Parallel Hash Join Algorithms
LAST CLASS

How to break up input data into smaller units (morsels) and allow workers to pull them from a global queue.

Distributed query scheduling is roughly the same as multi-core single node query scheduling.
There are two design decisions on how to handle queries that take longer to complete than expected.

**Approach #1: Dynamic Scaling**
→ Provision additional workers before a query starts.
→ Example: Snowflake Flexible Compute

**Approach #2: Work Stealing**
→ A worker takes tasks from a peer.
→ Example: Snowflake Scheduler
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 next class.

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...
HASHING VS. SORTING JOINS

1970s – Sorting
1980s – Hashing
1990s – Equivalent
2000s – Hashing
2010s – Hashing (Partitioned vs. Non-Partitioned)
2020s – Non-Partitioned Hashing
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 & 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 the development costs. Hard to know when to use it.
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.
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 due to cache misses.
**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.
PARTITION PHASE

Split the input relations into partitioned buffers by hashing the tuples’ join key(s).

→ Divide the inner/outer relations and redistribute among the CPU cores.
→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.

Explicitly partitioning the input relations before a join operator is sometimes called **Grace Hash Join**.
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.
NON-BLOCKING: SHARED 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>

*hash_p(key)*

**Global Partitions**

\[
\begin{align*}
P_1 & : \\
P_2 & : \\
\vdots & : \\
P_n & : \\
\end{align*}
\]
NON-BLOCKING: 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>

Global Partitions

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

\[ P_1 \]

\[ P_2 \]

\[ \vdots \]

\[ P_n \]
NON-BLOCKING: SHARED PARTITIONS

Data Table

Global Partitions

$\text{hash}_p(key)$
NON-BLOCKING: PRIVATE 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>

Local Partitions

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

\[\#_p\]

[Diagram showing partitions and hash function]
## Non-Blocking: 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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#(_p)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#(_p)</td>
</tr>
</tbody>
</table>

### Local Partitions

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

### Global Partitions

- \(P_1\)
- \(P_2\)
- \(P_n\)
**NON-BLOCKING: 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#p</td>
</tr>
</tbody>
</table>

### Local Partitions

<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th></th>
<th>P_2</th>
<th></th>
<th>P_n</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
<td></td>
<td>#</td>
</tr>
</tbody>
</table>

### Global Partitions

---
**NON-BLOCKING: PRIVATE 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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Local Partitions**

```
hash_p(key)
```

```
P_1
P_2
P_n
```

**Global Partitions**

```
P_1
```

CMU-DB 15-721 (Spring 2024)
NON-BLOCKING: PRIVATE PARTITIONS

Data Table

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

Local Partitions

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

Global Partitions

\[ P_1 \]

\[ P_2 \]
NON-BLOCKING: PRIVATE PARTITIONS

Data Table

Local Partitions

Global Partitions

A | B | C
---|---|---
0 | 0 | 0

hash_p(key)

P_1  
P_2  
\vdots
P_n

P_1  
P_2  
\vdots
P_n
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**.
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.
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.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Radix</th>
<th>Histogram (Key → Count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 12 23 08 11 04</td>
<td>1 1 2 0 1 0</td>
<td>0→2 1→3 2→1</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix Sum</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>
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.
Step #1: Inspect input, create histograms

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

<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>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step #1: Inspect input, create histograms

Partition 0: 2
Partition 1: 2

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

Step #2: Compute output offsets

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

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

Step #2: Compute output offsets

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

Partition 1:
- CPU 0
- Partition 1: 2
- Partition 1: 3

Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
RADX PARTITIONS

Step #3: Read input and partition

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

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
Partition 0:
- 2

Partition 1:
- 2
- 3

Step #3: Read input and partition

- Partition 0, CPU 0
- Partition 0, CPU 1
- Partition 1, CPU 0
- Partition 1, CPU 1
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

<table>
<thead>
<tr>
<th>Partition 0: 2</th>
<th>Partition 1: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>07</td>
<td>18</td>
</tr>
<tr>
<td>07</td>
<td>19</td>
</tr>
<tr>
<td>07</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition 0: 1</th>
<th>Partition 1: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

#hash_p(key)
Recursively repeat until target number of partitions have been created.
# 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>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></td>
<td>#p</td>
<td>07</td>
<td>19</td>
<td>15</td>
<td>10</td>
<td>03</td>
<td>11</td>
<td>15</td>
<td>10</td>
<td>03</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3
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 \( R \).

For each tuple, hash the join key attribute(s) 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.
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/smiths/S])[MHasher](https://github.com/smiths/S) for a comprehensive hash function benchmark suite.
HASH FUNCTIONS

CRC-32/64 (1975)
→ Modern CPUs have explicit CRC instructions. Some DBMSs use this for hashing integers.

MurmurHash (2008)
→ Designed as 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 FUNCTIONS

Modern CPUs have explicit CRC instructions. Some DBMSs use this for hashing integers.

- **CityHash** (2011): Designed to be faster for short keys (<64 bytes).
- **XXHash** (2012): From the creator of zstd compression.
- **FarmHash** (2014): Newer version of CityHash with better collision rates.

**smhasher**

**SMhasher**

<table>
<thead>
<tr>
<th>Hash function</th>
<th>MiB/sec</th>
<th>cycl./hash</th>
<th>cycl./map</th>
<th>size</th>
<th>Quality problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>donthing32</td>
<td>11149469.06</td>
<td>4.00</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>donthing64</td>
<td>11787670.42</td>
<td>4.00</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>donthing128</td>
<td>11745060.76</td>
<td>4.06</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>NOP_OAAT_read64</td>
<td>11372846.37</td>
<td>14.00</td>
<td>-</td>
<td>47</td>
<td>test NOP</td>
</tr>
<tr>
<td>BadHash</td>
<td>769.94</td>
<td>73.97</td>
<td>-</td>
<td>47</td>
<td>bad seed 0, test FAIL</td>
</tr>
<tr>
<td>sumhash</td>
<td>10629.57</td>
<td>29.53</td>
<td>-</td>
<td>363</td>
<td>bad seed 0, test FAIL</td>
</tr>
<tr>
<td>sumhash32</td>
<td>42877.79</td>
<td>23.12</td>
<td>-</td>
<td>863</td>
<td>UB, test FAIL</td>
</tr>
<tr>
<td>multiply_shift</td>
<td>8026.77</td>
<td>26.05</td>
<td>226.80 (2)</td>
<td>345</td>
<td>bad seeds &amp; 0xffffffff, fails most tests</td>
</tr>
<tr>
<td>pair_multiply_shift</td>
<td>3716.35</td>
<td>40.22</td>
<td>186.34 (3)</td>
<td>609</td>
<td>fails most tests</td>
</tr>
<tr>
<td>crc32</td>
<td>383.12</td>
<td>134.21</td>
<td>257.60 (11)</td>
<td>452</td>
<td>insecure, 8590x collisions, distr. PerlinNoise</td>
</tr>
<tr>
<td>md5_32</td>
<td>350.53</td>
<td>644.31</td>
<td>894.12 (10)</td>
<td>4419</td>
<td></td>
</tr>
</tbody>
</table>

Some table cells contain special symbols or text, indicating quality problems or performance metrics.
HASH FUNCTIONS

SMhasher

<table>
<thead>
<tr>
<th>Hash function</th>
<th>MiB/sec</th>
<th>cycl./hash</th>
<th>cycl./map</th>
<th>size</th>
</tr>
</thead>
</table>
| do nothing 32  | 11149469.06 | 4.00       | -         | 13   | bad
| do nothing 64  | 11787670.42  | 4.00       | -         | 13   | bad
| do nothing 128 | 11745060.76  | 4.06       | -         | 13   | bad
| NOP_OAAT_read64 | 11972846.37 | 14.00     | -         | 47   | test
| BadHash       | 769.94   | 73.97      | -         | 47   | bad
| sumhash       | 10629.57 | 29.53      | -         | 363  | bad
| sumhash32     | 42877.79 | 23.12      | -         | 863  | UB
| multiply_shift| 8026.77  | 26.05      | 226.80 (8) | 345  | bad
| pair_multiply_shift | 3716.95 | 40.22      | 186.34 (3) | 609  | fail
| crc32         | 383.12   | 134.21     | 257.60 (11) | 452  | in
| md5_32        | 350.53   | 644.31     | 894.12 (10) | 4419 | in

Summary


So the fastest hash functions on x86_64 without quality problems are:

- xxh3low
- Wyhash
- ahash64
- t1ha2_anonce
- komihash
- FarmHash (not portable, too machine specific: 64 vs 32bit old gcc, …)
- haltime_hash128
- Spooky32
- pengyhash
- nmhash32
- mx3
- MUM/mir (different results on 32/64-bit archs, lots of bad seats to filter out)
- lasthash32
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

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

Put A

Bucket Pointers

A \mid val

Buckets
CHAINED HASHING

$hash(key) \% N$

Put A
Put B

Bucket Pointers

B | val

A | val

Buckets
CHAINED HASHING

hash(key) % N

Put A
Put B
Put C

Bucket Pointers

B | val
A | val
C | val

Buckets
CHAINED HASHING

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

Put A  
Put B  
Put C  
Put D

Bucket Pointers

B | val

A | val

C | val

Buckets
CHAINED HASHING

hash(key) % N

Put A
Put B
Put C
Put D

Bucket Pointers

B | val
A | val
C | val
D | val
**CHAINED HASHING**

- \( \text{hash(key)} \% N \)
- Put A
- Put B
- Put C
- Put D
- Put E

Bucket Pointers

\[
\begin{align*}
\text{A} & \quad \text{val} \\
\text{B} & \quad \text{val} \\
\text{C} & \quad \text{val} \\
\text{D} & \quad \text{val}
\end{align*}
\]
CHAINED HASHING

hash(key) \% N

Put A
Put B
Put C
Put D
Put E

Bucket Pointers

\begin{align*}
A & | val \\
B & | val \\
C & | val \\
D & | val \\
E & | val \\
\end{align*}
CHAINED HASHING

hash(key) \% N

- Put A
- Put B
- Put C
- Put D
- Put E
- Put F

Bucket Pointers

B | val
A | val
C | val
F | val

D | val
E | val
CHAINED HASHING

\( \text{hash(key)} \mod N \)

**Bucket Pointers**

- **Bloom Filter**
- **Bloom Filter**
- **Bloom Filter**

**HyPer**

- **64-bit Bucket Pointers**
- **48-bit Pointer**
- **16-bit Bloom Filter**

- **B** | val
- **A** | val
- **C** | val
- **F** | val
- **D** | val
- **E** | val
CHAINED HASHING

hash(key) \% N

Get G

Does key 'G' exist?

Bucket Pointers

Bloom Filter

Bloom Filter

Bloom Filter

D | val

E | val

F | val

A | val

C | val

B | val
OPEN-ADDRESSING HASHING

Single giant table of slots. Resolve collisions by 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.

Different probing schemes:
→ **Linear**: Scan slots sequentially to find entry / empty slot.
→ **Quadratic**: Jump to slots based on quadratic equation.
LINEAR PROBE HASHING

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

\[ \text{A} \ | \text{val} \]
LINEAR PROBE HASHING

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

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

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

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

```
A | val
B | val
```

63
LINEAR PROBE HASHING

hash(key) % N

A | val
B | val
A | val
C | val
D
E
F
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

B | val
A | val
C | val
D | val
LINEAR PROBE HASHING

hash(key) \% N

A
B
C
D
E
F
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

B | val
A | val
C | val
D | val
E | val
F | val
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.

Some research claims this is the best approach. Real-world results vary.
ROBIN HOOD HASHING

Variant of linear probe hashing that steals slots from "rich" keys and gives them to "poor" keys.
→ Each key tracks the distance to its optimal position.
→ 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.

Some research claims this is the best approach.
Real-world results vary.

Building a faster hash table for high performance SQL joins

November 23, 2023 · 11 min read
Andrey Pechkurov
Core Database Engineer

If you run a JOIN or a GROUP BY in a database of your choice, there is a good chance that there is a hash table at the core of the data processing. At QuestDB, we have FastMap, a hash table used for hash join and aggregate handling. While high performing, its design is a bit unconventional as it differs from most general-purpose hash tables.

In this article, we’ll tell you why hash tables are important to databases, how QuestDB’s FastMap works and why it speeds up SQL execution.
ROBIN HOOD HASHING

Variant of linear probing hashing that steals slots from "rich" keys and gives them to "poor" keys.

→ Each key tracks the number of positions it is farther from 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.

Some research claims this is the best approach.

Real-world results vary.

Building a faster hash table for high performance SQL joins

N.B. Thanks to the valuable feedback we received from the community after publishing this post, we've started experimenting with further optimizations. The most noticeable one is Robin Hood hashing, a linear probing enhancement aimed to minimize the number of look-up probes for the keys. If you prefer jumping to the code, the pull request is here.
Variants of linear probe hashing that steal slots from "rich" keys and give them to "poor" keys.

- Each key tracks the number of positions it is farther away from its optimal position in the hash 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.

Some research claims this is the best approach. Real-world results vary.
**ROBIN HOOD HASHING**

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

A
B
C
D
E
F

\[A \mid \text{val} \ [0]\]

# of "Jumps" From First Position
ROBIN HOOD HASHING

hash(key) % N

A
B
C
D
E
F

B | val [0]
A | val [0]
ROBIN HOOD HASHING

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

<table>
<thead>
<tr>
<th></th>
<th>\text{val[0]}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

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

\( hash(key) \mod N \)

- \( A \) \( \rightarrow \) \( B \) \( val[0] \)
- \( B \) \( \rightarrow \) \( A \) \( val[0] \)
- \( C \) \( \rightarrow \) \( C \) \( val[1] \)
- \( D \) \( \rightarrow \) \( D \) \( val[1] \)

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

\(\text{hash(key)} \mod N\)

\[
\begin{align*}
A & | \text{val [0]} \\
B & | \\
C & | \text{val [1]} \\
D & | \text{val [1]} \\
E & |
\end{align*}
\]

\(A[0] == E[0]\)
\(C[1] == E[1]\)
\(D[1] < E[2]\)
ROBIN HOOD HASHING

$hash(key) \% N$

A
B
C
D
E
F

<table>
<thead>
<tr>
<th>B</th>
<th>val [0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>val [0]</td>
</tr>
<tr>
<td>C</td>
<td>val [1]</td>
</tr>
<tr>
<td>E</td>
<td>val [2]</td>
</tr>
</tbody>
</table>

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

hash(key) % N

A | val [0]
B | val [0]
C | val [1]
D | val [2]
E | val [2]
F

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

$$hash(key) \mod N$$

<table>
<thead>
<tr>
<th></th>
<th><strong>val</strong> [0]</th>
<th><strong>val</strong> [1]</th>
<th><strong>val</strong> [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 (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

hash(key) % N

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

Neighborhood #1
HOPSCOTCH HASHING

Neighborhood Size = 3

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

A
B
C
D
E
F

Neighborhood #1
Neighborhood #2
Neighborhood #3
Neighborhood #4
\vdots
HOPSCOTCH HASHING

hash(key) % N

A
B
C
D
E
F

Neighborhood Size = 3

Neighborhood #1  Neighborhood #6
Neighborhood #2
Neighborhood #3
Neighborhood #4
::
Neighborhood #6
HOPSCOTCH HASHING

Neighborhood Size = 3

\( \text{hash}(key) \mod N \)

Neighborhood #3

\text{hash}(A) \mid A
HOPSCOTCH HASHING

Neighborhood Size = 3

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

<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></td>
<td></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>
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

hash(A) | A

hash(B) | B

Neighborhood #3
HOPSCOTCH HASHING

Hashing formula:

\[ hash(key) \mod N \]

Neighborhood Size = 3

A
B
C
D
E
F

Neighborhood #3:

\[ hash(A) \mid A \]

\[ hash(B) \mid B \]
HOPSCOTCH HASHING

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

$\text{hash}(A)$

$\text{hash}(B)$

$\text{hash}(C)$

$\text{hash}(D)$

$\text{hash}(E)$

$\text{hash}(F)$

Neighborhood Size = 3

Neighborhood #3
HOPSCOTCH HASHING

hash(key) % N

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C

Neighborhood #4

Neighborhood Size = 3
HOPSCOTCH HASHING

Neighborhood Size = 3

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

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

Neighborhood #4
HOPSCOTCH HASHING

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

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

Neighborhood Size = 3

Neighborhood #4
HOPSCOTCH HASHING

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

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

Neighborhood Size = 3

Neighborhood #3
HOPSCOTCH HASHING

Neighborhood Size = 3

$hash(key) \% N$

$hash(A) | A$

$hash(B) | B$

$hash(C) | C$

$hash(D) | D$

Neighborhood #3
HOPSCOTCH Hashing

hash(key) % N

Neighborhood Size = 3

Neighborhood #3
HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

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

Neighborhood #4
HOPSCOTCH HASHING

$hash(key) \mod N$

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

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

$Neighborhood\ Size = 3$

$Neighborhood\ \#4$
HOPSCOTCH HASHING

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

<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>$\text{hash}(A)$</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(B)$</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(C)$</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(E)$</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(D)$</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{Neighborhood Size} = 3$

$\text{Neighborhood #3}$
Hopscotch Hashing

Neighborhood Size = 3

\[ hash(key) \% N \]

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

Neighborhood Size = 3

hash(key) % N

<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>
</tbody>
</table>

Neighborhood #6
Hopscotch Hashing

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

<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>$\text{hash}(A)$</td>
<td>$A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(B)$</td>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(C)$</td>
<td>$C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(D)$</td>
<td>$D$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{hash}(E)$</td>
<td>$E$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Neighborhood #6

Neighborhood Size = 3
HOPSCOTCH HASHING

Neighborhood Size = 3

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

<table>
<thead>
<tr>
<th></th>
<th>hash(key) % N</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>hash(A)</td>
<td>A</td>
</tr>
<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>
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

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$
CUCKOO HASHING

Put A: $hash_1(A)$, $hash_2(A)$

A | val
CUCKOO HASHING

Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)
PUT A: $hash_1(A)$
   $hash_2(A)$

PUT B: $hash_1(B)$
   $hash_2(B)$
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
    $\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
    $\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
    $\text{hash}_2(C)$
Cuckoo Hashing

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
$\text{hash}_2(C)$
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
$\text{hash}_2(C)$
$\text{hash}_1(B)$
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
$\text{hash}_2(C)$
$\text{hash}_1(B)$
**Cuckoo Hashing**

Put A: $hash_1(A)$  
$hash_2(A)$

Put B: $hash_1(B)$  
$hash_2(B)$

Put C: $hash_1(C)$  
$hash_2(C)$  
$hash_1(B)$  
$hash_2(A)$

- **C** | **val**
- **B** | **val**
- **A** | **val**
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
$\text{hash}_2(C)$
$\text{hash}_1(B)$
$\text{hash}_2(A)$

Get B: $\text{hash}_1(B)$
$\text{hash}_2(B)$
HASH TABLE CONTENTS

Tuple Data vs. Pointers/Offsets to Data
→ Whether to store the tuple directly inside of the hash table.
→ Storing tuples inside of the table not possible in open-addressing if there is variable length data.

Join Keys-only vs. Join Keys + Hashes
→ Whether to only store the original join key(s) in the hash table or also include the computed hashed key.
→ Classic compute vs. storage trade-off.
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.

This enhancement is 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.

This enhancement is 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.

This enhancement is 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.

This enhancement is 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.

This enhancement is sometimes called sideways information passing.
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 ($R \bowtie S$)

### $4 \times$ Intel Xeon CPU E7-4870v2 (32 cores)

| Source | $|R| = 128M$, $|S| = 1280M$ |

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

### Source:
Stefan Schuh

Optimized

↑ Higher is Better
TPC-H Q19

4× Intel Xeon CPU E7-4870v4 (32 cores)
Scale Factor 100

- Join Operator
- Remaining Query

<table>
<thead>
<tr>
<th>No-Part (Linear)</th>
<th>No-Part (Array)</th>
<th>Radix (Linear)</th>
<th>Radix (Array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>279</td>
<td>301</td>
<td>285</td>
</tr>
<tr>
<td>13%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
</tr>
</tbody>
</table>

Source: Stefan Schuh
STRING HASH TABLES

2× Intel Xeon CPU E5-2460v4 (10 cores)
Join + Group By Microbenchmark

SAHA: A STRING ADAPTIVE HASH TABLE FOR ANALYTICAL DATABASES
APPL. SCI. 2020
PARTING THOUGHTS

Partitioned-based joins outperform no-partitioning algorithms in most settings, but it is non-trivial to tune it correctly.

AFAIK, most DBMSs picks one hash join implementation and does not try to be adaptive.
NEXT CLASS

Worst-Case Optimal Joins (aka multi-way joins)
Profiling with CPU Performance Counters