Goals for Today

- Explore the search space explosion for alternate query plans
- Understand the dynamic programming approach to exploring the (large!) space of query plans
- Reason about the heuristics used by the System R query optimizer to prune the space
  - Discuss some of the corners cut by query optimization algorithms like the System R approach

Logical Transformations:
Equivalent Relational Algebra Expressions

- Can write the same query multiple ways!
  - These alternate versions are akin to different possible logical query plans
- Good rules of thumb:
  - “Push” down selections
  - Avoid cross-products

Query Optimizer algorithm

- Goal: given a query, the optimizer wants to
  - Enumerate query plans to consider
  - Compare plans and choose the “best” one

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

**Selections:**
\[ \sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \quad \text{(Cascade)} \]
\[ \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \quad \text{(Commutate)} \]

**Projections:**
\[ \pi_{a_{n}}(R) \equiv \pi_{a_{1}}(\ldots \pi_{a_{n}}(R)) \quad \text{(Cascade)} \]
(if \( a_n \) includes \( a_{n-1} \) includes... \( a_1 \))

A projection could commute with a selection, e.g.,
\[ \pi_{a}(\sigma_{c}(R)) \equiv \sigma_{c}(\pi_{a}(R)) \quad \text{... if condition } c \text{ acts only on attributes in } a \]

---

R.A. Equivalences: Select & Project

- **Selection Push:** selection on attributes of \( R \) commutes with \( R \bowtie S \): \( \sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S \)

- **Projection Push:** A projection applied to join of \( R \) and \( S \) can be pushed before the join by:
  - retaining only attributes of \( R \) and \( S \) needed for the join,
  - or are kept by the projection
\[ \pi_{R,a,S,b}(R \bowtie_{R,a\Rightarrow S,c} S) \equiv (\pi_{R,a}(R)) \bowtie_{R,a\Rightarrow S,c} (\pi_{S,b}(S)) \]

---

R.A. Equivalences: Joins

\( (R \bowtie S) \equiv (S \bowtie R) \quad \text{(Commutative)} \)

\( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \text{(Associative)} \)

These mean we can switch join outer/inner relations and can do joins in any order!

-- If theta join, join condition must involve correct relations

---

Exercise 2-3

- Convert cross-product to join with \( R.a = S.c \)
- Commute the select condition \( R.a > 2 \) with join
- Note: *cannot* push projection \( R.c \) before join
  -- But could *cascade* the projection: project \( R.a,c \) before join, then project \( R.c \) after select

- Joining Boats and Sailors first would yield a lot of tuples, since this would become a cross-product!
Enumeration of Alternative Plans

- There are two main cases:
  - **Single-relation** plans (unary operators only)
  - **Multiple-relation** plans

- For unary operators:
  - For a scan, each available access path (sequential scan / index) is considered; one with the least *estimated* cost is chosen
  - Consecutive **Scan, Select, Project** and **Aggregate** operations can be typically *pipelined*

Enumerating Multi-Relation Plans

- Suppose we have N relations
  - Let’s ignore the space of different join algorithms for a moment
  - Recall: associative and commutative rules mean we can apply joins in any order

- How many join orders? **Example:** N=3, \{A,B,C\}
  - How many tree shapes?
  - Given a tree shape, how many leaf orderings?

For both tree shapes, can have 6 orderings of relations in the leaves

Exercise 4: Join Orders

- Leaf orderings given a shape? N!

- Tree shapes, for a fixed ordering of 4 relations
  - 1 left-deep and linear
  - 1 right-deep and linear
  - 1 bushy
  - 2 linear

Number of Join Orders

- Leaf order *permutations*: n!
- Tree shapes: Catalan numbers
  \[
  C(n) = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)! \cdot n!}
  \]
- Join orders(n) = n! * C(n-1)

<table>
<thead>
<tr>
<th>n</th>
<th>Join orders(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>1680</td>
</tr>
<tr>
<td>6</td>
<td>30,240</td>
</tr>
<tr>
<td>7</td>
<td>665,280</td>
</tr>
<tr>
<td>8</td>
<td>17,297,280</td>
</tr>
<tr>
<td>9</td>
<td>518,918,400</td>
</tr>
<tr>
<td>10</td>
<td>17,643,225,600</td>
</tr>
</tbody>
</table>

Dynamic Programming Approach

- Brute-force enumeration approach does not scale

- Observation: within the space of all possible plans, many plans share a common subplan

  $$\text{Best plan to join } A \text{ and } B \text{ can help us find the best plan to join } A, B, C, \text{ and } D$$

- Dynamic programming!
  - Cache best results for plans already considered

System R: Plans to Consider

- Fundamental decision in System R:
  - only left-deep join trees considered (1 tree shape)

- Left-deep trees allow us generate all fully pipelined plans
  - Note: Recall not all left-deep trees are fully pipelined (e.g., Sort-Merge join)

- Selections on a relation processed as part of access path, or on-the-fly with JOINs

More System R heuristics later...

Enumeration: Dynamic Programming (left-deep)

- Query plans differ by:
  - order of the N relations,
  - access method for each relation,
  - and the join method for each join

- Plans are enumerated in N passes, considering subsets of the N relations

- For each subset of relations, retain:
  - Cheapest plan overall (possibly unordered)

We’ll also hang onto the cheapest plans for ordered tuples! (Later)
Dynamic Programming Pseudocode

R $\leftarrow$ set of relations to join (e.g., ABCD)

for $\delta$ in $\{1 \ldots |R|\}$:
  for $S$ in $\{\text{all length } \delta \text{ subsets of } R\}$:
    $\text{optjoin}(S) = (S - a) \text{ join } a$

// where $a$ is the single relation that minimizes:
// $\text{cost}(\text{optjoin}(S - a)) +$
// min. cost to join $(S - a)$ to $a +$
// min. access cost for $a$

$\text{optjoin}(S - a)$ is cached from previous iteration

DP: Example (left-deep)

$\text{optjoin}(ABCD)$

$\delta = 1$

A = best way to access A
(e.g. sequential scan or index)
B = best way to access B
C = best way to access C
D = best way to access D

DP: Example (left-deep)

$\text{optjoin}(ABCD)$

$\delta = 2$

$\{A,B\} = AB$ or $BA$
(use pre-computed best way to access A and B)

$\{A,C\} = AC$ or $CA$

$\{A,D\} = AD$ or $DA$

$\{B,C\} = BC$ or $CB$

$\{B,D\} = BD$ or $DB$

$\{C,D\} = CD$ or $DC$

DP: Example (left-deep)

$\text{optjoin}(ABCD)$

$\delta = 3$

$\{A,B,C\} = \text{remove A, compare plans for } \{(B,C)\}$
$A$
remove B, compare plans for $\{(A,C)\}$
B
remove C, compare plans for $\{(A,B)\}$
C

$\{B,C,D\} = …$

$\{A,C,D\} = …$

$\{A,B,D\} = …$
DP: Example (left deep)

<table>
<thead>
<tr>
<th>Subplan</th>
<th>Best choice</th>
<th>Cost</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>index</td>
<td>150</td>
<td>1000</td>
</tr>
<tr>
<td>B</td>
<td>Seq scan</td>
<td>600</td>
<td>5000</td>
</tr>
<tr>
<td>{A,B}</td>
<td>BA</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>{B,C}</td>
<td>BC</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{A,B,C}</td>
<td>ACB</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>{B,C,D}</td>
<td>CBD</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

optjoin(ABCD)

δ=4

{A,B,C,D} = remove A, compare plans for \{\{B,C,D\}\} A remove B, compare plans for \{\{A,C,D\}\} B remove C, compare plans for \{\{A,B,D\}\} C remove D, compare plans for \{\{A,B,C\}\} D

Interesting Orders

- The output relation from a given operator could be ordered
  
  How?

- An intermediate result has an “interesting order” if it is returned in order of any of:
  - ORDER BY attributes
  - GROUP BY attributes
  - Join attributes of other joins

Why would we care?

DP Algorithm: Complexity (left-deep)

- Time complexity
  - For each pass k, consider all subsets of relations of size k \( \rightarrow N \text{ choose } k \) subsets
  - All subsets for N relations, less empty set: \( 2^N-1 \)

Power Set: the set of all subsets

- For each subset of size k, k ways to remove 1 join (k ≤ N)

Time complexity = O(N2^k)

System R: Plans Considered (Contd.)

- Only consider left-deep plans

- In DP algorithm, also keep in plan cache cheapest plan for each interesting order of the tuples

- Avoid Cross-products if possible
  - An i-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE clause have been used up

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an interestingly ordered plan or an additional sorting operator
Pass 1:
Reserves: Clustered B+ tree on bid matches bid=100, and is cheaper than file scan

Sailors: B+ tree matches rating>5, not very selective, and index is unclustered, so sequential file scan w/ select is likely cheaper. Also, Sailors.rating is not an interesting order.

Pass 2: We consider each Pass 1 plan as the outer:

Reserves as outer (using B+ Tree selection on bid):
Find lowest-cost join algorithm with Sailors as Inner

Sailors as outer (using Seq. File Scan w/selection on rating):
Find lowest-cost join algorithm with Reserves as Inner

Small Example

\[
\begin{align*}
\text{SELECT} & \quad S.\text{sname} \\
\text{FROM} & \quad \text{Sailors S, Reserves R} \\
\text{WHERE} & \quad S.\text{sid} = R.\text{sid} \\
\text{AND} & \quad S.\text{rating} > 5 \\
\text{AND} & \quad R.\text{bid} = 100
\end{align*}
\]

Indexes

Reserves:
Clustered B+ tree on bid

Sailors:
Unclustered B+ tree on rating

Bigger Example

\[
\begin{align*}
\text{Select} & \quad S.\text{sid}, \text{COUNT(*) AS numRedRes} \\
\text{FROM} & \quad \text{Sailors S, Reserves R, Boats B} \\
\text{WHERE} & \quad S.\text{sid} = R.\text{sid} \text{ AND R.bid = B.bid} \\
\text{AND} & \quad B.\text{color} = \text{"red"} \\
\text{GROUP BY} & \quad S.\text{sid}
\end{align*}
\]

Pass 1: Best plan(s) for accessing each relation (Ex. 5)
- Sailors: Seq File Scan; B+ tree on sid
- Reserves: Seq File Scan; B+ tree on sid
- Boats: Hash on color
  (note: given selection on color, clustered Hash is likely to be cheaper than file scan, so only it is retained)

Pass 2: For each of the plans in Pass 1, generate plans joining another relation as the inner (avoiding cross products)

Consider all join methods and every access path for the inner:
- File Scan Reserves (outer) with Boats (inner)
- File Scan Reserves (outer) with Sailors (inner)
- B+ on Reserves.sid (outer) with Boats (inner)
- B+ on Reserves.sid (outer) with Sailors (inner)
- File Scan Sailors (outer) with Reserves (inner)
- B+Tree Sailors.sid (outer) with Reserves (inner)
- Hash on Boats.color (outer) with Reserves (inner)

Retain both:
- cheapest plan for each pair of relations
- cheapest plan for each interesting order

Pass 3: For each of the plans retained from Pass 2, taken as the outer, generate plans for the remaining join
  - E.g.,
    Outer= Hash on Boats.color JOIN Reserves
    Inner = Sailors
    Join Method = Index NL using Sailors.sid B+Tree

Then, add the cost for doing the GROUP BY and aggregate:
- This is the cost to sort the result by sid, unless it has already been sorted by a previous operator.
- Then, choose the cheapest plan overall

Bigger Example: Pass 2

Bigger Example: Pass 3
Physical DB Design

- Query optimizer does what it can to use indexes, clustering, and operator implementations
- Database Administrator (DBA) is expected to set up physical design well
  - E.g., consider which indexes to create

**Good DBAs understand query optimizers very well!**

- Many DBMSs support a feature called **EXPLAIN**
  
  - Shows query plan the optimizer would choose
    - Use indexes or sequential scan?
    - Join order? Join algorithms?

---

SQLite Demo: EXPLAIN QUERY PLAN

```sql
sqlite> .read sailors.sql
sqlite> EXPLAIN QUERY PLAN SELECT * FROM Sailors WHERE age > 40;
selectid order from detail
---------- ---------- --------------
0 0 0 0 SCAN TABLE Sailors
sqlite> CREATE INDEX ageIndex ON Sailors(age);
sqlite> EXPLAIN QUERY PLAN SELECT * FROM Sailors WHERE age > 40;
selectid order from detail
---------- ---------- --------------
0 0 0 0 SEARCH TABLE Sailors USING INDEX ageIndex (age>?)
sqlite> CREATE INDEX ratingIndex ON Sailors(rating);
sqlite> EXPLAIN QUERY PLAN SELECT * FROM Sailors,Boats,Reserves
...> WHERE Sailors.sid = Reserves.sid
...> AND Boats.bid = Reserves.bid
...> AND Sailors.rating > 5;
selectid order from detail
---------- ---------- --------------
0 0 0 0 SEARCH TABLE Reserves
0 1 0 0 SEARCH TABLE Sailor
0 2 1 0 SEARCH TABLE Boats
sqlite> analyze; /* force sqlite to recompute statistics */
sqlite> EXPLAIN QUERY PLAN SELECT * FROM Sailors,Boats,Reserves
...> WHERE Sailors.sid = Reserves.sid
...> AND Boats.bid = Reserves.bid
...> AND Sailors.rating > 5;
selectid order from detail
---------- ---------- --------------
0 0 0 0 SEARCH TABLE Reserves USING INDEX ratingIndex (rating>?)
0 1 2 0 SEARCH TABLE Reserves USING COVERING INDEX sqlite_autoindex_reserves
0 2 1 0 SEARCH TABLE Boats USING INTEGER PRIMARY KEY (rowid=)```