood Size = 3

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

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

*Neighborhood Size = 3*

- **hash(key)**
  - A
  - B
  - C
  - D
  - E
  - F

- **hash(B)** | B
- **hash(A)** | A
- **hash(C)** | C
- **hash(D)** | D
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

hash(D) | D

Neighborhood #3
**Hopscotch Hashing**

- **hash(key)**
- **Neighborhood Size = 3**
- **Neighborhood #3**

- hash(A) | A
- hash(C) | C
- hash(D) | D
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

hash(D) | D

Neighborhood #3
HOPSCOTCH HASHING

**Neighborhood Size = 3**

- **hash(key)**
  - A
  - B
  - C
  - D
  - E
  - F

- **hash(B)** | B
- **hash(A)** | A
- **hash(C)** | C
- **hash(D)** | D
**Hopscotch Hashing**

Neighborhood Size = 3

- Hash function: `hash(key)`
- Neighborhood #3

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>hash(A)</code></td>
<td><code>hash(B)</code></td>
<td><code>hash(C)</code></td>
<td><code>hash(D)</code></td>
<td><code>hash(E)</code></td>
</tr>
</tbody>
</table>

```plaintext
hash(A) | A
hash(C) | C
hash(E) | E
hash(D) | D
```
Hopscotch Hashing

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

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

Neighborhood #6
**Hopscotch Hashing**

**Neighborhood Size = 3**

<table>
<thead>
<tr>
<th>Neighborhood #6</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(A)</td>
</tr>
<tr>
<td>hash(C)</td>
</tr>
<tr>
<td>hash(E)</td>
</tr>
<tr>
<td>hash(D)</td>
</tr>
<tr>
<td>hash(F)</td>
</tr>
</tbody>
</table>

**hash(key)**

- A
- B
- C
- D
- E
- F
C U C K O O H A S H I N G

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

Hash Table #2
Cuckoo Hashing

Hash Table #1

Insert X

hash_1(X)

hash_2(X)

Hash Table #2
Cuckoo Hashing

Hash Table #1

\[ \text{hash}_1(X) | X \]

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Hash Table #2

\[ \ldots \]

\[ \text{...} \]
CUCKOO HASHING

**Hash Table #1**

- Insert $X$
  - $\text{hash}_1(X)$
  - $\text{hash}_2(X)$

- $\text{hash}_1(X) | X$

**Hash Table #2**

- Insert $Y$
  - $\text{hash}_1(Y)$
  - $\text{hash}_2(Y)$
Cuckoo Hashing

Hash Table #1

Insert X
\[ hash_1(X) \quad hash_2(X) \]

Insert Y
\[ hash_1(Y) \quad hash_2(Y) \]

Hash Table #2

Hash Table #1

Hash Table #2
Cuckoo Hashing

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]
Cuckoo Hashing

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]

Hash Table #2

\[ \text{hash}_2(Y) \mid Y \]
**Cuckoo Hashing**

Hash Table #1

- Insert X
  - $\text{hash}_1(X)$
  - $\text{hash}_2(X)$

- Insert Y
  - $\text{hash}_1(Y)$
  - $\text{hash}_2(Y)$

- Insert Z
  - $\text{hash}_1(Z)$
  - $\text{hash}_2(Z)$
  - $\text{hash}_1(Y)$

Hash Table #2

- $\text{hash}_2(Z)$
  - Z
CUCKOO HASHING

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]
\[ \text{hash}_1(Y) \]

Hash Table #2

\[ \text{hash}_2(Z) \quad \text{Z} \]
Cuckoo Hashing

Hash Table #1

Insert X
\[\text{hash}_1(X) \quad \text{hash}_2(X)\]

Insert Y
\[\text{hash}_1(Y) \quad \text{hash}_2(Y)\]

Insert Z
\[\text{hash}_1(Z) \quad \text{hash}_2(Z)\]

\[\text{hash}_1(Y)\]

Hash Table #2

\[\text{hash}_2(Z) \mid Z\]

\[\vdots\]

\[\text{hash}_1(Y)\]
Cuckoo Hashing

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]
\[ \text{hash}_1(Y) \quad \text{hash}_2(X) \]

Hash Table #2

\[ \text{hash}_2(Z) \mid Z \]
\[ \text{hash}_2(X) \mid X \]
CUCKOO HASHING

Threads have to make sure that they 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 $S$. 
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.

→ Sometimes called *sideways information passing*. 
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.

→ Sometimes called *sideways information passing*. 
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.
→ Sometimes called *sideways information passing*. 
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.
→ Sometimes called *sideways information passing*.
### 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>Partitioning</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Input scans</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Sync during partitioning</td>
<td>_</td>
<td>Spinlock per tuple</td>
<td>Barrier, once at end</td>
<td>Barrier, 4 · #passes</td>
</tr>
<tr>
<td>Hash table</td>
<td>Shared</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
<tr>
<td>Sync during build phase</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sync during probe phase</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</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

<table>
<thead>
<tr>
<th></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>

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

Source: Spyros Blanas
**HASH JOIN – SKEWED DATA SET**

*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><strong>No Partitioning</strong></td>
<td>25.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shared Partitioning</strong></td>
<td>167.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private Partitioning</strong></td>
<td>56.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Radix</strong></td>
<td>50.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
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 DATA SET

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

<table>
<thead>
<tr>
<th>Cycles / Output Tuple</th>
<th>Partition</th>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Spyros Blanas

No Partitioning
RADIX HASH JOIN – UNIFORM DATA SET

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

Cycles / Output Tuple

Partition | Build | Probe

Intel Xeon CPU X5650 @ 2.66GHz

Varying the # of Partitions

Source: Spyros Blanas

15-721 (Spring 2020)
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

- No Partitioning
- Radix
- Ideal

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
TPC-H Q19

4× Intel Xeon CPU E7-4870v4
Scale Factor 100

Source: Stefan Schuh
PARTING THOUGHTS

Partitioned-based joins outperform no-partitioning algorithms is most 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 Joins