Overview of Query Evaluation

Chapter 12

Why Is This Important?

- Now that *we* know about the benefits of indexes, how does the *DBMS* know when to use them?
- An SQL query can be implemented in many ways, but which one is best?
  - Perform selection before or after join etc.
  - Many ways of physically implementing a join (or other relational operator), how to choose the right one?
- The DBMS does this automatically, but we need to understand it to know what performance to expect

Overview of Query Evaluation

- SQL query is implemented by a query plan
  - Tree of relational operators
    - `Pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
    - Can change structure of tree
    - Can choose different operator implementations
- Two main issues in query optimization:
  - 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.

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!*

Statistics and Catalogs

- Need information about the relations and indexes involved. Catalog typically contains:
  - #tuples (NTuples) and #pages (NPages) for each relation.
  - #distinct key values (NKeys), INPages, and low/high key values (ILow/IHigh) for each index.
  - Index height (IHeight) for each tree index.
  - Catalog data stored in tables; can be queried
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; costs are approximate anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) sometimes stored.

Access Paths

- Access path = way of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
  - Cost depends heavily on access path selected
- A tree index matches (a conjunction of) conditions that involve only attributes in a prefix of the search key.
  - E.g., Tree index on <a, b, c> matches "a=5 AND b=3" and "a=5 AND b>6", but not "b=3".
- A hash index matches (a conjunction of) conditions 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 not "b=3", "a=5 AND b=3", or "a>5 AND b=3 AND c=5".

A Note on Complex Selections

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- Selection conditions are first converted to conjunctive normal form (CNF):
  - E.g., (day<8/9/94 OR bid=5 OR sid=3) AND (rname='Paul' OR bid=5 OR sid=3)
- We only discuss case with no ORs; see text if you are curious about the general case.

Selectivity of Access Paths

- Selectivity = #pages retrieved (index + data pages)
- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - Terms that match the index reduce the number of tuples retrieved
  - Other terms are used to discard some retrieved tuples, but do not affect number of tuples fetched.
  - Consider "day < 8/9/94 AND bid=5 AND sid=3".
    - Can use B+ tree index on day; then check bid=5 and sid=3 for each retrieved tuple
    - Could similarly use a hash index on <bid,sid>; then check day < 8/9/94

Using an Index for Selections

- Without index on R.rname, have to scan
- Index cost depends on #qualifying tuples and clustering.
  - Cost of finding qualifying data entries (small) plus cost of retrieving records (could be large).
  - Data: 100K tuples on 1000 pages
  - Assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples).
  - Clustered index on rname: little more than 100 I/O
  - Unclustered index on rname: up to 10,000 I/O!

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


Projection

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

- The expensive part is removing duplicates.
  - DBMS does not remove duplicates by default.
- Sorting Approach
  - Sort on <sid, bid> and remove duplicates: scan of Reserves (1000 pages), plus 2-3 more passes of projected data set (~1000 pages)
- Hashing Approach
  - Hash on <sid, bid> to create partitions.
  - Load partitions into memory one at a time
    - Build in-memory hash structure, eliminate duplicates.
  - Scan of Reserves (1000 pages), plus write and read projected data (~500 I/O); but could be more
- If there is an index with all selected attributes in the search key, use index-only access on index leaves.

Join: Index Nested Loops

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

- Naïve implementation: scan S for each tuple in R
  - #pages(R) + |R| * #pages(S) page accesses
- Improved by block nested loops: foreach block of R, process each block from S
  - #pages(R) + #pages(R) * #pages(S) page accesses
- Can do even better with an index on the join column of one relation (say S) by making it the inner.
  - Cost: #pages(R) + (|R| * costOfFindingMatchingStuples)
  - For each R tuple, cost of probing S index is about 1.2 I/O 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: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Join Sailors and Reserves on sid
  - Assumption: R has 100K tuples on 1000 pages; S has 40K tuples on 500 pages
- Hash-index (Alt. 2) on sid of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100K tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in hash index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- Hash-index (Alt. 2) on sid of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 40K 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 from heap file is 2.5 I/Os. Total: 148,000 I/Os.

Join: Sort-Merge

- Sort R and S on the join column, then scan them to do a ``merge'' on join column, 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 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.
  - 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.)

Example of Sort-Merge Join

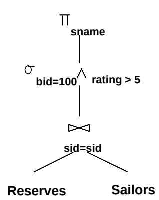
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

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

- Cost: $O(|S| \log|S| + |R| \log|R|) + \approx(|R|+|S|)$
  - Cost of scanning, usually $|R|+|S|$, could be $|R|*|S|$ (very unlikely)
- Assuming we can sort both R and S in two passes, sorting cost is 2*2*1000 I/Os for R and 2*2*500 I/Os for S
- Merge phase costs about 1000+500 I/Os
- Total cost: 4000+2000+1500 = 7500 I/Os.

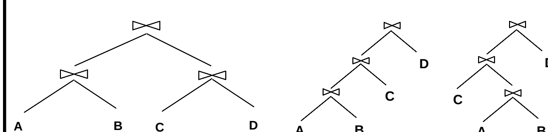
Highlights of System R Optimizer

- 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.



Plans Involving Joins

- 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.
      - But: sort-merge join implementation cannot be fully pipelined
  - Cartesian products avoided.



Cost Estimation

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

Size Estimation and Reduction Factors

- Consider a query block:

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

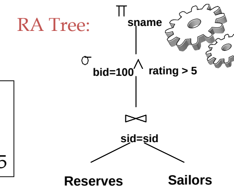
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size. Result cardinality = Max # tuples * product of all RF's.
  - Implicit assumption that terms are independent!
  - Term col=value has RF 1/NKeys(I), given index I on col
  - Term col1=col2 has RF 1/MAX(NKeys(I1), NKeys(I2))
  - Term col>value has RF (High(I)-value)/(High(I)-Low(I))

Schema for Examples

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

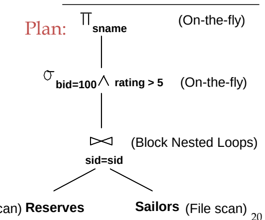
- Similar to old schema; rname added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Motivating Example

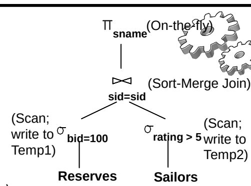


SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5

- Cost: 1000+1000*500 I/Os for join plus zero I/Os for on-the-fly computations
  - Total: 501,000 I/Os
- Misses several opportunities
  - Selections applied late.
  - No index used.
- Goal of optimization: Find more efficient plans that compute the same answer.

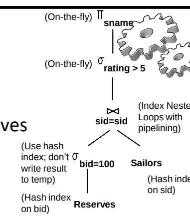


Alternative Plan 1 (No Indexes)



- Main idea: push selections.
- Cost of plan (with 5 buffers):
  - Scan Reserves (1000) + write Temp1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write Temp2 (250 pages, if we have 10 ratings).
  - Sort Temp1 (2*2*10), sort Temp2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os.
- Block nested loop (BNL) instead of sort-merge join
  - Buffer usage: 3 for Temp1 (hence only 10/3, i.e., 4 inner loops needed), 1 for Temp2, 1 for output
  - Join cost = 10+4*250, total cost = 2770 I/Os.
- Also `push' projections: Temp1 has only sid, Temp2 only sid and sname
  - Temp1 fits in the 3 buffer pages, cost of BNL drops to under 250 pages
  - Total < 2000 I/Os.

Alternative Plan 2 (With Indexes)



- Clustered hash index on bid of Reserves
  - 100,000/100 = 1000 selected tuples on 1000/100 = 10 consecutive pages.
- Index nested loops (INL) with pipelining (outer not materialized).
  - For each tuple returned by index on Reserves, find matches in Sailors by using the hash index.
    - Join column sid is a key for Sailors, hence at most one match.
- Why not push rating>5 before the join? Would prevent use of index on sid for Sailors for join!
- Cost: Find Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.
  - Assumption: hash index on Sailors uses Alternative 1, has 1.2 I/O average cost for retrieving matching tuple

Summary

- There are several alternative evaluation algorithms for each relational operator.
- A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.
- Must understand query optimization in order to fully 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.