Evaluating Relational Operators

Linda Wu

(CMPT 354 • 2004-2)

Topics

- Selection operation
- Projection operation
- Set operations
- Aggregate operations
- Join operation
- Impact of buffering

Relational Operations

- We will consider how to implement:
  1. Selection (σ): select a subset of rows from relation
  2. Projection (π): delete unwanted columns from relation
  3. Union (∪): tuples in relation 1 or 2
  4. Set-difference (–): tuples in relation 1 but not in relation 2
  5. Aggregation (SUM, MIN, etc) and GROUP BY
  6. Join (▷◁): combine two relations
     - Intersection(∩) and cross-product (×) are implemented as special cases of join

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

Reserves (*sid*: integer, *bid*: integer, *day*: date, *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

Selection σ

$\sigma_{R.attr\ \mathbf{op}\ value}(R)$

- No index on *R.attr*, *R* is not sorted on *attr*
  - Scan the entire relation *R*
  - Add tuples to the result if the condition is satisfied
- No index on *R.attr*, *R* is sorted on *attr*
  - Binary search to find the first qualifying tuple
  - From there, scan *R* till the condition is no longer satisfied
- B+ tree index on *R.attr*
  - Search the tree to find the first index pointing to a qualifying tuple of R
  - Scan the leaf pages to retrieve all qualifying data entries, and, the corresponding tuples if Alternative (2), (3)
- Hash index on *R.attr*, op is equality
  - Retrieve the correct bucket page in the index
  - Retrieve qualifying tuples from *R*

Selection σ (Cont.)

- If using an index for selections, the I/O cost depends on # of qualifying tuples and clustering
  - Cost of finding qualifying data entries (typically small) + cost of retrieving tuples (could be large without clustering)
  - Example

    SELECT * FROM Reserves R
    WHERE R.rname < 'C%'

    Assuming uniform distribution of names, about 10% of tuples qualify (100 pages; 10,000 tuples)

    If a clustered B+ tree index on *rname*, cost ≈ 100 I/Os; if an unclustered index, up to 10,000 I/Os (even worst than entire file scan!)

Selection σ (Cont.)

- Important refinement for unclustered indexes
  1. Find qualifying data entries
  2. Sort the rid's of the data records to be retrieved by their page-id component
  3. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering)

  Cost of retrieving tuples = # of pages containing qualifying tuples

Selection σ (Cont.)

- General selection conditions
  - A Boolean combination ($\wedge$,$\vee$) of terms
  - Terms have the form:

    attr op *constant*, or, *attr1* op *attr2*
  - Conditions expressed in conjunctive normal form (CNF)
    - A condition is a collection of conjuncts that are connected by $\wedge$
    - A conjunct consists of one or more terms connected by $\vee$
    - A conjunct containing $\vee$ is said to be disjunctive (contain disjunction)

Selection σ (Cont.)

- Selections without disjunction

  Approach 1: find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the access path
  - Most selective access path: an index or file scan that is estimated to require the fewest page I/Os
  - Terms that match the access path reduce # of tuples retrieved; other terms used to discard some retrieved tuples
  - *day<8/9/03 AND bid=5 AND sid=3*

    A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on <*bid, sid*> could be used; *day<8/9/03* must then be checked

Selection σ (Cont.)

Approach 2 (if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries)
- Retrieve the rids of data records using each matching index, then intersect these sets of rids (we'll discuss intersection soon)
- Retrieve the records and apply any remaining terms
- *day<8/9/03 AND bid=5 AND sid=3*

  If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can retrieve rids of records satisfying *day<8/9/03* using the first, rids of records satisfying *sid=3* using the second, intersect, retrieve records and check bid=5

* We don't discuss selections with disjunction

Projection π

$$\pi_{attr1,\ attr2,\ \ldots\ attrn}(R)$$

- Implementing projection
  - Remove unwanted attributes
  - Eliminate duplicates (based on sorting / hashing)

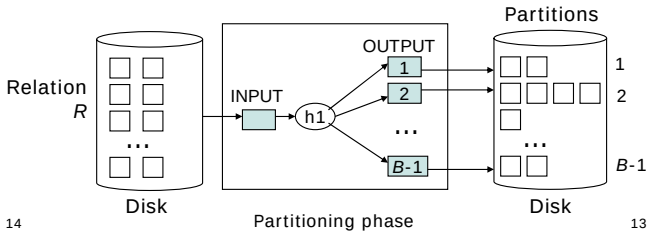
```
SELECT  DISTINCT R.sid, R.bid
FROM    Reserves R
```


Projection π (Cont.)

- Projection based on sorting
  - Modify pass 0 of external merge sort to eliminate unwanted fields
    - Runs of about 2B pages are produced, but tuples in runs are smaller than input tuples (size ratio depends on the total size of dropped fields)
  - Modify merging passes to eliminate duplicates
    - Less result tuples than input (difference depends on # of duplicates)
  - I/O cost
    - In Pass 0, read the original relation (*M* pages), write out the same number of smaller tuples
    - In merging passes, fewer pages are written out in each pass

Projection π (Cont.)

- Projection based on hashing
  - Used when # of buffer pages (B) is much larger than # of pages in relation R
  - Cost: read R for partitioning, write out each tuple with fewer fields (therefore fewer pages); they are read in the next phase, in-memory hash table is written out for each partition



Projection π (Cont.)

- *Phase 1: partitioning*

  Read R using one input buffer; for each tuple, discard unwanted fields, apply hash function *h1* to choose one of B-1 output buffers
  - Result is B-1 partitions (with no unwanted fields)
  - 2 tuples from different partitions must be distinct
- *Phase 2: duplicate elimination in each partition*

  For each partition: read it in, one page at a time; hash its tuples using *h2* ($\neq$ *h1*), then insert tuples into an in-memory hash table, discard duplicates; write hash table to the result file
  - If a tuple hashes to the same value as an existing tuple, compare the two to check whether duplicates

Projection π (Cont.)

- Sort-based approach is the standard
  - Better handling of non-uniformly distributed values
  - Result is sorted
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates

Set Operations

- R $\cap$ S, R $\times$ S, R $\cup$ S, R – S
- Intersection ($\cap$) and cross-product ($\times$) are implemented as special cases of join
  - $\cap$: equality on all fields as the join condition
  - $\times$: no join condition
- Implementation of union ($\cup$) and set-difference (–) are similar

Set Operations (Cont.)

Sort-based approach

- $R \cup S$
  - Sort both R and S (on the combination of all attributes)
  - Scan sorted R and S and merge them, eliminating duplicates
  - Alternative: merge runs from pass 0 for both relations
- $R - S$
  - During merging pass, write only tuples of R to the result, after checking they are not in S

Set Operations (Cont.)

Hash-based approach

- $R \cup S$
  - Partition R and S using hash function h
  - For each S-partition
    - Build in-memory hash table using h2 ($h2 \neq h$)
    - Scan corresponding R-partition, add tuples to the hash table of S-partition if they are not in it; discard them otherwise
    - Write out hash table
- $R - S$
  - For each tuple in R-partition, write it to the result if it is not in the hash table of S-partition

Aggregate Operations

- SUM, AVG, COUNT, MIN, MAX (without grouping)
  - In general, requires scanning the relation and maintaining some running information
  - Given an index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan

Operations	Running information
SUM	Total of the values retrieved
AVG	<total, count> of the values retrieved
COUNT	Count of values retrieved
MIN	Smallest value retrieved
MAX	Largest value retrieved

Aggregate Operations (Cont.)

- With grouping
  - Sorting approach

    Sort on group-by attributes, then scan relation and compute aggregate for each group (watch for group boundary)
  - Similar hashing approach on group-by attributes
- Using index
  - Given an index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan
  - Given a tree index, if group-by attributes form prefix of search key, can retrieve data entries in group-by order: sorting is avoided!

Join Operation

- Nested loops join
- Sort-merge join
- Hash join

Equality Joins With One Join Column

- Join operation R ⋈ S is very common; must be carefully optimized
- Implementing R ⋈ S as R × S followed by a selection is inefficient, since R × S is very large
- Assumption
  - R: M pages, p_R tuples / page
  - S: N pages, p_S tuples / page
  - R ⋈ S: join condition is $R_i = S_j$
- Cost metric: # of I/Os; ignore output costs

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


Simple Nested Loops Join

for each tuple ***r*** in R
  for each tuple ***s*** in S
    if $r_i == s_j$ then add <***r, s***> to result

- For each tuple in the outer relation R, scan the entire inner relation S
  - Cost = M + p_R * M * N
- Refinement: page-oriented Nested Loops join

  For each page of R, retrieve each page of S, and write out <r, s> for all qualifying tuples r ∈ R-page and S ∈ S-page
  - Cost = M + M*N
  - R ⋈ S = S ⋈ R: if smaller relation is outer, cost is smaller

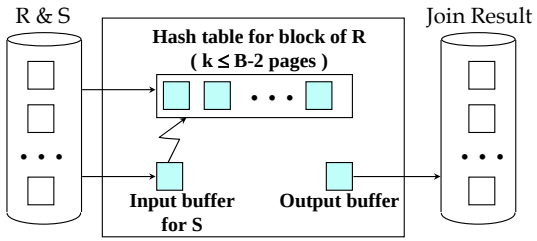
Block Nested Loops Join

for each block of B-2 page of R
  for each page of S
    for all matching in-memory tuples
    r ∈ R-block, ***s*** ∈ S-page
      add <***r, s***> to result

- One page as an input buffer for scanning the inner S, one page as output buffer, and use all remaining B-2 pages to hold block of outer R
  - For each matching tuple r ∈ R-block, s ∈ S-page, add <r, s> to result. Then read next R-block, scan S, etc.
  - R is scanned once; S is scanned multiple times

Block Nested Loops Join (Cont.)

- To find matching pairs of tuples
  - Build a in-memory hash table for the block of R
  - The effective block size of R (# of tuples /block) is reduced



Block Nested Loops Join (Cont.)

- # outer blocks = ⌈# outer pages / block size⌉
  = ⌈M / (B-2)⌉
  (Ignore the extra space for hash table)
- Cost = (scan cost of outer) + (# outer blocks) * (scan cost of inner) = M + N * ⌈M / (B-2)⌉
- Impact of blocked access
  - With blocked access considered, the best approach is to divide buffer pages evenly between R and S
  - More passes over the inner relation S, but seeking time and rotational delay are dramatically reduced

Index Nested Loops Join

for each tuple ***r*** in R
  for each tuple ***s*** in S where $r_i == s_j$
    add <***r***, ***s***> to result

Use index of S

- If there is an index on the join column of one relation (say S): make S the inner and exploit the index
  - Cost = M + (M*p_R * (cost of finding matching S tuples))
- For each R tuple, the cost to probe S index (alt. (2) or (3)) is about 1-2 for hash index, 2-4 for B+ tree; the cost to retrieve matching S tuples depends on clustering
  - Clustered index: 1 I/O (typical); unclustered: up to 1 I/O per matching S tuple

Index Nested Loops Join (Cont.)

- 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 cost = 1000 + (100*1000*2.2) = 221,000 I/Os
- 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/Os depending on whether the index is clustered

Sort-Merge Join

1. Sort R & S on the join column
   - To group all tuples with the same value in the join column
2. Scan R & S to do a "merge" on join column
   a) Scan R, S from 1st tuple; 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 (on join columns)
   b) At this point, all R tuples with same value in R_i (current R group) and all S tuples with same value in S_j (current S group) match; output <r, s> for all pairs of such tuples
   c) Resume scanning R and S

Sort-Merge Join (Cont.)

- R is scanned once; each S group is scanned once per matching R tuple
  - If # pages in a (repeated scanned) S group is small, it is very likely to find buffer pages for it
  - Otherwise, the 1st page of S group may no longer in the buffer pool when it is requested again
- I/O cost = (sorting cost on R, S) + (scan cost)
  = O(M logM) + O(N logN) + (M+N)
  - The scan cost is typically a single scan of each relation (M+N)
  - In the worst case, the scan cost could be M*N (very unlikely)

Sort-Merge Join (Cont.)

- Example: with 35, 100 or 300 buffer pages
  - Both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500
  - Compared with Blocked Nested Loops cost: 2500 ~ 15000 I/Os

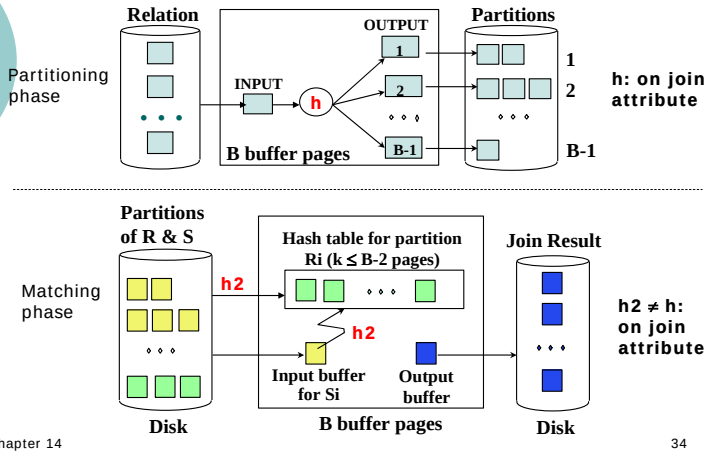
Sort-Merge Join (Cont.)

- Refinement: the merging phases in the sorting of R and S and be combined with the merging required for the join
  - With B > $\sqrt{L}$ (L = size of the larger relation), produce runs of length 2B in pass 0: # of runs of each relation < B/2; allocate 1 page per run of each relation; 2 passes for sorting
  - Merge runs of R, merge runs of S, and merge the resulting sorted R and S streams as they are generated while checking the join condition
  - Cost: read+write each relation in pass 0 + read each relation in the (only) merging pass = 3*(M+N)
  - In the previous example, the cost goes down from 7500 to 4500 I/Os

Hash Join

- Partition both relations using hash function h: R tuples in partition *i* (R_i) will only match S tuples in partition *i* (S_i)
- In practice, build an in-memory hash table to speed up the matching of tuples in R_i and S_i
  - A little more memory is needed
  - Read in a partition of R, hash it using h2 (≠ h): scan matching partition of S, search for matches
- I/O cost
  - In partitioning phase, read+write both relations: 2(M+N) I/Os
  - In matching phase, read both relations: M+N I/Os, if no partition overflow

Hash Join (Cont.)



Hash Join (Cont.)

- To decrease the chance of partition overflow, minimize the size of R partitions by maximizing # of partition
  - # of partitions k ≤ B-1, size of largest partition ≤ B-2. Assume uniformly-sized partitions. To maximize k:

    k= B-1, M/(B-1) ≤ B-2 ⇒ B > $\sqrt{M}$
- If the hash function does not partition R uniformly, one or more R partitions may not fit in memory
  - Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition

Hash Join (Cont.)

- Sort-Merge Join vs. Hash Join
  - Given a minimum amount of memory, both have a cost of 3(M+N) I/Os; Hash Join superior on this count if the relation sizes differ greatly
  - Hash join is shown to be highly parallelizable
  - Sort-Merge join is less sensitive to non-uniformly partitioning; the result is sorted

General Join Conditions

- Equalities over several attributes
  (e.g., R.sid=S.sid AND R.rname=S.sname)
  - Index nested loops join: build index on <R.sid, R.sname>, R is inner; or, use an existing index on sid or sname
  - Sort-merge and hash join: sort/partition on combination of the two join columns
  - Other join algorithms: unaffected

General Join Conditions (Cont.)

- Inequality conditions
  (e.g., R.rname < S.sname)
  - Index nested loops join: need clustered B+ tree index
    - Range probes on the inner; # of matches is likely to be much higher than for equality joins
  - Hash join, sort-merge join: not applicable
  - Other join algorithms: unaffected
- No join algorithm is uniformly superior to the others

Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork
- Repeated access patterns interact with buffer replacement policy
  - The inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold the inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (sequential flooding)
  - The replacement policy does not matter for Block Nested Loops

Summary

- A virtue of relational DBMSs: queries are composed of a few basic operators; the implementation of these operators can be carefully tuned (and it is important to do this!)
- Many alternative implementation techniques for each operator; no universally superior technique for most operators
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several operations