DBMS Internals
Execution and Optimization
May 10th, 2004

Agenda
- Questions on phase 2 of the project
- Today: DBMS internals part 2 —
  - Query execution
  - Query optimization
- Next week:
  - Thursday, not Monday.
  - Mostly Phil Bernstein on meta-data management.

Query Execution
- Query compiler
- Execution engine
- Index/record manager
- Buffer manager
- Storage manager

User/Application
- Query
- Record, index requests
- Read/write pages

Query Execution Plans
- SELECT S.sname
  FROM Purchase P, Person Q
  WHERE P.buyer=Q.name AND
  Q.city='seattle' AND
  Q.phone > '5430000'

Query Plan:
- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan, group) are not.

The Leaves of the Plan: Scans
- Table scan: iterate through the records of the relation.
- Index scan: go to the index, from there get the records in the file (w/ when would this be better?)
- Sorted scan: produce the relation in order. Implementation depends on relation size.

How do we combine operations?
- The front-end code. Each operation is in plan entered by 3 functions:
  - Open sets up the data structures and performs initializations
  - GetNext returns the next tuple of the result.
  - Close ends the operation. Clean up the data structures.
- Enables pipelining!
- Contributes with data-driven materialization code.
- Some 3-tuples it's the same (e.g., sorted scan).
Implementing Relational Operations

We will consider how to implement:

- **Selection (S)**: Selects a subset of rows from a relation.
- **Projection (P)**: Deletes unwanted columns from a relation.
- **Join (⋈)**: Allows us to combine two relations.
- **Set-difference**: Tuples in relation 1, but not in relation 2.
- **Union**: Tuples in relation 1 and in relation 2.
- **Aggregation** (SUM, AVG, etc.) and GROUP BY

### Schema for Examples

- **Purchase (buyer: string, seller: string, product: integer)**
- **Person (name: string, city: string, phone: integer)**

### Simple Selections

**SELECT * FROM Person P WHERE P.phone < '543%'**

**Result size estimation:**

\[ \text{Size of } P \times \text{reduction factor} \]

**M can be this later.**

### Using an Index for Selections

- **Cost** depends on number of qualifying tuples and clustering.
  - Cost of finding qualifying data entries typically an all plus cost of retrieving records.
  - In example, assuming uniform distribution of phones, about 54% of tuples qualify. 8000 pages, 50000 tuples. With a clustered index, cost is little more than 500 I/Os; if unclustered, up to 50000 I/Os!

**Important refinement for unclustered indexes:**

1. Find and sort the RIDs of the qualifying data entries.
2. Fetch RIDs in order. This ensures that each data page is looked at just once (though it is likely to be higher with clustering).

### Two Approaches to General Selections

**First approach:** Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don’t match the index:

- Cost is dependent on number of qualifying tuples and clustering.
- In example, assuming uniform distribution of phones, about 54% of tuples qualify. 8000 pages, 50000 tuples. With a clustered index, cost is little more than 500 I/Os; if unclustered, up to 50000 I/Os!

**Second approach:**

1. Get sets of RIDs of data records using each matching index.
2. Then intersect these sets of RIDs.
3. Retrieve the records and apply any remaining terms.
Implementing Projection

Two parts:
1. Remove unwanted attributes.
2. Remove duplicates from the result.

Refinements to duplicate removal:
- If an index on a relation contains all wanted attributes, then we can do an index-only scan.
- If the index contains a subset of the wanted attributes, you can remove duplicates locally.

Equality Joins With One Join Column

\[ \text{SELECT DISTINCT } \]
\[ \text{R.name, R.phone } \]
\[ \text{FROM Person R, Purchase S} \]
\[ \text{WHERE R.name=S.buyer} \]

Discussion

How would you implement join?

Simple Nested Loops Join

For each tuple \( r \) in \( R \) do
   for each tuple \( s \) in \( S \) do
      if \( r_i = s_j \) then add \( <r, s> \) to result

Index Nested Loops Join

For each tuple \( r \) in \( R \) do
   foreach tuple \( s \) in \( S \) where \( n = s_i \) 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:
- \( \text{Cost: } N + [\frac{M \cdot p_R}{p_S}] + \text{cost of finding matching } S \text{ tuples} \)
- 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.

Examples of Index Nested Loops

- Hash-index on name of Person (as inner):
  - Scan Purchase: 1000 page I/Os, 100*1000 tuples.
  - For each Person tuple: 1.2 I/Os to find index entry in index, plus 1 I/O to get the exactly one matching Person tuple. Total: 220,000 I/Os, 36 minutes

- Hash-index on buyer of Purchase (as inner):
  - Scan Person: 500 page I/Os, 80*500 tuples.
  - For each Person tuple: 1.2 I/Os to find index entry with data entries, plus cost of retrieving matching Purchase tuples. Assume uniform distribution, 2.5 purchases per buyer (100,000/40,000). Cost of retrieving them in 1 to 2.5 I/Os depending on clustering.
**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 ⊓ 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 >= 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 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: $M \log M + N \log N + (M+N)$
- But really, we can do it in $3(M+N)$ with some trickery.
  - The cost of scanning $M+N$, could be $M \cdot N$ (unlikely?)
- With 35, 100 or 300 buffer pages, both Person and Purchase can be sorted in 2 passes; total: 7500 (75 seconds).

**Cost of Hash-Join**

- In partitioning phase, read-write both relations; $2(M+N)$. In matching phase, read both relations; $M+N$ I/Os.
- In our running example, this is a total of 4500 I/Os. (45 seconds!)
- Sort-Merge Join vs. Hash Join:
  - Given a min join, an output on an empty outer have a cost of $3(M+N)$ I/Os. Hash Join superior on this count if relation sizes differ greatly. A lae, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

**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 (<> h1). Scan matching partition of S, search for matches.

**Double Pipelined Join (Tukwila)**

- Partially pipelined no output until lower read
- Assumes static: Square vs. outer - optimalization requires source behavior knowledge
- Outputs data in an efficient
- Requires less source knowledge to optimize
Query Optimization

Discussion

• How would you build a query optimizer?

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).

Building Blocks

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

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:

- $s_{C \cap \neg C}(R) = s_C(R) \cap s_{\neg C}(R)$
- $s_{C \cup \neg C}(R) = s_C(R) \cup s_{\neg C}(R)$
- $s_C(R \bowtie S) = s_C(R) \bowtie S$
  - When $C$ involves only attributes of $R$
  - $s_C(R \setminus S) = s_C(R) \setminus S$
  - $s_C(R \bowtie S) = s_C(R) \bowtie s_C(S)$
  - $s_C(R \cap S) = s_C(R) \cap S$

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

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

Laws involving projections

- $PM(R \bowtie S) = PN(PP(R) \bowtie PQ(S))$
  - Where $N, P, Q$ are appropriate subsets of attributes of $M$
- $PM(PN(R)) = PM,N(R)$

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

- $PA,B,G(R \bowtie S) =$

Query Rewrites: Sub-queries

```sql
SELECT Emp.Name
FROM Emp
WHERE Emp.Age < 30
AND Emp.DeptIn IN
(SELECT Dept.DeptID
FROM Dept
WHERE DeptLoc = "Seattle"
AND Emp.EmpID=Dept.Mgr)
```

The Un-Nested Query

```sql
SELECT Emp.Name
FROM Emp, Dept
WHERE Emp.Age < 30
AND Emp.DeptID=Dept.DeptID
AND DeptLoc = "Seattle"
AND Emp.EmpID=Dept.Mgr
```
Converting Nested Queries

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

Converting Nested Queries

Let's compute the complement first:

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

This becomes a SFW query:

```sql
Select distinct x.name, x.make
From product x, product y
Where x.color = 'blue'
AND x.make = y.make
AND y.color = 'blue'
AND x.price < y.price
```

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

Semi-Joins, Magic Sets

- You can't always unnest 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.
- A term called "sideways information passing".
- Great idea; reinvented every few years on a regular basis.

Rewrites: Magic Sets

Create View DepAvgSal AS

```sql
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 V did
From Emp E, Dept D
Where E.did=D.did
And E.age < 30 and D.budget > 100K

Supporting Views

1. Create View PartialResults
   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 FilterAS
   Select DISTINCT F.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:
  Join
  groupby(pid)
  Sum(units)
  Scan(Sales)
  Filter(date in Q2,2000)
  Products
  Filter (in NW)

  Join
  groupby(pid)
  Sum(units)
  Scan(Sales)
  Filter(store IN {CA, WA})

Rewrites: Operation Introduction

- Schema:
  - Category (pid, cid, details)
  - Sales(tid, date, store, pid, amount)

- Trees:
  Join
  groupby(cid)
  Sum(amount)
  Scan(Sales)
  Filter(store IN {CA, WA})

Schem a for Some Examples

Sailors (sid: integer, sname: string, rating: integer, age: real)
Reserves (sid: integer, bid: 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).
Query Rewriting: Predicate Pushdown

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

Advantages?

Disadvantages?

Query Rewrite: Predicate Movearound

Select sid, date
From V1, V2
Where V1.rating = V2.rating and V1.age = V2.age

Create View V1 AS
Select rating, Min(age)
From Sailors S
Where S.age < 20
Group By rating

Create View V2 AS
Select sid, rating, age, date
From Sailors S, Reserves R
Where R.sid=S.sid

Select sid, date
From V1, V2
Where V1.rating = V2.rating and V1.age = V2.age, age < 20

Sailing with 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.

Theorem 20?
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 okay in practice.
  - More sophisticated techniques known now.

Statistics and Catalogs

- Need information about the relations and indexes involved. Catalogs typically contain:
  - # tuples N(Tuples) and # pages N(Pages) for each relation.
  - # distinct key values N(Keys) and N(Pages) 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 okay, slight inconsistency okay.
  - More detailed information (e.g., histograms of the values in some field) are sometimes stored.

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/N(Keys(I)), given index I on col.
  - Term col1=col2 has RF 1/MAX(N(Keys(I1),N(Keys(I2)))
  - Term col1>value has RF (High(I)-value)/(High(I)-Low(I))

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

Histograms

- Key to obtaining good cost and size estimates.
- Come in several flavors:
  - Equi-depth
  - Equi-width
- Which is better?
  - Compare histogram sizes: special treatment of frequent values.

Employee(ssn, name, salary, phone)

Maintain a histogram on salary:

<table>
<thead>
<tr>
<th>Salary</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0-20k</td>
<td>500</td>
</tr>
<tr>
<td>$20.4k-40k</td>
<td>500</td>
</tr>
<tr>
<td>$40.4k-60k</td>
<td>500</td>
</tr>
<tr>
<td>$60.4k-80k</td>
<td>500</td>
</tr>
<tr>
<td>$80k-100k</td>
<td>500</td>
</tr>
<tr>
<td>&gt; 100k</td>
<td>500</td>
</tr>
</tbody>
</table>

Employee(ssn, name, salary, phone) = 25000, but now we know the distribution.
### Plans for Single-Relation Queries

**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

<table>
<thead>
<tr>
<th>Salary</th>
<th>Employee</th>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 100k</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>80k..100k</td>
<td>100</td>
<td>6500</td>
</tr>
<tr>
<td>60k..80k</td>
<td>80</td>
<td>12000</td>
</tr>
<tr>
<td>40k..60k</td>
<td>40</td>
<td>5000</td>
</tr>
<tr>
<td>20k..40k</td>
<td>20</td>
<td>800</td>
</tr>
<tr>
<td>0..20k</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Select the `salary` from the `Sailors S` where `S.rating = 8`

- **Determining Join Ordering**
  - **Left-deep:**

- **Types of Join Trees**
  - **Left-deep:**
Types of Join Trees

• Bushy:

![Bushy Join Tree Diagram]

• Right deep:

![Right Deep Join Tree Diagram]

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:
- Size(Q)
- A best plan for Q: Plan(Q)
- The cost of that plan: Cost(Q)

Dynamic Programming

Step 1: For each $\{R_i\}$ do:
- $\text{Size}(\{R_i\}) = B(R_i)$
- $\text{Plan}(\{R_i\}) = R_i$
- $\text{Cost}(\{R_i\}) = \text{cost of scanning } R_i$
Dynamic Programming

- **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 $Q', Q''$
    - $s_{ij} = Q' \cup Q''$
    - Compute $\text{cost}(\text{Plan}(Q') \times \text{Plan}(Q''))$
  - $\text{Cost}(Q)$ = the smallest such cost
  - $\text{Plan}(Q)$ = the corresponding plan

Dynamic Programming

**Summary:** computes optimal plans for subqueries:
- **Step 1:** $\{R_1\}, \{R_2\}, \ldots, \{R_n\}$
- **Step 2:** $\{R_1, R_2\}, \{R_1, R_3\}, \ldots, \{R_{n-1}, R_n\}$
- ...
- **Step n:** $\{R_1, \ldots, R_n\}$

- We used naive 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"