Evaluation of Relational Operations

Chapter 12, Part A

Overview of Query Optimization

- *Input: Sql query*
- *Output: Query Plan: Tree of Relational algebra operators, with choice of algorithm for each operator*
- Main issues:
  - For a given query, what plans are generated/considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
    - Using the cost formulas studied so far + assumptions
- Ideally: Want to find best plan.
- Practically: Avoid worst plans!
- We will study the System R approach.

Why System R Optimizer

- Most widely used currently; works well for < 10 joins.
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, are used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Types of Qeps

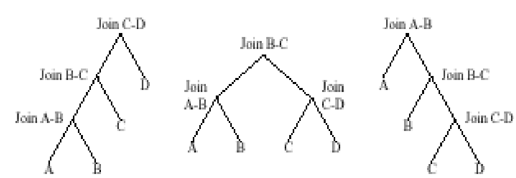


Figure 4. Left-Deep, Bushy, and Right-Deep Plans.

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Assumption: 4K page size

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: $R \bowtie S$. Common! Must be carefully optimized.
- $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs. We will also ignore cpu costs.

Simple Nested Loops Join

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

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + p_R * M * N = 1000 + 100*1000*500$ I/Os.
  - If 5 msec is the access time per page (and each tuple accesses a new page), the time taken is
    - 50,000,000 * 5/1000 which is 250,000 secs or 69 hours
  - Assumption of retrieving N pages for each of M* p_R tuples is not realistic

Page-oriented Nested Loops Join

- For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N = 1000 + 1000*500 = 501000
  - Time taken is: .69 hours
- Which relation should be chosen as outer/inner?
  - S outer, and R inner
  - Cost: N + N*M = 500 + 500*1000 = 500500
  - Time taken is: ~.69 hours

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)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.
  - Incremental (join only the tuples that are of interest)

Examples of Index Nested Loops

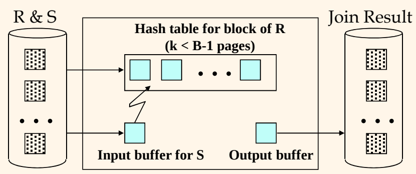
- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 221,000 I/Os.
  - M + M*Pr*1.2 +M*Pr*1
  - 1000 + 120,000 + 100,000
  - 0.3 hours (contrast this with 69 and .69 hours)

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/O's depending on whether the index is clustered or not.
  - 500 + 40,000*1.2 + 40,000*1 (assuming 1 I/O, clustered)
  - .13 hours -- clustered
  - 500 + 40,000*1.2 + 100,000 (assuming 2.5 I/O, not clustered)
  - or .2 hours -- unclustered
- Even with unclustered index, the cost is likely to be less (than simple nested loop join) if the number of matching tuples is small

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 = $\lceil \textit{\# of pages of outer / blocksize} \rceil$
  - Cost = M + $\lceil$M/(B-2)$\rceil$ * N
- If one relation can fit in the buffer, then the inner relation needs to be scanned only ONCE!
- The cost becomes M + N (Optimal!!!)
- If neither of the relations fit entirely in the buffer, we need to allocate the buffers judiciously.

Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \textit{\# of pages of outer / blocksize} \rceil$
- With Reserves (R) as outer, and 100 pages for R:
  - Assuming 102 buffer pages (100+1+1)
  - 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.
  - Cost = M + $\lceil$M/b$\rceil$*N
  - For the above example, it is
  - 1000 + 10*500 = 6000; That is 30 secs (assuming 5 msec per I/0)
- If 54 buffer pages (52+1+1), we would scan S 20 times
  - Cost: 1000 + 20*500 = 11000; that is 55 secs

Examples of Block Nested Loops

- 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.
  - Cost: 500 + 5*1000 = 5500
  - That is 27.5 secs
- With *sequential reads (blocked)* considered, analysis changes: may be best to divide buffers evenly between R and S.
- Double buffering can also be used.

Summary (join operation)

- If we can hold the smaller relation in memory + 2 buffers; cost = M + N I/Os (optimal)
- R as outer relation and S as inner; B buffers
  - Cost = M + ⌈M/(B-2)⌉*N
  - If M >>N, pick ***smaller*** as outer
  - When M=B–2, pick N as outer; only one scan of M
  - e.g., if we has 502 buffers, allocate 500 to S as outer, 1 to R as inner and 1 to output
  - Cost: 500 + 1*1000 = 500 + 1000 = 1500 (minimal cost)

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer!). Depends upon buffer management policy!!

Example of Sort-Merge Join

R

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

S

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

- Cost: Sort(R1) + sort(R2) + Merge (R1, R2)
  M log M + N log N + (M+N)
  - The cost of merging, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(*BNL cost: 2500 to 15000 I/Os*)

Sort-Merge Join

- With 35 buffers, we can sort both relations in 2 passes
  the cost of sort-merge is 7500,
  where as the cost of BNL is more than 15000
- With 100 buffers, we can sort both relations in 2 passes
  Cost of sort-merge: still 7500
  where as the cost of BNL is: 6500
- With 300 buffers,
  the cost of sort-merge is still 7500
  where as the cost of BNL drops to 2500.

- The number of buffers available makes a difference !!
- The worst case scenario is O(M*N) I/Os

Sort-Merge join Examples

- Assume 35 buffers.
- Sort of R (1000 pages)
  - Pass 0: 29 runs of 35 buffers each=2*1000 I/O's
  - Pass 1: Sort complete=2*1000 I/O's (29-way merge join)
  - Total: 4000 I/O's
- Sort of S (500 pages)
  - Pass 0: 15 runs of 35 buffers each;
  - Pass 1: 15-way join using 35 buffers
  - Total: 2*2*500= 2000 I/O's
- Merge= 1000+500= 1500 I/O's
- Total cost: 4000 + 2000 + 1500 = 7500

Sort-Merge join Examples

- Assume 100 buffers.
- Sort of R
  - Pass 0: 10 runs of 100 buffers each=2*1000 I/O's
  - Pass 1: Sort complete=2*1000 I/O's
  - Total: 4000 I/O's
- Sort of S
  - Pass 0: 5 runs of 100 buffers each;
  - Pass 1: 5-way join using 100 buffers
  - Total: 2*2*500= 2000 I/O's
- Merge= 1000+500= 1500 I/O's
- Total cost: 4000 + 2000 + 1500 = 7500

Examples (Contd…)

- With 35 buffers; cost of s-m is 7500
- Block Nested loop join cost is
  - $M + \lceil M/(B-2) \rceil * N$
  - 1000+ 31*500= 16500 I/O's; 1000/33 is 30.30
- With 300 buffers, the cost of sort-merge is still 7500 where as the cost of BNL drops to
  - 1000 + 1000/298 * 500 = 1000 + 4*500 = 3000 or
  - 500 + 500/298 * 1000 = 500 + 2*1000 = 2500
- The number of buffers available makes a difference !!
- The worst case scenario is O(M*N) I/Os

Two-Way External Merge Sort

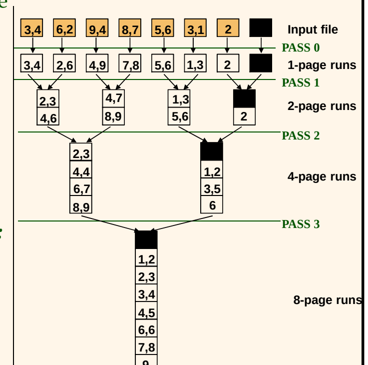
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

$$= \lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- *Idea:* ***Divide and conquer:*** sort subfiles and merge
- Can we improve upon this? How?



Refinement-1 of Sort-Merge Join

- We can combine the merging phase of *sorting* of R and S with the merging phase of join.
  - Let L be the size (in pages) of the larger relation
  - In order to manage L/B runs in pass 1, you need at least
    - L/B + 1 buffers
  - Hence, B > L/B or B^2 > L or B > $\sqrt{L}$
  - If the # of buffers available for the merge phase is 2$\sqrt{L}$, that is, more than the number of runs of R and S
    - We allocate one buffer for each run of R and one for each run of S
    - We then merge the runs of R and S streams as they are generated. we apply the join condition and discard tuples if they do not join.

Refinement-1 of Sort-Merge Join (Contd.)

- Cost: read+write each relation in Pass 0 + (only) read each relation in merging pass (+ writing of result tuples).
  - 3 * (M+N)
- In example, cost goes down from 7500 to 4500 I/Os
  - 3 * (1000+500) = 4500
- In practice, cost of sort-merge join, like the cost of external sorting, can be *linear*.

Refinement-2 of Sort-Merge Join

- This increases the number of buffers required to $\sqrt{2*L}$
- We apply the heapsort optimization to produce runs of size 2*B.
- Hence, we will have L /2*B runs of each relation, given the assumption that we have B buffers.
- Thus the number of buffers is B > L/2*b +1, or
- B > $\sqrt{L/2}$
- Hence we only need B > $\sqrt{2L}$ buffers instead of 2*$\sqrt{L}$ with this optimization.