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

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!

Example: SQL query

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

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

Graphical representation of query execution plan:

- Each operator has implementation choices
  - Choosing forms physical plan

DIAGRAM:

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

Example: SQL query

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

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

Graphical representation of query execution plan:

- Each operator has implementation choices
  - Choosing forms physical plan

DIAGRAM:

- Query converted to relational algebra expression
- Relational algebra converted to tree, joins as branches
- Operators can also be applied in different order!
### 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
  - some operators are blocking

### 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

```
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
- **Query Plan**: *(On-the-fly)*
  - Suppose there are **100 boats** (uniformly distributed)
  - Suppose there are **10 ratings** (uniformly distributed 1-10)

- **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)

Exercise 3: Estimate I/O cost (also try Exercise 4)

Alternative Plan 2: 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.
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!

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

• Cost of an operator depends on:
  – Algorithm used
  – Size of input

• Need to estimate the cardinality of an operator’s result, so its parent operator knows its input size(s)

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)
**Translating SQL to Relational Algebra**

```sql
SELECT S.sid, MIN(R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red" AND S.rating =
  (SELECT MAX(S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT(*) >= 2
```

**Query Optimization**

- **Idea:** for each SELECT-FROM-WHERE query block, determine a low-cost physical plan.

- **Two parts to optimizing a query:**
  1. Consider a set of alternative plans.
     - Must prune search space – more on this soon!
  2. 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.

**Pro-tip:** many DBMSs have an “explain” command that shows the chosen query plan for a query.

**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 Catalogs**

- **Need information about relations and indexes**

  - **Catalogs** typically contain at least:
    - # tuples (N_Tuples) and # pages (N_Pages) per relation.
    - # distinct key values (N_Keys) for each index.
    - low/high key values (Low/High) for each index.
    - Index height (Height) for each tree index.
    - Index size (N_Pages) for each index (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.

- **More detailed information (e.g., histograms of the values in some field) are sometimes stored.**
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 usually 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) = 
  \[ \#\ \text{input tuples} \times \text{product of all RF's} \]

  **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:**

<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>
<tr>
<td>col1 = col2</td>
<td>1 / MAX( NKeys(I1), Nkeys(I2) )</td>
</tr>
<tr>
<td>(useful for joins too)</td>
<td></td>
</tr>
</tbody>
</table>

  *Assumes each key in smaller index has a match in larger, uniform distribution*

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

Other Estimations

- **Disjunctive selections** (e.g., *cond1 OR cond2*)
  - Estimate cost of *all conditions* *not* true

- **Projection**
  - A tuple is a tuple, *cardinality* unchanged
  - (we know the size of fields using catalog)

Result Size Estimation for Joins

- Given an equi-join of R and S, what is the range of possible result sizes (in # of tuples)?
  - What if R and S have no join attribute values in common?
  - What if join attributes are a key for R
    - What if the join attributes are also a foreign key in S?

  **General case:** join attributes \( a \) in common, a key for neither
  - *Make 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
    \[
    \text{NTuples}(R) \times \text{NTuples}(S) / NKeys(S)
    \]
    - Reversing above assumption yields
      \[
      \text{NTuples}(R) \times \text{NTuples}(S) / Nkeys(R)
      \]
      *(use smaller of two if different)*