

Relational Query Optimization

(this time we really mean it)

R&G Chapter 15
Lecture 25



Administrivia

- Homework 5 mostly available
 - It will be due after classes end, Monday 12/8
- Only 3 more lectures left!
 - Next Tuesday: Physical Database Tuning
 - Following Tuesday: Real-World Databases
 - Following Thursday: Class wrap-up

Overview: Query Optimization

- ‘Explain’ exercise showed many ways to get same result, some more expensive than others
 - Access to table matters (index, non-index)
 - Order of operations matters
 - Join algorithm matters
- Goal of optimizer: find a good query plan
 - finding absolute best plan usually not feasible
 - sufficient to find non-bad plan

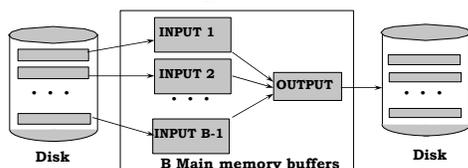
Review: Query Optimization

- Sorting data, even if it doesn’t fit in memory
- Different join algorithms
- Relational algebra equivalences
- Enumerating query plans
- Choosing the best query plan

Review: General External Merge Sort

➡ More than 3 buffer pages. How can we utilize them?

- To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
 - Pass 1, 2, ..., etc.: merge $B-1$ runs.



Review: Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = $2N * (\# \text{ of passes})$
- E.g., with 5 buffer pages, to sort 108 page file:
 - Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
- Now, do four-way ($B-1$) merges
 - Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)
 - Pass 2: 2 sorted runs, 80 pages and 28 pages
 - Pass 3: Sorted file of 108 pages



Review – Cost of Join Methods

- **Blocked Nested Loops**
 $M + \lceil M / B \rceil * N$
- **Indexed Nested Loops**
 $M + (M * p_R) * \text{cost to find matching tuples}$
- **Sort-Merge Join**
between $3(M+N)$ and $M*N$
- **Hash Join**
 $3(M+N)$ and higher, especially with skewed data



Review: Relational Algebra Equivalences

- **Selections:** $\sigma_{c_1 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\dots \sigma_{c_n}(R))$
(Cascade) $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$ (Commutate)
- **Projections:** $\pi_{a_1}(R) \equiv \pi_{a_1}(\dots (\pi_{a_n}(R)))$ (Cascade)
- **Joins:** $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (Associative)
 $(R \bowtie S) \equiv (S \bowtie R)$ (Commutate)



Review: More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma(R \bowtie S) \equiv \sigma(R) \bowtie S$)
- Similarly, if a projection follows a join $R \bowtie S$, we can 'push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.



Query Optimization

- **Process:**
 - Every SQL query can be translated to one or more relational algebra expression trees, a.k.a. *plans*
 - parts of nested queries usually considered separately
 - Consider some set of equivalent plans, evaluate the cost of each
 - Choose the best plan you find
- **Issues:**
 - What plans do you consider?
 - How do you evaluate the cost?
- **System R is approach we will examine**



Highlights of System R Optimizer

- **Impact:**
 - Most widely used currently; works well for < 10 joins.
- **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.
 - Cartesian products avoided.



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. (This is an over-simplification, but serves for now.)

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

Outer block Nested block

- For each block, the plans considered are:
 - All available access methods, for each reln in FROM clause.
 - All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)



Converting Query Blocks to Rel. Algebra

- We have 'extended' relational algebra
 - also include aggregate ops: group by, having

- How is this query block expressed?

```
SELECT S.sname
FROM Sailors S
WHERE S.age IN (constant set from subquery)
```

$$\Pi_{\text{sname}}(\sigma_{\text{age in set from subquery}}(\text{Sailors}))$$

- And this query block?

```
SELECT MAX (S2.age)
FROM Sailors S2
GROUP BY S2.rating
```

$$\Pi_{\text{Max(age)}}(\text{GroupBy}_{\text{Rating}}(\text{Sailors}))$$


What Query Plans do we get?

$$\Pi_{\text{sname}}(\sigma_{\text{age in set from subquery}}(\text{Sailors}))$$

$$\Pi_{\text{Max(age)}}(\text{GroupBy}_{\text{Rating}}(\text{Sailors}))$$

- These expressions are simple, no rewriting
- Must consider access plans to Sailors, though
 - $\sigma_{\text{age in set from subquery}}$ might use index
 - $\text{GroupBy}_{\text{Rating}}$ might benefit from clustered index



Enumeration of Alternative Plans

- There are two main cases:
 - Single-relation plans
 - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
 - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
 - 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).



Cost Estimation

- What factors does the cost of a sort depend on?
- What factors does the cost of each join method depend on?
- What factors does the cost of a selection depend on?



Cost Estimation

- For each plan considered, must estimate cost:
 - Must estimate *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.



Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
 - Cost is $\text{Height}(I)+1$ for a B+ tree, about 1.2 for hash index.
 - Clustered index I matching one or more selects:
 - $(\text{NPages}(I)+\text{NPages}(R))$ * product of RF's of matching selects.
 - Non-clustered index I matching one or more selects:
 - $(\text{NPages}(I)+\text{NTuples}(R))$ * product of RF's of matching selects.
 - Sequential scan of file:
 - $\text{NPages}(R)$.
- ☞ **Note:** Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)



Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- 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.



Example

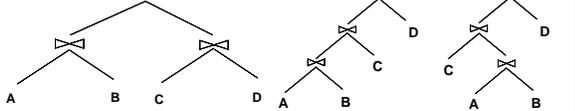
```
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

- If we have an index on *rating*:
 - $(1/NKeys(I)) * NTuples(R) = (1/10) * 40000$ tuples retrieved.
 - Clustered index: $(1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500)$ pages are retrieved. (This is the *cost*.)
 - Unclustered index: $(1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000)$ pages are retrieved.
- If we have an index on *sid*:
 - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.
- Doing a file scan:
 - We retrieve all file pages (500).



Queries Over Multiple Relations

- Fundamental decision in System R: only left-deep join trees are considered.
 - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space*.
 - Left-deep trees allow us to generate all *fully pipelined* plans.
 - Intermediate results not written to temporary files.
 - Not all left-deep trees are fully pipelined (e.g., SM join).



Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.
- Enumerated using **N passes** (if **N relations joined**):
 - Pass 1: Find best 1-relation plan for each relation.
 - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (*All 2-relation plans.*)
 - Pass N: Find best way 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 plan overall, plus
 - Cheapest plan for each *interesting order* of the tuples.



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.
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up.
 - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables.



Cost Estimation for Multirelation Plans

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

- 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. $Result\ cardinality = Max\ \# \ tuples * product\ of\ all\ RF's$.
- Multirelation plans are built up by joining one new relation at a time.
 - Cost of join method, plus estimation of join cardinality gives us both cost estimate and result size estimate



Example

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

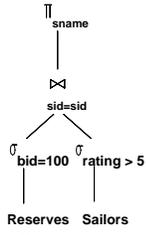
Reserves:
B+ tree on *bid*

• **Pass1:**

- *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:**

- We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
 - e.g., *Reserves* as outer: Hash index can be used to get *Sailors* tuples that satisfy *sid* = outer tuple's *sid* value.





Nested Queries

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

- **Nested block is optimized independently, with the outer tuple considered as providing a selection condition.**
- **Outer block is optimized with the cost of 'calling' nested block computation taken into account.**
- **Implicit ordering of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.***

Nested block to optimize:
SELECT *
FROM Reserves R
WHERE R.bid=103
AND S.sid= outer value

Equivalent non-nested query:
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
AND R.bid=103



General Optimization Strategies

- Using indexes good if term selective enough
- Because join and sort cost highly sensitive to size of input, often best to 'push' selections (and often projections) before join operations
- Nested queries often poorly optimized, write non-nested ones if possible
- Use 'Explain' when in doubt.



Summary

- Query optimization is an important task in a 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.



Summary (Contd.)

- **Single-relation queries:**
 - All access paths considered, cheapest is chosen.
 - *Issues:* Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.
- **Multiple-relation queries:**
 - All single-relation plans are first enumerated.
 - Selections/projections considered as early as possible.
 - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
 - Next, for each 2-relation plan that is 'retained', all ways of joining another relation (as inner) are considered, etc.
 - At each level, for each subset of relations, only best plan for each interesting order of tuples is 'retained'.