Chapter 14
Query Optimization

Chapter 14: Query Optimization

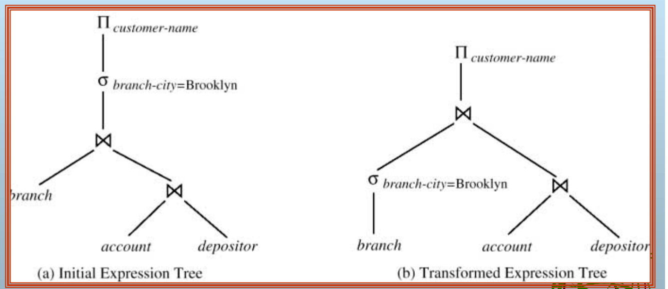
- Introduction
- Catalog Information for Cost Estimation
- Estimation of Statistics
- Transformation of Relational Expressions
- Dynamic Programming for Choosing Evaluation Plans

Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation (Chapter 13)
- Cost difference between a good and a bad way of evaluating a query can be enormous
  - Example: performing a $r \times s$ followed by a selection $r.A = s.B$ is much slower than performing a join on the same condition
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
    - E.g. number of tuples, number of distinct values for join attributes, etc.
  - Need to estimate statistics for intermediate results to compute cost of complex expressions

Introduction (Cont.)

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.



Introduction (Cont.)

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions
     - Use **equivalence rules** to transform an expression into an equivalent one.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on **estimated cost**
- The overall process is called **cost based optimization.**

Overview of chapter

- Statistical information for cost estimation
- Equivalence rules
- Cost-based optimization algorithm
- Optimizing nested subqueries
- Materialized views and view maintenance

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- s_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\Pi_A(r)$.
- *SC*(*A, r*): selection cardinality of attribute *A* of relation *r*; average number of records that satisfy equality on *A*.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Catalog Information about Indices

- f_i: average fan-out of internal nodes of index *i*, for tree-structured indices such as B+-trees.
- HT_i: number of levels in index *i* — i.e., the height of *i.*
  - For a balanced tree index (such as B+-tree) on attribute *A* of relation *r*, $HT_i = \lceil \log_{f_i}(V(A,r)) \rceil$.
  - For a hash index, HT_i is 1.
  - LB_i: number of lowest-level index blocks in *i* — i.e, the number of blocks at the leaf level of the index.

Measures of Query Cost

- Recall that
  - Typically disk access is the predominant cost, and is also relatively easy to estimate.
  - The *number of block transfers from disk* is used as a measure of the actual cost of evaluation.
  - It is assumed that all transfers of blocks have the same cost.
    - Real life optimizers do not make this assumption, and distinguish between sequential and random disk access
- We do not include cost to writing output to disk.
- We refer to the cost estimate of algorithm *A* as E_A

Selection Size Estimation

- **Equality selection** $\sigma_{A=v}(r)$
  - *SC(A, r)* : number of records that will satisfy the selection
  - $\lceil SC(A, r)/f_r \rceil$ — number of blocks that these records will occupy
  - E.g. Binary search cost estimate becomes

$$E_{a2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1$$

  - Equality condition on a key attribute: *SC(A,r)* = 1

Statistical Information for Examples

- $f_{account}$ = 20 (20 tuples of *account* fit in one block)
- *V(branch-name, account)* = 50 (50 branches)
- *V(balance, account)* = 500 (500 different *balance* values)
- $\pi_{account}$ = 10000 (*account* has 10,000 tuples)
- Assume the following indices exist on *account:*
  - A primary, B+-tree index for attribute *branch-name*
  - A secondary, B+-tree index for attribute *balance*

Selections Involving Comparisons

- Selections of the form $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
- Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - C = 0 if v < min(A,r)
    - C = $n_r . \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Implementation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i . If s_i is the number of satisfying tuples in *r*, the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$. The estimate for number of tuples in the result is: $n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$
- **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:
  $n_r - size(\sigma_\theta(r))$

Join Operation: Running Example

Running example:
depositor ⋈ *customer*

Catalog information for join examples:

- $n_{customer}$ = 10,000.
- $f_{customer}$ = *25,* which implies that
  $b_{customer}$ =10000/25 = 400.
- $n_{depositor}$ = 5000.
- $f_{depositor}$ = 50, which implies that
  $b_{depositor}$ = 5000/50 = 100.
- *V(customer-name, depositor)* = 2500, which implies that , on average, each customer has two accounts.

  Also assume that *customer-name* in *depositor* is a foreign key on *customer.*

Estimation of the Size of Joins

- The Cartesian product *r* x *s* contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then *r* ⋈ *s* is the same as *r* x *s*.
- If $R \cap S$ is a key for *R*, then a tuple of *s* will join with at most one tuple from *r*
  - therefore, the number of tuples in *r* ⋈ *s* is no greater than the number of tuples in *s*.
- If $R \cap S$ *in* S is a foreign key in *S* referencing *R*, then the number of tuples in *r* ⋈ *s* is exactly the same as the number of tuples in *s*.
  - The case for $R \cap S$ being a foreign key referencing *S* is symmetric.
- In the example query *depositor* ⋈ *customer, customer-name* in *depositor* is a foreign key of *customer*
  - hence, the result has exactly $n_{depositor}$ tuples, which is 5000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for *R* or *S*.
  If we assume that every tuple *t* in *R* produces tuples in *R* ⋈ *S,* the number of tuples in *R* ⋈ *S* is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(customer-name, depositor)* = 2500, and *V(customer-name, customer)* = 10000
  - The two estimates are 5000 * 10000/2500 - 20,000 and 5000 * 10000/10000 = 5000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\Pi_A(r)$ = $V(A,r)$
- Aggregation : estimated size of $_A\mathcal{g}_F(r)$ = $V(A,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1}\, \sigma_{\theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Size Estimation (Cont.)

- Outer join:
  - Estimated size of r ⟕ s = *size of* r ⋈ s + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of r ⟗ s = *size of* r ⋈ s + size of r + size of s

Estimation of Number of Distinct Values

Selections: $\sigma_\theta(r)$

- If θ forces *A* to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., *A* = 3
- If θ forces A to take on one of a specified set of values: $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., (*A* = 1 *V* *A* = 3 *V* *A* = 4)),
- If the selection condition θ is of the form *A op r* estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
  - where *s* is the selectivity of the selection.
- In all the other cases: use approximate estimate of $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: r ⋈ s

- If all attributes in *A* are from *r* estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If *A* contains attributes *A*1 from *r* and *A*2 from *s*, then estimated $V(A,r \bowtie s)$ =
  $\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_{A(r)}$ as in *r*.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(*A*) and max(*A*), the number of distinct values can be estimated as min(V(*A*,*r*), *V*(*G*,*r*)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use *V*(*G*,*r*)

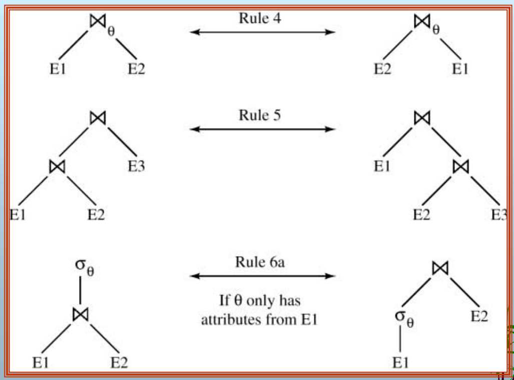
Transformation of Relational Expressions

- Two relational algebra expressions are said to be equivalent if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples
- An equivalence rule says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
   $$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$
2. Selection operations are commutative.
   $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
   $$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$
4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

8. The projections operation distributes over the theta join operation as follows:

   (a) if Π involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.
   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

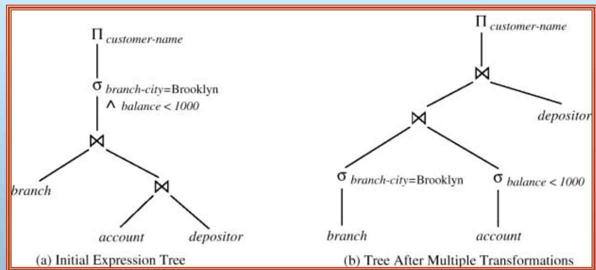
Transformation Example

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.
  $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch \bowtie (account \bowtie depositor)))$
- Transformation using rule 7a.
  $\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch)) \bowtie (account \bowtie depositor))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.
  $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$
- Transformation using join associatively (Rule 6a):
  $\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie (account)) \bowtie depositor)$
- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression
  $\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch) \bowtie \sigma_{balance > 1000}(account)$
- Thus a sequence of transformations can be useful

Multiple Transformations (Cont.)



(a) Initial Expression Tree

(b) Tree After Multiple Transformations

Projection Operation Example

$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie\ account) \bowtie\ depositor)$

- When we compute

  $(\sigma_{\text{branch-city = “Brooklyn”}}\ (branch) \bowtie account\ )$

  we obtain a relation whose schema is:
  (branch-name, branch-city, assets, account-number, balance)
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

  $\Pi_{\ customer\text{-}name}$ ((
  $\Pi_{\ account\text{-}number}$ ($(\sigma_{\text{branch-city = “Brooklyn”}}\ (branch) \bowtie account\ ))$
  $\bowtie depositor)$

Join Ordering Example

- For all relations $r_1, r_2,$ and r_3,

  $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3\ )$
- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

  $(r_1 \bowtie r_2) \bowtie r_3$

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

  $\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch))$
  $\bowtie \text{account} \bowtie \text{depositor})$
- Could compute *account* $\bowtie$ *depositor* first, and join result with

  $\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)$

  but *account* $\bowtie$ *depositor* is likely to be a large relation.
- Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute

  $\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie account$

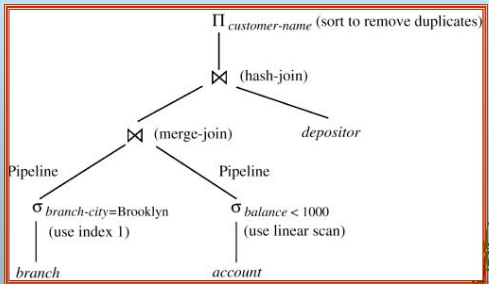
  first.

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to systematically generate expressions equivalent to the given expression
- Conceptually, generate all equivalent expressions by repeatedly executing the following step until no more expressions can be found:
  - for each expression found so far, use all applicable equivalence rules, and add newly generated expressions to the set of expressions found so far
- The above approach is very expensive in space and time
- Space requirements reduced by sharing common subexpressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared
    - E.g. when applying join associativity
- Time requirements are reduced by not generating all expressions
  - More details shortly

Evaluation Plan

- An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans: choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
  - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

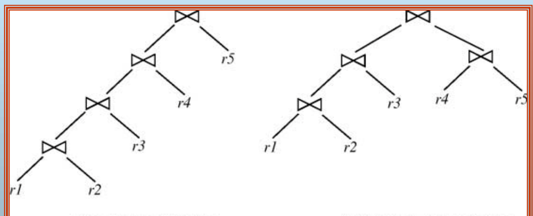
- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
  - Recursively compute costs for joining subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives.
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Join Order Optimization Algorithm

procedure findbestplan(*S*)
    if (*bestplan*[*S*].*cost* ≠ ∞)
        return *bestplan*[*S*]
    // else *bestplan*[*S*] has not been computed earlier, compute it now
    for each non-empty subset *S*1 of *S* such that *S*1 ≠ *S*
        P1= findbestplan(*S*1)
        P2= findbestplan(*S* - *S*1)
        A = best algorithm for joining results of *P*1 and *P*2
        cost = *P*1.*cost* + *P*2.*cost* + cost of *A*
        if *cost* < *bestplan*[*S*].*cost*
            bestplan[*S*].*cost* = cost
            bestplan[*S*].*plan* = "execute *P*1.*plan*; execute *P*2.*plan*;
                    join results of *P*1 and *P*2 using *A*"
    return *bestplan*[*S*]

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep Join Tree   (b) Non-left-deep Join Tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the n alternatives.
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Orders in Cost-Based Optimization

- Consider the expression $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation.
  - Generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common only r_1 and r_2 is not useful.
  - Using merge-join to compute $r_1 \bowtie r_2 \bowtie r_3$ may be costlier, but may provide an output sorted in an interesting order.
- Not sufficient to find the best join order for each subset of the set of n given relations; must find the best join order for each subset, for each interesting sort order
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Steps in Typical Heuristic Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining).

Structure of Query Optimizers

- The System R/Starburst optimizer considers only left-deep join orders. This reduces optimization complexity and generates plans amenable to pipelined evaluation.
  System R/Starburst also uses heuristics to push selections and projections down the query tree.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these.
- For scans using secondary indices, some optimizers take into account the probability that the page containing the tuple is in the buffer.
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
- This expense is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.

Optimizing Nested Subqueries**

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- E.g.
  select *customer-name*
  from *borrower*
  where exists (**select** *
  from *depositor*
  where *depositor.customer-name* = *borrower.customer-name*)
- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Optimizing Nested Subqueries (Cont.)

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.: earlier nested query can be rewritten as
  select *customer-name*
  from *borrower, depositor*
  where *depositor.customer-name* = *borrower.customer-name*
  - Note: above query doesn't correctly deal with duplicates, can be modified to do so as we will see
- In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
  - A temporary relation is created instead, and used in body of outer level query

Optimizing Nested Subqueries (Cont.)

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** …
  from L_1
  where P_1 **and exists** (**select** *
  from L_2
  where P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2
  where P_2^1
  select …
  from L_1, t_1
  where P_1 **and** P_2^2
  - P_2^1 contains predicates in P_2 that do not involve any correlation variables
  - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries (Cont.)

- In our example, the original nested query would be transformed to
  create table t_1 **as**
  select distinct *customer-name*
  from *depositor*
  select *customer-name*
  from *borrower*, t_1
  where t_1.*customer-name* = *borrower.customer-name*
- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated when
  - the nested subquery uses aggregation, or
  - when the result of the nested subquery is used to test for equality, or
  - when the condition linking the nested subquery to the other query is **not exists**,
  - and so on.

Materialized Views**

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  create view *branch-total-loan*(*branch-name, total-loan*) **as**
  select *branch-name*, **sum**(*amount*)
  from *loan*
  groupby *branch-name*
- Materializing the above view would be very useful if the total loan amount is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Supported directly by the database

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - $R = (A,B)$, and $r(R) = \{ (a,2), (a,3)\}$
  - $\Pi_A(r)$ has a single tuple (a).
  - If we delete the tuple $(a,2)$ from r, we should not delete the tuple (a) from $\Pi_A(r)$, but if we then delete $(a,3)$ as well, we should delete the tuple
- For each tuple in a projection $\Pi_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\Pi_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\Pi_A(r)$
    - if the count becomes 0, we delete the tuple from $\Pi_A(r)$

Aggregation Operations

- count : $v = {}_A g_{count(B)}(r)$.
  - When a set of tuples i_r is inserted
    - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted
    - for each tuple t in i_r we look for the group $t.A$ in v, and subtract 1 from the count for the group.
      - If the count becomes 0, we delete from v the tuple for the group $t.A$
- sum: $v = {}_A g_{sum\ (B)}(r)$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)
- To handle the case of **avg**, we maintain the **sum** and **count** aggregate values separately, and divide at the end

Aggregate Operations (Cont.)

- **min**, **max**: $v = {}_A g_{min\ (B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of r that are in the same group to find the new minimum

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in r we check if it is present in s, and if so we add it to v.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to s are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work
  - we leave details to you.

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g. consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that s has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace v by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: "What is the best set of views to materialize?".
  - This decision must be made on the basis of the system **workload**
- Indices are just like materialized views, problem of **index selection** is closely related, to that of materialized view selection, although it is simpler.
- Some database systems, provide tools to help the database administrator with index and materialized view selection.

End of Chapter

(Extra slides with details of selection cost estimation follow)

Selection Cost Estimate Example

$\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}}(account)$

- Number of blocks is $b_{account}$ = 500: 10,000 tuples in the relation; each block holds 20 tuples.
- Assume *account* is sorted on *branch-name.*
  - *V(branch-name,account)* is 50
  - 10000/50 = 200 tuples of the *account* relation pertain to Perryridge branch
  - 200/20 = 10 blocks for these tuples
  - A binary search to find the first record would take $\lceil log_2(500) \rceil$ = 9 block accesses
- Total cost of binary search is 9 + 10 -1 = 18 block accesses (versus 500 for linear scan)

Selections Using Indices

- **Index scan** – search algorithms that use an index; condition is on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition $E_{A3} = HT_i + 1$
- **A4** (*primary index on nonkey, equality)* Retrieve multiple records. Let the search-key attribute be *A*.

$$E_{A4} = HT_i + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil$$

- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key $E_{A5} = HT_i + 1$
  - Retrieve multiple records (each may be on a different block) if the search-key is not a candidate key. $E_{A3} = HT_i + SC(A,r)$

Cost Estimate Example (Indices)

Consider the query is $\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}}(account)$, with the primary index on *branch-name.*

- Since *V(branch-name, account)* = 50, we expect that 10000/50 = 200 tuples of the *account* relation pertain to the Perryridge branch.
- Since the index is a clustering index, 200/20 = 10 block reads are required to read the *account* tuples.
- Several index blocks must also be read. If B$^+$-tree index stores 20 pointers per node, then the B$^+$-tree index must have between 3 and 5 leaf nodes and the entire tree has a depth of 2. Therefore, 2 index blocks must be read.
- This strategy requires 12 total block reads.

Selections Involving Comparisons

selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using a linear file scan or binary search, or by using indices in the following ways:

- **A6** (*primary index, comparison).* The cost estimate is:

$$E_{AB} = HT_i + \left\lceil \frac{c}{f_r} \right\rceil$$

where *c* is the estimated number of tuples satisfying the condition. In absence of statistical information *c* is assumed to be $n_r/2$.

- **A7** (*secondary index, comparison*). The cost estimate:

$$E_{A7} = HT_i + \frac{LB_i \cdot c}{n_r} + c$$

where *c* is defined as before. (Linear file scan may be cheaper if *c* is large!).

Example of Cost Estimate for Complex Selection

- Consider a selection on *account* with the following condition: **where** *branch-name* = "Perryridge" **and** *balance* = 1200
- Consider using algorithm A8:
  - The *branch-name* index is clustering, and if we use it the cost estimate is 12 block reads (as we saw before).
  - The *balance* index is non-clustering, and *V(balance, account* = 500, so the selection would retrieve 10,000/500 = 20 accounts. Adding the index block reads, gives a cost estimate of 22 block reads.
  - Thus using *branch-name* index is preferable, even though its condition is less selective.
  - If both indices were non-clustering, it would be preferable to use the *balance* index.

Example (Cont.)

- Consider using algorithm A10:
  - Use the index on *balance* to retrieve set S_1 of pointers to records with *balance* = 1200.
  - Use index on *branch-name* to retrieve-set S_2 of pointers to records with *branch-name* = Perryridge".
  - $S_1 \cap S_2$ = set of pointers to records with *branch-name* = "Perryridge" and *balance* = 1200.
  - The number of pointers retrieved (20 and 200), fit into a single leaf page; we read four index blocks to retrieve the two sets of pointers and compute their intersection.
  - Estimate that one tuple in 50 * 500 meets both conditions. Since $n_{account}$ = 10000, conservatively overestimate that $S_1 \cap S_2$ contains one pointer.
  - The total estimated cost of this strategy is five block reads.