Relational Query Optimization

UMass Amherst
March 23, 2010
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?
    - Huge number of alternative, semantically equivalent plans.
  - **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**

```sql
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:**
  1. Take Cartesian product (a.k.a. cross-product) of relns in FROM, projecting only to those columns that appear in other clauses
  2. If a WHERE clause exists, apply all filters in it
  3. If a GROUP BY clause exists, form groups on the result
  4. If a HAVING clause exists, filter groups with it
  5. If an ORDER BY clause exists, make sure output is in the right order
  6. If there is a DISTINCT modifier, remove duplicates
Basics of Query Optimization

- Convert selection conditions to **conjunctive normal form (CNF)**:
  - \((\text{day}<8/9/94 \text{ OR } \text{bid}=5 \text{ OR } \text{sid}=3) \text{ AND } (\text{rname}='Paul' \text{ OR } \text{sid}=3)\)

- 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.
Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of query blocks, and these are optimized one block at a time.

- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (More discussion later.)

```
SELECT S.sname
FROM   Sailors S
WHERE  S.age IN
       (SELECT MAX (S2.age)
        FROM   Sailors S2
        GROUP BY S2.rating)
```
System Catalog

- System information: buffer pool size and page size.
- For each relation:
  - relation name, file name, file structure (e.g., heap file)
  - attribute name and type of each attribute
  - index name of each index on the relation
  - integrity constraints...
- For each index:
  - index name and structure (B+ tree)
  - search key attribute(s)
- For each view:
  - view name and definition
System Catalog (Contd.)

- 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). More on this later…

- **Statistics updated periodically.**
  - Updating whenever data changes is costly; lots of approximation anyway, so slight inconsistency ok.

- Intensive use in query optimization! Always keep the catalog in memory.
Schema for Examples

Sailors \((s\text{id}: \text{integer}, \ s\text{name}: \text{string}, \ r\text{ating}: \text{integer}, \ a\text{ge}: \text{real})\)
Reserves \((s\text{id}: \text{integer}, \ b\text{id}: \text{integer}, \ d\text{ay}: \text{dates}, \ r\text{name}: \text{string})\)

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

\[ \pi_{sname}(\sigma_{bid=100 \land rating>5}(Reserves \bowtie_{sid=sid} Sailors)) \]
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:** $500 + 500 \times 1000$ I/Os

- **Misses several opportunities:**
  - Selections could have been ‘pushed’ earlier.
  - No use is made of any available indexes.
  - More efficient join algorithm…

```
\begin{align*}
\text{Reserves} & \quad \text{Sailors} \\
\text{(File scan)} & \quad \text{(File scan)} \\
\text{sid=sid} & \quad \text{bid=100} \\
\text{(Simple Nested Loops)} & \quad \text{(On-the-fly)} \\
\text{rating > 5} & \quad \text{sname} \\
\text{(On-the-fly)} & \quad \prod
\end{align*}
```
Relational Algebra Equivalences

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

- **Selections:** \( \sigma_{c_1 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \) (Cascade)

- \( \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \) (Commute)

- **Projections:** \( \pi_{a_1}(R) \equiv \pi_{a_1}(\ldots (\pi_{a_n}(R))) \) (Cascade)

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

\( (R \bowtie S) \equiv (S \bowtie R) \) (Commute)
More Equivalences

- A projection $\pi$ commutes with a selection $\sigma$ that only uses attributes retained by $\pi$, i.e., $\pi_a(\sigma_c(R)) = \sigma_c(\pi_a(R))$.

- Selection between attributes of the two relations of a cross-product converts cross-product to a join, i.e., $\sigma_c(R \times S) = R \bowtie_c S$

- A selection on attributes of $R$ commutes with $R \bowtie S$, i.e., $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$.

- Similarly, if a projection follows a join $R \bowtie S$, we can `push' it by retaining only attributes of $R$ (and $S$) that are (1) needed for the join or (2) kept by the projection.
Alternative Plan 1 (Selection Pushed Down)

- **Push selections below the join.**
- **Materialization:** store a temporary relation T, if the subsequent join needs to scan T multiple times.
  - The opposite is pipelining.

- 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-Merge join: Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250), total = 4060 page I/Os.
  - BNL join: join cost = 10+4*250, total cost = 2770.
Alternative Plan 2 (Using Indexes)

- **Selection using index**: clustered index on *bid* of Reserves.
  - Retrieve 100,000/100 = 1000 tuples in 1000/100 = 10 pages.

- Indexed NLJ: **pipelining** the outer and **indexed lookup** on the inner.
  - The outer: scanned only once, pipelining, no need to materialize.
  - The inner: join column *sid* is a key for Sailors; at most one matching tuple, unclustered index on *sid* OK.

- Push *rating* > 5 before the join? Need to use search arguments
  More on this later…

- **Cost**: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.
SELECT
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
FROM
  customer,
  orders,
  lineitem
WHERE
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date('1995-03-15')
  and l_shipdate > date('1995-03-15')
GROUP BY
  l_orderkey,
  o_orderdate,
  o_shippriority
ORDER BY
  revenue desc,
  o_orderdate
LIMIT 10
Postgres EXPLAIN output

Limit (cost=425583.61..425583.63 rows=10 width=28)
  -> Sort (cost=425583.61..426332.86 rows=299703 width=28)
      Sort Key: sum((lineitem.l_extendedprice * (1::double precision - lineitem.l_discount))), orders.o_orderdate
  -> GroupAggregate (cost=374986.77..383977.86 rows=299703 width=28)
      -> Sort (cost=374986.77..375736.03 rows=299703 width=28)
          Sort Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
      -> Hash Join (cost=66349.40..333381.03 rows=299703 width=28)
          Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
          -> Seq Scan on lineitem (cost=0.00..214592.79 rows=3151149 width=20)
              Filter: (l_shipdate > '1995-03-15'::date)
          -> Hash (cost=63869.47..63869.47 rows=142635 width=12)
              -> Hash Join (cost=6272.32..63869.47 rows=142635 width=12)
                  Hash Cond: (orders.o_custkey = customer.c_custkey)
                  -> Seq Scan on orders (cost=0.00..48025.25 rows=724048 width=16)
                      Filter: (o_orderdate < '1995-03-15'::date)
                  -> Hash (cost=5902.96..5902.96 rows=29549 width=4)
                      -> Seq Scan on customer (cost=0.00..5902.96 rows=29549 width=4)
                          Filter: (c_mktsegment = 'BUILDING'::bpchar)
Pipelined Evaluation

- **Materialization**: Output of an op is saved in a temporary relation for uses (multiple scans) by the next op.

- **Pipelining**: No need to create a temporary relation. Avoid the cost of writing it out and reading it back. Can occur in two cases:
  - **Unary operator**: when the input is pipelined into it, the operator is applied on-the-fly, e.g. selection on-the-fly, project on-the-fly.
  - **Binary operator**: e.g., the outer relation in indexed nested loops join.
Iterator Interface for Execution

- A query plan, i.e., a tree of relational ops, is executed by calling operators in some (possibly interleaved) order.
- **Iterator Interface** for simple query execution:
  - Each operator typically implemented using a uniform interface: `open`, `get_next`, and `close`.
  - Query execution starts top-down (*pull-based*). When an operator is ``pulled’ for the next output tuples, it
    1. `pulls’ on its inputs (opens each child node if not yet, gets next from each input, and closes an input if it is exhausted),
    2. computes its own results.

- **Encapsulation**
  - Encapsulated in the operator-specific code: access methods, join algorithms, and materialization vs. pipelining…
  - Transparent to the query executer.
Highlights of System R Optimizer

- **Impact:** most widely used; works well for < 10 joins.
- **Cost of a plan:** approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate *cost of operations* and *result sizes*.
  - Considers combination of CPU and I/O costs.
- **Plan Space:** too large, must be pruned.
  - Only considers the space of *left-deep plans*.
    - Left-deep plan: a tree of joins in which the inner is a base relation.
    - Left-deep plans naturally support *pipelining*.
  - Avoids cartesian products!
- **Plan Search:** dynamic programming (prunes useless subtrees).
Plan Space

For each block, the plans considered are:

- All available access methods, for each reln in 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.
  - Consider all permutations of N relns, # of plans is N factorial!

Left-deep

Bushy

Bushy
Plan Space

- For each block, the plans considered are:
  - All available access methods, for each reln in 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, N factorial!
  - All join methods, for each join in the tree.
  - Appropriate places for selections and projections.
Cost Estimation

- For each plan considered, must estimate its cost.
- Estimate *cost* of each operation in a plan tree:
  - Depends on input cardinalities.
  - 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:
  - Use information about the input relations.
  - For selections and joins, assume independence of predicates and uniform distribution of values.
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). More on this later…
Size Estimation & Reduction Factors

- Consider a query block:

- 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!
  - Term col=value: RF = 1/NKeys(I), given index I on col
  - Term col>value: RF = (High(I)-value)/(High(I)-Low(I))
  - Term col1=col2: RF = 1/MAX(NKeys(I1), NKeys(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 NTuples(S)/NKeys(I2) matches, a RF of 1/NKeys(I2).
Size Estimation & Reduction Factors

- Consider a query block: 
  ```sql
  SELECT attribute list
  FROM relation list
  WHERE term1 AND ... AND termk
  ```

- **Reduction factor (RF) or Selectivity of each term:**
  - Assumption 1: uniform distribution of the values!
  - Term `col=value`: RF = $1/N\text{Keys}(I)$, given index $I$ on $col$
  - Term `col>value`: RF = $(\text{High}(I)-\text{value})/((\text{High}(I)-\text{Low}(I))$
  - Term `col1=col2`: RF = $1/\max(N\text{Keys}(I1), N\text{Keys}(I2))$

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

- **Result cardinality** = Max # tuples * product of all RF’s.
  - Assumption 2: terms are independent!
Cost Estimation for Multi-relation Plans

- Consider a query block:

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

- Reduction factor (RF) is associated with each term.

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

- Result cardinality = max # tuples * product of all RF’s.

- Multi-relation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimate of join cardinality gives us both cost estimate and result size estimate.
Queries Over Multiple Relations

- As the number of joins increases, the number of alternative plans grows rapidly.

- System R: (1) use only left-deep join trees, where the inner is a base relation, (2) avoid cartesian products.
  - Allow pipelined plans; intermediate results not written to temporary files.
  - Not all left-deep trees are fully pipelined!
    - Sort-Merge join (the sorting phase)
    - Two-phase hash join (the partitioning phase)
Plan space 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 common prefixes, 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 substructure 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. Include index scans available on “sargable” predicates.
  - **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.
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.
- A $k$-way ($k<N$) plan is not combined with an additional relation unless there is a join condition between them.
  - Do it until all predicates in WHERE have been used up.
  - That is, avoid Cartesian products if possible.
- In spite of pruning plan space, still creates an exponential number of plans.
Example

Sailors:
- B+ tree on rating
- Hash on sid

Reserves:
- B+ tree on bid

Pass 1

- **Sailors:**
  - B+ tree matches 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 rating order).

- **Reserves:** B+ tree on bid matches bid = 500; cheapest.
Pass 2

- Consider each plan retained from Pass 1 as the *outer*, and consider how to join it with the (only) other relation.

- *Reserves as outer*: Hash index can be used to get Sailors tuples that satisfy $sid = \text{outer tuple's} \, sid$ value.
  - rating > 5 is a *search argument* pushed to the index scan on Sailors.
System R: Limitation 1

- Uniform distribution of values:
  - Term \( \text{col}=\text{value} \) has RF \( 1/N\text{Keys}(I) \), given index \( I \) on \( \text{col} \)
  - Term \( \text{col}>\text{value} \) has RF \( (\text{High}(I)-\text{value})/(\text{High}(I)-\text{Low}(I)) \)

- Often causes highly inaccurate estimates
  - E.g., distribution of gender: male (40), female (4)
  - E.g. distribution of age:
    0 (2), 1 (3), 2 (3), 3 (1), 4 (2), 5 (1), 6 (3), 7 (8), 8 (4), 9 (2),
    10 (0), 11 (1), 12 (2), 13 (4), 14 (9). \( N\text{Keys}=15 \), count = 45.
    Reduction factor of age=14: 1/15? 9/45!

- **Histogram**: approximates a data distribution
## Histograms

### Equiwidth: buckets of equal size

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/3</td>
<td>8</td>
</tr>
<tr>
<td>4/3</td>
<td>4</td>
</tr>
<tr>
<td>15/3</td>
<td>15</td>
</tr>
<tr>
<td>3/3</td>
<td>3</td>
</tr>
<tr>
<td>15/3</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

Still not accurate for value 14: 5/45

### Equidepth: equal counts of buckets favoring frequent values

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/4</td>
<td>9</td>
</tr>
<tr>
<td>10/4</td>
<td>10</td>
</tr>
<tr>
<td>10/2</td>
<td>10</td>
</tr>
<tr>
<td>7/4</td>
<td>7</td>
</tr>
<tr>
<td>9/1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

Small errors for infrequent items: tolerable.

Now accurate for value 14: 9/45
System R: Limitation 2

- Predicates are independent:
  - \( \text{Result cardinality} = \text{max # tuples} \times \text{product of Reduction Factors of matching predicates.} \)
- Often causes highly inaccurate estimates
  - E.g., Car DB: 10 makes, 100 models. RF of make='honda' and model='civic' >> than 1/10 * 1/100!
- Multi-dimensional histograms [PI’97, MVW’98, GKT’00]
  - Maintain counts and frequency in multi-attribute space.
- Dependency-based histograms [DGR’01]
  - Learn dependency between attributes and compute conditional probability \( P(\text{model}='\text{civic}' \mid \text{make}='\text{honda}') \)
Nested Queries With No Correlation

- **Nested query (block):** a query that appears as an operand of a predicate of the form “expression operator query”.

- **Nested query with no correlation:** the nested block does not contain a reference to tuple from the outer.
  - A nested query needs to be evaluated **only once**.
  - The optimizer arranges it to be evaluated before the top level query.

```sql
SELECT S.sname
FROM    Sailors S
WHERE  S.rating >
   (SELECT Avg(rating)
    FROM Sailors)

(SELECT Avg(rating)
 FROM Sailors)

SELECT S.sname
FROM    Sailors S
WHERE  S.rating > value
```
Nested Queries With Correlation

- Nested query with correlation: the nested block contains a reference to a tuple from the outer.
  - Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
  - The nested block is executed using nested iteration, a tuple-at-a-time approach.

```
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)
```
Query Decorrelation

- Implicit ordering of nested blocks means *nested iteration* only.
- The equivalent, non-nested version of the query is typically optimized better, e.g. *hash join* or *sort-merge*.
- *Query decorrelation* is an important task of optimizer.

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

```
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 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.
Many other research directions

- Extensible query optimizers
- Optimization of expensive predicates
- Multiple-query optimization
- Adaptive query processing