Overview of Query Evaluation

Chapter 12
Overview of Query Evaluation

- **Query:**
  
  ```
  SELECT S.sname 
  FROM Reserves R, Sailors S
  WHERE R.sid = S.sid 
  AND R.bid = 100 AND S.rating > 5
  ```

- **Plan:** Tree of relational algebra ops, with an algorithm for each
  
  - Each “pulls” tuples from tables via “access paths”
  - An access path might involve an index, iteration, sorting, or other approaches.

- Two main issues in query optimization:
  
  - For a given query, what plans are considered?
  - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?

- **Ideally:** Want to find optimal plan.
- **Practically:** Want to avoid poor plans!
Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - **Indexing:** Can use WHERE conditions to retrieve small subset of tuples (selections, joins)
  - **Iteration:** Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - **Partitioning:** By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

* Watch for these techniques as we discuss query evaluation!
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.
Today’s Working Example

- Consider database with the following two tables:

  Sailors\((\text{sid}: \text{integer}, \text{sname}: \text{string}, \text{rating}: \text{integer}, \text{age}: \text{real})\)

  Reserves\((\text{sid}: \text{integer, bid}: \text{integer, day}: \text{date, rname}: \text{string})\)

- Assume each tuple of Reserves is 40 bytes, a page holds, at most, 100 records, each Sailors’ tuple is 50 bytes, and a page holds no more than 80 records

- Furthermore, assume

  1000 pages of Reserves (< 100,000 records), and

  500 pages of Sailors (< 40,000 records)
Example’s Catalog

- The system catalog is itself a collection of relations/tables (ex. Table attributes, table statistics, etc.)
- Catalog tables can be queried just like any other table
- Relational algebra operations can be used to examine Query evaluation tradeoffs

<table>
<thead>
<tr>
<th>Attribute_Cat</th>
<th>attr_name</th>
<th>rel_name</th>
<th>type</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>rel_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>Attribute_Cat</td>
<td>integer</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>sid</td>
<td>Sailors</td>
<td>integer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sname</td>
<td>Sailors</td>
<td>string</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>rating</td>
<td>Sailors</td>
<td>integer</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>age</td>
<td>Sailors</td>
<td>real</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>sid</td>
<td>Reserves</td>
<td>integer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>bid</td>
<td>Reserves</td>
<td>integer</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>day</td>
<td>Reserves</td>
<td>date</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>rname</td>
<td>Reserves</td>
<td>string</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Access Paths

- An **access path** is a method of retrieving tuples:
  - File scan, or index search that **matches** the given query’s selection

- A tree index **matches** (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <a, b, c> matches the selection a=5 AND b=3, and a=5 AND b>6, but not b=3.

- A hash index **matches** (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on <a, b, c> matches a=5 AND b=3 AND c=5; but it does not match b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5.
A Note on Complex Selections

\[(\text{day}<8/9/94 \text{ OR } \text{bid}=5 \text{ OR } \text{sid}=3) \text{ AND } (\text{rname}=\text{‘Paul’} \text{ OR } \text{bid}=5 \text{ OR } \text{sid}=3)\]

- Selection conditions are first converted to “product of sums” form

\[(\text{day}<8/9/94 \text{ AND } \text{rname}=\text{‘Paul’}) \text{ OR } \text{bid}=5 \text{ OR } \text{sid}=3\]

- “AND” terms allow us to optimally choose indices
- “OR” terms can be tested sequentially in iterations.
One Approach to Selections

- Find the *most selective access path*, retrieve tuples using it, and apply any remaining unmatched terms
  - *Most selective access path*: Either an index traversal or file scan that we *estimate* requires the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other unmatched terms are used to discard tuples, but do not affect number of tuples/pages fetched.
  - Consider \( \text{day}<8/9/94 \text{ AND bid}=5 \text{ AND sid}=3 \).
    - A B+ tree index on \( \text{day} \) can be used; then, \( \text{bid}=5 \) and \( \text{sid}=3 \) checked for each retrieved tuple.
    - Similarly, a hash index on \(<\text{bid}, \text{sid}>\) could be used; then \( \text{day}<8/9/94 \) checked. *Which is faster?*
Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - For example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

```
SELECT * 
FROM Reserves R 
WHERE R.rname < 'C%'
```
Projection

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection

- Expensive part is eliminating duplicates.
  - SQL systems don’t remove duplicates unless the keyword DISTINCT is specified in a query.

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection

- Expensive part is eliminating duplicates.
  - SQL systems don’t remove duplicates unless the keyword DISTINCT is specified in a query.

- Sorting Approach
  - Sort on <sid, bid> and remove duplicates.
    (Can optimize by dropping unwanted attributes while sorting.)

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection

Expensive part is eliminating duplicates.
- SQL systems don’t remove duplicates unless the keyword DISTINCT is specified in a query.

Sorting Approach
- Sort on <sid, bid> and remove duplicates.
  (Can optimize by dropping unwanted attributes while sorting.)

Hashing Approach
- Hash on <sid, bid> during scan to create partitions.
  Ignore hash-key collisions.

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection

- Expensive part is eliminating duplicates.
  - SQL systems don’t remove duplicates unless the keyword DISTINCT is specified in a query.

- Sorting Approach
  - Sort on <sid, bid> and remove duplicates.
    (Can optimize by dropping unwanted attributes while sorting.)

- Hashing Approach
  - Hash on <sid, bid> during scan to create partitions. Ignore hash-key collisions.

- With an index containing both R.sid and R.bid, you can step through the leaves (if tree) compressing duplicates, or directory of a Hash, however, may be cheaper to sort data entries!

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Join: Index Nested Loops

- If there is an index on the join attribute of one relation (say S), can make it the inner loop to exploit the index.
  - Cost: $M + (M \cdot p_R) \cdot \text{cost of finding matching } S \text{ tuples}$
  - $M=$#pages of $R$, $p_R=$# $R$ tuples per page

- For each $R$ tuple, cost of probing $S$ index is $\sim 1.2$ for hash index, 2-4 for B+ tree. Cost of then finding $S$ tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O total (typical)
  - Unclustered: upto 1 I/O per matching $S$ tuple.

```python
foreach tuple r in R:
    foreach tuple s in S:
        if r\_i op s\_j add <r, s> to result
```
Examples of Index Nested Loops

- Hash-index (Alt. 2) on sid of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (exactly one) matching Sailors tuple.
  - Total: 1000 + (1+1.2)*100000 = 221,000 I/Os.
Examples of Index Nested Loops

- **Hash-index (Alt. 2) on sid of Sailors (as inner):**
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (exactly one) matching Sailors tuple.
  - Total: $1000 + (1+1.2)\times100000 = 221,000$ I/Os.

- **Hash-index (Alt. 2) on sid of Reserves (as inner):**
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor ($100,000 / 40,000$). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
  - Total: $500 + (1.2 + 1)\times40000 = 88,500$ I/Os (clustered)
    $500 + (1.2 + 2.5)\times40000 = 148,500$ I/Os (unclustered)
Join: Sort-Merge ($R \bowtie_i S$)

- Sort $R$ and $S$ on the join column
Join: Sort-Merge ($R \bowtie_i S$)

- Sort $R$ and $S$ on the join column
- Scan them while “merging” (on join col.) and outputting resulting tuples.
  - 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 in $R_i$ (current $R$ group) and all $S$ tuples with same value in $S_j$ (current $S$ group) match; output $<r, s>$ for all pairs of such tuples.
  - Then resume scanning $R$ and $S$. 
Join: Sort-Merge ($R \bowtie_i S$)

- Sort $R$ and $S$ on the join column
- Scan them while “merging” (on join col.) and outputting resulting tuples.
  - 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 in $R_i$ (current $R$ group) and all $S$ tuples with same value in $S_j$ (current $S$ group) match; output $<r, s>$ for all pairs of such tuples.
  - Then resume scanning $R$ and $S$.

- $R$ is scanned once; each $S$ group is scanned once per matching $R$ tuple. (Multiple scans of an $S$ group are likely to find needed pages in buffer.)
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>uppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

Note importance of out-of-core external sorting (Next lecture’s topic)

- Cost: \( M \log M + N \log N + (M+N) \)
  - The cost of scanning, \( M+N \), could be \( M*N \) (very unlikely!)
- With 35, 100, or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
Highlights of Query Optimization

- Cost estimation: 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 the space of left-deep plans is considered.
    - Left-deep plans allow output of each operator to be pipelined into the next operator without storing it in a temporary relation.
  - Actual Cartesian products avoided.
Cost Estimation

- For each plan considered, we must estimate cost:
  - **Cost** of each operation in plan tree.
    - Depends on input cardinalities.
    - We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also *estimate size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND R.bid=100 AND S.rating>5
```

**RA Tree:**

```
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \land \sigma_{\text{sid} = \text{sid}} \]
```

- **Reserves**
- **Sailors**
Size Estimation and Reduction Factors

- Consider a query block:

\[
\text{SELECT attribute list FROM relation list WHERE } \text{term}_1 \text{ AND ... AND } \text{term}_k
\]

- 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.

\[
\text{Result cardinality} = \text{Max # tuples} \times \text{RF}_1 \times \text{RF}_2 \times \ldots \times \text{RF}_k.
\]

  - Implicit assumption that terms are independent!
  - Term \( \text{col}=\text{value} \) has RF \( 1/N\text{Keys(I)} \), given index I on \text{col}
  - Term \( \text{col}_1=\text{col}_2 \) has RF \( 1/\text{MAX}(N\text{Keys(I}_1), N\text{Keys(I}_2)) \)
  - Term \( \text{col}>\text{value} \) has RF \( (\text{High(I)}-\text{value})/(\text{High(I)}-\text{Low(I)}) \)
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 Plan 1 (No Indexes)

- **Main difference:** Push selects.
- With 5 buffers, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, assumes uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os.
Alternative Plan 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.
Practical Example

$ sqlite3 actors.db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> EXPLAIN QUERY PLAN
   ...>  SELECT C.role, A.name, M.title
   ...>  FROM Casts C, Actors A, Movies M
   ...>  WHERE C.aid=A.aid AND C.mid=M.mid AND C.role like "%Batman%";
0|0|0|SCAN TABLE Casts AS C (~500000 rows)
0|1|1|SEARCH TABLE Actors AS A USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
0|2|2|SEARCH TABLE Movies AS M USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite> EXPLAIN QUERY PLAN
   ...>  SELECT C.role, A.name, M.title
   ...>  FROM Casts C, Actors A, Movies M
   ...>  WHERE C.aid=A.aid AND C.mid=M.mid AND M.title="Batman";
0|0|2|SCAN TABLE Movies AS M (~100000 rows)
0|1|0|SEARCH TABLE Casts AS C USING AUTOMATIC COVERING INDEX (mid=?) (~7 rows)
0|2|1|SEARCH TABLE Actors AS A USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite>
Summary

- There are several alternative evaluation algorithms for each relational operator.

- A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.

- Must understand query optimization in order to fully 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.