Acknowledgements: portions based on slides by Raghu Ramakrishnan and Johannes Gehrke, as well as slides by Zack Ives.
Class Agenda

• Last time:
  – Quiz #1
  – Query optimization

• Today:
  – Query optimization, continued

• Reading
  – Chapter 15 of Ramakrishnan and Gehrke (or Chapter 14 of Silberschatz et al)
Announcements

Reminder: DavisDB Part 2 due Sunday @11:59pm

DavisDB Part 3: out Sunday night, due Sunday 5/8 @11:59pm

Statistics for Quiz #1:
- avg: 15.9/19
- median: 16/19
- std: 2.5
- min: 10/19
- max: 20/19
Reminder: Implementation Hints for DavisDB Part 2

• Handling duplicates: what if many records have the same key value? Can circumvent by including (internally) the record id as part of the key
  – i.e., "key" becomes a pair <key, recordID>
  – No duplicates, by construction!

• Handling deletions: you are permitted to just use tombstones
  – When an entry is deleted, replace by a special marker indicating an empty slot (which may be reused later)
  – Internal nodes are never deleted or merged!
Relational Query Optimization, Continued
Query Optimization

• Given a SQL query:
  – Build a *logical query plan*: tree of algebraic operations
  – Transform into "better" logical plan
  – Convert into a *physical query plan*, using implementations of operators we've seen in the previous lectures

• Goal: find the physical query plan that has minimum cost
  – In practice: avoid the plans with the highest costs
  – Sources of cost: Interactions with other concurrent tasks; sizes of intermediate results; choices of algorithms, access methods; I/O and CPU; properties of data such as skew, order, placement; ...
Optimization Strategies

• Many possible strategies, all boil down to a search over the space of possible plans
  – Super-exponential complexity in the # of operators
  – Hence, exhaustive search generally not feasible

• What can you do?
  – Heuristics only: INGRES, Oracle until the mid-90s
  – Randomized, simulated annealing, ... : many efforts in the mid-90s
  – Heuristics plus cost-based join enumeration: System R
  – Stratified search (heuristics plus cost-based enumeration of joins and a few other operators): Starbust
  – Unified search (full cost-based search): EXODUS, Volcano, Cascades
Highlights of System R Optimizer

• Historically, the most influential optimizer design

• Cost estimation: approximate art at best
  – Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes
  – Considers combination of CPU and I/O costs

• Plan space: too large, must be pruned using heuristics
  – Only the space of left-deep plans is considered
  – Pipelined execution model: output of each operator is pipelined into the next operator, without storing it in a temporary relation
  – Cartesian products avoided

• Dynamic programming approach
**Query Blocks: Units of Optimization in System R**

SQL query parsed into a collection of *query blocks*, to be optimized one block at a time.

Nested blocks treated as calls to a subroutine, made once per outer tuple.

For each block, the 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, with the inner relation in the `from` clause, considering all join order permutations and join methods.
Left-Deep Join Trees

• Left-deep join tree:
  
  \[
  \begin{array}{c}
  & \& \\
  & \& U \\
  & \& T \\
  R & S
  \end{array}
  \]

• "Bushy" join tree:
  
  \[
  \begin{array}{c}
  & \& \\
  & \& & \& \\
  R & S & T & U
  \end{array}
  \]
Relational Algebra Equivalences

• Allow us to choose different join orders; to "push" selections and projections ahead of joins; etc

1. $\sigma_{F_1} (\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$

2. $\sigma_F(E_1 [\cup, \cap, \neg] E_2) \equiv \sigma_F(E_1) [\cup, \cap, \neg] \sigma_F(E_2)$

3. $\sigma_F(E_1 \times E_2) \equiv \sigma_{F_0}(\sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2))$
   \[ F \equiv F_0 \wedge F_1 \wedge F_2, \text{Fi contains only attributes of } E_i, i = 1, 2. \]

4. $\sigma_{A=B}(E_1 \times E_2) \equiv E_1 \bowtie_{A=B} E_2$

5. $\pi_A(E_1 [\cup, \cap, \neg] E_2) \equiv \pi_A(E_1) [\cup, \cap, \neg] \pi_A(E_2)$
Relational Algebra Equivalences (2)

6. \( \pi_A(E_1 \times E_2) \equiv \pi_{A_1}(E_1) \times \pi_{A_2}(E_2), \)
   with \( A_i = A \cap \{ \text{attributes in } E_i \}, i = 1, 2. \)

7. \( E_1 [\cup, \cap] E_2 \equiv E_2 [\cup, \cap] E_1 \)
   \( (E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3) \) (the analogous holds for \( \cap \))

8. \( E_1 \times E_2 \equiv \pi_{A_1,A_2}(E_2 \times E_1) \)
   \( (E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3) \)
   \( (E_1 \times E_2) \times E_3 \equiv (E_1 \times E_3) \times E_2 \)

9. \( E_1 \bowtie E_2 \equiv E_2 \bowtie E_1 \)
   \( (E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3) \)

(Theoretical aside: is this set of equivalences complete?)
Enumeration of Alternative Plans

• There are two main cases:
  – Single-relation plans
  – Multiple-relation plans

• Single-relation plans: queries consist of a combination of selections, projections, and aggregates (no joins)
  – Each available access path (file or index scan) is considered, and the one with the least estimated cost is chosen
  – The different operations are carried out together in a pipeline (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)
Cost Estimation

- Must estimate cost of each plan considered
- To do this, must estimate cost of each operation in plan tree
  - Depends on input cardinalities, statistical properties, etc
- Must also estimate size of result for each operation in tree!
  - Use information about the input relations
  - For selections and joins, assume independence of predicates
- Dirty little secret of DBMS world: estimation works well for simple plans, but poorly for complex plans
Cost Estimation for Single-Relation Plans

• Clustered index $I$ matching one or more selections:
  - cost $\approx (\# \text{ pages in } I) \times \text{product of RF's* of matching selects}$

• Non-clustered index $I$ matching one or more selections:
  - cost $\approx (\# \text{ pages in } I + \# \text{ tuples in } R) \times \text{product of RF's of matching selects}$

• Sequential scan of file:
  - cost $\approx \# \text{ of pages in } R$

• Extra cost for duplicate elimination if user says select distinct

* RF is "reduction factor" : what % of the data passes the selection condition
Queries Over Multiple Relations

- Fundamental heuristic in System R: *only left-deep join trees are considered*
- As the # of joins increases, the # of alternative plans grows very rapidly; we need to restrict the search space
- Left-deep join trees allow us to generate all *fully pipelined* plans
  - i.e., intermediate results not written to temporary files (not "materialized")
  - not all left-deep physical plans are fully pipelined
- Bushy join trees: can't have fully pipelined plans
  - Inner table must always be materialized for each tuple of the outer table
  - So, a plan in which the inner table is the result of a join forces us to materialize the result of that join
Enumeration of Left-Deep Plans

• Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.

• Enumeration via dynamic programming strategy: $n$ passes, where $n = \# \text{ relations joined}$
  
  – Pass 1: find best 1-relation plan for each relation
  
  – Pass 2: find best way to join result of each 1-relation plan (as outer) to another relation
  
  – Pass $n$: find best way to join result of each $(n-1)$-relation plan (as outer) to the $n$th relation

• For each subset of relations, retain only:
  
  – Cheapest plan overall, plus
  
  – Cheapest plan for each "interesting order" of the tuples
Enumeration of Plans (2)

• order by, group by, aggregates, etc. handled as a final step, using either an "interestingly ordered" plan or an additional sorting operator

• An \((n-1)\)-way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in \textit{where} have been used up
  
  – i.e., avoid Cartesian products if possible

• In spite of pruning plan space, this approach is still exponential in the \# of tables
• Assume: B+ tree index on Sailors.rating; hash index on Sailors.sid; B+ tree index on Reserves.bid (all unclustered)
Enumeration of Plans Example: Pass 1

• Consider access path methods for single relations
  – **Sailors**: three access methods (B+ tree, hash index, sequential scan), taking into account selection $\sigma_{\text{rating}>5}$.
    • B+ tree? Yes, matches $\sigma$; also returns tuples sorted by rating
    • Hash index? Sequential scan? More costly than B+ tree
  => **B+ tree preferred, with tuples sorted by rating**
  – **Reserves**: two access methods (B+ tree, sequential scan), taking into account selection $\sigma_{\text{bid}=100}$.
    • B+ tree? yes, matches $\sigma$
    • Sequential scan? Slower than B+ tree
  => **B+ tree preferred**
Enumeration of Plans Example: Pass 2

- Consider all two-relation plans, using access method from Pass 1 for outer relation in join

**Reserves outer, Sailors inner:**
- Need only `Sailors` tuples that satisfy $\sigma_{\text{rating}>5}$ and $\sigma_{\text{sid}=\text{value}}$ where `value` is some value from an outer tuple
- Access method for `Sailors`:
  - B+ tree? Yes, matches $\sigma_{\text{rating}>5}$
  - Hash index? Yes, matches $\sigma_{\text{sid}=\text{value}}$; = more selective than $>$
  => **Hash index preferred**
- Alternative join methods: all are considered, e.g.,
  - Sort-merge join: inputs must be sorted by `sid`; no single-relation access method returns them sorted this way, so requires extra sort
  - Index nested loops: can use, since have hash index on `Sailors.sid`
  - etc
  => **Index nested loops join preferred**
Enumeration of Plans Example: Pass 2 (cont)

Sailors outer, Reserves inner:

– Need only Reserves tuples that satisfy \( \sigma_{bid=100} \) and \( \sigma_{sid=value} \) where value is some value from an outer tuple

– Choose access method for Reserves

  • ...

– Choose preferred join algorithm

  • ...

– Retain cheapest plan overall: e.g.,

  Index nested loops join with Reserves outer, Sailors inner preferred

• Pass 2 is the last pass, so we output this as the plan
Enumeration of Plans Example (2): Pass 3

```
SELECT S.sid, B.bid
FROM Reserves R, Sailors S, Boats B
AND B.color = "red"
```

- For each plan retained in Pass 2, taken as the outer relation, consider how to join the remaining relation as the inner one
  - \{Reserves, Sailors\} outer, Boats inner
  - \{Reserves, Boats\} outer, Sailors inner: not considered!
    - no join condition for \{Reserves, Boats\}
  - \{Sailors, Boats\} outer, Reserves inner: also not considered!
    - no join condition for \{Sailors, Boats\}
Cost Estimation for Multi-Relation Plans

SELECT attribute-list
FROM relation-list
WHERE term₁ AND ... AND termₖ

• Key issue: estimating cardinalities of intermediate results
• Maximum # tuples in result is the product of the cardinalities of relations in the from clause
• *Reduction factor* (RF) associated with each *term* reflects the impact of the term in reducing result size. Result cardinality \( \approx \max \# \text{ tuples} \times \text{product of all RF's} \)
• Multirelation plans are built up by joining one new relation at a time
  – Cost of join method plus estimation of join cardinality gives us both cost estimate and result size estimate
• Errors at each step are compounded!
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 consideration.

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

```sql
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:
```sql
SELECT *
FROM Reserves R
WHERE R.bid = 103
AND R.sid = outer value
```

Equivalent non-nested query:
```sql
SELECT S.name
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
AND R.bid = 103
```
Summary

• Query optimization: crucial task in relational DBMS
  – Declarative query language requires powerful optimizer

• Even an end-user (DBA) must understand optimization in order to understand the performance impact of a given database design (schema, indices, etc) on a workload (expected queries and updates)

• Two parts to optimizing a query:
  – Explore the space of alternative plans
    • Must prune search space; System R considers 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, indices, operator implementations
Summary (continued)

• 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: greedy, inductive approach
  – Base case: All single-relation plans are first enumerated
    • Selections/projections considered as early as possible
  – Inductive case: for each $n$-relation plan, all ways of joining another relation (as inner) are considered, to produce an $n+1$ relation plan
  – At each level, for each subset of relations, only best plan (for each "interesting order" of tuples) is retained