Evaluating Relational Operators: Part II

Relational Operators

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Example

```
SELECT  *
FROM    Reserves R, Sailor S,
WHERE   R.sid = S.sid
```

- No indices on Sailor or Reserves

Tuple Nested Loop Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if r.sid == s.sid then add <r, s> to result
```

- R is “outer” relation
- S is “inner” relation

Analysis

- Assume
  - M pages in R, p_R tuples per page
    - M = 1000, p_R = 100
  - N pages in S, p_S tuples per page Select
    - N = 500, p_S = 80
- Total cost = $M + p_R * M * N$
  - Ignore cost of writing out result
  - Same for all join methods
- Main problem: depends on # tuples per page

Page Nested Loop Join

```
foreach page p1 in R do
    foreach page p2 in S do
        foreach r in p1 do
            foreach s in p2 do
                if r.sid == s.sid then add <r, s> to result
```

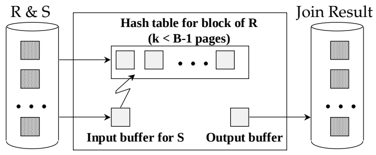
- R is “outer” relation
- S is “inner” relation

Analysis

- Assume
  - M pages in R, p_R tuples per page
    - M = 1000, p_R = 100
  - N pages in S, p_S tuples per page Select
    - N = 500, p_S = 80
- Total cost = M + M * N
- Note: Smaller relation should be "outer"
  - Better for S to be "outer" in this case!
- Main problem: does not use all buffer pages

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \# \ of \ pages \ of \ outer \ / \ blocksize \rceil$
- With Reserves (R) as outer, and 100 page blocks:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

Example

```
SELECT *
FROM   Reserves R, Sailor S,
WHERE  R.sid = S.sid
```

- Hash index on Sailor.sid

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- Cost of finding matching tuples depends on type of index
  - B+-tree or hash
  - Clustered or unclustered

Example

```
SELECT *
FROM   Reserves R, Sailor S,
WHERE  R.sid > S.sid
```

- B+-tree index on Sailor.sid

Example

SELECT *
FROM Reserves R, Sailor S,
WHERE R.sid = S.sid

- No indices on Sailor or Reserves

Sort-Merge Join

- Sort R on the join attributes
- Sort S on the join attributes
- Merge sorted relations to produce join result
  - Advance r in R until r.sid >= s.sid
  - Advance s in S until s.sid >= r.sid
  - If r.sid = s.sid
    - All R tuples with same value as r.sid is *current R group*
    - All S tuples with same value as s.sid is *current S group*
    - Output all <rg, sg> pairs, where rg is in current R group, sg is in current S group
  - Repeat

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Analysis

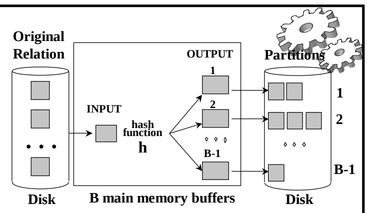
- Assume
  - M pages in R, p_R tuples per page
  - N pages in S, p_S tuples per page Select
- Total cost = M log M + N log N + (M + N)
- Note: (M + N) could be (M * N) in worst case
  - Unlikely!
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes
  - Total join cost: 7500
  - Equivalent BNL cost: 2500 to 15000

Refinement of Sort-Merge Join

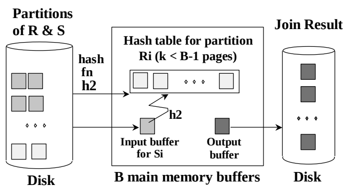
- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - Assume B > $\sqrt{L}$, where *L* is the size of *larger* relation
  - Use refinement that produces runs of length 2B in Phase 1
  - #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation (only) in merging pass = 3 (M + N)
  - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Analysis (without recursive paritioning)

- Assumptions
  - # partitions = B –1
  - B-2 > size of largest partition (to avoid partitioning again)
- Required memory
  - M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$
  - M corresponds to *smaller* relation
- In partitioning phase, read+write both relns: 2(M+N)
- In matching phase, read both relns: M+N
- Total cost = 3 (M + N)
- In our running example, this is a total of 4500 I/Os

Hash-Join vs. Sort-Merge Join

- Given a minimum amount of memory, both have cost of 3 (M + N)
- Benefits of hash join
  - Superior if relation sizes differ greatly
  - Highly parallelizable
- Sort merge join
  - Less sensitive to data skew
  - Result is sorted

General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

Relational Operators

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar; we'll do union.
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0 for *both* relations.
- Hash based approach to union:
  - Partition R and S using hash function *h*.
  - For each S-partition, build in-memory hash table (using *h2*), scan corr. R-partition and add tuples to table while discarding duplicates.

Relational Operators

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Example

SELECT MAX(S.age)
FROM Sailor S

- Sequential scan
- Index-only scan (given index on age)

Example

SELECT MAX(S.age)
FROM Sailor S
GROUP BY S.rating

- Sort on rating, then aggregate
- Hash on rating, then aggregate
- Index-only scan (given B+ tree index on rating, age)