CS 443
Query Optimization
Chapter 15

Slides adapted from
Ramakrishnan & Gerhke
pages.cs.wisc.edu/~dbbook/

Renée J. Miller

Query Evaluation

- How could we evaluate the following query?
  - select Date
  - from Reserves R, Sailors S
  - where R.SID = S.SID and S.Rating = 10

  - R = Reserves[Day,BID, SID]
  - S = Sailors[SID, Name, Rating, Age]

  - $\pi_{Date}(\sigma_{R.SID=S.SID \text{ and } Rating = 10}\ (R \times S))$
  - other options?

Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

** Watch for these techniques as we discuss query evaluation!*

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, 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.

Overview of Query Optimization

5

- *Plan: Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan. Practically: Avoid worst plans!
- We will study the System R approach.

Schema for Examples

6

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.

Query Blocks: Units of Optimization

7

- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an over-simplification, but serves for now.)

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
  (SELECT  MAX (S2.age)
   FROM  Sailors S2
   WHERE S.Rating=S2.Rating )
```

Outer block   Nested block

- For each block, the plans considered are:
  - All available access methods, for each reln in FROM clause.
  - All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)

Relational Algebra Equivalences

8

- Permits the choice of different join orders and to `push' selections and projections ahead of joins.
- *Selections*: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ (Cascade)
  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (Commute)
- *Projections*: $\pi_A(R) \equiv \pi_A(\ldots(\pi_{ABC}(R)))$ (Cascade)
- *Joins*: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (Associative)
  $(R \bowtie S) \equiv (S \bowtie R)$ (Commute)

☞ Show that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

More Equivalences

9

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with R $\bowtie$ S. (i.e., σ (R $\bowtie$ S) $\equiv$ σ (R) $\bowtie$ S)
- Similarly, if a projection follows a join R $\bowtie$ S, we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

Enumeration of Alternative Plans

10

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

Statistics and Catalogs

11

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Cost Estimation

12

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - *Cost is Height(I)+1 for a B+ tree, about 1.2 for hash index.*
- Clustered index I matching one or more selects:
  - *(NPages(I)+NPages(R)) * product of RF's of matching selects.*
- Non-clustered index I matching one or more selects:
  - *(NPages(I)+NTuples(R)) * product of RF's of matching selects.*
- Sequential scan of file:
  - *NPages(R).*

☞ ***Note:*** *Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)*

Access Paths

- An <u>access path</u> is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
- A tree index <u>*matches*</u> (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index <u>*matches*</u> (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

One Approach to Selections

- 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 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<8/9/94* must then be checked.

Using an Index for Selections

- Cost depends on #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 unclustered, upto 10000 I/Os!

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

Projection

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


- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- If there is an index with both R.sid and R.bid in the search key, may be cheaper to use data entries!

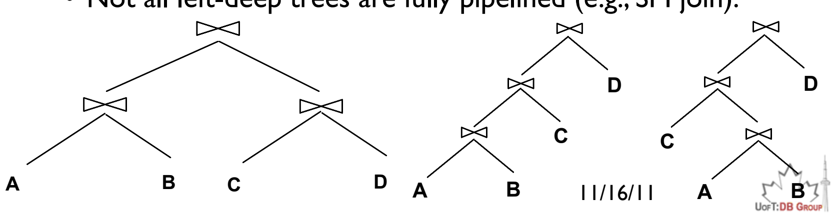
Example

```
SELECT  S.sid
FROM  Sailors S
WHERE  S.rating=8
```


- If we have an index on *rating*:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) pages are retrieved. (This is the ***cost***.)
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) pages are retrieved.
- If we have an index on *sid*:
  - Would have to retrieve all tuples/pages in file (500). Compare with a clustered index *rating*, the cost is 55, with unclustered index, 5+4000.
- Doing a file scan:
  - We retrieve all file pages (500).

Queries Over Multiple Relations

- Fundamental decision in System R: *<u>only left-deep join trees</u>* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



Enumeration of Left-Deep Plans

- Left-deep plans differ only in: order of relations, access method for each relation, and join method for each join.
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.

Enumeration of Plans (Contd.)

21

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator.
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, (the one exception being where all predicates in WHERE have been used up meaning the query contains a cartesian product)
  - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables.

Cost Estimation for Multi-relation Plans

22

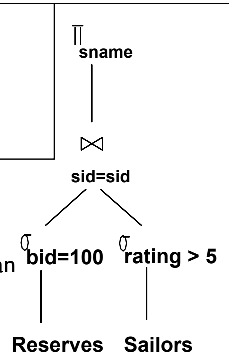
- Consider a query block:

```
SELECT attribute list
FROM relation list
WHERE term1 AND ... AND termk
```

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size.
  - *Result cardinality* = Max # tuples * product of all RF's.
- Multi-relation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimation of join cardinality gives us both cost estimate and result size estimate.


```
SELECT sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid and
  bid = 100 and rating > 5
```

Sailors:
  B+ tree on *rating*
  Hash on *sid*
Reserves:
  B+ tree on *bid*



23

- Pass1:
  - *Sailors*: B+ tree matches *rating>5*, probably cheapest. However, if selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
    - *sid* is an interesting order, so hash on *sid* kept even if higher cost than *rating* index
  - *Reserves*: B+ tree on *bid* matches *bid=100*; cheapest.
- Pass 2:

Consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.

Reserves as outer: Hash index can be used to get Sailors tuples that satisfy *sid* = outer tuple's *sid* value (selection on rating moved **after** join)
Alternative is BNL with $\sigma_{rating>5}$(Sailors)

Sailors as outer: block-nested loop to join with $\sigma_{bid=100}$(Reserves)

Nested Queries

24

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT *
   FROM Reserves R
   WHERE R.bid=103
    AND R.sid=S.sid)
```

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of `calling' nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

Nested block to optimize:

```
SELECT *
FROM Reserves R
WHERE R.bid=103
  AND R.sid= outer value
```

Equivalent non-nested query:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
  AND R.bid=103
```

Summary

25

- Query optimization is an important task in a relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Summary (Contd.)

26

- Single-relation queries:
  - All access paths considered, cheapest is chosen.
  - *Issues*: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.
- Multiple-relation queries:
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained'.