Introduction

- Today’s topic: QUERY PROCESSING
- Some database operations are EXPENSIVE
- Can greatly improve performance by being "smart"
  - e.g., can speed up 1,000,000x over naïve approach
- Main weapons are:
  1. clever implementation techniques for operators
  2. exploiting relational algebra "equivalences"
  3. using statistics and cost models to choose among these.

A Really Bad Query Optimizer

- For each Select-From-Where query block
  - Create a plan that:
    - Forms the cross product of the FROM clause
    - Applies the WHERE clause
- Then, as needed:
  - Apply any projections and output expressions
  - Apply duplicate elimination and/or ORDER BY

Cost-based Query Sub-System

The Query Optimization Game

- Goal is to pick a "good" plan
  - Good = low expected cost, under cost model
  - Degrees of freedom:
    - access methods
    - physical operators
    - operator orders
- Roadmap for this topic:
  - First: implementing individual operators
  - Then: optimizing multiple operators

Relational Operations

- We will consider how to implement:
  - Selection (σ) Select a subset of rows.
  - Projection (π) Remove unwanted columns.
  - Join (⋈) Combine two relations.
  - Set-difference (-) Tuples in reln. 1, but not in reln. 2.
  - Union (∪) Tuples in reln. 1 and in reln. 2.

Q: What about Intersection?
**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.
- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - \([S] = 500, p_s = 80.\)
- **Reserves:**
  - Each tuple is 40 bytes, 100 tuples per page, 1000 pages.
  - \([R] = 1000, p_R = 100.\)

**Simple Selections**

$$\sigma_{R.\text{rname} < 'C%' \land \text{day} < 8/9/94 \lor \text{bid} = 5 \lor \text{sid} = 3} \text{ Reserves } R$$

- How best to perform? Depends on:
  - what indexes are available
  - expected size of result
- Size of result approximated as
  \((\text{size of } R) \times \text{selectivity}\)
  - selectivity estimated via statistics – we will discuss shortly.

**Our options ...**

- If no appropriate index exists:
  Must scan the whole relation
  \(\text{cost} = [R].\) For "reserves" \(= 1000 \text{ I/Os}.\)

**Our options ...**

- With index on selection attribute:
  1. Use index to find qualifying data entries
  2. Retrieve corresponding data records
  Total cost = cost of step 1 + cost of step 2
  - For "reserves", if selectivity = 10% (100 pages, 10000 tuples):
    - If clustered index, cost is a little over 100 I/Os;
    - If unclustered, could be up to 10000 I/Os! ... unless ...

**Refinement for unclustered indexes**

1. Find qualifying data entries.
2. Sort the rid’s of the data records to be retrieved.
3. Fetch rids in order.
   Each data page is looked at just once (though # of such pages likely to be higher than with clustering).

**General Selection Conditions**

- \((\text{day} < 8/9/94 \text{ AND rname} = 'Paul' \text{ OR bid} = 5 \text{ OR sid} = 3)\)
- First, convert to **conjunctive normal form (CNF):**
  \((\text{day} < 8/9/94 \text{ OR bid} = 5 \text{ OR sid} = 3) \text{ AND (rname} = 'Paul' \text{ OR bid} = 5 \text{ OR sid} = 3)\)
- We only discuss the case with no ORs
- Terminology:
  - A B-tree index matches terms that involve only attributes in a prefix of the search key, e.g.:
  - Index on \(<a, b, c>\) matches \(a = 5\) and \(b = 3\), but not \(b = 3.\)
2 Approaches to General Selections

**Approach I:**
1. Find the **cheapest access path**
2. retrieve tuples using it
3. Apply any remaining terms that don’t match the index
   - **Cheapest access path:** An index or file scan that we estimate will require the fewest page I/Os.
   - (Why not say “that will return the fewest tuples”?)

**Cheapest Access Path - Example**

query: \( \text{day} < 8/9/94 \ \text{AND} \ \text{bid} = 5 \ \text{AND} \ \text{sid} = 3 \)

some options:
- \( \text{B+tree index on } \text{day}; \text{check bid}=5 \ \text{and} \ \text{sid}=3 \ \text{afterward}. \)
- \( \text{hash index on } \langle \text{bid, sid} \rangle; \text{check day}<8/9/94 \ \text{afterward}. \)

- How about a B+tree on \( \langle \text{name, day} \rangle \)?
- How about a B+tree on \( \langle \text{day, name} \rangle \)?
- How about a Hash index on \( \langle \text{day, name} \rangle \)?

**Approach II:** use 2 or more matching indexes.
1. From each index, get set of **rids**
2. Compute **intersection** of rid sets
3. Retrieve records for rid sets in intersection
4. Apply any remaining terms

**Projection**

**Issue is removing duplicates.**

**Use sorting!!**
1. Scan \( R \), extract only the needed attributes into \( R' \)
2. Sort \( R' \)
3. Read \( R' \) and remove adjacent duplicates as you return

**Cost:**
- Reserves with size ratio 0.25 = 250 pages.
- With 20 buffer pages can sort in 2 passes, so:
  \[ 1000 + 250 + 2 \times 250 + 250 = 2500 \ I/Os \]

**Projection -- improved**

- Modify the external sort algorithm:
  - Modify Pass 0 to
    - eliminate unwanted fields.
    - generate initial sorted runs.
  - Modify Passes 1+ to eliminate duplicates.

**Cost:**
- Reserves with size ratio 0.25 = 250 pages.
- With 20 buffer pages can sort in 2 passes, so:
  1. Read 1000 pages
  2. Write 250 (in runs of 40 pages each)
  3. Read and merge runs
- Total cost = \( 1000 + 250 + 250 = 1500 \).

**Other Projection Tricks**

- **If an index search key contains all wanted attrs:**
  - Do **index-only scan**
    - Apply projection techniques to data entries (much smaller!)
- **If a B+Tree index search key prefix has all wanted attrs:**
  - Do **in-order index-only scan**
    - Compare adjacent tuples on the fly (no sorting required!)
Query Execution Framework

```
SELECT DISTINCT name, gpa
FROM Students
```

One possible query execution plan:

```
HeapScan
Sort
Distinct
```

Iterators

- Relational operators are all subclasses of the class `iterator`:
  ```
  class iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
  }
  ```

- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Any iterator can be input to any other!

Example: Sort

```
class Sort extends iterator {
  void init();
  tuple next();
  void close();
  iterator &inputs[1];
  int numberOfRuns;
  DiskBlock runs[];
  RID nextRID[];
}
```

```
init():
  - generate the sorted runs on disk (passes 0 to n-1)
  - Allocate runs[] array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to first RID of each run

next():
  - nextRID array tells us where we’re “up to” in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- End of Fun -- if no tuples remain)

close():
  - deallocate the runs and nextRID arrays
```

Postgres Version

- `src/backend/executor/nodeSort.c`
  - `ExecInitSort` (init)
  - `ExecSort` (next)
  - `ExecEndSort` (close)

- The encapsulation stuff is hardwired into the Postgres C code
  - Postgres predates even C++!
  - See `src/backend/execProcNode.c` for the code that “dispatches the methods” explicitly!

Joins

```
SELECT * 
FROM Reserves R1, Sailors S1 
WHERE R1.sid=S1.sid
```

- Joins are very common.
- R x S is large; so, R x S followed by a selection is inefficient.
- Many approaches to reduce join cost.

Join techniques we will cover today:
1. Nested-loops join
2. Index-nested loops join
3. Sort-merge join

Simple Nested Loops Join

```
R x S: foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_i then add <r, s> to result
```

Cost = \( (p_R|R|)*|S| + |R| = 100*1000*500 + 1000 \text{ IOs} \)
- At 10ms/IO, Total time: ???

- What if smaller relation (S) was “outer”?
- What assumptions are being made here?
- What is cost if one relation can fit entirely in memory?
Page-Oriented Nested Loops Join

\[ \text{Cost} = [R] \times [S] + [R] = 1000 \times 500 + 1000 \]

- If smaller relation (S) is outer, \( \text{cost} = 500 \times 1000 + 500 \)
- Much better than naïve per-tuple approach!

Block Nested Loops Join

- Page-oriented NL doesn’t exploit extra buffers :(
- Idea to use memory efficiently:

\[ \text{Cost} = \text{Scan outer} + (#outer \text{ blocks} \times \text{scan inner}) \]

Input buffer for S, Output buffer

Examples of Block Nested Loops Join

- Say we have B = 100+2 memory buffers
- Join cost = [outer] + (#outer blocks * [inner])
  \[ \#outer \text{ blocks} = [outer] / 100 \]

- With R as outer (|R| = 1000):
  - Scanning R costs 1000 IO’s (done in 10 blocks)
  - Per block of R, we scan S; costs 10*500 I/Os
  - Total = 1000 + 10*500.

- With S as outer (|S| = 500):
  - Scanning S costs 500 IO’s (done in 5 blocks)
  - Per block of S, we can R; costs 5*1000 IO’s
  - Total = 500 + 5*1000.

Index Nested Loops Join

\[ \text{Cost} = [R] + ([R] \times [p_R]) \times \text{cost to find matching S tuples} \]

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.
- For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 I/Os for B+Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
     - Clusters/index: 1 I/O per page of matching S tuples.
     - Unclustered: up to 1 I/O per matching S tuple.
- What assumption is made here?

Sort-Merge Join

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

Example:

```
SELECT * 
FROM Reserves R1, Sails S1 
WHERE R1.sid=S1.sid 
```
Cost of Sort-Merge Join

• Cost: Sort R + Sort S + ([R]+[S])
  – But in worst case, last term could be [R]*[S] (very unlikely!)
  – Q: what is worst case?

Suppose B = 35 buffer pages:
• Both R and S can be sorted in 2 passes
• Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500

Suppose B = 300 buffer pages:
• Again, both R and S sorted in 2 passes
• Total join cost = 7500

Block-Nested-Loop cost = 2500 ... 15,000

Other Considerations ...

1. An important refinement:
   Do the join during the final merging pass of sort!
   • If have enough memory, can do:
     1. Read R and write out sorted runs
     2. Read S and write out sorted runs
     3. Merge R-runs and S-runs, and find R>S matches
   Cost = 3*[R] + 3*[S]
   Q: how much memory is "enough" (will answer next time ...)

2. Sort-merge join an especially good choice if:
   – one or both inputs are already sorted on join attribute(s)
   – output is required to be sorted on join attributes(s)
   Q: how to take these savings into account? (stay tuned ...)

Summary

• A virtue of relational DBMSs:
  queries are composed of a few basic operators
  – The implementation of these operators can be carefully tuned
  • Many alternative implementation techniques for each operator
  – No universally superior technique for most operators.
  • Must consider available alternatives
  – Called "Query optimization" -- we will study this topic soon!

The Exam is next Tuesday, Feb 26

• 1 Hour
• During class
• TAs available for questions 12:30-12:50
• Exam 12:50-1:50
• Cheat Sheet Allowed
  – 1 single-sided page
  – hand written
  – must be turned in
• All material before SQL Queries

Topics

• ER Graphs
• DB Design (Tables from ER graph)
• File organization (slotted pages, clustered vs. not)
• XQuery & XPath
• B-Trees
• Sorting
• Relational Algebra
• Relational Calculus