Relational Query Optimization

Yanlei Diao
UMass Amherst
April 1 and 3, 2007

Slide Content Courtesy of R. Ramakrishnan, J. Gehrke, and J. Hellerstein
Overview of Query Evaluation

- **Query Evaluation Plan**: tree of relational algebra (R.A.) operators, with choice of algorithm for each operator.

- Three main issues in query optimization:
  - Plan space: for a given query, what plans are considered?
  - Plan cost: how is the cost of a plan estimated?
  - Search algorithm: search the plan space for the cheapest (estimated) plan.

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

```
SELECT   {DISTINCT} <list of columns>
FROM      <list of relations>
{WHERE    <list of "Boolean Factors">}
{GROUP BY <list of columns>}
{HAVING   <list of Boolean Factors>}
{ORDER BY <list of columns>};
```

- **Query Semantics:**
  - Take Cartesian product (a.k.a. cross-product) of relns in FROM
  - If a WHERE clause exists, apply all filters in it
  - If a GROUP BY clause exists, form groups on the retained tuples
  - If a HAVING clause exists, filter groups with it
  - If an ORDER BY clause exists, make sure tuples are output in the right order
  - Project tuples onto a set of columns. If there is a DISTINCT modifier, remove duplicates
Basics of Query Optimization

- Convert the WHERE condition to conjunctive normal form:
  - \((\text{day}<8/9/94 \text{ OR bid}=5 \text{ OR sid}=3) \text{ AND } (\text{rname}=\text{'Paul'} \text{ OR sid}=3)\)
  - Why not disjunctive normal form?
- Interleave FROM and WHERE into a plan tree for optimization.
- Apply GROUP BY, HAVING, DISTINCT and ORDER BY at the end, pretty much in that order.
Schema for Examples

Sailors \((sid: \text{integer}, \ sname: \text{string}, \ rating: \text{integer}, \ age: \text{real})\)
Reserves \((sid: \text{integer}, \ bid: \text{integer}, \ day: \text{dates}, \ rname: \text{string})\)

- **Sailors:**
  - 80 tuples per page, 500 pages.

- **Reserves:**
  - 100 tuples per page, 1000 pages.
The algebraic expression partially specifies how to evaluate the query:

- Compute the natural join of Reserves and Sailors
- Perform the selections
- Project the `sname` field

Expression in Relational Algebra (RA):

\[\pi_{\text{sname}} (\sigma_{\text{bid}=100 \land \text{rating}>5} (\text{Reserves} \bowtie_{\text{sid}=\text{sid}} \text{Sailors}))\]
Query Evaluation Plan

- **Query evaluation plan** is an extended RA tree, with additional annotations:
  - *access method* for each relation;
  - *implementation method* for each relational operator.

- Cost of the query plan (I/Os):
  - \[500 + 500 \times 1000 = 500,500\] I/Os

- Missed several opportunities:
  - Selections could have been `pushed` earlier.
  - No use is made of any available indexes.
  - More efficient join algorithm…
Relational Algebra Equivalences

- Allow us to (1) choose different join orders and to (2) ‘push’ selections and projections ahead of joins.

- **Selections:** \( \sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \) (Cascade)
  \[\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))\] (Commutative)

- **Joins:** \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T\) (Associative)
  \[ (R \bowtie S) \equiv (S \bowtie R)\] (Commutative)

- Show that: \( R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S\)
More Equivalences

- A projection $\pi$ commutes with a selection $\sigma$ that only uses attributes retained by $\pi$.

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

- Selection between attributes of the two relations of a cross-product converts cross-product to a join.

$$\sigma_c(R \times S) \equiv R \bowtie c S$$

- A selection on attributes of $R$ commutes with $R \bowtie S$.

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$
Alternative Plan 1 (Selection Pushed Down)

- **Push selections below the join.**
  - Can use index scan to retrieve tuples, e.g. hash index or B+tree index, and then feed them to the join.
  - Retrieve 100,000/100 = 1000 tuples (assume 100 boats).
  - Clustered index: 1000/100=10 I/Os
  - Unclustered index: 1000 I/Os

- **Cost of the query plan:**
  - Selection of Reserves tuples: 10 I/Os
  - Write tuples to a temp file: 10 I/Os
  - Join: 10+10*500 = 5,010 I/Os
  - Total = 5,030 I/Os
Access Methods

- An access method (path) is a method of retrieving tuples:
  - File scan, or index scan with the search key matching selection conditions in the query.
- Generally, a tree index matches a conjunction of terms if the attributes in the terms form a prefix of the search key.
  - E.g., Tree index on <a, b, c> matches the selection $a=5 \text{ AND } b=3$, and $a=5 \text{ AND } b>6$, but not $b=3$. Why?
- A hash index matches a conjunction of terms if there is an equality term $attribute = value$ for every attribute in the search key of the index.
  - E.g., Hash index on <a, b, c> matches $a=5 \text{ AND } b=3 \text{ AND } c=5$; but it does not match $b=3$, or $a=5 \text{ AND } b=3$, or $a>5 \text{ AND } b=3 \text{ AND } c=5$. 
Alternative Plan 2 (Using Indexes)

- **Selection using index**: retrieving 1000 tuples in 10 pages.

- **Indexed NLJ**: pipelining the outer and indexed lookup on the inner.
  - The outer: scanned only once, pipelining.
  - The inner: join column `sid` is a key for Sailors; *at most one* matching tuple, unclustered index on `sid` OK.

- **Cost of the query plan**:
  - Selection of Reserves tuples (10 I/Os).
  - For each, must get matching Sailors tuple (1000*(1.2+1)).
  - Total = 10+1000*1.2 = 2,210 I/Os.
Cost Estimation

- For each plan considered, must estimate its cost.
- Estimate cost of each operation in a plan tree:
  - Need cardinality (# tuples) and size (# pages) of each input relation.
  - We’ve discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
- Estimate size of result for each operation in tree:
  - Produce cardinality and size for cost estimation of a downstream operator.
Statistics in System Catalog

- Statistics about each relation (R) and index (I):
  - **Cardinality**: # tuples (NTuples) in R.
  - **Size**: # pages (NPages) in R.
  - **Index Cardinality**: # distinct key values (NKeys) in I.
  - **Index Size**: # pages (INPages) in I.
  - **Index height**: # nonleaf levels (IHeight) of I.
  - **Index range**: low/high key values (Low/High) in I.
  - More detailed info, e.g., histograms, …
Reduction Factors

Consider a query block:

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

Reduction factor (RF) or Selectivity of each term reflects the impact of the term in reducing result size.

- **Assumption 1:** uniform distribution of the values!
- Column = value: RF = \(1/N\text{Keys}(I)\), given index I on the column
- Column > value: RF = \((\text{High}(I) - \text{value})/(\text{High}(I) - \text{Low}(I))\)
- Column1 = Column2: RF = \(1/\text{MAX}(N\text{Keys}(I1), N\text{Keys}(I2))\)
  - Each value from R with the smaller index I1 has a matching value in S with the larger index I2.
  - Values in S are evenly distributed.
  - So each R tuple has \(N\text{Tuples}(S)/N\text{Keys}(I2)\) matches, a RF of \(1/N\text{Keys}(I2)\).
Size Estimation

- Consider a query block and effects of all terms:

- Max. number of tuples in result = the product of the cardinalities of relations in the FROM clause.

- Result cardinality = Max # tuples * product of Reduction Factors of all (conjunctive) terms.
  - Assumption 2: terms are independent!
Plans for Single Relation Queries

- Queries over a single relation can consist of selection, projection, and aggregation.

- **Enumeration of alternative plans:**
  - Each available access path (file/index scan) is considered, the one with least estimated cost is chosen.
  - The various operations are often carried out together:
    - If an index is used for a selection, projection can be done for each retrieved tuple.
    - The resulting tuples can be pipelined into the aggregate computation in the absence of GROUP BY; otherwise, hashing or sorting is needed for GROUP BY.
Cost Estimates for Single-Relation Plans

- **Index I on primary (candidate) key** matches selection:
  - Cost of lookup = $\text{Height}(I)+1$ for a B+ tree, $\approx 1.2$ for hash index.
  - Cost of record retrieval = 1

- **Clustered index I** matching one or more selections:
  - Cost of lookup + $$(\text{INPages'}(I)+\text{NPages}(R)) \times \text{product of RF's of matching selections}.$$ (Treat INPages' as the number of leaf pages in the index.)

- **Non-clustered index I** matching one or more selections:
  - Cost of lookup + $$(\text{INPages'}(I)+\text{NTuples}(R)) \times \text{product of RF's of matching selections}.$$  

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

- May add extra costs for GROUP BY and duplicate elimination in projection (if a query says DISTINCT).
Plans for Multiple Relation Queries

- As the number of joins increases, the number of alternative plans grows rapidly.
- Intuition: use only left-deep join trees
  - Join relns one at a time with inner as a base relation.
Plans for Multiple Relation Queries

- As the number of joins increases, the number of alternative plans grows rapidly.
- Intuition: use only left-deep join trees
  - Join relns one at a time with inner as a base relation.
  - Allow pipelining: intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined!
    - E.g., sort-merge join (the sorting phase), two-phase hash join (the partitioning phase)

Diagram:

- Left-deep tree structure with nodes A, B, C, D
Plan Space

- For each query block, the plans considered are:
  - All *available access methods*, for each reln in the FROM clause.
  - All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
    - Considering all permutations of N relns!
    - Avoid cartesian products before exhausting join predicates!
      e.g. R.a = S.a and R.b = T.b, how many left-deep trees?
  - All *join methods*, for each join in the tree.
  - *Appropriate places for selections and projections.*
Place Search

- Left-deep join plans differ in:
  - the order of relations,
  - the access path for each relation, and
  - the join method for each join.

- Many of these plans share subplans, so don’t enumerate all of them. This is a job for...

- **Dynamic Programming**
  
  “a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructures that takes much less time than naive methods.”
Enumeration of Left-Deep Plans

- Enumerate using \( N \) passes (if \( N \) relations joined):
  - Pass 1: Find best 1-relation plan for each relation, including file and index scans.
  - Pass 2: Find best ways to join result of each 1-relation plan (as outer) to another relation.  (All 2-relation plans.)
  - ...
  - Pass \( N \): Find best ways 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 unordered plan, and
  - cheapest plan for each interesting order of the tuples, and discard all others.
Outline

- Query Optimization
- Index Design and Tuning
Decisions to Make

- What are the common query workloads?
  - Common selection criteria
  - Common join operators

- What indexes should we create?
  - Which relations should have indexes?
  - What field(s) should be the search key?
  - Should we build several indexes?
  - For each index, what kind of an index should it be?
    - Hash/Tree? Clustered?

- Our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. Have to understand optimizers!
Index Selection for Joins

- When considering a join condition:
  - **Hash index** on inner is very good for Index Nested Loops join.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - **Clustered B+ tree** on join column(s) is good for Index Nested Loops join and Sort-Merge join.
  - **Unclustered B+ tree** is good for Index Nested Loops join if it is on the primary key of the inner relation.

(We discussed indexes for single-table queries in query optimization.)
Example 1

Hash index on \( D.dname \) supports ‘Toy’ selection.
- Given this, index on \( D.dno \) is not needed.
- Because selection makes the join index inapplicable.

Hash index on \( E.dno \) allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.

```sql
SELECT E.ename, D.mgr
FROM   Emp E, Dept D
WHERE  D.dname='Toy' AND E.dno=D.dno
```
### Example 2

<table>
<thead>
<tr>
<th>SELECT E.ename, D.mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM Emp E, Dept D</td>
</tr>
<tr>
<td>WHERE D.dname='Toy' AND E.dno=D.dno</td>
</tr>
</tbody>
</table>

- **What if WHERE included:    `... AND E.age=25`?**
  - Could retrieve Emp tuples using index on *E.age*, then join with Dept tuples satisfying *dname* selection.
  - So, if *E.age* index is already created, this query provides much less motivation for adding an *E.dno* index.

- **Summary:**
  - Use only one between selection and join indexes over one relation;
  - Use selection index if it makes a join input very small.
Example 3

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
    AND E.hobby='Stamps'
    AND E.dno=D.dno
```

- Clearly, Emp should be the outer relation.
  - Suggests that we build a hash index on \textit{D.dno}.
- What index should we build on Emp?
  - B+ tree on \textit{E.sal} or an index on \textit{E.hobby} could be used.
  - Which is better depends upon selectivity of the conditions.
  - As a rule of thumb, equality selections more selective than range selections.
Clustering and Joins

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
```

- Clustering is especially important when accessing inner tuples in INL.
  - Should make index on E.dno clustered.

- Suppose that the WHERE clause is instead:
  ```
  WHERE E.hobby='Stamps' AND E.dno=D.dno
  ```
  - If many employees collect stamps, Sort-Merge join may be worth considering. A clustered index on D.dno would help.

- **Summary:** Clustering is useful whenever many tuples are to be retrieved.
Create Index in SQL

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name ON table (column [,...n])

- Only table owner can create indexes on that table.
- The owner of a table can create an index at any time, whether or not there is data in the table.
- Other features:
  - Can add DESC or ASC keyword after each column
  - Can specify the fill factor, …
Outline

- Query Optimization
- Index Design and Tuning
- Tuning Queries and Views
Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-built, or if statistics are too old.

- Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:
  - Selections involving null values.
  - Selections involving arithmetic or string expressions.
  - Selections involving OR conditions.
  - Lack of evaluation features like index-only strategies or certain join methods or poor size estimation.

- Check the plan that is being used! Then adjust the choice of indexes or rewrite the query/view.
Rewriting SQL Queries

- **Guideline:** Use only one “query block”, if possible.

```sql
SELECT DISTINCT *
FROM Sailors S
WHERE S.sname IN
    (SELECT Y.sname
     FROM YoungSailors Y)
```

- Not always possible ...

```sql
SELECT *
FROM Sailors S
WHERE S.sname IN
    (SELECT DISTINCT Y.sname
     FROM YoungSailors Y)
```

- Complicated by interaction of **NULLs, duplicates, aggregation, subqueries.**
More Guidelines for Query Tuning

- Minimize the use of **DISTINCT**: don’t need it if duplicates are acceptable, or if answer contains a key.
- Minimize the use of **GROUP BY** and **HAVING**:

```
SELECT MIN (E.age) FROM Employee E
GROUP BY E.dno
HAVING E.dno = 102
```

```
SELECT MIN (E.age) FROM Employee E WHERE E.dno = 102
```
Guidelines for Query Tuning (Contd.)

- Avoid using intermediate relations:

  ```sql
  SELECT E.dno, AVG(E.sal)
  FROM Emp E, Dept D
  WHERE E.dno=D.dno
  AND D.mgrname='Joe'
  GROUP BY E.dno
  ```

  vs.

  ```sql
  SELECT T.dno, AVG(T.sal)
  FROM Temp T
  GROUP BY T.dno
  ```

- Does not materialize the intermediate reln Temp.

- If there is a dense B+ tree index on <dno, sal>, an index-only plan can be used to avoid retrieving Emp tuples in the second query!
Questions