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

Xiangyao Yu
9/9/2020
Announcements

Piazza

Sample reviews and exam questions

Lectures after the exam: state-of-the-art research in database systems

Email me if you have problems submitting the review
Today’s Paper: Join

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 assumptions
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
  • Avoids sync complexity
  • Could be built on systems of the day

Memory
  • Tens of Megabytes

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 $|R| \times F$ blocks
Join Algorithms
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

Unsorted R and S
Sorted runs of 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

Unsorted R and S
Sorted runs of R and S
Find matches in sorted runs
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

Memory layout in Phase 1

- Priority queue (heap)
- Memory
- Input buffer
- Output buffer
Naïve solution:
- Load \(| M | \) blocks
- Sort
- Output \(| M | \) blocks

Each run contains \(| M | \) blocks
Sort Merge Join – Replacement Selection

Replacement selection:
• load | M | I blocks and sort

While heap is not empty
  Output one tuple and load one tuple from input buffer
  If the new tuple < any tuple in output
    save the tuple for next run (heap size reduces)
  else
    heap reorder
Replacement selection:
• load |M| blocks and sort

While heap is not empty
  Output one tuple and load one tuple from input buffer
  If the new tuple < any tuple in output
    save the tuple for next run (heap size reduces)
  else
    heap reorder

Each run contains $2 \times |M| \times |M|$ blocks

Replacement selection:
• load | M | blocks and sort

While heap is not empty
- Output one tuple and load one tuple from input buffer
- If the new tuple < any tuple in output
  - save the tuple for next run (heap size reduces)
- else
  - heap reorder

Each run contains $2 \times |M| \times |M|$ blocks


Total number of runs

$$\leq \frac{|S|}{2 \times |M|} + \frac{|R|}{2 \times |M|} \leq \frac{|S|}{|M|}$$
Sort Merge Join – Phase 2

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

- One input buffer required for each run

Memory layout in Phase 2

<table>
<thead>
<tr>
<th>in-buf</th>
<th>in-buf</th>
<th>...</th>
<th>in-buf</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0$</td>
<td>$R_1$</td>
<td></td>
<td>$R_n$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>$S_1$</td>
<td></td>
<td>$S_m$</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 )
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 = $|R| \times F$)

write back to disk

Memory

$S$
Simple Hash Join – 1\textsuperscript{st} 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 – 1\textsuperscript{st} pass

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

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

Memory

write back to disk

write back to disk
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

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

\[
\text{Memory}
\]

\[
\text{S}
\]

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

\[
\text{write back to disk}
\]
Simple Hash Join – 3\textsuperscript{rd} pass

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

Hash table on $\mathbf{R}$
(size = $|\mathbf{R}| \times F$)
GRACE Hash Join

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

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

Memory layout in Phase 1
GRACE Hash Join

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

Phase 2: Separately join each pairs of partitions

Memory layout in Phase 2
GRACE Hash Join

Assume $k$ partitions for $R$ and $S$

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

$$k \leq |M|$$
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 \( R \) 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

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

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

\[ |R| \leq \frac{|M|}{F} k \leq \frac{|M|^2}{F} \]

\[ |M| \geq \sqrt{|R| \times F} \]
GRACE vs. Simple Hash Join

When $|R| \times |F| < |M|$:
- Simple hash join incurs no IO traffic
- GRACE hash join writes and reads each table (i.e., the partitions) once

When $|R| \times |F| \gg |M|$
- Simple hash join incurs significant IO traffic
- GRACE hash join writes and reads each table (i.e., the partitions) once
Hybrid Hash Join

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

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

<table>
<thead>
<tr>
<th>Memory layout in Phase 1 of GRACE hash join</th>
</tr>
</thead>
<tbody>
<tr>
<td>out-buf R₀</td>
</tr>
</tbody>
</table>
Hybrid Hash Join

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

Memory layout in Phase 1 of hybrid hash join

<table>
<thead>
<tr>
<th>out-buf</th>
<th>out-buf</th>
<th>...</th>
<th>out-buf</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>R₂</td>
<td>...</td>
<td>Rₖ</td>
</tr>
</tbody>
</table>

Hash table for R₀
Hybrid Hash Join

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

Case 1: $|R| \times F < |M|$
Identical to simple hash join

Memory layout in Phase 1 of hybrid hash join
Hybrid Hash Join

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

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

Identical to simple hash join

Case 2: $|R| \times F \gg |M|$

Similar to GRACE hash join

Memory layout in Phase 1 of hybrid hash join
Evaluation

• 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
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)
- Set a bit for each R tuple
- Use to filter S during initial scan, discard tuple if missing in array

Semijoin
- 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
Why \( \sqrt{325 \text{ MB}} \) is 4 MB?
  - \( \sqrt{325\text{MB} / \text{block}_\text{size}} = 4 \text{ MB} / \text{block}_\text{size} \)

Modern systems using Babb array?
Join in in-memory database?
Evaluation on real, parallel systems?
Babb filter vs. Bloom filter
Is it possible to make GRACE hash join work when $|M| < \sqrt{|R| \times F}$? For example, $|M| = 10$, $F = 1$, $|R| = 1000$. You may modify the GRACE hash join algorithm as described in the paper.

Is it possible for a sort-merge join algorithm to outperform a hash-based join algorithm? If yes, when can this happen?
Before Next Lecture

Submit discussion summary to https://wisc-cs764-f20.hotcrp.com
  • Title: Lecture 2 discussion. group ##
  • Authors: Names of students who joined the discussion

Deadline: Thursday 11:59pm

Submit review for