Overview of Query Evaluation

Midterm next Tuesday, in class 80 min, open book, open internet, no communication

I expect to post the grades for problem sets #1 and #2 before class on Thursday

Change of plan for Thursday
Overview of Query Evaluation

- **Query:**
  
  SELECT C.name, D.race, D.sex, D.count  
  FROM County C, Demographics D  
  WHERE C.fips=D.fips  
  AND D.year=2020 AND C.region LIKE "Western %"

- **Plan:** Tree of operations with an algorithm for each
  - Each operation "pulls" tuples from "relations" 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 an efficient plan.
  - How is the **cost of a plan** estimated?

- **Ideally:** Want to find the optimal plan.

- **Reality:** Want to avoid poor plans!
Some Common Techniques

- Algorithms for evaluating queries use the same simple ideas extensively:
  - **Indexing**: Can use WHERE conditions to retrieve a subset of tuples (selections, joins)
  - **Iteration**: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the search keys of 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 all the tables 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 and high key values (Low/High) for each tree index.

- Catalogs are updated regularly.
  - 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 a simplified database with the following two tables:

  County\((fips: \text{int}, \ name: \text{text}, \ region: \text{text})\)
  Demographics\((fips: \text{int}, \ year: \text{int}, \ race: \text{text}, \ sex: \text{text}, \ count: \text{int})\)

- Assume each tuple of County is 200 bytes, a page holds, at most, 20 rows, each Demographics tuple is 50 bytes, and a page holds no more than 80 rows

- Furthermore, assume 6 pages of County (< 120 records), and 200 pages of Demographics (< 16,000 records)
Example’s Catalog

Attribute_Cat(attr_name: string, rel_name: string, type: string, position: integer)

- 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
- These queries can be used to examine Query evaluation tradeoffs

<table>
<thead>
<tr>
<th>Attribute_Cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_name</td>
</tr>
<tr>
<td>attr_name</td>
</tr>
<tr>
<td>rel_name</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>position</td>
</tr>
<tr>
<td>fips</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>region</td>
</tr>
<tr>
<td>fips</td>
</tr>
<tr>
<td>year</td>
</tr>
<tr>
<td>race</td>
</tr>
<tr>
<td>sex</td>
</tr>
<tr>
<td>count</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

\[ \text{year} > 2010 \ \text{AND} \ \text{race} = \text{"aian" AND (fips=37001 OR fips=37063)} \]

- Selection conditions are first converted to “sum-of-products” form (ORs of AND clauses)
  \[ (\text{year} > 2010 \ \text{AND} \ \text{race} = \text{'aian'} \ \text{AND} \ \text{fips=37001}) \ \text{OR} \ (\text{year} > 2010 \ \text{AND} \ \text{race} = \text{'aian'} \ \text{AND} \ \text{fips=37063}) \]

- “AND” terms allow us to optimally choose indices “OR” terms can be generated as independent query evaluations over the same tables or a subset
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 $year > 2010$ AND $race = 'aian'$
    - A B+ tree index on $year$ can be used; then, $sex$ would be checked for each retrieved tuple.
    - Similarly, a hash index on $race$ could be used; then $year > 2010$ 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 if table isn't clustered on search key).

- Assume 50% of demographics records are 2010 or after
  - If the table is clustered by *year*, the cost is little more than $(0.5 \times 200) = 100$ I/Os
  - If table isn't clustered by year (say sex), then there are likely 40 per page requiring us to read all 200 pages!
  - In reality, demographics probably are clustered by the year, so the 100 I/Os might not be that far off
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 if table isn't clustered on search key).
  - There are 2097 demographics records for race='aian'.
    - A single hash leads us to a hash bucket linked to 2 overflow buckets with these record's page ids
    - In the worst-case these 2097 records are spread across all 200 Demographics table pages.
    - The hash index on Player.name is not very selective for this query
    - However, if records are clustered by <year,race>, we might find all the "aian" records in a subset of pages, getting us back to 100.
Selection

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

- Sorting Approach
  - Sort on <pid, tid> and remove duplicates.
    (Can optimize by dropping unneeded attributes while sorting.)

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

- With an index containing both pid and tid, you can step through the leaves (if tree) compressing duplicates, or directory of a Hash, however, may be cheaper to sort data entries!
Join: Index Nested Loops

foreach tuple r in R:
    foreach tuple p in P:
        if $r_i \text{ op } p_j$:
            add <$r_i$, $p_j$> to result

foreach tuple p in P:
    foreach tuple r in R:
        if $r_i \text{ op } p_j$:
            add <$r_i$, $p_j$> to result

- If there is an index on the attribute of one relation (say P), if we make it the inner loop to exploit the index.
  - Cost: $M + (M^*_p R) \times \text{cost of finding matching P tuples}$
  - $M = \# \text{pages of R, } p_R = \# \text{tuples per R page}$

- For each R tuple, cost of probing S index is ~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.
Examples of Index Nested Loops

❖ Hash-index on race:
  - Scan County: 6 pages
  - Use index on Demographics:
    • 1.2 I/Os to get page index, plus 120 I/Os
      to get "aian" Demographic records assumes some clustering
    • check year > 2010
  - 6 + (1+1.2) + 120 = 128 I/Os.

❖ Tree-index on year:
  - Scan County: 6 pages
  - Use B+ tree index on Demographics (3 levels + 110 pages)
  - check race = "aian"
  - Total: 3 + 110 = 113 I/Os
Join: Sort-Merge (R JOIN S ON i=j)

- First, Sort R and S on the join attribute
- Scan both sorted tables while "merging" to output result tuples.
  - Advance scan of R until current R-tuple >= current P tuple, then advance scan of P until current P-tuple >= 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_i, s_j> 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. (Repeated scanning of S group is likely to find needed pages in buffer.)
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>college</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>29010</td>
<td>Austin Shepherd</td>
<td>Alabama</td>
<td>1992-05-28</td>
</tr>
<tr>
<td>29011</td>
<td>Josh Shirley</td>
<td>Nevada-Las Vegas</td>
<td>1992-01-04</td>
</tr>
<tr>
<td>29012</td>
<td>Jameill Showers</td>
<td>Texas-El Paso</td>
<td>--</td>
</tr>
<tr>
<td>29013</td>
<td>Trevor Siemian</td>
<td>Northwestern</td>
<td>1991-12-26</td>
</tr>
<tr>
<td>29014</td>
<td>Ian Silberman</td>
<td>Boston College</td>
<td>1992-10-10</td>
</tr>
<tr>
<td>29015</td>
<td>Shayne Skov</td>
<td>Stanford</td>
<td>1990-07-09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pid</th>
<th>tid</th>
<th>year</th>
<th>starts</th>
</tr>
</thead>
<tbody>
<tr>
<td>29010</td>
<td>1032</td>
<td>2015</td>
<td>0</td>
</tr>
<tr>
<td>29011</td>
<td>1006</td>
<td>2015</td>
<td>0</td>
</tr>
<tr>
<td>29011</td>
<td>1001</td>
<td>2016</td>
<td>0</td>
</tr>
<tr>
<td>29012</td>
<td>1012</td>
<td>2015</td>
<td>0</td>
</tr>
<tr>
<td>29013</td>
<td>1004</td>
<td>2015</td>
<td>0</td>
</tr>
<tr>
<td>29013</td>
<td>1004</td>
<td>2016</td>
<td>14</td>
</tr>
<tr>
<td>29013</td>
<td>1004</td>
<td>2017</td>
<td>10</td>
</tr>
<tr>
<td>29013</td>
<td>1032</td>
<td>2018</td>
<td>0</td>
</tr>
<tr>
<td>29013</td>
<td>1019</td>
<td>2019</td>
<td>0</td>
</tr>
</tbody>
</table>

We'll use "out-of-core" external sorting (Next lecture's topic)

Pass 1: Read P in 10, 50 block chunks, sort each one, and then write them back, then read R in 8, 50 block chunks, sort each, and write them back (2(400+500))

Pass 2: Read in the head blocks of the 10 sorted P chunks and the heads of 8 sorted R chunks. Merge the tops of the 10 into one block and the tops of the 8 into another (refill any head block when it is exhausted). These two merged blocks are then scanned for matching keys (400+500).

❖ Cost: M log M + N log N + (M+N)
  ▪ The cost of scanning, M+N, could be M*N (very unlikely!)
❖ Using only 20 buffer pages, 200 Demographics pages can be sorted in 2 passes; total join cost: 10*20+10 = 210 I/Os.
Highlights of Query Optimization

- **Cost estimation:** Approximations are an art.
  - 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.

Alternate Evaluation Trees:

```
SELECT C.name, D.race, D.sex, D.count
   D.year>2010 AND D.race='aian'
   C JOIN D ON P.fips = D.fips
```

Load 6 County blocks and scan 200 Demographics blocks 206 I/Os using only 7 buffer pages
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.