
Join Processing

Christos Faloutsos
www.cs.cmu.edu/~christos

Roadmap
1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
   OS
   R-trees
   z-ordering
   buffering
   q-opt: joins
   system R q-opt
3) Transactions: locking, recovery
4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort

Example #1

1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
   OS
   R-trees
   z-ordering
   buffering
   q-opt: joins
   system R q-opt
3) Transactions: locking, recovery
4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort

Example #1

- 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

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?

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</td>
</tr>
<tr>
<td>2</td>
<td>Jane</td>
</tr>
<tr>
<td>3</td>
<td>Sally</td>
</tr>
<tr>
<td>4</td>
<td>Sandy</td>
</tr>
</tbody>
</table>

Example #1 - Joined

Joining Customer.c_id = Purchase.c_id

<table>
<thead>
<tr>
<th>Customer.c_id</th>
<th>Customer.c_name</th>
<th>Purchase.e_id</th>
<th>Purchase.c_id</th>
<th>Purchase.c_name</th>
<th>item_id</th>
<th>item_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob Smith</td>
<td>0400</td>
<td>1</td>
<td>Bob Smith</td>
<td>0001</td>
<td>5.40</td>
</tr>
<tr>
<td>2</td>
<td>Jane Smith</td>
<td>0012</td>
<td>3</td>
<td>Sandy</td>
<td>0012</td>
<td>9.70</td>
</tr>
<tr>
<td>3</td>
<td>Sally</td>
<td>0000</td>
<td>1</td>
<td>Bob Smith</td>
<td>0001</td>
<td>5.40</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>Employee</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_id</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>Bob</td>
</tr>
<tr>
<td>2</td>
<td>Sally</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Sandy</td>
<td>Sandy</td>
</tr>
<tr>
<td>4</td>
<td>John</td>
<td>John</td>
</tr>
<tr>
<td>5</td>
<td>Brandon</td>
<td>Brandon</td>
</tr>
<tr>
<td>6</td>
<td>Brian</td>
<td>Brian</td>
</tr>
</tbody>
</table>

Example #2 - Joined

Employee E1, Manager E2

<table>
<thead>
<tr>
<th>Employee</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_id</td>
<td>Name</td>
<td>mgr_id</td>
</tr>
<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>3</td>
</tr>
<tr>
<td>5</td>
<td>Brandon</td>
<td>4</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>R</th>
<th>S</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 Doe</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Overview

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

Sorting

- Assume br=|R| blocks of rel. ‘R’, and
- only M (cbr) buffers in main memory
- Q: how to sort (‘external sorting’)?

Q: how to sort (‘external sorting’)?

A:
- create sorted runs of size M
- merge
Sorting

- create sorted runs of size M (how many?)
- merge them (how?)

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)

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

Hash Join – “Classic”

- Perform well in DSS situations (both R and S not necessarily sorted)
- Hash joins important in today’s DB

Hash Join

- Low memory approach, common to upcoming methods:
  - Partition relations into subsets with hash function h1(); use h2() for memory h.f. (or, use one h.f., and choose subsets of its values for h1())
  - Join each subset
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 $h_1()$
  - 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?)

Hash join - “Simple”

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

Hash join - “Simple”

- 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
- Twice as much work when memory is large
- 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?
- A: sqrt(|R|) (as in sorting, so that we have 2 passes - assume that sqrt(|R|) fits in memory)
Hash join - GRACE

In more detail:

Hash Join – GRACE

Choose hash function \( h \) to partition \( R \) into \( |F \cup R| \) subsets

Scan \( R \), then \( S \), placing into output buffer. When full, flush buffer to disk.

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

I/O cost?

A:

\(- |R| + |S| \) to partition

\(- |R| + |S| \) to join

\(- \text{minus some, if extra memory is available} \)

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 a single buffer in main memory (IMi-B)
- Use remaining B for partition buffers

Hash Join – Hybrid

- Select B: IMi-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

| Buffers – One per partition (B partitions) | Hash Table – One Partition |

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:
  - \((IRi+SI) \times (1-q)\) write from the B output buffers
  - \((IRi+SI)/(1-q)\) to read them back
  - \(q = |R0|/|R|\) percentage of ‘privileged’ first bucket
  - \(q=1\): ‘zero’ cost, like “simple”
  - \(q \to 0\): two passes, like GRACE
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 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
- see [Shapiro] for how to modify ‘hybrid’ and sort-merge.
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