System R Optimizer

Read the paper (available at the course web page):
The System R query optimizer is the most widely used currently. It works well for <10 joins.

The original prototype (Phase 0, 1974-1975) was single-user, no locking, no recovery. Plan costing was based on # of tuples accessed only (rather than # of blocks, plus CPU time).

Phase 1 (1975-1978) handled memory management, locking, logging:

- **Storage manager: RSS (Research Storage System)**
  - RSI interface provides tuple-at-a-time operators on base relations
  - RSS scan (operations: OPEN, NEXT, CLOSE), which can be either a segment scan (SARS), where each page is touched once, or an index scan (SARGS) for $B^+$-tree indexes.

- **Query processor: RDS (Research Data System)**
  - Chooses a low cost plan from those provided by RSS
  - Does query analysis, catalog lookup, authorization, query optimization.
Access Methods

A *sargable predicate* takes the form:

\[ \text{TABLE.COLUMN compare value} \]

and may specified on any of the scans to filter the stream records.

A *SARG* is a disjunctive normal form with sargable predicates.

Storage structures:

- Data are stored in a set of logical spaces, called *segments*, used for the control of physical clustering. They contain user data, indexes, the user catalog, intermediate query results.
- A segment is a set of pages and may contain tuples from many relations. A relation though does not cross segment boundaries.
- Each segment has a mapping table that translates logical addresses to physical addresses.
Query Optimization

• Exhaustive search of the solution space to select the best evaluation plan from nearly all possible plans

• It considers both I/O and CPU costs:
  \[ \text{cost} = (\text{page fetches}) + W \times (\text{RSI calls}) \]
  where (RSI calls) is the number of tuples processed
  and W is the weighting factor between I/O and CPU (typically, 0.1 to 0.3)

• Instead of using heuristic rules, it uses statistical information

• It takes into account the order of the result tuples of an operator to select the next operator

• \textit{Selectivity}: expected fraction of tuples that satisfy a predicate:
  \[ F(\text{pred}) = |\sigma_{\text{pred}}(R_1 \times \ldots \times R_n)| / |(R_1 \times \ldots \times R_n)| \]
  given the input relations, \( R_1, \ldots, R_n \), of the predicate

• Uses table/index cardinality and \# of unique values in an attribute.
Statistics

- NCARD(T)  cardinality of relation T
- TCARD(T)  size of T in pages
- P(T)      percentage of non-empty pages for T
- ICARD(I)  cardinality of index I (# of distinct keys)
- NINDEX(I)  size of I in pages
- high/low key values of index I

Selectivity of a predicate $F(\text{pred})$ is calculated in terms of the above statistics.

If there is an index I on column T.A, then:

$$F(T.A = \text{value}) = \frac{1}{ICARD(I)}$$

assuming an even distribution of tuples among the index keys.

Otherwise (if there is no index), $F(T.A = \text{value}) = \frac{1}{10}$  (why?)
More on Selectivities

- $F(R.A=S.B)$
  
  - $= \frac{1}{\max(\text{ICARD}(I_R),\text{ICARD}(I_S))}$ if there are indexes $I_R/I_S$ for $R.A/S.B$
  
  - $= \frac{1}{\text{ICARD}(I)}$ if there is an index $I$ for $R.A$ or $S.B$
  
  - $= \frac{1}{10}$ otherwise

- $F(T.A > \text{value})$ or $\geq, <, \leq$
  
  - $= \frac{\text{high key value} - \text{value}}{\text{high key value} - \text{low key value}}$
  
  - $= \frac{1}{3}$ otherwise

- $F(T.A \text{ between value1 and value2})$
  
  - $= \frac{\text{value2} - \text{value1}}{\text{high key value} - \text{low key value}}$
  
  - $= \frac{1}{4}$ otherwise

- $F(T.A \text{ in } [\text{value}_1,\ldots,\text{value}_n])$
  
  - $= n \times F(T.A = \text{value})$

- $F(T.A \text{ in subquery})$
  
  - $= F(\text{subquery})$
More on Selectivities

• $F(\text{pred}_1 \text{ or } \text{pred}_2) = F(\text{pred}_1) + F(\text{pred}_2) - F(\text{pred}_1) \times F(\text{pred}_2)$

• $F(\text{pred}_1 \text{ and } \text{pred}_2) = F(\text{pred}_1) \times F(\text{pred}_2)$

• $F(\text{not } \text{pred}) = 1 - F(\text{pred})$

For a query $Q$: select * from $T_1, \ldots, T_n$ where pred

The Query Cardinality:

$$Q\text{CARD}(Q) = \prod_i \text{NCARD}(T_i) \times F(\text{pred})$$

$$R\text{SI}\text{CARD}(Q) = \prod_i \text{NCARD}(T_i) \times F(\text{sargable predicates})$$

(for RSI calls).
Approach

• Concept of *interesting order*: order specified by GROUP-BY or ORDER-BY or requested by a merge-join.

• Use of dynamic programming to limit the number of alternative plans.

• Support for equivalent classes for access plans to avoid repeating search (memoization).

• Use of pruning to reduce the search space.
  – Prune uninteresting plans (plans that do not deliver interesting order)
  – Don’t prune operator implementations based on cost, because they may result to an overall cheaper plan
  – Avoid cross products
  – Prune join implementations
  – Prune within a class
  – Merge equivalent classes.
Single Relation Queries

If it is a group-by or an order-by query, the interesting order is the group-by/order-by attributes.

Access Paths:
• Examine the cheapest access path which produces tuples in each interesting order.
• Examine the cheapest unordered access path.

Cost Formulas:
• Unique index matching a equal predicate: $1 + 1 + W$
• Matching clustered index I of table T:
  \[ F(\text{preds}) \times (\text{NINDX}(I) + \text{TCARD}(T)) + W \times \text{RSICARD} \]
• Matching non-clustered index I of table T:
  \[ F(\text{preds}) \times (\text{NINDX}(I) + \text{NCARD}(T)) + W \times \text{RSICARD} \]
• Non-matching index: use $F(\text{preds}) = 1$
• Segment scan: $\frac{\text{TCARD}(T)}{P(T)} + W \times \text{RSICARD}$
Joins

- Joins considered: nested loops and merging scans (for equi-joins only).
- Only left-deep trees are considered. Join order permutations:
  - For n tables, there are n! permutations
  - The problem is now finding an order for n tables, t₁,…,tn, with minimal cost.
- Assumption: once the first k relations are joined, the method to join the result to the k+1’th relation is independent of the order of joining the first k relations.
  - Is it valid?
  - Is it realistic?
  - Optimal substructure => dynamic programming.
Dynamic Programming

- A join order heuristic is used to prune out cartesian products:
  
  We examine join orders $t_1,\ldots,t_n$ if for each $j$:
  - either $t_j$ has a join predicate with $t_k$, for some $k<j$
  - or $t_j$ does not have a join predicate, for all $k>j$

  e.g. query graph: $T_1 \rightarrow T_2 - T_3$,
  
  join orders not considered: $T_1, T_3, T_2$ and $T_3, T_1, T_2$

- Interesting orders: group-by attributes, order-by attributes, and every join attribute (since merging scan needs its inputs joined by the join attributes).

- Needs to store $2^n$ solutions, one for each subset of $n$ tables, times the number of applicable interesting orders plus one (for the unorder case). Each solution has a cost, cardinality, and delivered order. For each subset of $k$ relations, we join the composite result with one of the $n-k$ remaining relations and store the result.
Plan Costing

Cardinality of outer relation: \( N \) (estimated using selectivities)

Cost of joins:

- Nested loops: \( \text{cost(outer)} + N \times \text{cost(inner)} \)
- Merge scan join: \( \text{cost(outer)} + \text{cost(inner)} + \text{sorting-cost} \)
Example

```
select name, title, sal, dname
from emp, dept, job
where title = 'clerk' and loc = 'denver'
    and emp.dno = dept.dno and emp.job = job.job
```

Interesting orders: dno and job.

Access paths for single relations:
- EMP: index scan using emp.dno, index scan using emp.job, segment scan (pruned)
- DEPT: index scan using dept.dno, segment scan (pruned)
- JOB: index scan using job.job, segment scan (cheaper, so not pruned).
Example (cont.)

Two relations:

- **emp*dept:**
  - Nested loop: index scan on emp.dno, index scan on dept.dno,
  - Nested loop: index scan on emp.job, index scan on dept.dno,
  - Merge join: index scan on emp.dno, merge emp.dno with dept.dno
  - Merge join: sort emp.job by dno into L1, merge L1 with dept.dno

- **job*emp**
  - Nested loop: segment scan job, index scan on emp.job
  - Merge join: segment scan on job, sort by job.job into L2, merge L2 with emp.job
  - Merge join: index scan on job.job, merge L2 with emp.job

- **emp*job**
  - ...

- **dept*emp**
  - ...

- **dept*job**
  - ...

- **job*emp**
  - ...

- **job*dept**
  - ...

- **dept*job**
  - ...

- **dept*dept**
  - ...

- **job*job**
  - ...

- **job*dept**
  - ...

- **dept*job**
  - ...

- **dept*dept**
  - ...

- **job*job**
  - ...

- **dept*dept**
  - ...

- **job*job**
  - ...
Example (cont.)

Three relations:

• (emp*dept)*job
  – Get (emp*dept) with job order, sort job segment scan by job into L2, merge the two results
  – ...

• (emp*job)*dept
  – Get the (emp*job) with job order, sort it by dno into L5, merge L5 with dept.dno scan
  – ...

Nested Queries

Classification:

- *Uncorrelated subquery*: single-shot evaluation subquery
  
  ```sql
  select name
  from employee
  where salary = (select avg(salary) from employee)
  ```

  If the subquery returns a value, replace it with that value:
  
  ```sql
  select name
  from employee
  where salary = K
  ```

  where \( K \) is the average salary. If the subquery returns a set, evaluate the subquery and store the result in a table:
  
  ```sql
  select name
  from employee
  where dept in (select dno from dept where location = 'denver')
  ```
Nested Queries (cont.)

- **Correlated subqueries**: when it refers to variables/values obtained from a higher-level query block:
  
  ```sql
  select name
  from employee x
  where salary > (select salary from employee where num = x.manager)
  ```

  Naïve nested-loop evaluation: for each employee x, evaluate the inner query (very expensive). Better evaluation:

  ```sql
  select x.name
  from employee x, employee m
  where m.num = x.manager
  and x.salary > m.salary
  ```