CAS CS 460/660
Introduction to Database Systems
Query Optimization
Review

- Implementation of Relational Operations as Iterators
  - Focus largely on External algorithms (sorting/hashing)
- Choices depend on indexes, memory, stats,…
- Joins
  - Blocked nested loops:
    - simple, exploits extra memory
  - Indexed nested loops:
    - best if 1 rel small and one indexed
  - Sort/Merge Join
    - good with small amount of memory, bad with duplicates
  - Hash Join
    - fast (enough memory), bad with skewed data
    - Relatively easy to parallelize
- Sort and Hash-Based Aggs and DupElim
Query Optimization Overview

- Query can be converted to relational algebra
- Rel. Algebra converted to tree, joins as branches
- Each operator has implementation choices
- Operators can also be applied in different order!

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

\[
\pi_{\text{sname}}(\sigma_{\text{bid}=100 \land \text{rating}>5}(\text{Reserves} \bowtie\bowtie \text{Sailors}))
\]
Recall:

• Relational operators at nodes support uniform *iterator* interface:
  
  $\text{Open}(), \text{get\_next}(), \text{close}()$

• Unary Ops – On Open() call Open() on child.

• Binary Ops – call Open() on left child then on right.

• By convention, outer is on left.

Alternative is pipelining (i.e. a “push”-based approach).

Can combine push & pull using special operators.
Query Optimization Overview (cont)

- **Logical Plan:** Tree of R.A. ops
- **Physical Plan:** Tree of R.A. ops, with choice of algorithm for each operator.

Two main issues:
- For a given query, what plans are considered?
  - Algorithm to search plan space for cheapest (estimated) plan.
- How is the cost of a plan estimated?

- Ideally: Want to find best plan.
- Reality: Avoid worst plans!
Cost-based Query Sub-System

Queries

Select *
From Blah B
Where B.blah = blah

Usually there is a heuristics-based rewriting step before the cost-based steps.
As seen in previous lectures…

Reserves:
- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Let’s say there are 100 boats.

Sailors:
- Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Let’s say there are 10 different ratings.

Assume we have 5 pages in our buffer pool.
Motivating Example

- **Cost**: $500 + 500 \times 1000$ 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, etc.

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

**SQL Query**:

```sql
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND R.bid=100 AND S.rating>5
```
Alternative Plans – Push Selects (No Indexes)

- Sailors
  - sid=sid
  - rating > 5
  - bid=100

- Reserves
  - sid=sid
  - rating > 5
  - bid=100

- (Page-Oriented Nested loops)
- (On-the-fly)

500,500 IOs

250,500 IOs
Alternative Plans – Push Selects
(No Indexes)

```
Sailors
    sid=sid
        bid=100
            (On-the-fly)
        sid=sid
            (Page-Oriented Nested loops)
            rating > 5 (On-the-fly) Reserves
        rating > 5 (On-the-fly) Sailors

Reserves
    sid=sid
        bid=100 (On-the-fly)
    rating > 5 (On-the-fly) Sailors
        Reserves
```

250,500 IOs

250,500 IOs
Alternative Plans – Push Selects (No Indexes)

250,500 IOs

6000 IOs
Alternative Plans – Push Selects
(No Indexes)

Reserves

6000 IOs

Sailors

4250 IOs

1000 + 500 + 250 + (10 * 250)
Alternative Plans – Push Selects
(No Indexes)

\[
\begin{align*}
\text{Reserves} & \quad \text{Sailors} \\
\text{Reserves} & \quad \text{Sailors}
\end{align*}
\]

![Decision tree diagram with operations and counts:](image)

- \( \text{sid}=\text{sid} \)
- \( \text{rating} > 5 \)
- \( \text{bid}=100 \)

4250 IOs

4010 IOs

\[500 + 1000 +10 +(250 \times 10)\]
Alternative Plans 1
(No Indexes)

Main difference: Sort Merge Join

With 5 buffers, cost of plan:

- Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
- Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
- Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
- Total: 4060 page I/Os. (note: T2 sort takes 4 passes with B=5)

If use BNL join, join = 10+4*250, total cost = 2770.

Can also `push' projections, but must be careful!

- T1 has only sid, T2 only sid, sname:
- T1 fits in 3 pgs, cost of BNL under 250 pgs, total < 2000.
**Alt Plan 2: Indexes**

- With clustered hash index on \( bid \) of Reserves, we get \( 100,000/100 = 1000 \) tuples on \( 1000/100 = 10 \) pages.

- INL with outer not materialized.

  - Projecting out unnecessary fields from outer doesn’t help.

  - Join column \( sid \) is a key for Sailors.
    
    At most one matching tuple, **unclustered index** on \( sid \) OK.

  - Decision not to push \( rating > 5 \) before the join is based on availability of \( sid \) index on Sailors.

  - **Cost:** Selection of Reserves tuples (10 I/Os); then, for each, must get matching Sailors tuple (1000*1.2); total **1210 I/Os.**
What is needed for optimization?

- Iterator Interface
- Cost Estimation
- Statistics and Catalogs
- Size Estimation and Reduction Factors
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

```
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
```

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.
Translating SQL to Relational Algebra

```
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
```

\[ \pi \text{S.sid, MIN(R.day)} (\text{HAVING COUNT(*)} \geq 2 (\text{GROUP BY S.Sid} (\sigma \text{B.color = "red"} \land \text{S.rating = val} (\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))))) \]
Relational Algebra Equivalences

- Allow us to choose different operator orders and to `push’ selections and projections ahead of joins.

**Selections:**

\[ \sigma_{c_1 \land \ldots \land c_n} (R) \equiv \sigma_{c_1} (\ldots \sigma_{c_n} (R)) \]  
(Cascade)

\[ \sigma_{c_1} (\sigma_{c_2} (R)) \equiv \sigma_{c_2} (\sigma_{c_1} (R)) \]  
(Commute)

**Projections:**

\[ \pi_{a_1} (R) \equiv \pi_{a_1} (\ldots (\pi_{a_n} (R))) \]  
(Cascade)

(if an includes an-1 includes… a1)

**Joins:**

\[ R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \]  
(Associative)

\[ (R \bowtie S) \equiv (S \bowtie R) \]  
(Commute)

These two mean we can do joins in any order.
More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.

- **Selection Push**: selection on \( R \) attrs commutes with

\[
R \bowtie S: \quad \sigma(R \bowtie S) \equiv \sigma(R) \bowtie S
\]

- **Projection Push**: A projection applied to \( R \bowtie S \) can be pushed before the join by retaining only attributes of \( R \) (and \( S \)) that are needed for the join or are kept by the projection.
Query optimization is an important task in a relational DBMS.

Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).

Two parts to optimizing a query:

1. Consider a set of alternative plans.
   - Must prune search space; typically, left-deep plans only.

2. Must 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.
The “System R” Query Optimizer

■ Impact:
  ➤ Inspired most optimizers in use today
  ➤ Works well for small-med 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.
  ➤ More sophisticated techniques known now.

■ Plan Space: Too large, must be pruned.
  ➤ Only the space of left-deep plans is considered.
  ➤ Cartesian products avoided.
Cost Estimation

- To estimate cost of a plan:
  - Must estimate cost of each operation in plan tree and sum them up.
    - Depends on input cardinalities.

  - So, must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

- In System R, cost is boiled down to a single number consisting of
  
  \#I/O ops + \textit{factor} \times \#CPU instructions
Statistics and Catalogs

Need information about the relations and indexes involved.

Catalogs typically contain at least:

- # tuples (NTuples) and # pages (NPages) per rel’n.
- # distinct key values (NKeys) for each index.
- low/high key values (Low/High) for each index.
- Index height (IHeight) for each tree index.
- # index pages (INPages) for each index.

Stats 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.
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 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) =
    # 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 columns
   (modern systems have removed this restriction)

Term \( col=value \)
    \[ RF = \frac{1}{N\text{Keys}(l)} \]

Term \( col1=col2 \) (This is handy for joins too…)
    \[ RF = \frac{1}{\text{MAX}(N\text{Keys}(l1), N\text{Keys}(l2))} \]

Term \( col>value \)
    \[ RF = \frac{(\text{High}(l)-value)}{(\text{High}(l)-\text{Low}(l))} \]

Note, In System R, if missing indexes, assume 1/10!!!
For better RF estimation, many systems use histograms:

### Equiwidth

<table>
<thead>
<tr>
<th>No. of Values</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>1</th>
<th>8</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0-.99</td>
<td>1-1.99</td>
<td>2-2.99</td>
<td>3-3.99</td>
<td>4-4.99</td>
<td>5-5.99</td>
<td>6-6.99</td>
</tr>
</tbody>
</table>

### Equidepth

<table>
<thead>
<tr>
<th>No. of Values</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0-.99</td>
<td>1-1.99</td>
<td>2-2.99</td>
<td>3-4.05</td>
<td>4.06-4.67</td>
<td>4.68-4.99</td>
<td>5-6.99</td>
<td></td>
</tr>
</tbody>
</table>
Postgres uses equidepth histograms (need to store just the boundaries) and Most Common Values (MCV).

Example:

```sql
most_common_vals l
{EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA}
most_common_freqs l {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

The estimator uses both histograms (for range queries) and MCVs for exact match queries (equality).

Sometimes, we both to estimate range queries and join results.

See more:

Q: Given a join of R and S, what is the range of possible result sizes (in #of tuples)?

- Hint: what if R and S have no attributes in common?
- Join attributes are a key for R (and a Foreign Key in S)?

General case: join attributes in common but a key for neither:

- estimate each tuple r of R generates $\frac{N_{Tuples}(S)}{N_{Keys}(A,S)}$ result tuples, so result size estimate:
  
  $$\frac{N_{Tuples}(R) \times N_{Tuples}(S)}{N_{Keys}(A,S)}$$

- but can also estimate each tuple s of S generates $\frac{N_{Tuples}(R)}{N_{Keys}(A,R)}$ result tuples, so:
  
  $$\frac{N_{Tuples}(R) \times N_{Tuples}(S)}{N_{Keys}(A,R)}$$

- If these two estimates differ, take the lower one!
Enumeration of Alternative Plans

There are two main cases:

- Single-relation plans (unary ops) and Multiple-relation plans

For unary operators:

- For a scan, each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.

- consecutive Scan, Select, Project and Aggregate operations can be essentially carried out together

  (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are pipelined into the aggregate computation).
I/O Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - Cost is $\text{Height}(I) + 1$ for a $B+$ tree, about 1.2 for hash index

- Clustered index I matching one or more selects:
  - $(\text{NPages}(I) + \text{NPages}(R)) \times \text{product of RF's of matching selects.}$

- Non-clustered index I matching one or more selects:
  - $(\text{NPages}(I) + \text{NTuples}(R)) \times \text{product of RF's of matching selects.}$

- Sequential scan of file:
  - $\text{NPages}(R)$.

- **Note:** Must also charge for duplicate elimination if required
Schema for Examples


- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. 100 distinct bids.

- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. 10 Ratings, 40,000 sids.
Example

- If we have an index on `rating`:
  - Cardinality: \((1/N\text{Keys}(I)) \times \text{NTuples}(S) = (1/10) \times 40000\) tuples retrieved.
  - Clustered index: \((1/N\text{Keys}(I)) \times (\text{NPages}(I)+\text{NPages}(S)) = (1/10) \times (50+500) = 55\) pages are retrieved.
  - Unclustered index: \((1/N\text{Keys}(I)) \times (\text{NPages}(I)+\text{NTuples}(S)) = (1/10) \times (50+40000) = 4005\) pages are retrieved.

- If we have an index on `sid`:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000. No reason to use this index! (see below)

- Doing a file scan:
  - We retrieve all file pages (500).
Cost-based Query Sub-System

Queries

Select *
From Blah B
Where B.blah = blah

Usually there is a heuristics-based rewriting step before the cost-based steps.
System R - Plans to Consider

For each block, plans considered are:

- All available access methods, for each relation in FROM clause.

- All left-deep join trees
  - i.e., all ways to join the relations one-at-a-time, considering all relation permutations and join methods.

(note: system R originally only had NL and Sort Merge)
Highlights of System R Optimizer

■ Impact:
  ➤ Most widely used currently; works well for < 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 combination of CPU and I/O costs.
    ▪ For simplicity we ignore CPU costs in this discussion
  ➤ More sophisticated techniques known now.

■ Plan Space: Too large, must be pruned.
  ➤ Only the space of left-deep plans is considered.
  ➤ Cartesian products avoided.
Queries Over Multiple Relations

- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined plans.*

- Intermediate results not written to temporary files.
- Not all left-deep trees are fully pipelined (e.g., SM join).
Plans differ by: order of the N relations, access method for each relation, and the join method for each join.

- maximum possible orderings = N! (but delay X-products)

Enumerated using N passes

For each subset of relations, retain only:

- Cheapest plan overall (possibly unordered), plus
- Cheapest plan for each *interesting order* of the tuples.
Enumeration: Dynamic Programming

- **Pass 1**: Find best 1-relation plans for each relation.

- **Pass 2**: Find best ways to join result of each 1-relation plan as outer to another relation. *(All 2-relation plans.)*
  
  *consider all possible join methods & inner access paths*

- **Pass N**: Find best ways to join result of a (N-1)-rel’n plan as outer to the N’th relation. *(All N-relation plans.)*
  
  *consider all possible join methods & inner access paths*
Interesting Orders

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
An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up.

i.e., avoid Cartesian products if possible.

ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered` plan or an additional sorting operator.

In spite of pruning plan space, this approach is still exponential in the # of tables.

COST = #IOs + (inst_per_IO * CPU Inst)
Example (modified from book ch 15)

Indexes

<table>
<thead>
<tr>
<th>Reserves:</th>
<th>Sailors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered B+ tree on bid</td>
<td>Unclust B+ tree on rating</td>
</tr>
</tbody>
</table>

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

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 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 (B+Tree selection on bid):**
Use Sort Merge to join with Sailors as inner

**Sailors as outer (File Scan w/select on rating):**
Use BNL on result of selection on Reserves.bid
Example (modified from book ch 15)

Select S.sid, COUNT(*) AS numredres
FROM  Sailors S, Reserves R, Boats B
AND B.color = “red”
GROUP BY S.sid

- **Pass1: Best plan(s) for accessing each relation**
  - Sailors: File Scan; B+ on sid
  - Reserves: File Scan; B+ on bid, B+ 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.bid (outer) with Boats (inner)
  - B+ on Reserves.bid (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 cheapest plan for each pair of relations plus cheapest plan for each interesting order.
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.
**Nested Queries**

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.

- Outer block is optimized with the cost of `calling` nested block computation taken into account.

- Implicit ordering of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

```
SELECT  S.sname
FROM    Sailors S
WHERE   EXISTS
    (SELECT  *
     FROM    Reserves R
     WHERE   R.bid=103
              AND  R.sid=S.sid)
```

Nested block to optimize:
```
SELECT  *
FROM    Reserves R
WHERE   R.bid=103
        AND  R.sid= outer value
```

Equivalent non-nested query:
```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid
        AND  R.bid=103
```
Points to Remember

- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).

- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must 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.
Points to Remember

Single-relation queries:

- All access paths considered, cheapest is chosen.

  *Issues*: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.
Multiple-relation queries:

- All single-relation plans are first enumerated.
  - Selections/projections considered as early as possible.
- Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
- Next, for each 2-relation plan that is `retained’, all ways of joining another relation (as inner) are considered, etc.
- At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained’.
Summary

- Performance can be dramatically improved by changing access methods, order of operators.
- Iterator interface
- Cost estimation
  - Size estimation and reduction factors
- Statistics and Catalogs
- Relational Algebra Equivalences
- Choosing alternate plans
- Multiple relation queries
- We focused on “System R”-style optimizers
  - New areas: Rule-based optimizers, random statistical approaches (eg simulated annealing), adaptive/dynamic optimization.