COURSE 8

Evaluation of Relational Operators 2

Distributed Queries

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
  AND E.salary < 7000
```

- Horizontally Fragmented:

Tuples with Salary < 5000 at Shanghai, >= 5000 at Tokyo.

  - Must compute SUM(*age*), COUNT(*age*) at both sites.
  - If WHERE contained just E.*salary*>6000, just one site.

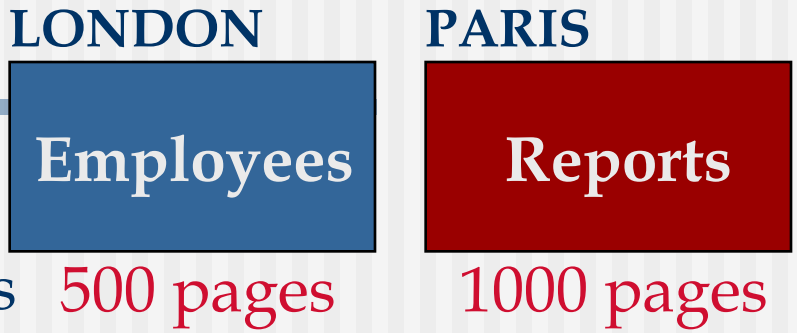
- Vertically Fragmented: *title* and *salary* at Shanghai, *ename* and *age* at Tokyo, *id* at both.

  - Must reconstruct relation by join on *id*, then evaluate the query.

- Replicated: Employees copies at both sites.

  - Choice of site based on local costs, shipping costs.

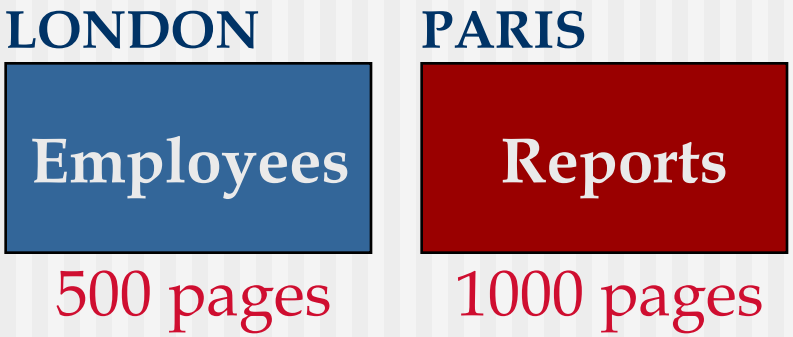
Distributed Joins



for Employees 1 page = 80 tuples
for Reports 1 page = 100 tuples

- Fetch as Needed, Page oriented nested loops, Employees as outer (for each Employee page fetch all Reports pages from Paris):
  - **Cost:** 500 D + 500 * 1000 (D+S)
  - **D** is cost to read/write page; **S** is cost to ship page.
  - If query was not submitted at London, must add cost of shipping result to query site.
  - Can also do INL (indexed nested loops) at London, fetching matching Reports tuples to London as needed.

Distributed Joins



- Ship to One Site: Ship Reports to London.
  - Cost: 1000 S + 4500 D (sort-merge join; cost = 3*(500+1000))
  - If result size is very large, may be better to ship both relations to result site and then join them!
- Ship Employees to Paris
  - Cost: 500 S + 4500 D

Semijoin

- At London, project Employees onto join columns and ship this to Paris.
- At Paris, join Employees projection with Reports.
  - Result is called **reduction** of Reports with respect to Employees .
- Ship reduction of Reports to London.
- At London, join Employees with reduction of Reports.
- Idea: Tradeoff the cost of computing and shipping projection and computing and shipping reduction for cost of shipping full Reports relation.
- Especially useful if there is a selection on Employees, and answer desired at London.

Bloomjoin

- At London, compute a bit-vector of some size k:
  - Hash join column values into range 0 to k-1.
  - If some tuple hashes to *i*, set bit *i* to 1 (*i* from 0 to k-1).
  - Ship bit-vector to Paris.
- At Paris, hash each tuple of Reports similarly, and discard tuples that hash to 0 in Employees bit-vector.
  - Result is called **reduction** of Reports wrt Employees.
- Ship bit-vector reduced Reports to London.
- At London, join Employees with reduced Reports.
- Bit-vector cheaper to ship, almost as effective.

Distributed Query Optimization

- Cost-based approach; consider all plans, pick cheapest; similar to centralized optimization.
  - Difference 1: Communication costs must be considered.
  - Difference 2: Local site autonomy must be respected.
  - Difference 3: New distributed join methods.

- Query site constructs global plan, with suggested local plans describing processing at each site.
  - If a site can improve suggested local plan, free to do so.

Simple Selections

```
SELECT  *
FROM    Students S
WHERE   S.sname < 'C%'
```

- Of the form $\sigma_{R.attr\ OP\ value}(S)$
- Size of result approximated as *size of S * reduction factor*; we will consider how to estimate reduction factors later.
- With no index, unsorted: Must essentially scan the whole relation; cost is N (number of pages in S)
- With no index, sorted: binary search to locate the first tuple satisfying the condition; cost is $Log_2 N$
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records.

Using an Index for Selections

- Cost depends on number of qualifying tuples and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if un-clustered, up to 10000 I/Os!
- *Important refinement for un-clustered indexes*:
  1. Find qualifying data entries.
  2. Sort the rid's of the data records to be retrieved.
  3. Fetch rids in order. This ensures that each data page is looked at just once (though number of such pages likely to be higher than with clustering).

General Selection Conditions

(day<8/9/94 AND grade=10) OR cid=5 OR sid=3

- Such selection conditions are first converted to *conjunctive normal form* (CNF):

  (day<8/9/94 OR cid=5 OR sid=3) AND

  (grade=10 OR cid=5 OR sid=3)

- We only discuss the case with no ORs (a conjunction of *terms* of the form *attr* ***op*** *value*).
- An index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - Index on <*a, b, c*> matches *a=5 AND b= 3*, but not *b=3*.

Approaches to General Selections

- First approach: Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND cid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *cid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on <*cid, sid*> could be used; *day<8/9/94* must then be checked.

Approaches to General Selections (cont)

- Second approach (if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries):
  - Get sets of 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.
  - Consider *day<8/9/94 AND cid=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<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *cid=5*.

The Projection Operation

- Projection query: $\pi_{cid, sid}$Evaluations

```
SELECT   DISTINCT
         E.sid, E.cid
FROM  Evaluations E
```

- To implement projection
  - Remove unwanted attributes
  - Eliminate any duplicate tuples
- Sort-based projection
- Hash-based projection

Projection Based on Sorting

- Step 1 - Scan E to produce a set of tuples containing only the desired attributes
  - Cost = N I/Os to scan E, N is number of pages of E + T I/Os to write temp relation E', T is number of pages of E'
- Step 2 - Sort tuples using combination of attributes as key
  - Cost = $O(T\log_2 T)$
- Step 3 - Scan sorted result, compare adjacent tuples and discard duplicates
  - Cost =T

Projection Based on Sorting - Improved

- Modify Pass 0 of external sort to eliminate unwanted fields. Thus, runs of about 2B pages are produced, but tuples in runs are smaller than input tuples. (Size ratio depends on number and size of fields that are dropped.)
- Modify merging passes to eliminate duplicates. Thus, number of result tuples smaller than input. (Difference depends on number of duplicates.)
- Cost: In Pass 0, read original relation (size M), write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass. Using Evaluations example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25

Projection Based on Sorting - Example

- Projection on *Evaluations* relation
- Sort-based projection
  - Step 1:
    - Scan *Evaluations* with 1000 I/Os
    - If a tuple in E' is 10bytes, 250 I/O to write out E'
  - Step 2:
    - Given 20 buffer pages, sort E' in two passes at a cost of 2*2*250 I/Os
  - Step 3:
    - 250 I/Os to scan for duplicates
  - Total cost: 2500 I/Os

Projection Based on Sorting - Example

- Projection on *Evaluations* relation
- Improved sort-based projection
  - First Pass:
    - Scan *Evaluations* with 1000 I/Os
    - Write out E' with 250 I/Os
    - With 20 buffer pages, the 250 pages are written out as 7 internally sorted runs, each about 40 pgs
      - Refinement of external sort, write out 2*B internally sorted pages
  - Second pass:
    - Read the runs at a cost of 250 I/Os and merge them
  - Total cost: 1500 I/Os

Projection Based on Hashing

- *Partitioning phase*: 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 (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.
- *Duplicate elimination phase*: For each partition, read it and build an in-memory hash table, using hash fn *h2* (<> *h1*) on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

Projection Based on Hashing - Cost

- Partitioning
  - Read E = N I/Os
  - Write E′ = T I/Os
- Duplicate elimination
  - Read in partitions = T I/Os
- Total cost = N + 2T I/Os
- *Evaluations* example = 1000 + 2*250 = 1500 I/Os

Discussion of Projection

- Sort-based approach is the standard; better handling of skew and 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

- 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 corresponding R-partition and add tuples to table while discarding duplicates.

Aggregate Operations (AVG, MIN etc.)

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.
- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

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.
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops? Sort-Merge Join?