Parallel Hash Join Algorithms
Project #2:
→ Feedback Submission: **Saturday April 1st**
→ Final Submission: **Monday May 1st**
→ Sign up for a system if you haven't yet!

Project #3
→ Proposal Presentation: **Wednesday March 1st**
→ Status Update Presentation: **Monday April 3rd**
→ Final Presentations: TBA
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.
→ We will discuss multi-way joins in Lecture #13.

Two main approaches:
→ Hash Join
→ Sort-Merge Join

We won't discuss nested-loop joins because an OLAP DBMS almost never wants to use this...
Many OLTP DBMSs do not implement hash join.

But an **index nested-loop join** is conceptually equivalent to a hash join.

→ Index NL joins typically means using an existing B+Tree.
→ Hash join will build a hash table (index) on the fly and then discard immediately after the operation is complete.
HASHING VS. SORTING JOINS

1970s – Sorting
1980s – Hashing
1990s – Equivalent
2000s – Hashing
2010s – Hashing (Partitioned vs. Non-Partitioned)
2020s – Non-Partitioned Hashing
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!

→ Benefits of Radix Hash Join aren't worth engineering costs.
JOIN ALGORITHM DESIGN GOALS

These goals matter whether the DBMS is using a hardware-conscious vs. hardware-oblivious algorithm for joins.

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.
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 one of the most important operators in a DBMS for OLAP workloads.
→ But it is still not the dominant cost.

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 disjoint subsets using a hash function 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.
PARTITIONING PHASE

Approach #1: Implicit Partitioning
→ The data was partitioned on the join key when it was loaded into the database.
→ No extra pass over the data is needed.

Approach #2: Explicit Partitioning
→ Divide only the outer relation and redistribute among the different CPU cores.
→ Can use the same radix partitioning approach we talked about last time.
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 *Grace 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 and let other threads build hash table at the same time.

**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.
SHARED PARTITIONS

Data Table

\( \text{hash}_p(key) \)
Global Partitions

Data Table

$\text{hash}_p(key)$

Global Partitions
**Data Table**

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

**Global Partitions**

$\text{hash}_p(key)$

$P_1 \rightarrow \text{Data Partition}$

$P_2 \rightarrow \text{Data Partition}$

$P_n \rightarrow \text{Data Partition}$
**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>
</tbody>
</table>

Global Partitions

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

\[
\#_p
\]

\[
P_1
\]

\[
P_2
\]

\[
P_n
\]

\[
\vdots
\]
PRIVATE PARTITIONS

Data Table

Local Partitions

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

Data Table: A, B, C

Local Partitions: $\text{hash}_p(key)$

Global Partitions: $P_1, P_2, P_n$
### PRIVATE PARTITIONS

**Data Table**

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

**Local Partitions**

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

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>![P1]</td>
<td>![P2]</td>
<td>![Pn]</td>
</tr>
</tbody>
</table>

**Global Partitions**

<table>
<thead>
<tr>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>![P1]</td>
</tr>
</tbody>
</table>
### PRIVATE PARTITIONS

#### Data Table

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

#### Local Partitions

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

- \( P_1 \)
- \( P_2 \)
- \( P_n \)

#### Global Partitions

- \( P_1 \)
- \( P_2 \)
- \( P_n \)
PRIVATE PARTITIONS

Data Table

Local Partitions

hash_p(key)

Global Partitions

A  B  C

P1  P2  Pn

P1  P2  Pn

P1  P2  Pn

P1  P2  Pn
RADIX PARTITIONING

Scan the input relation multiple times to generate the partitions.

Two-pass algorithm:

→ **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 per-thread 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).
→ Efficient to compute with bitshifting + multiplication.
The radix of a key is the value of an integer at a position (using its base).
→ Efficient to compute with bitshifting + multiplication.
RADIX

The radix of a key is the value of an integer at a position (using its base).
→ Efficient to compute with bitshifting + multiplication.
The radix of a key is the value of an integer at a position (using its base).
→ Efficient to compute with bitshifting + multiplication.

Compute radix for each key and populate histogram of counts per radix.
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.
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 $(x_0, x_1, ..., x_n)$ 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.

**Input**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

**Prefix Sum**

1  
2  
3  
6  
10  
15  
21  
28  
36  
45
RAIDX 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>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
<td>_p #</td>
</tr>
</tbody>
</table>

![Diagram showing the process of radix partitioning]

- Inspect the input values and create a histogram for each digit.
- Use a hash function to map the input values to specific buckets or partitions.
- Process the input data by partitioning it based on the digit and corresponding buckets.

This approach simplifies the sorting process, especially for large datasets, by breaking down the sorting task into smaller, more manageable parts.
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>
</table>
| Partition 0: 2
| Partition 1: 2 |
| Partition 0: 1
| Partition 1: 3 |
RADIX PARTITIONS

Step #2: Compute output offsets

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

<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>

Partition 0: 2
Partition 1: 2
Partition 0: 1
Partition 1: 3

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
## RADIUS PARTITIONS

### Step #2: Compute output offsets

| hash_p(key) |  
|----------|---
| 07       | 2 Partition 0: 2
| 18       | 2 Partition 1: 2
| 19       | 3 Partition 0: 1
| 07       | 1 Partition 1: 3
| 03       |  
| 11       |  
| 15       |  
| 10       |  

- **Partition 0, CPU 0**: 2
- **Partition 0, CPU 1**: 2
- **Partition 1, CPU 0**: 1
- **Partition 1, CPU 1**: 3
RADIX PARTITIONS

Step #3: Read input and partition

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

hash_p(key)

<table>
<thead>
<tr>
<th>#p</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>07</td>
</tr>
<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>

Partition 0: 2
Partition 1: 2
Partition 0: 1
Partition 1: 3
**Step #3: Read input and partition**

- **Partition 0, CPU 0**: 07
- **Partition 0, CPU 1**: 03
- **Partition 1, CPU 0**: 11
- **Partition 1, CPU 1**: 15, 10
**RADIX PARTITIONS**

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

<table>
<thead>
<tr>
<th>Hash(key)</th>
<th>Partition 0</th>
<th>Partition 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p 7</td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>#p 18</td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>#p 19</td>
<td>03</td>
<td>18</td>
</tr>
<tr>
<td>#p 07</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>#p 03</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>#p 11</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>#p 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#p 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Partition 0, CPU 0:** 07, 07
- **Partition 0, CPU 1:** 03, 18
- **Partition 1, CPU 0:** 19, 11
- **Partition 1, CPU 1:** 15, 10

### Notes
- **Radix partitions** refer to dividing data into partitions based on their keys.
- **Hash function** is used to map keys to partitions.
- The diagram illustrates the partitioning process with specific keys and their corresponding partitions.
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>hash_p(key)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>#p</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>#p</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Partition 0: 2**
- #p = 0
- #p = 7

**Partition 1: 2**
- #p = 1
- #p = 8
- #p = 19

**Partition 0: 1**
- #p = 1

**Partition 1: 3**
- #p = 1
- #p = 5
- #p = 15
**OPTIMIZATIONS**

**Software Write Combine Buffers:**
→ Each worker maintains local output buffer to stage writes.
→ When buffer full, write changes to global partition.
→ Similar to private partitions but without a separate write phase at the end.

**Non-temporal Streaming Writes**
→ Workers write data to global partition memory using streaming instructions to bypass CPU caches.
BUILD PHASE

The threads are then to scan either the tuples (or partitions) of $\mathbf{R}$.

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.
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](#) for a comprehensive hash function benchmark suite.
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://github.com/rurban/smhasher) 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

Throughput (MB/sec)

Key Size (bytes)

Source: Fredrik Widlund
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)

Bucket Pointers

A
B
C
D
E
F

Buckets
CHAINED HASHING

A B C D

hash(key)

A

B C D E F

hash(A) | A

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

Buckets

hash(A) | A
hash(B) | B
hash(C) | C
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C

Buckets
**CHAINED HASHING**

The diagram illustrates the chaining method in hashing. Each key (A, B, C, D, E, F) is hashed to a bucket. If a bucket is already occupied, the next bucket is used (chained). For example:

- hash(A) is A
- hash(B) is B
- hash(C) is C
- hash(D) is D
- hash(E), hash(F) are not shown.

The diagram shows how chaining resolves collisions by linking elements together in a chain of buckets.
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

hash(key)

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

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

\[
\begin{array}{c}
\text{hash}(D) | D \\
\text{hash}(E) | E \\
\end{array}
\]
CHAINED HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A
hash(C) | C

hash(D) | D
hash(E) | E
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)
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B
LINEAR PROBE HASHING

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

\[
\begin{array}{c|c}
\text{hash}(A) & A \\
\text{hash}(B) & B \\
\end{array}
\]
LINEAR PROBE HASHING

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

- **hash(A) | A**
- **hash(B) | B**
- **hash(C) | C**
LINEAR PROBE HASHING

A
B
C
D
E
F

hash(key)

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

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

\[
\begin{array}{c|c}
 \text{A} & \text{hash(A)} \\
 \text{B} & \text{hash(B)} \\
 \text{C} & \text{hash(C)} \\
 \text{D} & \text{hash(D)} \\
 \text{E} & \\
 \text{F} & \\
\end{array}
\]
LINEAR PROBE HASHING

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

\[
\begin{array}{c|c}
A & \text{hash(A)} | A \\
B & \text{hash(B)} | B \\
C & \text{hash(C)} | C \\
D & \text{hash(D)} | D \\
E & \text{hash(E)} | E \\
\end{array}
\]
LINEAR PROBE HASHING

hash(key)

A

B

C

D

E

F

hash(A) | A

hash(B) | B

hash(C) | C

hash(D) | D

hash(E) | E

hash(F) | F
To reduce the number of wasteful comparisons during the build/probe phases, it is important to avoid collisions of hashed keys.

This requires a hash table with \( \sim 2 \times \) 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

$\text{hash(key)}$

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

# of "Jumps" From First Position
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A [\emptyset]

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

hash(key)

<table>
<thead>
<tr>
<th>A</th>
<th>B [0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A [0]</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

hash(B) == \( B[0] \)

hash(A) == \( A[0] \)

hash(C) == \( C[1] \)

\( A[0] == C[0] \)
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]
C[1] == E[1]
ROBIN HOOD HASHING

hash(key)

\[
\begin{align*}
\text{hash(A)} & | A[0] \\
\text{hash(B)} & | B[0] \\
\text{hash(C)} & | C[1] \\
\text{hash(E)} & | E[2] \\
\end{align*}
\]

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

hash(key)

A
B
C
D
E
F

hash(B) | B [\emptyset]

hash(A) | A [\emptyset]

hash(C) | C [1]

hash(E) | E [2]

hash(D) | D [2]

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

hash(key)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

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 a contiguous range of slots in the table.
→ The size of a neighborhood is a configurable constant (ideally a single cache-line).
→ A key is guaranteed to be in its neighborhood or not exist in the table.

The goal is to have the cost of accessing a neighborhood to be the same as finding a key.
HOPSCOTCH HASHING

Neighborhood Size = 3

Neighborhood #1
HOPSCOTCH HASHING

hash(key)

A  B  C  D  E  F

Neighborhood Size = 3

Neighborhood #1
Neighborhood #2
Neighborhood #3
Neighborhood #4

⋮
HOPSCOTCH HASHING

Neighborhood Size = 3

- Neighborhood #1
- Neighborhood #2
- Neighborhood #3
- Neighborhood #4
- \vdots
- Neighborhood #6

hash(key)

A
B
C
D
E
F
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

hash(A) | A

Neighborhood #3
HOPSCOTCH HASHING

Neighborhood Size = 3

Neighborhood #1

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>
</table>

hash(A) | A

hash(B) | B
HOPSCOTCH HASHING

Neighborhood Size = 3

Neighborhood #3

hash(key)

A
B
C
D
E
F

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

hash(key)
HOPSCOTCH HASHING

hash(key)

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

Neighborhood Size = 3

Neighborhood #3

\[
\begin{array}{c|c}
| hash(B) & B \\
| hash(A) & A \\
\end{array}
\]
HOPSCOTCH HASHING

**Neighborhood Size = 3**

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

```
hash(key)
```

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

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

```
Neighborhood #3
```
HOPSCOTCH HASHING

neighborhood #3

= 3

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C
HOPSCOTCH HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

Neighborhood Size = 3

Neighborhood #4
HOPSCOTCH HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B

hash(A) | A

hash(C) | C

Neighborhood Size = 3

Neighborhood #4
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 #4
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

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

Neighborhood Size = 3

$hash(key)$

$hash(A) | A$

$hash(B) | B$

$hash(C) | C$

$hash(D) | D$
HOPSCOTCH HASHING

hash(key)

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

Neighborhood Size = 3

Neighborhood #3
**Hopscotch Hashing**

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

**Neighborhood Size = 3**

**Neighborhood #3**

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

Neighborhood Size = 3

Neighborhood #4

hash(key)

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

\[\begin{array}{c}
hash(A) | A \\
hash(B) | B \\
hash(C) | C \\
hash(D) | D \\
\end{array}\]
**HOPSCOTCH HASHING**

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

- Neighborhood Size = 3

**Neighborhood #4**

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

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

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

Neighborhood Size = 3

Neighborhood #3

hash(key)

A
B
C
D
E
F

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

hash(key)

A
B
C
D
E
F

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

Neighborhood Size = 3
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

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

Neighborhood #6
**Hopscotch Hashing**

- **Neighborhood Size = 3**

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

- **Neighborhood #6**
  - 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

Neighborhood #6
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key)

A
B
C
D
E
F

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

Neighborhood #6
HOPSCOTCH HASHING

Neighborhood Size = 3

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>hash(D)</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>hash(E)</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>hash(F)</td>
<td>F</td>
</tr>
</tbody>
</table>

Neighborhood #6
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

Insert X

$\text{hash}_1(X)$  $\text{hash}_2(X)$
Cuckoo Hashing

**Hash Table #1**

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

**Hash Table #2**

**Insert X**

hash₁(X)  hash₂(X)
Cuckoo Hashing

Hash Table #1

- Insert X
  - $hash_1(X)$
  - $hash_2(X)$

Hash Table #2

- Insert Y
  - $hash_1(Y)$
  - $hash_2(Y)$
CUCKOO HASHING

Hash Table #1

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

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

Hash Table #2

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

Hash Table #2

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

\[ \text{hash}_1(X) \mid X \]
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**
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]

Hash Table #1

Hash Table #2

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

Hash Table #1

- **Insert X**
  - hash₁(X) → hash₂(X)

- **Insert Y**
  - hash₁(Y) → hash₂(Y)

- **Insert Z**
  - hash₁(Z) → hash₂(Z)
  - hash₁(Y)

Hash Table #2

- **Insert Z**
  - hash₂(Z) → Z

- **Hash Table #1**
  - hash₁(X) → X

- **Hash Table #2**
  - hash₂(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

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

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

Hash Table #2

\[ \text{hash}_2(Z) \quad Z \]
\[ \text{hash}_2(X) \quad X \]
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*. 

**PROBE PHASE – BLOOM FILTER**
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*.
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*. 
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

Implemented multiple variants of hash join algorithms based on previous literature and compare unoptimized vs. optimized versions.

Core approaches:
→ No Partitioning Hash Join
→ Concise Hash Table Join
→ 2-pass Radix Hash Join (Chained vs. Linear)

Special Case: Arrays for monotonic primary keys.
JOIN COMPARISON (R⨝S)

4× Intel Xeon CPU E7-4870v2 (Only 32 cores)

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>128M</td>
<td>1280M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Throughput (M tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort-Merge</td>
<td>518</td>
</tr>
<tr>
<td>Concise Hash</td>
<td>556</td>
</tr>
<tr>
<td>Radix-Part (Chained)</td>
<td>456</td>
</tr>
<tr>
<td>No-Part (Linear)</td>
<td>806</td>
</tr>
<tr>
<td>No-Part (Array)</td>
<td>1145</td>
</tr>
<tr>
<td>Radix-Part (Chained)</td>
<td>1449</td>
</tr>
<tr>
<td>Radix-Part (Linear)</td>
<td>1454</td>
</tr>
<tr>
<td>Radix-Part (Array)</td>
<td>1449</td>
</tr>
</tbody>
</table>

Source: Stefan Schuh

Optimized

Better Performance
TPC-H Q19

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

Source: Stefan Schuh
PARTING THOUGHTS

Partitioned-based joins outperform no-partitioning algorithms in 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