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?

Ideally: find the best query plan
Reality: avoid the worst plans!
**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!**

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

Each operator has implementation choices → Choosing forms physical plan

**Query Optimizer algorithm**

- **Goal**: given a 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!

**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 costs for each operator in plan

```
π_{sname}(σ_{bid=100 \land rating > 5}(Reserves \bowtie◁Sailors))
```

Left branch is the “outer” relation
Exercise 2: Reasoning about cost

- 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 (PNLJ)
  - 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?

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

   NEED INPUT SIZE TO ORDER BY, DETERMINED BY OUTPUT SIZE OF JOIN

Exercise 3: 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

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

- Operator algorithm(s) must support pipelining for this to work!
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](image1.png)

Alternative Plans – Push SELECTs (No Indexes)

![Diagram](image2.png)
Exercise 4-5: Estimate I/O cost

<table>
<thead>
<tr>
<th>Joins</th>
<th>Operations</th>
<th>I/O Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Scan &amp; Write to temp T</td>
<td>4250 IOs</td>
</tr>
<tr>
<td>1</td>
<td>σ_{bid=100}</td>
<td>1000 + 500 + 250 + (10 * 250)</td>
</tr>
<tr>
<td>2</td>
<td>σ_{rating &gt; 5}</td>
<td>6000 IOs</td>
</tr>
</tbody>
</table>

Exercise 5: Estimate I/O cost

Query Blocks: Units of Optimization

- An SQL query is parsed into a set 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)

**Alternative Plan: Indexes**

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

- Accessing Reserves, bid=100:
  - Get 100,000/100 boats = 1000 records → 1000/100 pages = 10 pages
  - Since using clustered index

- Cost: Selection on Reserves (10 I/Os); then, for each tuple, get [one] matching Sailors tuple (1000*1.2+1) = 2210 I/Os

- Typo: Join column sid is a key for Sailors!

- Join column sid is a key for Sailors!
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: 

```sql
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>$col = value$</td>
<td>$1 / Nkeys(I)$</td>
</tr>
<tr>
<td>$col &gt; value$</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 6

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