Goals for Today

- Reason about the stages of query optimization
- Understand how to estimate the cost of a full query plan
  - Pipelining vs. materialization
  - Intermediate result sizes

Cost-based Query Sub-System

- What plans are considered?
- How is the cost of a plan estimated?

Adminstrivia

- Lab 3 starts next week
- No problem set out this week
- Updated grutoring hours
- Take poll on Piazza for my office hours timing!
Query Optimization Overview

- Query converted to relational algebra expression
- Relational algebra converted to tree, joins as branches
- **Operators can also be applied in different order!**

Each operator has implementation choices ➔ Choosing forms physical plan

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

• Left branch is the “outer” relation

π_sname(σ_{bid=100 ∧ rating > 5}(Reserves ▶◃ Sailors))

Query Optimizer algorithm

- **Goal: given a query**, the optimizer wants to
  - Decide which query plans to consider
  - Compare plans and choose the “best” one
    (best = shortest time to run)

- **How about this algorithm?**
  - Step 1: enumerate the space of all possible plans
  - Step 2: run each query plan, measure its runtime
  - Step 3: choose the plan that ran the fastest!

Query Optimizer algorithm

- **Goal: given a query**, the optimizer wants to
  - Decide which query plans to consider
  - Compare plans and choose the “best” one
    (best = shortest time to run)

- **Algorithm**
  - Step 1: consider a set of possible plans
  - Step 2: estimate cost for each plan
  - Step 3: choose the plan with lowest cost

Estimating Cost

- Don’t want to execute a plan to figure out its run-time!
  - Instead estimate cost of the plan
  - Use cost as a proxy for run-time

- Cost of a plan = sum of the costs for each operator in the plan
Reasoning about operator cost

- For questions below, assume:
  - Each relation is 5 pages and stored as a heap file, no indexes
  - Buffer pool has 4 frames
  - Join algorithm is page-nested-loop-join (PNLI)
  - Order by operator uses general external merge-sort

1. (Review) What is the cost in I/Os for this plan, ignoring cost of final output?

```
A  B
```

2. Now what about the cost of this plan? *What information are you missing?*

```
ORDER BY(A.foo)
A  B
```

Pipelined vs. Materialized

- Query plan operator’s output could be generated in either *materialized* or *pipelined* fashion

- **Materialized**
  - Output of an operator *written back to disk* as a temporary file before its parent reads it in

- **Pipelining** (“on-the-fly”)
  - Output of *operator immediately given to parent* as input

Pipelining

- Parent and child operators *executing concurrently*
  - Iterator model
  - Parent calls next() on child/children
  - (As needed) child calls next() on its child/children

- *Savings compared to materialization*
  - No write I/O cost for child’s output
  - No read I/O cost for parent’s input

- Algorithms of operators must support pipelining for this to work

Exercise 2: Pipelining

- Use Page-Nested-Loop joins for the join algorithm

- Some examples:
  - (A join B) join C
    - Pipelined
  - C join (A join B)
    - Since (A join B) is the inner relation for the second join, need to materialize it
Schema for Examples

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

- **Reserves:**
  - Each record is 40 bytes long
  - 100 record per page
  - 1000 pages
- **Sailors:**
  - Each record is 50 bytes long,
  - 80 record per page
  - 500 pages

Suppose there are **100 boats** (uniformly distributed)

Suppose there are **10 ratings** (uniformly distributed 1-10)

Motivating Example

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

- Cost: 500+500*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...
- **Goal of optimization:** To find more efficient plans that compute the same answer.

Alternative Plans – Push SELECTs (No Indexes)

![Diagram](image)

- Sailors
- Reserves

500,500 I/Os

Alternative Plans – Push SELECTs (No Indexes)

![Diagram](image)

- Sailors
- Reserves

250,500 I/Os

- Sailors
- Reserves

4010 I/Os

500 + 1000 + 10 + (250 * 10)
Exercise 3-4: Estimate I/O cost

Alternative Plan: Indexes

- Suppose have these indexes:
  - Clustered Alt 1 hash index on bid of Reserves
  - Unclustered Alt 2 hash index on sid of Sailors

- Getting Reserves with bid=100:
  - Using index, we get 100,000/100 boats = 1000 records on 1000/100 = 10 pages

- **Cost**: Selection on Reserves (10 I/Os); then, for each tuple, get [one] matching Sailors tuple (1000*1.2) = 1210 I/Os.
  - Due to index on sid, decide not to push down rating >5
  - Join column sid is a key for Sailors!

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.

- Inner blocks are usually treated as subroutines

- Computed:
  - once per query (for uncorrelated sub-queries)
  - or once per outer tuple (for correlated sub-queries)
The System R aka “Selinger-style” Query Optimizer

- **Impact:**
  - Inspired most optimizers in use today
  - Works well for small-medium complexity queries (< 10 joins)

- **Cost estimation:**
  - Very inexact, but works ok in practice.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers a simple combination of CPU and I/O costs.

- **Plan Space:** Too large, must be pruned!

Statistics and cardinality estimation

- **Catalogs** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) per relation
  - and for each index:
    - # distinct key values (Nkeys).
    - low/high key values (Low/High).
    - Index height (Height) for each tree index.
    - Index size (NPages) (e.g., # leaf pages for tree).

- Statistics in catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

Size Estimation and Reduction Factors

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

- **Reduction factor (RF)** associated with each term reflects the impact of the term in reducing result size

- **RF is also called “selectivity”**

- How to predict size of output?
  - Need to know/estimate input size
  - Need to know/estimate RFs
  - Need to know/assume how terms are related

Result Size Estimation for Selections

- Result cardinality (for conjunctive terms) = # input tuples * product of all RF’s

Assumptions:

1. Values are uniformly distributed and terms are independent!
2. In System R, stats only tracked for indexed attributes (modern systems have removed this restriction)

<table>
<thead>
<tr>
<th>Term</th>
<th>Reduction Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>col = value</code></td>
<td>1 / Nkeys(I)</td>
</tr>
<tr>
<td><code>col &gt; value</code></td>
<td>(High(I)-value) / (High(I)-Low(I))</td>
</tr>
</tbody>
</table>

*Note: in System R, if missing indexes, assume RF = 1/10*
Exercise 5

• RF = 16/40 * 1/10 = 1/25
  – Result size: 20 pages or 1600 tuples

Result Size Estimation for Joins

• For equi-join of R and S... range of result sizes (in # of tuples)?
  – If R and S have no join attribute values in common?
  – If join attributes are a key for S?
    • And if the join attributes are also a foreign key in R?

• General case: join attributes a in common, a key for neither
  – Assumption: the set of distinct R.a values is contained in S.a
  – Idea: each tuple of R has a 1/NKeys(S) chance of joining with a tuple in S

\[
\text{NTuples}(R) \times \text{NTuples}(S) / \text{NKeys}(S)
\]

  – Reversing above assumption yields

\[
\text{Ntuples}(S) \times \text{Ntuples}(R) / \text{Nkeys}(R)
\]

  (use smaller of two if different)