CS 764: Topics in Database Management Systems
Lecture 2: Join

Xiangyao Yu
9/12/2021
Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO
North Dakota State University

We study algorithms for computing the equijoin of two relations in a system with a standard architecture but with large amounts of main memory. Our algorithms are especially efficient when the main memory available is a significant fraction of the size of one of the relations to be joined; but they can be applied whenever there is memory equal to approximately the square root of the size of one relation. We present a new algorithm which is a hybrid of two hash-based algorithms and which dominates the other algorithms we present, including sort-merge. Even in a virtual memory environment, the hybrid algorithm dominates all the others we study.

Finally, we describe how three popular tools to increase the efficiency of joins, namely filters, Babb arrays, and semijoins, can be grafted onto any of our algorithms.

Categories and Subject Descriptors: H.2.0 [Database Management]: General; H.2.4 [Database Management]: Systems—query processing; H.2.6 [Database Management]: Database Machines

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hash join, join processing, large main memory, sort-merge join
Agenda

System architecture and notations

Join algorithms
  • Sort merge join
  • Simple hash join
  • GRACE hash join
  • Hybrid hash join

Partition overflow and additional techniques
Agenda

System architecture and notations

Join algorithms
  • Sort merge join
  • Simple hash join
  • GRACE hash join
  • Hybrid hash join

Partition overflow and additional techniques
System Architecture and Assumptions

CPU: uniprocessor
  • No multi-core synchronization complexity
  • Could be built on systems of the day

Memory
  • Tens of Megabytes
  • Good for both sequential and random accesses
  • Capacity is smaller than disk

Disk
  • Good for only sequential accesses
Notation

**Relations**: $R, S \ (|R| < |S|)$

**Join**: $S \bowtie R$

**Memory**: $M$

**$|R|**: number of blocks in relation $R$ (similar for $S$ and $M$)

**$F$**: hash table for $R$ occupies $|R| \times F$ blocks

Focus only on equi-join
Notation

**Relations**: R, S (|R| < |S|)

**Join**: S \(\bowtie\) R

**Memory**: M

| R |: number of blocks in relation R (similar for S and M)

F: hash table for R occupies |R| * F blocks

```
SELECT *
FROM R, S
WHERE R.C3 = S.C5
```
answer = {}
for t₁ in R do
  for t₂ in S do
    if R.C3 = S.C5
      then answer = answer U {(C1,…,C8)}
return answer

Vanilla query executor

Relation R

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relation S

<table>
<thead>
<tr>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SELECT *
FROM R, S
WHERE R.C3 = S.C5
answer = {}

for t₁ in R do
  for t₂ in S do
    if R.C3 = S.C5
      then answer = answer ∪ {(C1,...,C8)}

return answer

Vanilla query executor

Key question: How to execute a join fast?

Relation R

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td>C4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relation S

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>C6</td>
<td>C7</td>
</tr>
<tr>
<td>C8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SELECT *
FROM R, S
WHERE R.C3 = S.C5
Agenda

System architecture and notations

**Join algorithms**
- Sort merge join
- Simple hash join
- GRACE hash join
- Hybrid hash join

Partition overflow and additional techniques
Sort Merge Join

**Key idea:** sort both relations based on join attributes, then traverse both relations in the sorting order
Sort Merge Join

**Key idea:** sort both relations based on join attributes, then traverse both relations in the sorting order.

**Challenge:** If a relation does not fit in memory, need to sort data on disk.
Sort Merge Join

Phase 1: Produce sorted runs of S and R
Phase 2: Merge runs of S and R, output join result

Unsorted R and S
Sort Merge Join

Phase 1: Produce sorted runs of S and R
Phase 2: Merge runs of S and R, output join result

Each sorted run can fit in memory
Sort Merge Join

Phase 1: Produce sorted runs of S and R

Phase 2: Merge runs of S and R, output join result
Sort Merge Join – Phase 1

Phase 1: Produce sorted runs of S and R

• Each run of S will be \(2 \times |M|\) average length
Sort Merge Join – Phase 1

Phase 1: Produce sorted runs of S and R
  • Each run of S will be $2 \times |M|$ average length

Q: Where does 2 come from?
A: Replacement selection
Naïve solution:
- Load $|M|$ blocks
- Sort
- Output $|M|$ blocks

Each run contains $|M|$ blocks
Replacement selection:

- load | M | blocks and sort

While heap is not empty

  If new tuple ≥ all tuples in output
    add new tuple to heap
  else
    save new tuple for next run
Replacement selection:
  • load \( |M| \) blocks and sort

While heap is not empty
  If new tuple \( \geq \) all tuples in output
    add new tuple to heap
  else
    save new tuple for next run

A run contains \( 2 \times |M| \) blocks on average

Replacement selection:
- load $|M|$ blocks and sort

While heap is not empty
- If new tuple $\geq$ all tuples in output
  - add new tuple to heap
- else
  - save new tuple for next run

A run contains $2 \times |M|$ blocks on average

Total number of runs:
\[
\frac{|S|}{2 \times |M|} + \frac{|R|}{2 \times |M|} \leq \frac{|S|}{|M|}
\]

Sort Merge Join – Phase 2

Phase 2: Merge runs of S and R, output join result

- One input buffer required for each run

Find matches in sorted runs

Memory layout in Phase 2

<table>
<thead>
<tr>
<th>Memory</th>
<th>in-buf</th>
<th>in-buf</th>
<th>...</th>
<th>in-buf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R₀</td>
<td>R₁</td>
<td>...</td>
<td>Rₙ</td>
</tr>
<tr>
<td></td>
<td>S₀</td>
<td>S₁</td>
<td>...</td>
<td>Sₘ</td>
</tr>
</tbody>
</table>
Sort Merge Join – Phase 2

Phase 2: Merge runs of S and R, output join result
  • One input buffer required for each run

Requirement

\[ |M| \geq \text{total number runs} \]

Satisfied if

\[ |M| \geq \frac{|S|}{|M|} \]

namely

\[ |M| \geq \sqrt{|S|} \]
Hash Join

Build a hash table on the smaller relation (R) and probe with larger (S)
Hash tables have overhead, call it F
When R doesn’t fit fully in memory, partition hash space into ranges

Hash table on R
(size = |R| x F)
Agenda

System architecture and notations

Join algorithms
- Sort merge join
- Simple hash join
- GRACE hash join
- Hybrid hash join

Partition overflow and additional techniques
Simple Hash Join

• Build a hash table on \( R \)

Hash table on \( R \)
(size = \(|R| \times F\) )

Memory

\( S \)
Simple Hash Join – 1st pass

- Build a hash table on $R$
- If $R$ does not fit in memory, find a subset of buckets that fit in memory

Hash table on $R$
(size = $|R| \times F$)

Memory

write back to disk

$S$
Simple Hash Join – 1st pass

• Build a hash table on \( R \)
• If \( R \) does not fit in memory, find a subset of buckets that fit in memory
• Read in \( S \) to join with the subset of \( R \)

Hash table on \( R \)
(size = |\( R \)| \( \times \) \( F \))

write back to disk

Memory

\( S \)
Simple Hash Join – 1st pass

- Build a hash table on \( R \)
- If \( R \) does not fit in memory, find a subset of buckets that fit in memory
- Read in \( S \) to join with the subset of \( R \)
- The remaining tuples of \( S \) and \( R \) are written back to disk

\[
\text{Hash table on } R \\
\text{(size } = |R| \times F)
\]

\[
\text{Memory}
\]

\[
\text{write back to disk}
\]

\[
\text{write back to disk}
\]

\[
S
\]
Simple Hash Join – 2\textsuperscript{nd} pass

- Build a hash table on $R$
- If $R$ does not fit in memory, find a subset of buckets that fit in memory
- Read in $S$ to join with the subset of $R$
- The remaining tuples of $S$ and $R$ are written back to disk

\[
\text{Hash table on } R \quad \text{(size } = |R| \times F)\]

\[
\text{Memory}
\]

\[
\text{S}
\]

write back to disk

write back to disk
Simple Hash Join – 3rd pass

- Build a hash table on \( R \)
- If \( R \) does not fit in memory, find a subset of buckets that fit in memory
- Read in \( S \) to join with the subset of \( R \)
- The remaining tuples of \( S \) and \( R \) are written back to disk

Hash table on \( R \)
(size = \( |R| \times F \))

Memory

\( S \)
Agenda

System architecture and notations

Join algorithms

• Sort merge join
• Simple hash join
• GRACE hash join
• Hybrid hash join

Partition overflow and additional techniques
GRACE Hash Join

Phase 1: Partition both R and S into pairs of k shards
Phase 2: Separately join each pairs of partitions
GRACE Hash Join

Phase 1: Partition both R and S into pairs of k shards

Phase 2: Separately join each pairs of partitions
GRACE Hash Join

Phase 1: Partition both R and S into pairs of k shards

Phase 2: Separately join each pairs of partitions

Memory layout in Phase 2

Hash table for $R_i$

input buffer for S

Memory
GRACE Hash Join

Assume \( k \) partitions for \( R \) and \( S \)

In phase 1, needs one output buffer (i.e., block) for each partition

\[ k \leq |M| \]
GRACE Hash Join

Assume \( k \) partitions for \( R \) and \( S \)

In phase 1, needs one output buffer (i.e., block) for each partition

\[
k \leq |M|
\]

In phase 2, the hash table of each shard of \( R \) must fit in memory

\[
\frac{|R|}{k} \times F \leq |M|
\]
**GRACE Hash Join**

**Assume** \( k \) **partitions for** \( \textbf{R} \) **and** \( \textbf{S} \)**

**In phase 1**, needs one output buffer (i.e., block) for each partition

\[
k \leq |M|
\]

**In phase 2**, the hash table of each shard of \( \textbf{R} \) must fit in memory

\[
\frac{|R|}{k} \times F \leq |M|
\]

The maximum size of \( \textbf{R} \) to perform Grace hash join:

\[
|R| \leq \frac{|M|}{F} k \leq \frac{|M|^2}{F} \quad \Rightarrow \quad |M| \geq \sqrt{|R| \times F}
\]
GRACE vs. Simple Hash Join

When $|R| \times F < |M|$
- Simple hash join incurs no IO traffic (better)
- GRACE hash join writes and reads each table once
- Trivial optimization to GRACE: use simple hash join when $|R| \times F < |M|$

When $|M|^2 \geq |R| \times F >> |M|$
- Simple hash join incurs significant IO traffic
- GRACE hash join writes and reads each table once (better)
GRACE vs. Simple Hash Join

When $|R| \times F < |M|$
- Simple hash join incurs no IO traffic (better)
- GRACE hash join writes and reads each table once
- Trivial optimization to GRACE: use simple hash join when $|R| \times F < |M|$

When $|M|^2 \geq |R| \times F >> |M|$
- Simple hash join incurs significant IO traffic
- GRACE hash join writes and reads each table once (better)

Discussion Question:
What if $|R| \times F > |M|^2$?
Agenda

System architecture and notations

Join algorithms
  • Sort merge join
  • Simple hash join
  • GRACE hash join
  • Hybrid hash join

Partition overflow and additional techniques
Hybrid Hash Join

When two algorithms are good in different settings, create a hybrid!
Hybrid Hash Join

When two algorithms are good in different settings, create a hybrid!

**Key observation**: when $|R|$ is relatively small (e.g., $|R| = 2 |M|$), significant memory capacity is unused in Phase 1 of GRACE join.
Hybrid Hash Join

When two algorithms are good in different settings, create a hybrid!

**Key observation**: when $|R|$ is relatively small (e.g., $|R| = 2|M|$), significant memory capacity is unused in Phase 1 of GRACE join.

**Key idea**: Use the otherwise-unused memory to build hash table for $R_0$.

---

**Memory layout in Phase 1 of GRACE hash join**

<table>
<thead>
<tr>
<th>out-buf</th>
<th>out-buf</th>
<th>...</th>
<th>out-buf</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0$</td>
<td>$R_1$</td>
<td></td>
<td>$R_k$</td>
</tr>
</tbody>
</table>

Hash table for $R_0$
Hybrid Hash Join

Case 1: $|R| \times F < |M|$

- No need to partition $R$
- Identical to simple hash join

Memory layout in Phase 1 of hybrid hash join

Hash table for $R_0$
Hybrid Hash Join

Case 1: $|R| \times F < |M|$
- No need to partition $R$
- Identical to simple hash join

Case 2: $|R| \times F = \alpha |M| (\alpha \text{ is small})$
- $R_0$ is a significant fraction of $R$
- $R_0$ is not written to disk
- Performance is like simple hash join

Memory layout in Phase 1 of hybrid hash join

<table>
<thead>
<tr>
<th>out-buf</th>
<th>Hash table for $R_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td></td>
</tr>
<tr>
<td>$R_2$</td>
<td></td>
</tr>
</tbody>
</table>
Hybrid Hash Join

Case 1: \( |R| \times F < |M| \)
- No need to partition R
- Identical to simple hash join

Case 2: \( |R| \times F = \alpha |M| \) (\( \alpha \) is small)
- \( R_0 \) is a significant fraction of R
- \( R_0 \) is not written to disk
- Performance is like simple hash join

Case 3: \( |R| \times F >> |M| \)
- \( R_0 \) is an insignificant fraction of R
- Performance is like GRACE hash join

Memory layout in Phase 1 of hybrid hash join
**Conclusion 1:** Hash join is generally better than sort-merge join.

**Conclusion 2:** Hybrid hash join is strictly better than simple and GRACE hash joins.
Agenda

System architecture and notations

Join algorithms

• Sort merge join
• Simple hash join
• GRACE hash join
• Hybrid hash join

Partition overflow and additional techniques
Partition Overflow

So far we assume uniform random distribution for $R$ and $S$

What if we guess wrong on size required for $R$ hash table and a partition does not fit in memory?

**Solution**: further divide into smaller partitions range
Additional Techniques

Babb array (or bitmap filter)

- One bit per hash bucket in R
- Set the bit if a tuple in R maps to the bucket
- When scanning S, if a tuple hashes to a bucket where the bit is unset, can discard the tuple immediately
Additional Techniques

Babb array (or bitmap filter)

- One bit per hash bucket in R
- Set the bit if a tuple in R maps to the bucket
- When scanning S, if a tuple hashes to a bucket where the bit is unset, can discard the tuple immediately

Semi-join

- Project join attributes from R, join to S, then join that result back to R
- Useful if full R tuples won’t fit into memory, but join will be selective and filter many S tuples
- Can be added to any join algorithm above
Join – Comments and Q/A

- How will the join algorithms change in parallel system?
- Is simple hash better since modern systems have large memories?
- Is the assumption $|M| > \sqrt{|S|}$ realistic?
- How to select a good hash function?
- Babb arrays used in practice?
- How do new storage devices (e.g., PM, SSD, tiered memory) change the story?
- Difficult to understand math.
- Lack of experiments.
In some modern in-memory DBMSs, the entire database can fit in memory. In such a system, can similar optimizations be applied to on-chip SRAM caches vs. DRAM? What are the key challenges compared to a DRAM vs. Disk setting?
Submit review for

Peter Boncz, et al., Database Architecture Optimized for the new Bottleneck: Memory Access. VLDB, 1999