Chapter 18 - Query Processing and Optimization

Scanner - identifies the language components (tokens) in the query
Parser - checks the query syntax to determine if it matches with the rules of the language grammar
- we won't go into detail on these - covered in compiler course

Validate - check that all relation and attrib names are valid and semantically meaningful

Internal representation of query
- query tree or query graph

Determine execution strategy - for retrieving results of query
- not unique
- must choose most efficient one (query optimization)
- trade-off between optimal query and time needed to find optimal

In hierarchical and network DB's, user plans execution strategy - low level DML

In relational DB, DML is declarative - tells what to compute not how to compute
- optimizer determines execution strategy

- Two approaches to choosing execution strategy:
1) heuristic rules for ordering operations in query execution
2) systematically estimating cost of different execution strategies

Implementing basic relational operations:

SELECT: $\sigma_{<\text{selection-condition}>}(<\text{relation-name}>)$

Single condition queries

1) Linear search - used if the attribute in the selection condition is not an index attribute or an ordering attribute

ex: $\sigma_{<\text{TOY\_NAME='DOLL'}>}(\text{TOY})$

2) Binary search - used if equality condition and if attrib is ordered through index or storage strategy (key)

ex: $\sigma_{<\text{CUST\_NUM='001'}>}(<\text{CUSTOMER}>)$

3) Primary index or hash key to retrieve a single record - if selection condition is equality on a key attrib with primary index or hash key

ex: $\sigma_{<\text{CUST\_NUM='001'}>}(<\text{CUSTOMER}>)$

- where we have an index on CUST_NUM

4) Primary index to retrieve multiple records - <, >, >=, <= on primary key
- do equality first and then search above or below

ex: $\sigma_{\text{<ORDER\_NUM > '5000'>}}(\text{<ORDER>})$

5) Clustering index to retrieve multiple records - equality comparison on non-key attrib with clustering index

ex: $\sigma_{\text{<NAME='Karen Smith'>}}(\text{CUSTOMER})$

6) Secondary (B+-tree) index -

ex: $\sigma_{\text{<DATE\_ORDERED>'09/23/91'>}}(\text{ORDER})$

- assuming a secondary index is created on DATE_ORDERED

Conjunctive condition queries:

- in general - perform search for any conjuncts having key or ordered attribs first (see above) - then search among the records chosen for the other condition(s)

ex: $\sigma_{\text{<(MAN\_ID='FP') AND (MSRP>30.00)>}}(\text{TOY})$

- assuming an index is defined on MAN_ID

- the optimizer should choose the access path that retrieves the fewest records in the most efficient way

- must consider selectivity of a condition (#records retrieved / #in file)
- estimates may be stored in DBMS catalog

JOIN - $R \; |X|_{A=B} \; S$

- most time-consuming operation - we only talk about two-relation equijoin (natural joins) - any more relations involved get combinatorial explosion of possibilities for order of join

1) Nested (inner loop) - least efficient

```
for r in R
   for s in S
      if r[A]=s[B]
         include (r,s)
      endif
   endfor
endfor
```

2) Use existing access structure

- assume index (or hash key) exists on attrib B of S

```
for r in R
   use S's access structure to get all s in S such that s[B] = r[A]
```

3) Sort-merge join: both files physically sorted by A (for R) and B (for S)

- scan both files in order of join attribute - match records that have the same values

```
i, j := 0;
while (i<=n and j<=m)
    if R(i)[A] < S(j)[B]
        i:=i+1;
    elseif R(i)[A] > S(j)[B]
        j:=j+1;
    else
        include the tuple <R(i), S(j)>
        output other tuples that match R(i) -incrementing j
        output other tuples that match S(j) - incrementing i
    endif
endwhile
```

4) Hash-join: R and S hashed to same hash file with same hash function with A and B as hash keys
   - find matching buckets and compare using sort-merge join within buckets

PROJECT: $\pi_{<attrib\text{-}list>}(<relation\text{-}name>)$

- straightforward unless the attribute list does not include a key
- if not, remove duplicates by sorting list and removing dups

Set Operations:

- Cartesian product is expensive because of number of tuples - avoid by using other operations in query optimization
- other three ops apply only to union compatible relations
- sort both relations on same attributes and scan through each to get results

- hashing to implement union, intersection and difference - hash both files to the same hash file buckets

Combining operations:

- reduce the number of temporary files - create algorithms for combining operations
- ex: join can be combined with two selects on the input files and a final project on the result - two input files and a single output file (no temps)

Heuristics for Query Optimization:

- use heuristic to modify internal representation of a query
  - internal representation usually a tree or graph
  - we will look at trees - more used for relational algebra
  - look at graphs on your own

- main heuristic - apply select and project before joins (or other binary ops)
  - select and project make results smaller
  - joins make results bigger
  - reduce size of join files by selecting and projecting first

- query trees
  - corresponds to a relational algebra expression
  - input relations at leaf nodes
  - operations at internal nodes
  - execution of a query tree - executing an internal node and replacing internal node by the relation that results
  - execution terminates when root node is executed and produces the final result

  - there can be many equivalent query trees for a query
    BUT - parser typically generates a standard canonical query tree to correspond to an SQL query (see example)

  - canonical tree is inefficient - optimizer has to transform tree into a more efficient form

General Transformation Rules

1) Cascade of sigma:

2) Commutativity of sigma:

3) Cascade of pi:

4) Commuting sigma with pi

5) Commutativity of |X| (or X ) (notice that since order does not matter, the relations are still equivalent)

6) Commuting sigma with |X| (or X)

7) Commuting pi with |X| (or X):

8) Commutativity of intersection and union (not difference)

9) Associativity of |X|, X, union and intersection

10) Commuting sigma with set operations:

Heuristic that uses the above transformations to make a more efficient tree:
(go through example for each step)

1) R1 - break up SELECT ops into a cascade of SELECTs

2) R2, R4, R6, R10 - move each SELECT op as far down the query tree as is permitted

3) R9 - rearrange leaf nodes so that leaf node with most restrictive SELECT ops are executed first - ones that produce the fewest tuples

4) Combine X with a subsequent SELECT to make a JOIN op

5) R3, R4, R7, R11 - break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT ops as needed

6) Identify subtrees that represent groups ops that can be executed by a single algorithm

Cost Estimates in Query Optimization

- query optimizer must estimate the cost of executing a query using a specific execution strategy - choose strategy with lowest cost estimate
- because estimates are used (not actual costs) optimizer may not choose optimal execution strategy

- this approach most suitable for compiled queries

- for interpreted queries (optimization occurs at runtime) can be too slow

What to consider when estimating cost of an execution strategy:

1) access cost to secondary storage
   - search for, read, write data blocks
   - depends on access structures, contiguous? file blocks
   - emphasis is here for large DBs
2) storage of intermediate files
3) computation cost - cost of performing in-memory operations
   - emphasis can be here for small (mostly main mem) DBs
4) communications cost - cost of shipping query and result from database site to original site of query
   - emphasis may be here for distributed DBs

- Data used to determine the cost function may be kept in the DB catalog:

- size of each file
- number of records r
- number of blocks b

- blocking factor bfr
- primayr access method, primary access attributes
- number of levels of a multilevel index
- number of first level index blocks b11
- number of distinct values (d) of an indexing attribute
- selection cardinality s (average number of records that will satisfy an equality selection)
  - key: s=1
  - = nonkey: s=r / d

- Because some of this data changes, optimizer will need close values (if not exact)

Let's look at example cost functions for the operations we discussed above

SELECT - recall the 8 selection algorithms discussed above
- these use number of block transfers to estimate cost (ignore all other parameters)

S1) Linear search:
- C = b
- for equality on a key only half on average are searched; C = b / 2

S2) Binary search:
- C = log2 b + ceiling(s / bfr) - 1
- reduces to log b for equality on a key since s = 1

S3) Primary index:
- C = x + 1 (one more block than the number of index levels)

Hash key:
- C = approx 1

S4) Ordering index to retrieve multiple records:
- <, >, <=, >= approx half will satisfy condition; C = x + b / 2

S5) Clustering index to retrieve multiple records:
- C = x + ceiling(s / bfr)
- s is selection cardinality of the indexing attribute

S6) Secondary B+tree index:
- equality condition: C = x + s
  since each record may reside in a different block (nonclustering index)
- inequality condition:
  - half first level index blocks are accessed
  - half file records via the index
  - very approximately: C = x + (b11 / 2) + (r / 2)

Example: TOY relation has r=10000 records; b=2000 disk blocks; bfr=5 records / block

- clustering index on MSRP, levels x=3, selection cardinality s=20
- secondary index on TOY_NUM, x=4, s=1

- secondary index on MAN_ID, x=2, first level index blocks b11=4, distinct values d=125; s=r / d=80

1) $\sigma_{TOY\_NUM}$(TOY)
   - use method S6 - cost estimate is C=4+1 = 5

2) $\sigma_{MAN\_ID>324}$(TOY)
   - use S6b; C=x+(b11 / 2)+r / 2 = 2 + 4 / 2 + 10000 / 2 = 5004 OR
   - use S1; C=b=2000
   - choose S1

3) $\sigma_{MAN\_ID=324}$(TOY)
   - use S6a; C=x+s=2+80=82 OR
   - use S1; C=2000
   - choose S6a

4) $\sigma_{MSRP>45.00 \text{ AND } MAN\_ID=325}$(TOY)
   - esitmate the cost of using any one of the three components of the selection condition plus the brute force method
   - S1: C=2000
   - use condition MAN_ID=325: C=82 (as above)
   - use condition MSRP>45.00: using S4 - C=x+b / 2=3+2000 / 2=1003
   - thus, do MAN_ID=325 first, then linear search for the other condition

JOIN

- need estimate of the number of tuples fo the file that results after the join
- ratio of the number of tuples in te join file to the size of the Cartesian product file - join selectivity (js) - let |R| be the number of tuples in R

js = |(R |X| c S)| / |(R X S)| = |R |X| c S)| / (|R| * |S|)

in general, 0<= js <= 1

- for c: R.A = S.B we have two special cases:

1) if A is a key of R, then |(R |X| c S)| <= |S|, so js <= (1 / |R|)
2) if B is a key of S, then |(R |X| c S)| <= |R|, so js <= (1 / |S|)

- store an estimate of join selectivity for commonly used join conditions, we can use the formula:

|(R |X| c S)| = js * |R| * |S|

- use this the estimate size of join file for different implementations of the join operator: R $|X|_{A=B}$ S
   - $b_R$ = #blocks in R; $b_S$ = # blocks in S; $bfr_{RS}$ = blocking factor for the result
- find cost estimate for each

J1) Nested loop approach - use R for outer loop, assume two memory buffers
   C = $b_R$ + ($b_R$*$b_S$) + (js * |R| * |S|) / $bfr_{RS}$

- last part due to writing file to disk

J2) Access structure

a) secondary index for B of S, s=selection cardinality for B

$C = bR + (|R| * (x + s)) + (js * |R| * |S|) / bfr_{RS}$

b) clustering index for B of S

$C = b_R + (|R| * (x + s / bfr_B)) + (js * |R| * |S|) / bfr_{RS}$

c) primary index for B of S

$C = b_R + (|R| * (x+1) + (js * |R| * |S|) / bfr_{RS}$

d) hash key for B of S, h >= 1, average number of block accesses to retrieve a record, given hash key value

$C = b_R + (|R| * h) + (js * |R| * |S|) / bfr_{RS}$

J3) Sort-merge-join

$C = b_R + b_S + (js * |R| * |S|) / bfr_{RS}$

Example:

TOY file as in earlier example; MANUFACTURER file with $r_M$=125 records, $b_M$=13 disk blocks

TOY $|X|_{MAN\_ID=MAN\_ID}$ MANUFACTURER

J1) with TOY as outer loop

$C = b_T + (b_M*b_T) + (js * r_T * r_M) / bfr_{MT} =$
2000+(2000*13)+((1 / 125)*10000*125) / 4 = 30,500

J1) with MAN as outer loop

$C= b_M + (b_M*b_T) + (js * r_T * r_M) / bfr_{MT} =$
13+(2000*13)+((1 / 125)*10000*125) / 4 = 28,513

J2) with TOY as outer loop

$C = bT + (rT * (x + 1)) + (js * r_T * r_M) / bfr_{MT} =$
2000 + (10000 * 2) + ((1 / 125)*10000*125) / 4 = 24,500

J2) with MAN as outer loop

$C = bM + (rM * (x + s)) + (js * r_T * r_M) / bfr_{MT} =$
13 + (125*(2*80)) + ((1 / 125)*10000*125) / 4 = 12,763

- choose this one because it has the lowest cost estimate