System R Optimization (contd.)

Instructor: Sharma Chakravarthy
sharma@cse.uta.edu
The University of Texas @ Arlington
Optimization

Criteria

- number of page accesses + cpu cost

Factors influencing this criteria

- the size of each relation involved
- the type of scan (heap, hash, b+tree, etc.)
- Join algorithm used (nested-loop, sort-sort merge, hash-based, Index nested loop)
- Tuple order of the inputs
- Join sequence
Concept

- interesting output order
  - the order of the data that can be used by some join implementation or is the final output order
    - Merge join
    - Group by or order by
Example:
```sql
select *
from R, S
where ..... 
orderby R.A
```

Suppose R is sorted on A (clustered on A or index on A), then that is considered an interesting order
Approaches

- Dynamic programming
  - limit the number of implementations to consider
  - Reduces the number of partial access plans to consider
  - The ability to prune an entire class of partial access plans
  - Use the concept of equivalence class
Dynamic programming

Number of nodes: 30
Number of nodes evaluated: 10
Approaches contd

- Techniques used in dynamic programming
  - Pruning of Un-interesting Scans
  - Avoidance of Cross Product
  - Pruning within Equivalence Class
  - Pruning within Similar Class
  - Pruning of Join Implementations
Pruning of uninteresting scans

- **Effect**
  - retrieval cost
  - Output (interesting order)

- **Strategy**
  - if a scan (1) has un-interesting order and (2) is not the cheapest scan then prune it
  - for scans with interesting orders and are not the cheapest, we do not prune them because they might help reduce the cost of later joins
Avoidance of cross product

- Cross product
  - The most expensive operation
- Strategy
  - Never perform a cross product between two (composite) relations if some join predicates still exist.
- Situations
  - Eliminating
  - Delaying
Elimination

- Eliminate the cross product all together

- Example

  select T1.a, T2.b, T3.c
  from T1, T2, T3
  where T1.x=T2.y
  AND T2.r=T3.s
Elimination contd.

- Cross product
  - Between T1 and T3
- Acceptable join sequences:
  - (T1,T2), T3
  - (T2,T3), T1
- Pruned join sequences:
  - (T1,T3), T2
  - (T3,T1), T2
Delay cross products

- Delay the performance of the cross product
  
  Select T1.a, T2.b, T3.c
  
  From T1, T2, T3
  
  Where T1.x = T2.y

- Cross product
  
  - Between T1 and T3
  
  - Between T2 and T3

- Acceptable join sequences:
  
  (T1, T2), T3    (T2, T1), T3
  
- Pruned join sequences
  
  - (T2, T3), T1    (T3, T2), T1    (T1, T3), T2    (T3, T1), T2
Pruning within equivalence class

Equivalence Class Concept

- Two partial access plans belong to the same equivalence class if they affect the costs of the later joins in the same way.

Two partial access plans are in the same equivalence class if

1. the relations involved are the same
2. the sizes of their associated composite relations are the same
3. the interesting order are the same
4. the states (stream or in disk) of the composite relations are the same, and
5. the access structures available to the composite relations are the same.
Pruning within equivalence class contd.

- **Strategy**
  - Among the partial access plans belonging to the same equivalence class, keep the cheapest partial access plan, and prune the remaining partial access plans.
Pruning within similar class

- Concepts
  - locally interesting order
    - Tuple orders that are interesting to later joins or is the final output order
  - Similar class
    - Partial access plans belong to the same similar class if
      1. the relation involved are the same and
      2. the sizes of the composite relations are the same
Pruning within similar class contd.

- **Strategy**
  - As we further extend the partial access plans, some tuple orders that used to be interesting will become locally uninteresting.
  - Within each similar class, we prune all the partial access plans with locally uninteresting tuple orders unless they are the cheapest partial access plans within the similar class.
Cost-based optimization

- For Single relation queries, the cheapest access path is obtained by evaluating the cost for each available access path.
- For each access path, a predicted cost is computed along with the ordering of tuples it will produce.
- We need to choose the “cheapest” one which is either
  - the one that produces an interesting order
  - The one that is “unordered” + cost of providing an interesting order if necessary.
Enumeration of Alternative Plans

- 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 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).
Cost Estimates for Single-Relation Plans

- **Index I on primary key matches selection:**
  - Cost is $\text{Height}(I) + 1$ for a B+ tree, about 1.2 for hash index.

- **Clustered index I matching one or more selects:**
  - $(N\text{Pages}(I) + N\text{Pages}(R)) \times \text{product of RF’s of matching selects}.$

- **Non-clustered index I matching one or more selects:**
  - $(N\text{Pages}(I) + N\text{Tuples}(R)) \times \text{product of RF’s of matching selects}.$

- **Sequential scan of file:**
  - $N\text{Pages}(R)$.

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

- If we have an index on rating:
  - \( (1/N\text{Keys}(I)) \times NTuples(R) = (1/10) \times 40000 \) tuples retrieved.
  - Clustered index: \( (1/N\text{Keys}(I)) \times (N\text{Pages}(I)+N\text{Pages}(R)) = (1/10) \times (50+500) \) pages are retrieved. (This is the cost.)
  - Unclustered index: \( (1/N\text{Keys}(I)) \times (N\text{Pages}(I)+NTuples(R)) = (1/10) \times (50+40000) \) pages are retrieved.

- If we have an index on sid:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.

- Doing a file scan:
  - We retrieve all file pages (500).

```
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```
Access path selection for joins

- **System R uses 2 methods** for performing 2-way joins
  - **Nested loops:**
    - uses scans (can be one of the scans mentioned earlier), in any order, on the inner and the outer relation
  - **Merging Scans:**
    - requires the inner and outer relations to be scanned in the join column order.
    - Introduces interesting order for later computations.
    - If there are more than one predicate, one of them is used as a join predicate and the others are treated as ordinary predicates
    - This is applied to equi-joins (although it could be applied to any other types)
    - If one or both of the relations has no indexes on the join column, must be sorted into a temp which is ordered by the join column (clustering will further reduce the number of page scans)

- **Current optimizers use all supported join algorithms during optimization**
Queries Over Multiple Relations

- Fundamental decision in System R: only left-deep join trees 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).
N-way joins

- N-way joins are computed using a sequence of 2-way joins. A composite relation is the result of a join and is always used as the outer relation.

- In System R
  - Composite relations are not materialized unless they need to be sorted
  - Predicates are classified into:
    - Sargable – applied by RSS scan during the scan
    - Residual – e.g., arithmetic, subqueries which require repeated evaluation are applied after a tuple has been retrieved by RSS scan but before participating in any join
    - Local predicates – references columns of a relation
More on Joins

- Important: The cardinality of the join of n relations is the same regardless of join order. However, the cost of joining in different orders can be substantially different.
- If a query has n relations in its from list, then there are n! (factorial) permutations (not left-deep trees) of relation join orders.
- Once the first k relations are joined, the method to join the composite to the k+1st relation is independent of the order of the first k.
- That is, the eligible predicates are same, the set of interesting orderings in the possible join methods are the same.
Query: retrieve into result (T1.a1)
where
T1.a1 = T2.a2 and
T1.a1 = T3.a3 and
T1.a1 = T4.a4

Figure 3-5. Four table query and join orderings evaluated by JOIN. Trees labeled (e) are not evaluated by System R.
Search space

- The search tree is constructed by iteration on the number of relations joined so far.
- First, the best way is found to access each single relation for each interesting order and for each unordered case.
- Next, the best way of joining any relation to these is found, subject to the heuristics of join order. This produces solutions for joining pairs of relations.
- The best way of joining sets of three relations is found by considering all sets of two relations and joining each third relation permitted by the join order heuristic.
- For each plan, a join of a set of relations, the order of composite result is kept in the tree.
Search space

- Consider finding the best join-order for $r_1$ join $r_2 \cdots r_n$.
- There are $(2(n - 1))/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
To find best left-deep join tree for a set of $n$ relations:
- Consider $n$ alternatives with one relation as right-hand side input and the other relations as left-hand side input.
- Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the $n$ alternatives.

- If only left-deep trees are considered, time complexity of finding best join order is $O(n \ 2^n)$
- Space complexity remains at $O(2^n)$
Search space

- The number of solutions which must be stored is at most $2^n$ (the number of subsets of n tables) times the number of interesting result orders. The computation time to generate the tree is approximately proportional to the same number.
- This number is frequently reduced substantially by the join order heuristic.
- joins of 8 tables have been optimized in a few seconds (on a 370/158 cpu)
Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the 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.)

- 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, unless all predicates in WHERE have been used up.
  - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables/relations.
Example

- **Pass 1:**
  - **Sailors:** B+ tree matches \( \text{rating} > 5 \), and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
    - Still, B+ tree plan kept (because tuples are in \( \text{rating} \) order).
  - **Reserves:** B+ tree on \( \text{bid} \) matches \( \text{bid} = 500 \); cheapest.

- **Pass 2:**
  - We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
    - e.g., **Reserves as outer**: Hash index can be used to get Sailors tuples that satisfy \( \text{sid} = \) outer tuple’s \( \text{sid} \) value.

Database Management Systems, S. Chakravarthy
Example

<table>
<thead>
<tr>
<th>Name</th>
<th>DNO</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>50</td>
<td>12</td>
<td>8500</td>
</tr>
<tr>
<td>JONES</td>
<td>50</td>
<td>5</td>
<td>15000</td>
</tr>
<tr>
<td>DOE</td>
<td>51</td>
<td>5</td>
<td>9500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>MFG</td>
<td>DENVER</td>
</tr>
<tr>
<td>51</td>
<td>BILLING</td>
<td>BOULDER</td>
</tr>
<tr>
<td>52</td>
<td>SHIPPING</td>
<td>DENVER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JOB</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>CLERK</td>
</tr>
<tr>
<td>6</td>
<td>TYPIST</td>
</tr>
<tr>
<td>9</td>
<td>SALES</td>
</tr>
<tr>
<td>12</td>
<td>MECHANIC</td>
</tr>
</tbody>
</table>
Example contd

- Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver

```sql
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

- No Group By or Order By
```
Given

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
<th>Segment scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP</td>
<td>EMP.DNO</td>
<td>EMP</td>
</tr>
<tr>
<td>3 access</td>
<td>N1</td>
<td>N1 (unordered, not in interesting order)</td>
</tr>
<tr>
<td>paths</td>
<td>C(EMP.DNO)</td>
<td>C(EMP seg scan) pruned</td>
</tr>
<tr>
<td>DEPT</td>
<td>DEPT.DNO</td>
<td>C(DEPT seg. scan) pruned</td>
</tr>
<tr>
<td>cheaper</td>
<td>N2</td>
<td></td>
</tr>
<tr>
<td>JOB</td>
<td>JOB.JOB</td>
<td>C(JOB seg. scan)</td>
</tr>
<tr>
<td></td>
<td>N3</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3. Search tree for single relations
Figure 4. Extended search tree for second relation (nested loop join)
Database Management Systems, S. Chakravarthy 38

(EMP, DEPT)  (EMP, JOB)  (DEPT, EMP)  (JOB, EMP)

Index            Index          Index        Index                    Index                   Index scan
(EMP.DNO)       (EMP.JOB)      (EMP.DNO)     (EMP.JOB)    (DEPT.DNO)       (JOB.JOB)          (JOB)
N1                        N1                 N1       N1                          N2                       N3                     N3

Index                          Index                  Index       Index                     Index               Index      Index
(DEPT.DNO)          (DEPT.DNO) (JOB.JOB)       (JOB.JOB)        (EMP.DNO)        (EMP.JOB)      (EMP.JOB)
N4                         N4                  N5       N5                           N4                  N5 N5

+                              +                     + +                              +           +                         +
N1CE(D.DNO)     N1CE(D.DNO)    N1CE(J.JOB)   N1CE(J.JOB)    N2CD(E.DNO)   N3CJ(E.DNO)  N3CJ(E.DNO)

DNO order             JOB order           DNO order       JOB order        DNO order        JOB order        unordered

Equivalence classes

Equivalence classes

Segment scan

(EMP, DEPT) (EMP, JOB) (DEPT, EMP) (JOB, EMP)
Figure 5. Extended search tree for second relation (merge join)
Figure 6. Extended search tree for third relation
Nested Queries

- Classification
  - single-shot evaluation of sub/nested queries
  - Example
    Select Name
    From Employee
    Where Salary = (Select Ave (salary)
    From Employee)
  - The subquery returns a single value.
  - Hence the predicate can be modified as
    Salary = “K” where k is the result of the sub query
Nested queries contd.

- Sub/nested query can return a set
  - Example
    ```sql
    Select Name
    From Employee
    Where Dept_number IN
        (Select Dept_Number
         From Dept
         Where location='Denver')
    
    - Modify the predicate appropriately at run time.
      (as an IN clause which has an explicit set as part of the query)
Nested queries contd.

- The previous query can be re-written without nesting
  - Example
    ```sql
    Select Name
    From Employee e, Dept d
    Where e.dept_number = d.dept_number
    and d.location='Denver'
    ```
  - The optimizer does not do this transformation
Nested queries contd.

“Select names of Employees who earn more than their managers”
- In principle, a correlation query must be reevaluated for each candidate tuple from the referenced query block
Nested queries contd.

- Correlated sub query
  - a sub query that contains reference to a value obtained from a candidate tuple of a higher level query block
  - Example
    
    ```sql
    Select Name
    From Employee X
    where Salary > (Select Salary
                     From Employee
                     where Employee.Num=X.Manager)
    ```
Multi level correlated subqueries

Select names of employee’s who earn more than their manager’s manager.

L1

select name
from Employee X
where salary >

L2

(select salary
from Employee
where Employee.number =

L3

(select manager
From Employee
where Employee.number = X.Manager))

Since L3 query references a L1 value, L3 need to be evaluated for each L1 tuple. Since it does not reference any L2 value, it need not be evaluated for every L2 candidate tuple.
Multi level correlated subqueries

It may be useful to sort Emp on manager attribute to avoid multiple evaluations for more than one emp having the same manager
Nested Queries

- 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.

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

Nested block to optimize:
SELECT *
FROM Reserves R
WHERE R.bid=103
AND S.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

- 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.)

- **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`. 