Lecture #7

Query Optimization
May 16th, 2002

Agenda/Administration

- Last homework handed out by the weekend.
- Projects status?
- Trip Report
- Query optimization

Query Optimization

Goal:
Declarative SQL query → Imperative query execution plan:

```
SELECT S.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
  Q.city='seattle' AND
  Q.phone > '5430000'
```

Inputs:
- the query
- statistics about the data
  (indexes, cardinalities, selectivity factors)
- available memory

Ideally: Want to find best plan. Practically: Avoid worst plans!

How are we going to build one?

- What kind of optimizations can we do?
- What are the issues?
- How would we architect a query optimizer?

Discussion

How Would You Do It?

Schema for Some Examples

Sailors (sid: integer, sname: string, rating: integer, age: real)
Reserves (tid: integer, sid: integer, day: date, rname: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (4000 tuples)
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (4000 tuples).
Motivating Example

Cost: $500+500 \times 1000$ I/Os

By no means the worst plan!

Misses several opportunities:
- Selections could have been `pushed' earlier, no use is made of any available indexes, etc.

Goal of optimization: To find more efficient plans that compute the same answer.

Alternative Plans 1

- **Main difference:** push selects.
- With 5 buffers, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 ($2 \times 2 \times 10$), sort T2 ($2 \times 3 \times 250$), merge (10+250), total=1800
  - Total: 3560 page I/Os.
- If we used BNL join, join cost = 10+4*250, total cost = 2770.
- If we `push' projections, T1 has only sid, T2 only sid and sname:
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.

Alternative Plans 2

With Indexes

- With clustered index on bid of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- **INL with pipelining (outer is not materialized).**
  - Join column sid is a key for Sailors.
    - At most one matching tuple, unclustered index on sid OK.
  - Decision not to push rating>5 before the join is based on availability of sid index on Sailors.
  - **Cost:** Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Building Blocks

- Algebraic transformations (many and wacky).
- Statistical model: estimating costs and sizes.
- Finding the best join trees:
  - Bottom-up (dynamic programming): System-R
- **Newer architectures:**
  - Starburst: rewrite and then tree find
  - Volcano: all at once, top-down.

Query Optimization Process (simplified a bit)

- Parse the SQL query into a logical tree:
  - identify distinct blocks (corresponding to nested sub-queries or views).
- Query rewrite phase:
  - apply algebraic transformations to yield a cheaper plan.
  - Merge blocks and move predicates between blocks.
- Optimize each block: join ordering.
- Complete the optimization: select scheduling (pipelining strategy).

Key Lessons in Optimization

- There are many approaches and many details to consider in query optimization
  - Classic search/optimization problem!
  - Not completely solved yet!
- **Main points to take away are:**
  - Algebraic rules and their use in transformations of queries.
  - Deciding on join ordering: System-R style (Selinger style) optimization.
  - Estimating cost of plans and sizes of intermediate results.
Operations (revisited)

- Scan ([index], table, predicate):
  - Either index scan or table scan.
  - Try to push down sargable predicates.
- Selection (filter)
- Projection (always need to go to the data?)
- Joins: nested loop (indexed), sort-merge, hash, outer join.
- Grouping and aggregation (usually the last).

Algebraic Laws

- Commutative and Associative Laws
  - \( R \cup S = S \cup R, R \cup (S \cup T) = (R \cup S) \cup T \)
  - \( R \cap S = S \cap R, R \cap (S \cap T) = (R \cap S) \cap T \)
  - \( R \bowtie S = S \bowtie R, R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T \)
- Distributive Laws
  - \( R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T) \)

Laws involving selection:

- \( \sigma_{C \land C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R) \)
- \( \sigma_{C \lor C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R) \)
- \( \sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \)

  - When \( C \) involves only attributes of \( R \)
- \( \sigma_C(R - S) = \sigma_C(R) - S \)
- \( \sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_{C(S)} \)
- \( \sigma_C(R \cap S) = \sigma_C(R) \cap S \)

Example: \( R(A, B, C, D), S(E, F, G) \)

- \( \sigma_{F=3}(R \bowtie S) = ? \)
- \( \sigma_{A=5 \land G=9}(R \bowtie S) = ? \)

Laws involving projections

- \( \Pi_M(R \bowtie S) = \Pi_M(\Pi_M(R) \bowtie \Pi_M(S)) \)
  - Where \( N, P, Q \) are appropriate subsets of attributes of \( M \)
- \( \Pi_M(\Pi_M(R)) = \Pi_{M\cap M}(R) \)

Example \( R(A, B, C, D), S(E, F, G) \)

- \( \Pi_{A,B,C,D}(R \bowtie S) = \Pi_{A,B,C,D}(R) \bowtie \Pi_{E,F,G}(S) \)

Query Rewrites: Sub-queries

```
SELECT Emp.Name
FROM Emp
WHERE Emp.Age < 30
   AND Emp.Dept# IN
      (SELECT Dept.Dept#
       FROM Dept
       WHERE Dept.Loc = "Seattle"
          AND Emp.Emp#=Dept.Mgr)
```
The Un-Nested Query

```
SELECT Emp.Name
FROM Emp, Dept
WHERE Emp.Age < 30
    AND Emp.Dept#=Dept.Dept#
    AND Dept.Loc = "Seattle"
    AND Emp.Emp#=Dept.Mgr
```

Converting Nested Queries

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
AND x.price >= ALL (Select y.price
From product y
Where x.maker = y.maker
AND y.color= "blue")
```

How do we convert this one to logical plan?

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
AND x.price < SOME (Select y.price
From product y
Where x.maker = y.maker
AND y.color= "blue")
```

Let’s compute the complement first:

```
Select distinct x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
    AND y.color="blue" AND x.price < y.price
```

This one becomes a SFW query:

```
Select distinct x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
    AND y.color="blue" AND x.price < y.price
```

This returns exactly the products we DON’T want, so...

```
(Select x.name, x.maker
From product x
Where x.color = "blue")
EXCEPT
(Select x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
    AND y.color="blue" AND x.price < y.price)
```

Semi-Joins, Magic Sets

- You can’t always un-nest sub-queries (it’s tricky).
- But you can often use a semi-join to reduce the computation cost of the inner query.
- A magic set is a superset of the possible bindings in the result of the sub-query.
- Also called “sideways information passing”.
- Great idea; reinvented every few years on a regular basis.
Rewrites: Magic Sets

Create View DepAvgSal AS
(Select E.did, Avg(E.sal) as avgsal
From Emp E
Group By E.did)

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
And E.age < 30 and D.budget > 100k
And E.sal > V.avgsal

Rewrites: SIPS

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
And E.age < 30 and D.budget > 100k
And E.sal > V.avgsal

DepAvgSal needs to be evaluated only for
departments where V.did IN
Select E.did
From Emp E, Dept D
Where E.did=D.did
And E.age < 30 and D.budget > 100K

Supporting Views

1. Create View PartialResult as
(Select E.eid, E.sal, E.did
From Emp E, Dept D
Where E.did=D.did
And E.age < 30 and D.budget > 100K)

2. Create View Filter AS
Select DISTINCT P.did FROM PartialResult P.

2. Create View LimitedAvgSal as
(Select F.did Avg(E.Sal) as avgSal
From Emp E, Filter F
Where E.did=F.did
Group By F.did)

And Finally…

Transformed query:

Select P.eid, P.sal
From PartialResult P, LimitedAvgSal V
Where P.did=V.did
And P.sal > V.avgsal

Rewrites: Group By and Join

• Schema:
  – Product (pid, unitprice,…)
  – Sales(tid, date, store, pid, units)

• Trees:
  
<table>
<thead>
<tr>
<th>Group By (pid)</th>
<th>Sum (units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td></td>
</tr>
<tr>
<td>Products</td>
<td></td>
</tr>
<tr>
<td>Filter (in NW)</td>
<td></td>
</tr>
<tr>
<td>Filter(date in Q2,2000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group By (pid)</th>
<th>Sum (units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td></td>
</tr>
<tr>
<td>Products</td>
<td></td>
</tr>
<tr>
<td>Filter (in NW)</td>
<td></td>
</tr>
<tr>
<td>Filter(date in Q2,2000)</td>
<td></td>
</tr>
</tbody>
</table>

Rewrites: Operation Introduction

• Schema: (pid determines cid)
  – Category (pid, cid, details)
  – Sales(tid, date, store, pid, amount)

• Trees:
  
<table>
<thead>
<tr>
<th>Group By (pid)</th>
<th>Sum (amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td></td>
</tr>
<tr>
<td>Filter (…)</td>
<td></td>
</tr>
<tr>
<td>Filter (store IN {CA,WA})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group By (pid)</th>
<th>Sum (amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td></td>
</tr>
<tr>
<td>Filter (…)</td>
<td></td>
</tr>
<tr>
<td>Filter (store IN {CA,WA})</td>
<td></td>
</tr>
</tbody>
</table>
Query Rewriting: Predicate Pushdown

The earlier we process selections, less tuples we need to manipulate higher up in the tree. Disadvantages?

Query Rewrites: Predicate Pushdown (through grouping)

Select bid, Max(age) From Reserves R, Sailors S Where R.sid=S.sid GroupBy bid Having Max(age) > 40 GroupBy bid

• For each boat, find the maximal age of sailors who’ve reserved it.
• Advantage: the size of the join will be smaller.
• Requires transformation rules specific to the grouping/aggregation operators.
• Will it work if we replace Max by Min?

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

Then, move them down.

Query Rewrite Summary

• The optimizer can use any semantically correct rule to transform one query to another.
• Rules try to:
  – move constraints between blocks (because each will be optimized separately)
  – Unnest blocks
• Especially important in decision support applications where queries are very complex.
• In a few minutes of thought, you’ll come up with your own rewrite. Some query, somewhere, will benefit from it.
• Theorems?
Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
  - Depends on input cardinalities.
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
- We’ll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

Statistics and Catalogs

- 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 Model for Our Analysis

- As a good approximation, we ignore CPU costs:
  - B: The number of data pages
  - P: Number of tuples per page
  - D: (Average) time to read or write disk page
  - Measuring number of page I/O’s ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.

Simple Nested Loops Join

- For each tuple r in R do
  - for each tuple s in S do
    - if r == s then add <r, s> to result
      - For each tuple in the outer relation R, we scan the entire inner relation S.
      - Cost: M + (P_r * M) * N.
- Page-oriented Nested Loops join: For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N.

Index Nested Loops Join

- foreach tuple r in R do
  - foreach tuple s in S where r_i == s_i do
    - add <r, s> to result
  - If there is an index on the join column of one relation (say S), can make it the inner.
    - Cost: M + ((M*P_s) * cost of finding matching S tuples)
  - For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
    - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold “block” of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.
Sort-Merge Join \((R \bowtie S)\)

- Sort \(R\) and \(S\) on the join column, then scan them to do a “merge” on the join column.
  - Advance scan of \(R\) until current \(R\)-tuple \(\geq\) current \(S\)-tuple, then advance scan of \(S\) until current \(S\)-tuple \(\geq\) current \(R\)-tuple; do this until current \(R\) tuple = current \(S\) tuple.
  - At this point, all \(R\) tuples with same value and all \(S\) tuples with same value match; output \(<r, s>\) for all pairs of such tuples.
  - Then resume scanning \(R\) and \(S\).

**Cost of Sort-Merge Join**

- \(R\) is scanned once; each \(S\) group is scanned once per matching \(R\) tuple.
- Cost: \(M \log M + N \log N + (M+N)\)
  - The cost of scanning, \(M+N\), could be \(M \times N\) (unlikely!)

Hash-Join

- Partition both relations using hash fn \(h\). \(R\) tuples in partition \(i\) will only match \(S\) tuples in partition \(i\).
- Read in a partition of \(R\), hash it using \(h2(\leftrightarrow h1)\). Scan matching partition of \(S\), search for matches.

**Cost of Hash-Join**

- In partitioning phase, read+write both relations; \(2(M+N)\). In matching phase, read both relations; \(M+N\) I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of \(3(M+N)\) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Size Estimation and Reduction Factors

- Consider a query block:
  - 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 \((\text{High}(I)-\text{value})/(\text{High}(I)-\text{Low}(I))\)

Histograms

- Key to obtaining good cost and size estimates.
- Come in several flavors:
  - Equi-depth
  - Equi-width
- Which is better?
- Compressed histograms: special treatment of frequent values.
**Histograms**

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

**Employee(ssn, name, salary, phone)**

Maintain a histogram on salary:

- T(Employee) = 25000, but now we know the distribution

<table>
<thead>
<tr>
<th>Salary</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..20k</td>
<td>200</td>
</tr>
<tr>
<td>20k..40k</td>
<td>800</td>
</tr>
<tr>
<td>40k..60k</td>
<td>6500</td>
</tr>
<tr>
<td>60k..80k</td>
<td>12000</td>
</tr>
<tr>
<td>&gt;80k</td>
<td>6500</td>
</tr>
<tr>
<td>&gt;100k</td>
<td>500</td>
</tr>
</tbody>
</table>

- T(Employee) = 25000, but now we know the distribution

---

**Histgrams**

**Ranks(rankName, salary)**

- Estimate the size of Employee >= salary Ranks

<table>
<thead>
<tr>
<th>Employee</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>&gt;80k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>800</td>
<td>6500</td>
<td>12000</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ranks</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>&gt;80k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>30</td>
<td>80</td>
<td>80</td>
<td>2</td>
</tr>
</tbody>
</table>

- Assume: 
  - V(Employee, Salary) = 200
  - V(Ranks, Salary) = 250
- Then T(Employee >= salary Ranks) =
  \[ T_{i=1}^{16} T_i T_i' / 250 \]
  \[ = (200x8 + 800x20 + 5000x40 + 12000x80 + 6500x100 + 500x2)/250 \]

---

**Plans for Single-Relation Queries**

(Prep for Join ordering)

- **Task**: create a query execution plan for a single Select-project-group-by block.
- **Key idea**: consider each possible access path to the relevant tuples of the relation. Choose the cheapest one.
- 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).

**Example**

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

- **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 (≈ 55).
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) pages are retrieved.
- **Index on sid**: 
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500.
- **Doing a file scan**: we retrieve all file pages (500).
Determining Join Ordering

- $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
- Join tree:

Types of Join Trees

- Left deep:

Types of Join Trees

- Bushy:

Types of Join Trees

- Right deep:

Problem

- Given: a query $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of $\{R_1, \ldots, R_n\}$, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for $\{R_1\}, \{R_2\}, \ldots, \{R_n\}$
  - Step 2: for $\{R_1,R_2\}, \{R_1,R_3\}, \ldots, \{R_{n-1}, R_n\}$
  - ...
  - Step n: for $\{R_1, \ldots, R_n\}$
- A subset of $\{R_1, \ldots, R_n\}$ is also called a subquery
Dynamic Programming

- For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$
  - compute the following:
    - $\text{Size}(Q)$
    - A best plan for $Q$: $\text{Plan}(Q)$
    - The cost of that plan: $\text{Cost}(Q)$

**Step 1**: For each $\{R_i\}$ do:

- $\text{Size}(\{R_i\}) = B(R_i)$
- $\text{Plan}(\{R_i\}) = R_i$
- $\text{Cost}(\{R_i\}) = \text{cost of scanning } R_i$

**Step i**: For each $Q \subseteq \{R_1, \ldots, R_n\}$ of cardinality $i$ do:

- Compute $\text{Size}(Q)$ (later…)
- For every pair of subqueries $Q'$, $Q''$
  - s.t. $Q = Q' \cup Q''$
  - compute $\text{cost}(\text{Plan}(Q') \text{ Plan}(Q''))$
- $\text{Cost}(Q) = \text{the smallest such cost}$
- $\text{Plan}(Q) = \text{the corresponding plan}$

**Step n**: $\text{Return Plan}(\{R_1, \ldots, R_n\})$

Summary: computes optimal plans for subqueries:

- **Step 1**: $\{R_1\}$, $\{R_2\}$, $\ldots$, $\{R_n\}$
- **Step 2**: $\{R_1, R_2\}$, $\{R_1, R_3\}$, $\ldots$, $\{R_{n-1}, R_n\}$
- $\ldots$
- **Step n**: $\{R_1, \ldots, R_n\}$

- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next)
  - heuristics for Reducing the Search Space
  - Restrict to left linear trees
  - Restrict to trees "without cartesian product"
  - need more than just one plan for each subquery:
    - "interesting orders"

Completed Physical Query Plan

- Choose algorithm to implement each operator
  - Need to account for more than cost:
    - How much memory do we have ?
    - Are the input operand(s) sorted ?

- Decide for each intermediate result:
  - To materialize
  - To pipeline