Data Processing on Modern Hardware

Jana Giceva

Lecture 4: In memory Joins
In-memory joins

After plain select queries, let us now look at join queries:

```
SELECT COUNT(*)
FROM orders, lineitem
WHERE o_orderkey = l_orderkey
```

We want to ignore result materialization for now, thus only count the result tuples.

Furthermore, we assume:
- No exploitable order
- No exploitable indices (input might be an intermediate result), and
- An equality join predicate (as above).
- No prior knowledge about key distribution
History of join processing: hashing vs. sorting

1970s – sorting
1980s – hashing
1990s – equivalent
2000s – hashing
2010s – hashing
2020s – ???

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

→ Trade-offs between partitioning & non-partitioning Hash-Join.

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

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

→ New optimizations and results for Radix Hash Join.

→ Hold up everyone! Let's look at everything more carefully!

src: Andy Pavlo (Advanced Databases) CMU
Hash Join

Hash Join is a good match for the equi-join example earlier.

To compute $R \bowtie S$,

1. **Build a hash table** on the outer join relation $R$
2. **Scan** the inner relation $S$, and **probe** into the hash table for each tuple $s \in S$.

```plaintext
1 function: hash_join(R, S)
   // Build phase
2   for each tuple $r \in R$ do
3       insert $r$ into hash table $H$
   // Join phase
4   for each tuple $s \in S$ do
5     probe $H$ and append matching tuples to result
```

$$\pi \quad R.id = S.id$$
Hash Join

1. build

2. probe

Complexity O(N)

Build is easy to parallelize

Probe needs no synchronization
Parallel Hash Join

Key characteristics:

- **Split** the input relations into chunks

- **Build:**
  - Each thread operates on its own input chunk and writes to a shared hash table
  - The shared hash table is protected using locks
  - Usually very low contention

- **Probe:**
  - Multiple readers – no synchronization needed
  - Each thread probes the hash table for its own chunk’s tuples
  - Passes on the matched tuples
Algorithm design goals for modern hardware:

- Minimize synchronization
  - avoid taking latches during execution
- Minimize memory access cost
  - ensure that data is local to worker thread
  - reuse data while it is still in the cache

The naïve parallel hash join has a lot of random accesses

- For large relations, every hash table access will likely be a cache miss
- The better the hash function, the more random the distribution of keys

Cost per tuple (build phase):

- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

hash join is severely latency bound
Hardware-oblivious vs conscious dilemma

- **Hardware-conscious:**
  - Best performance can be achieved by fine-tuning to the underlying architecture:
    - Cache hierarchy, translation lookaside buffer (TLB), non-uniform memory accesses (NUMA), etc.

- **Hardware-oblivious:**
  - Algorithms can be efficient while remaining hardware oblivious because modern hardware hides the performance loss inherent in the multi-layer memory hierarchy with hyper-threads
    - Easily portable to different hardware
    - More robust to data-skew
Quick recap of virtual memory and address translation
Memory translation

- Request is virtual address (VA), want physical address (PA)
- Use look-up table that we call **page table (PT)**

1. Processor sends virtual address to MMU
2,3. MMU fetches PTE from page table in cache/memory
4. MMU sends physical address to cache/memory requesting data
5. Cache/memory sends data to processor

![Diagram of memory translation]

- VA = Virtual Address
- PA = Physical Address
- PTEA = Page Table Entry Address
- PTE = Page Table Entry
- Data = Content of memory stored at VA originally requested by CPU
Translation Lookaside Buffer (TLB)

A TLB hit eliminates a memory access!
A TLB miss incurs additional memory access (the PTE)
The standard page size on x86 is 4kB, but larger sizes 2MB and 1GB are also available (i.e., huge pages).

Two level TLB caches
- Separate L1 for instruction and data
- Unified L2 for both
Back to hash joins
Factors that affect cache misses in a DBMS:
- Cache + TLB capacity
- Locality (temporal + spatial)

Key approaches to use:

- **Sequential** (strided) access (e.g., table scan):
  - Cluster and align data to a cache line
  - Execute more operations per cache line

- **Random** access (e.g., index look-ups):
  - Pre-fetch data from memory manually
  - Use the blocking technique – partition data to fit in cache
  - Watch-out for the TLB cache

src: Johannes Gehrke (Main-memory database systems) http://www.cs.cornell.edu/courses/cs632/2001sp/slides/Main-memory%20database%20systems.ppt
Hashing schemes

- **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

- **Open addressing:**
  - Use a single giant table of slabs
  - Linear probing (LP) – resolve collisions by linearly searching for the next free slot in the table
  - Other probe sequences (e.g., quadratic, robin-hood, hopscotch, etc.)

- **Different trade-offs:**
  - **Locality:** pointer chasing vs. sequential access
    - Chaining better performance during build phase
    - LP better throughput during probe phase
  - **Robustness:** on high load factors, LP suffers from primary clustering
Hash Table implementation

- Even for a simple chain hashing scheme, there are many things to consider.

- Naïve implementation:
  - Hash table is an array of head pointers, each of which points to the head of a linked bucket chain.
  - Each bucket is implemented as a 48-byte record:
    - `free` points to the next available tuple space,
    - `next` pointer leads to the next overflow buffer
    - The bucket holds two 16-byte tuples.
  - Since it is a shared hash table, latches are needed for synchronization. Implemented as a separate latch array.
  - 3 separate cache lines

src: Blanas et al. Design and evaluation of main memory hash join algorithms for multi-core CPUs SIGMOD 2011
Hash Table implementation

- An alternative chain hashing scheme:
  - The main hash table is a contiguous array of buckets.
  - Header contains 1-byte for latch, and a 7-byte counter indicating the number of tuples in the bucket.
  - Contains two 16-byte tuples.
  - For overflow, additional buckets are allocated outside the main hash table, referenced by the next pointer.
  - Fits in 1 cache line

Contiguous memory block can reduce the number of cache misses significantly.

src: Balkesen et al. Main-memory Hash Joins on Modern Processor Architectures ICDE 2014
Performance impact of HT implementation

cycles per output tuple

number of threads

Naïve HT implementation
Alternative HT implementation
Build

src: Balkesen et al. Main-memory Hash Joins on Modern Processor Architectures ICDE 2014
The **hash join** has inherently a lot of **random accesses**, which is a problem when the **data** is large and **does not fit in the cache**.

There are two main options one could take:

- **Pre-fetching**
  - Recall assignment 1 → the hardware pre-fetcher cannot help with random accesses
  - But: a software pre-fetcher can issue memory requests ahead of time and hide latencies [1]

- **Partitioning**
  - Recall blocked matrix multiplication example →
  - Split the input relations into cache-resident buffers by hashing the tuples’ join key(s) [2]
  - Insight: the cost of partitioning is often less than the overhead of cache misses for build and probe

Case 1: Software based prefetching

- To hide cache miss latencies in hash joins, one can use **software pre-fetching**.
- Modify the source code using **special instructions** (compiler **intrinsic**) on any pointer in the program.

\[
\text{__mm_prefetch(} \text{void *p, enum __mm_hint h)}; \]

**Group pre-fetching**
- Modified forms of compiler transformations called **strip mining** and **loop distributions**
- Restructure the code so that hash probe accesses resulting from groups of G consecutive probe tuples can be pipelined

**Software pipelining**
- Generate efficient schedules for loops by overlapping the execution of operations from different iterations of the loop.
- Assume there are no inter-tuple dependencies (for simplicity)

src: Chen et al. Improving Hash Join Performance through Prefetching. ICDE 2004
Group pre-fetching (example)

for $i=0$ to $N-1$ do
  code 0;
  visit ($m_i^1$); code 1;
  visit ($m_i^2$); code 2;
  ... 
  visit ($m_i^k$); code $k$;
end for

for $j=0$ to $N-1$ step $G$ do
  for $i=j$ to $j+G-1$ do
    code 0;
    prefetch ($m_i^1$);
    end for
  for $i=j$ to $j+G-1$ do
    visit ($m_i^1$); code 1;
    prefetch ($m_i^2$);
    end for
  for $i=j$ to $j+G-1$ do
    visit ($m_i^2$); code 2;
    prefetch ($m_i^3$);
    end for
  ... 
  for $i=j$ to $j+G-1$ do
    visit ($m_i^k$); code $k$;
    end for
end for

src: Chen et al. Improving Hash Join Performance through Prefetching. ICDE 2004
Software-pipelined pre-fetching

```
for i=0 to N-1 do
  code 0;
  visit \( m_i^1 \); code 1;
  visit \( m_i^2 \); code 2;
  ...
  visit \( m_i^k \); code k;
end for
```

```
for j=0 to N-kD-1 do
  i=j+kD;
  code 0 for element i;
  prefetch \( m_i^1 \);
  i=j+(k-1)D;
  visit \( m_i^1 \); code 1 for element i;
  prefetch \( m_i^2 \);
  i=j+(k-2)D;
  visit \( m_i^2 \); code 2 for element i;
  prefetch \( m_i^3 \);
  ...
  i=j;
  visit \( m_i^k \); code k for element i;
end for
```

\( D \) is the prefetching distance.

src: Chen et al. Improving Hash Join Performance through Prefetching. ICDE 2004
Group vs software-pipelined pre-fetching

**Software-pipelined:**
- Can always hide miss latencies
- But, has a larger book-keeping overhead and larger maintained state

**Group:**
- Easier to implement
- Not all cache misses can be hidden (esp. when code 0 is empty)
  - Can be amortized with large group of elements

src: Chen et al. Improving Hash Join Performance through Prefetching. *ICDE 2004*
Impact of prefetching on join performance

Big reduction of d-cache stalls
Eliminate TLB stalls

src: Chen et al. Improving Hash Join Performance through Prefetching. ICDE 2004
Case 2: partitioning

- Recall the *blocking* matrix multiplication example?

- In *blocking*, an *algorithm* is restructured to reuse chunks of data that fit in the cache.

For (i=0; i<M; i++)
  for (j=0; j<N, j++)
    process(a[i][j]);

For (b=0; b<N/B; b++)
  for (i=0; i<M, i++)
    for (j=b*B; j<(b+1)*B; j++)
      process(a[i][j]);

- In *partitioning*, the *layout* of the *input data* is reorganized to make maximum use of the cache
  - Make sure that partitions fit in the cache

Quicksort(relation[N])

Partition relation into blocks < cache size
For each partition r
  Quicksort(relation[PARTITIONSIZE]);
Merge all partitions

Partitioned Hash Join

1. partition
2. build
3. probe

Parallelism: assign partitions to threads → no synchronization needed

Cache-sized chunks

one hash table per partition

$R$ scan $h_1 \rightarrow h_2$ $r_1 \rightarrow h_2 \rightarrow h_2$ $r_2 \rightarrow h_2 \rightarrow h_2$ $r_3 \rightarrow h_2 \rightarrow h_2$ $r_4 \rightarrow h_2 \rightarrow h_2$

$h_2 \rightarrow h_1$ $s_1 \rightarrow h_1$ $s_2 \rightarrow h_1$ $s_3 \rightarrow h_1$ $s_4 \rightarrow h_1$

$S$ scan

1. partition
Cache analysis of Partitioned Hash Joins

- **Build / Probe are now contained within the caches:**
  - From 34 down to 15/21 instructions per tuple (build/probe)
  - From 1.5 down to 0.01 cache misses per tuple
  - From 3.3 down to almost no TLB misses

- Joining two relations with 8B key+payload and 128M tuples (total size 977MB)
- Measured on 3 different machines

- **Partitioning** is now critical
  - Many partitions are far apart
  - Each one will reside on its own page
  - Run out of **TLB entries** (100-500)

src: Balkesen et al. Main-memory Hash Joins on Modern Processor Architectures ICDE 2014
Partitioning is expensive beyond $2^8 - 2^9$ partitions.

Due to TLB thrashing.

```
for all input tuples t do
    h ← hash(t.key)
    out[ pos[h] ] ← t
    pos [h] ← pos[h] + 1
end for
```
Radix partitioning (basic)

// Build a histogram
for i = 0 to N – 1 do
    + + histogram[h(input[i])];

// Calculate prefix-sum
offset = 0;
for i = 0 to num_partitions – 1 do
    dest[i] = offset;
    offset += histogram[i];

// Partition the data
for i = 0 to N – 1 do
    bucket_num = h(input[i]);
    output[dest[bucket_num]] = input[i];
    + + dest[bucket_num];

Partition a dataset into $2^R$ partitions.

- In the **first pass** over the data, for each partition we count the entries that will be sent to it.

- From this histogram, we calculate the start index of each partition (i.e., prefix sum).

- The **second pass** over the data copies the entries to their designated partition.
Optimizing the radix sort - partitioning

It’s an art in itself and was studied extensively

- Single vs. multi-pass partitioning
- Software Write-Combine Buffers
- Non-temporal Streaming
- Using huge page tables
- NUMA awareness → covered in two weeks

Creating too many partitions can easily thrash the TLB cache.
Thus, do a multi-pass partitioning, and limit the fan-out of each partitioning pass.
Multi-pass partitioning

![Graph showing the comparison between single-pass and two-pass partitioning](image)

- **Single-pass partitioning**
- **Two-pass partitioning**

Throughput [million tuples/sec]

Radix bits

- Values range from 4 to 16.
Software managed buffers

Naïve partitioning

```plaintext
for all input tuples t do
    h ← hash(t.key)
    copy t to out[pos[h]]
    pos[h] ← pos[h] + 1
end for
```

- TLB miss only every bufsize tuples
- Choose bufsize to match cache line size

Software managed buffers

```plaintext
for all input tuples t do
    h ← hash(t.key)
    copy t to out[pos[h]]
    pos[h] ← pos[h] + 1
end for
```

```plaintext
for all input tuples t do
    h ← hash(t.key)
    buf[h][pos[h] mod bufsize] ← t
    if pos[h] mod bufsize = 0 then
        copy buf[h] to out[pos[h] − bufsize]
    end if
    pos[h] ← pos[h] + 1
end for
```
Software managed buffers – suitable bufsize

![Graph showing the relationship between number of cache-lines per partition buffer and Radix partitioning time.]

src: Schuhknecht et al. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning VLDB 2015
Non-temporal Streaming Stores

Key idea: keep the working set warm in cache, and issue memory writes that bypass the cache

Method: non-temporal streaming stores

\[ \text{\_mm256\_stream\_si256}(\text{\_m256i}\* \text{mem, \_m256i a}) \]

- This AVX intrinsic writes 4 buffered 64-bit entries to a partition at once (i.e., half a cache line).
- The processor tries to fill a cache line in its own write-combine buffer before writing to memory.
- As soon as it is filled, it is flushed out without reading the corresponding cache-line from memory.
- Caveat: the memory address must be aligned to 32 Bytes = 256 bits
- For AVX 512, we can fill a full cache line per call 😊
Partitioning performance

![Radix partitioning time vs Number of partitions]

- Original
- SW Buffers
- SW Buffers + Streaming store
- Original (prefetched)
- SW Buffers + Streaming store (prefetched)

Source: Schuhknecht et al. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning *VLDB 2015*
Results

Radix join (basic)

No partitioned hash join with prefetching

Radix join (with chain hashing) with optimizations

Radix join (with linear hashing) with optimizations

Radix join (with LH) with optimizations and NUMA awareness

Lighter shade is with huge pages

src: Schuh et al. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory SIGMOD 2016
So far, join on narrow tuples

- If optimized well, with prefetching or SWWCB and streaming instructions, the join quickly becomes memory bound.

- A simple analytical model can tell us when to use which type of join (no-partitioning, or radix-join).

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Model for memory bandwidth consumed per tuple for suboperations of hash join algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bytes read</td>
</tr>
<tr>
<td>Out-of-cache build</td>
<td>$CL + t$</td>
</tr>
<tr>
<td>Out-of-cache probe</td>
<td>$CL \cdot \left\lceil \frac{t}{CL} \right\rceil + t$</td>
</tr>
<tr>
<td>In-cache build</td>
<td>$t$</td>
</tr>
<tr>
<td>In-cache probe</td>
<td>$t$</td>
</tr>
<tr>
<td>Partition</td>
<td>$t$</td>
</tr>
</tbody>
</table>

src: Makreshanski et al. Many-query join: efficient shared execution of relational joins on modern hardware *VLDBJ 2018*
References

- Various papers cross-referenced in the slides
  - Chen et al. Improving Hash Join Performance through Prefetching. *ICDE 2004*
  - Shatdal et al. Cache conscious algorithms for relational query processing. *VLDB 1994*
  - Balkesen et al. Design and evaluation of main memory hash join algorithms for multi-core CPUs *SIGMOD 2011*
  - Blanas et al. Main-memory Hash Joins on Modern Processor Architectures *ICDE 2014*
  - Polychroniou and Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison and radix-sort. *SIGMOD 2014*
  - Schuhknecht et al. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning *VLDB 2015*
  - Schuh et al. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory *SIGMOD 2016*
  - Makreshanski et al. Many-query join: efficient shared execution of relational joins on modern hardware *VLDBJ 2018*

- Lecture: *Database Systems on Modern CPU Architectures* by Prof. Thomas Neumann (TUM)
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Advanced Databases* by Prof. Andy Pavlo (CMU)
- Book: *Computer Systems: A Programmer’s Perspective* 3rd edition by Bryant and O’Hallaron
- Book: *What every programmer should know about memory* by Ulrich Drepper

- Intel manuals for software write combining, streaming instructions, software-based prefetching
- Check out the code from Cagri Balkesen for high performance radix join implementation:
  - https://www.systems.ethz.ch/node/334
Appendix – Address Translation
How do we perform the virtual $\rightarrow$ physical address translation?

CPU Chip

Virtual address (VA)

0x4100

MMU

Physical address (PA)

0x4

Main memory

0:
1:
2:
3:
4:
5:
6:
7:

Data (int/float)

M-1:
Address Translation: Page Hit

- Request is virtual address (VA), want physical address (PA)
- Use look-up table that we call **page table (PT)**

1. Processor sends virtual address to MMU
2. MMU fetches PTE from page table in cache/memory
3. MMU sends physical address to cache/memory requesting data
4. Cache/memory sends data to processor

**Diagram:**

1) VA → MMU
2) PTEA → MMU
3) PTE → MMU
4) PA → MMU
5) Data → CPU

**Abbreviations:**
- VA = Virtual Address
- PA = Physical Address
- PTEA = Page Table Entry Address
- PTE = Page Table Entry
- Data = Content of memory stored at VA originally requested by CPU
Address Translation: Page Fault

1. Processor sends virtual address to MMU
2,3. MMU fetches PTE from page table in cache/memory
4. Page is not there, MMU triggers page fault exception
5. Handler identifies victim (and, if dirty, pages it out to disk)
6. Handler pages in new page and updates PTE in memory
7. Handler returns to original process, restarting faulting instructions
Address Translation

Virtual Address

TLB Lookup

TLB miss
Check the Page Table

Page not in Memory

Page Fault (OS loads page)

Find in Disk

TLB hit
PTE
Protection Check

PTE

Access Denied
Protection Fault

SIGSEGV

Access Permitted
Physical Address

Check cache

Page in Memory

Update TLB

Find in Mem