Join Processing

Instructor: Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

Reference
Leonard Shapiro: Join Processing in Database Systems with Large Main Memories, ACM TODS 11(3), Sept. 1986

Overview
- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join
- Performance trade-offs
What is Join Processing?

- Combine two relations
- Most common: equijoin
e.g., person.car_id=car.id
- Goal: Maximize performance
  - Some methods require additional indexing, or are only useful on indexed fields

Example #1

- Customer (c_id, c_name)
- Purchase (c_id, item_id, item_price)

SELECT c_name, item_price
FROM Customer C JOIN Purchase P
ON C.c_id = P.c_id
WHERE P.item_id = 1

Assume |C| \(\cong\) 10,000; |P| \(\cong\) 1,000,000

Example #1 - Data

<table>
<thead>
<tr>
<th>Customer</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_id</td>
<td>c_name</td>
</tr>
<tr>
<td>1</td>
<td>Bob Smith</td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
</tr>
<tr>
<td>3</td>
<td>Sara Dane</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>item_id</th>
<th>item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>3.49</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>2.50</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>3.49</td>
</tr>
<tr>
<td>2</td>
<td>1012</td>
<td>9.79</td>
</tr>
<tr>
<td>3</td>
<td>1001</td>
<td>2.79</td>
</tr>
<tr>
<td></td>
<td>1012</td>
<td>9.69</td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>1012</td>
<td>9.79</td>
</tr>
</tbody>
</table>
Example #1 - Joined

Joining Customer.c_id = Purchase.c_id

<table>
<thead>
<tr>
<th>Customer.c_id</th>
<th>c_name</th>
<th>Purchase.c_id</th>
<th>item_id</th>
<th>item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob Smith</td>
<td>1</td>
<td>1001</td>
<td>2.79</td>
</tr>
<tr>
<td>1</td>
<td>Bob Smith</td>
<td>1</td>
<td>1012</td>
<td>9.49</td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
<td>2</td>
<td>1000</td>
<td>3.49</td>
</tr>
<tr>
<td>3</td>
<td>Sara Dunn</td>
<td>3</td>
<td>1001</td>
<td>2.79</td>
</tr>
</tbody>
</table>

Example #2

- Employee (e_id, name, mgr_id)

SELECT E1.name, E2.name
FROM Employee E1 JOIN Employee E2
ON E1.e_id = E2.mgr_id

- Assume |Employee| ≈ 2000

Example #2 - Data

<table>
<thead>
<tr>
<th>e_id</th>
<th>Name</th>
<th>mgr_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sally</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Sandy</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Brandon</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Brian</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example #2 - Joined

Employee E1, Manager E2

<table>
<thead>
<tr>
<th>E1.Name</th>
<th>E2.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Brian</td>
</tr>
<tr>
<td>Sandy</td>
<td>Sally</td>
</tr>
<tr>
<td>Brandon</td>
<td>Bob</td>
</tr>
<tr>
<td>[-]</td>
<td>[-]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>mgr_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sally</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Sandy</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>John</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Brandon</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Brian</td>
<td>6</td>
</tr>
</tbody>
</table>

Traditional Join Algorithms

- Nested Loops/Index Nested Loops
- Sort Merge
- Hashing Algorithms
- Hash-Join
  - GRACE
  - Hybrid
  - Adaptive
- In general: Outer relation smaller than Inner

Symbols + assumptions

- \(|M|\): # pages/buffers in memory
- \(|R|, |S|\): #pages of ‘R’ and ‘S’
- Assume that \(|R| < |S|\)
Nested Loops & co.

- For each row of outer R, scan inner S for matching rows
- If join column of S is indexed, row lookups via index
- Improvement: **Block nested-loop** join (S scanned once for every block of R)
- Essentially $O(n^2)$

---

Nested Loops & co.

<table>
<thead>
<tr>
<th>c_id</th>
<th>c_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob Smith</td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
</tr>
<tr>
<td>3</td>
<td>Sara Dane</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item_id</th>
<th>item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2.50</td>
</tr>
<tr>
<td>1012</td>
<td>0.79</td>
</tr>
<tr>
<td>100</td>
<td>3.49</td>
</tr>
<tr>
<td>101</td>
<td>3.79</td>
</tr>
</tbody>
</table>

Option Index

---

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join
  - Performance trade-offs
Sorting

Q: how to sort ('external sorting')?

A:
- create sorted runs of size |M|=B
- merge

Memory  R
1
B

Sorting
- create sorted runs of size |M|
- merge first |M|-1 runs into a sorted run of (|M|-1)*|M|,

External Merge Sort
- To sort a relation R with |R|=N pages using |M|=B buffer pages:
  - Pass 0: use B buffer pages. Produce \( \lceil N/B \rceil \) sorted runs of B pages each.
  - Pass 1, 2, ..., etc.: merge B-1 runs.

Cost of External Merge Sort
- How many passes do we need?
- What is the overall cost?
Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_2 \left( \frac{N}{B} \right) \rceil$
- Cost = $2N \ast$ (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\left\lceil \frac{108}{5} \right\rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\left\lceil \frac{22}{4} \right\rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Creating sorted runs

- Quicksort is a fast way to sort in memory.
- Alternative? (what is the objective about number/length of sorted runs?)

Internal Sort Algorithm

- Keep heap $H$ in memory
  - read $B-2$ pages of records, inserting into $H$;
  - while (records left) {
    - $m = H.$removeMinUnmarked (); append $m$ to output buffer;
    - if ($m$ is NULL) => all entries in $H$ are marked {
      - $H.$unmark(all);
      - start new output run;
    } else {
      - read in new record $r$ (use 1 buffer for input pages);
      - $H.$insert($r$ at $m$'s position);
      - if ($r < m$) $H.$mark($r$);
    }
  }
More on Heapsort

- Fact: average length of a run in heapsort is $2(B-2)$
  - The "snowplow" analogy
- Worst-Case:
  - What is min length of a run?
  - How does this arise?
- Best-Case:
  - What is max length of a run?
  - How does this arise?
- Quicksort is faster, but ... longer runs often means fewer passes!

Sort Merge Join

- (external) sort both R and S
- Scan through each sequentially and output results
- Notice: Must “backup” S if R key not unique
- Essentially $O(n \log n)$

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_id</td>
<td>c_name</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Bob Smith</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sara Dane</td>
<td></td>
</tr>
<tr>
<td>Sorted</td>
<td>Sorted</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_id</th>
<th>Item_id</th>
<th>Item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>3.49</td>
<td>2.50</td>
</tr>
<tr>
<td>1001</td>
<td>9.79</td>
<td>3.49</td>
</tr>
<tr>
<td>1012</td>
<td>9.49</td>
<td>2.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_id</th>
<th>Item_id</th>
<th>Item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.79</td>
<td>1001</td>
<td>9.49</td>
</tr>
<tr>
<td>3.49</td>
<td>1000</td>
<td>9.79</td>
</tr>
<tr>
<td>2.50</td>
<td>1000</td>
<td>3.49</td>
</tr>
</tbody>
</table>
Sort Merge Join

- Cost: Assuming sqrt(|S|) buffers, only two passes - I/O cost:
  - |R|+|S| to read each in
  - |R|+|S| to write runs
  - |R|+|S| to read runs and merge
  - minus some I/O, if more memory exists

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join
  - Performance trade-offs

Hash Joins

- Hash joins important in today’s DB
- Good when no sorted index on R and no output sorting on join attribute
- Performs well in DSS situations (both R and S large)
- Good in low memory situations
Hash Join – “Classic”

- Create hash table on join attribute of R
- Sequentially scan S and lookup in table
- Best when the hash table fits in main memory

<table>
<thead>
<tr>
<th>c_id</th>
<th>c_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob Smith</td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
</tr>
<tr>
<td>3</td>
<td>Sara Dane</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item_id</th>
<th>item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>3.49</td>
</tr>
<tr>
<td>1002</td>
<td>2.79</td>
</tr>
<tr>
<td>1012</td>
<td>9.49</td>
</tr>
</tbody>
</table>

Build Table R

S

Best when the hash table fits in main memory
Hash Join

- Low memory approach, common to upcoming methods:
  - Partition relations into subsets with hash function $h_1();$ use $h_2()$ for memory h.t. (or, use one h.f., and choose subsets of its values for $h_1()$
  - Join each subset

Equivalently [Shapiro]: ONE only h.f., whose buckets we group in 'partitions'

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
    - Hash-join: "classic", "simple", GRACE, hybrid
  - Performance trade-offs
Hash join - “Simple”

If |M| < |R| (<|S|), what would you do? (say, half of R fits in memory - how will "classic" go?)

1. choose h.f. $h_1()$ so that each bucket of R fits in memory
2. scan R; keep the contents of first bucket in memory; write out the rest on a new file
3. hash the contents of first bucket, using $h_2()$, again, in memory
4. Scan S; for each tuple, probe, or write-out

Identical to "classic" if R fits in memory
performs well if R almost fits in memory
but poorly otherwise (too many passes)

Q1: cost?
Q2: how would you fix it?
Hash join - “Simple”

- Cost: say we need \( A \) pieces for ‘R’ (\( A = |R| / |M| \))
- each piece requires one more pass over R and S - \( i \)-th pass needs
  - \((A - i)\) pieces of R read; \(A - i - 1\) written
  - similarly for S

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join: "classic", "simple", GRACE, hybrid
  - Performance trade-offs

Hash Join – GRACE

- Better than simple hashing, esp. when little memory is available
- Requires partitioning
- Scan once for partitioning, once for hash/join
Hash Join - GRACE

- hash 'R' into (1, ..., 'max') partitions
- hash 'S' into partitions (same hash function)
- join each pair of matching partitions

Hash Join - GRACE

- how to join each pair of partitions Hr-i, Hs-i?
- A: build another hash table for Hr-i, and probe it with each tuple of Hs-i

Hash Join - GRACE

- How many partition 'max' to use?
Hash Join - GRACE

- How many partitions ‘max’ to use?
  - A: \( \sqrt{|R|} \) (as in sorting, so that we have 2 passes - assume that \( \sqrt{|R|} \) fits in memory)

![Diagram of Hash Join]

Hash join - GRACE

- In more detail:
  - Choose hash function \( h1() \) to partition \( R \) into \( \sqrt{|R|} \) subsets
  - Scan \( R \), then \( S \), placing into output buffer. When full, flush buffer to disk.

![Hash Join Example]

Hash Join – GRACE

- Choose hash function \( h1() \) to partition \( R \) into \( \sqrt{|R|} \) subsets
- Scan \( R \), then \( S \), placing into output buffer. When full, flush buffer to disk.

![Hash Join Example]
Hash Join – GRACE

- For each R buffer, read from disk, make hash table (use $h2()$) (assume it fits in memory)
- Read S buffer, lookup each tuple in hash, output on match

Hash Join - GRACE

- I/O cost?

- A:
  - $|R| + |S|$ to partition
  - $|R| + |S|$ to join
  - minus some, if extra memory is available
  - Minimum two passes – what if R fits in memory?
Hash joins

- "simple" is good for large memories;
- GRACE, for small memories
- Q: can we get the best of both worlds? How?

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join: "classic", "simple", GRACE, hybrid
- Performance trade-offs

Hash Join – Hybrid

- High-level description: Like "simple", but hash the passed-over tuples of 'R'
- In high-memory situations, identical to simple hashing
- Hash one buffer in main memory (|M| - B)
- Use remaining B for partition buffers
Hash Join – Hybrid

- Select B: |M|-B blocks are needed to create a hash table for each partition
- B partitions
- Read R. If tuple hashes to first partition, add to hash table.
- Else, add to appropriate output buffer.

Memory Layout

<table>
<thead>
<tr>
<th>Hash Table – One Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers – One per partition (B partitions)</td>
</tr>
</tbody>
</table>

Hash Join – Hybrid

- Flush R buffers.
- Read S. If tuple hashes to first partition, check against hash table and output if joins.
- Else, add to output buffer.
- Repeat for all partitions (read partition from disk/from memory, build hash table, join.)

Memory Layout

<table>
<thead>
<tr>
<th>Hash Table – One Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers – One per partition (B partitions)</td>
</tr>
</tbody>
</table>

Hash Join - Hybrid

- Hybrid performs best, within certain bounds of memory availability – need enough room for hash table, obviously
- I/O Cost?
Hash Join - Hybrid

- Cost?
- A:
  - \((|R|+|S|) \times (1-q)\) write from the B output buffers
  - \((|R|+|S|)(1-q)\) to read them back
  - \(q = |R_0| / |R|\) percentage of 'privileged' first bucket
  - \(q \rightarrow 0\): two passes, like GRACE
  - \(q=1\): 'zero' cost, like "simple"

Shapiro – Performance Notes

| R | 20MB |
| S | 40MB |

- a) Sort-Merge
- b) Simple Hash
- c) GRACE
- d) Hybrid

Overview

- Join Processing – what is it?
- Traditional Join Algorithms
- Performance trade-offs
  - Partition overflow
  - Conclusions
Partition Overflow

- Observation: we need only worry about 'R'
- two types of overflow:
  - disk
  - memory

Partition Overflow to disk

- Hybrid/GRACE: If a partition on disk \( R_i \) is too large to fit in memory, repartition (as well as \( S_i \))

Partition Overflow to memory

- Problem: in-memory hash table might be too big - what to do?
- Simple: hash buckets may need to be reverted to "passed over" and written to the file on disk
- Hybrid: re-assign some buckets of \( R_0 \) to some other partition (say, \( R_i, R_j \) etc)
Memory management strategies

- So far we assumed that $|M|$ pages are dedicated to us
- unrealistic: instead, ‘hot-set’+VM model
- Details: see Shapiro for how to modify ‘hybrid’ and sort-merge.

Adaptive Hash Join

- Works well when static memory allocation is not available
- Based upon available memory, algorithm tunes:
  - Number of memory-resident buckets
  - Size of buckets (splitting/merging)
  - Buffer size of temp file reads/writes
- Implemented in Tandem NonStop SQL
- Better performance than simple hashing

Dynamic Memory Hybrid Hash Join

- Combines GRACE simplicity with memory-consciousness of Adaptive
- Begin hashing $R$ relation into buckets
- When we run out of memory (or memory shrinks)
  - select a target, write all bucket contents to disk
  - save one page as an output buffer
  - (flush output buffer as full)
- As needed, bring buckets back from disk, hash, and probe $S$ relation
Overview

- Join Processing – what is it?
- Traditional Join Algorithms
  - Nested-loops
  - Sort-merge
  - Hash-join
- Performance trade-offs and tools

Other tools

- Babb arrays (hashed bit-maps)
- Semijoins (similar philosophy: quickly detect non-matching tuples)

Conclusions

- Various join processing algorithms
  - Nested-loops; Sort-merge; Hash-join
  - Hybrid hash: best of hashing ones
  - Performance related to replacement strategies