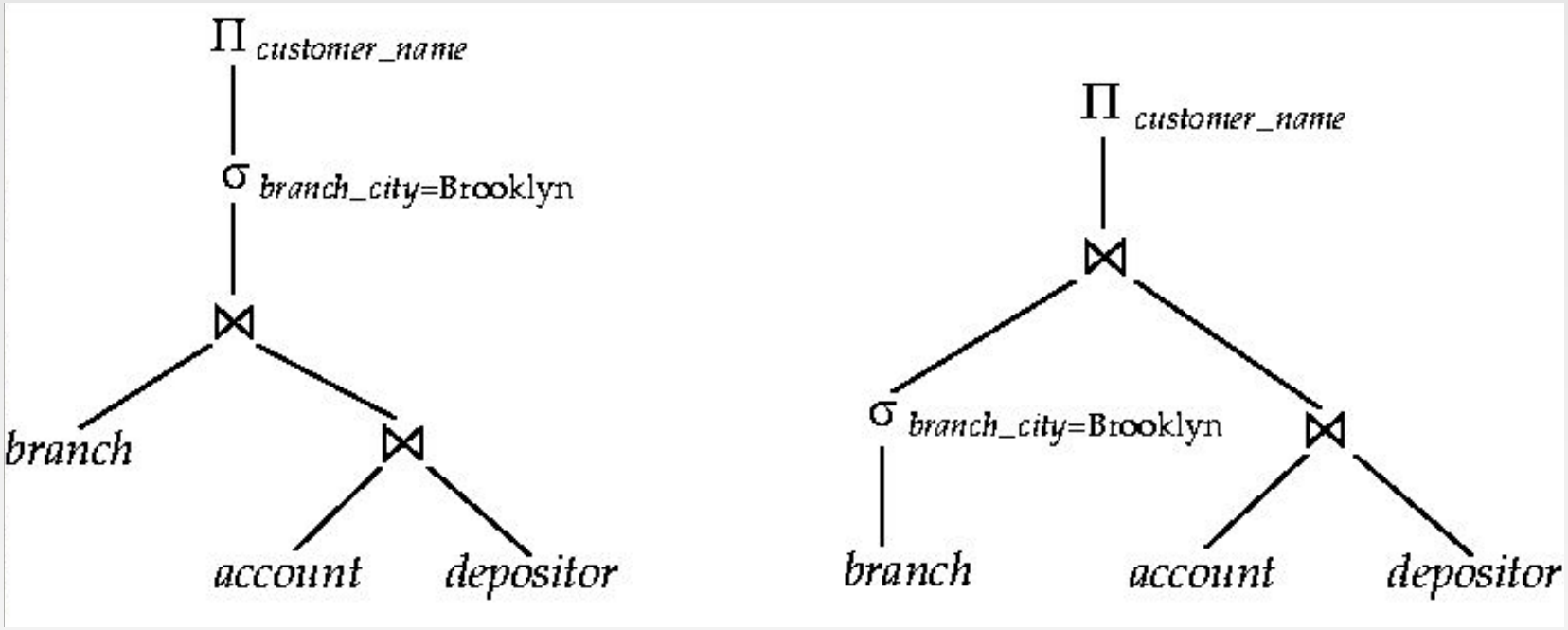
Query Optimization

Query Optimization

- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Materialized views

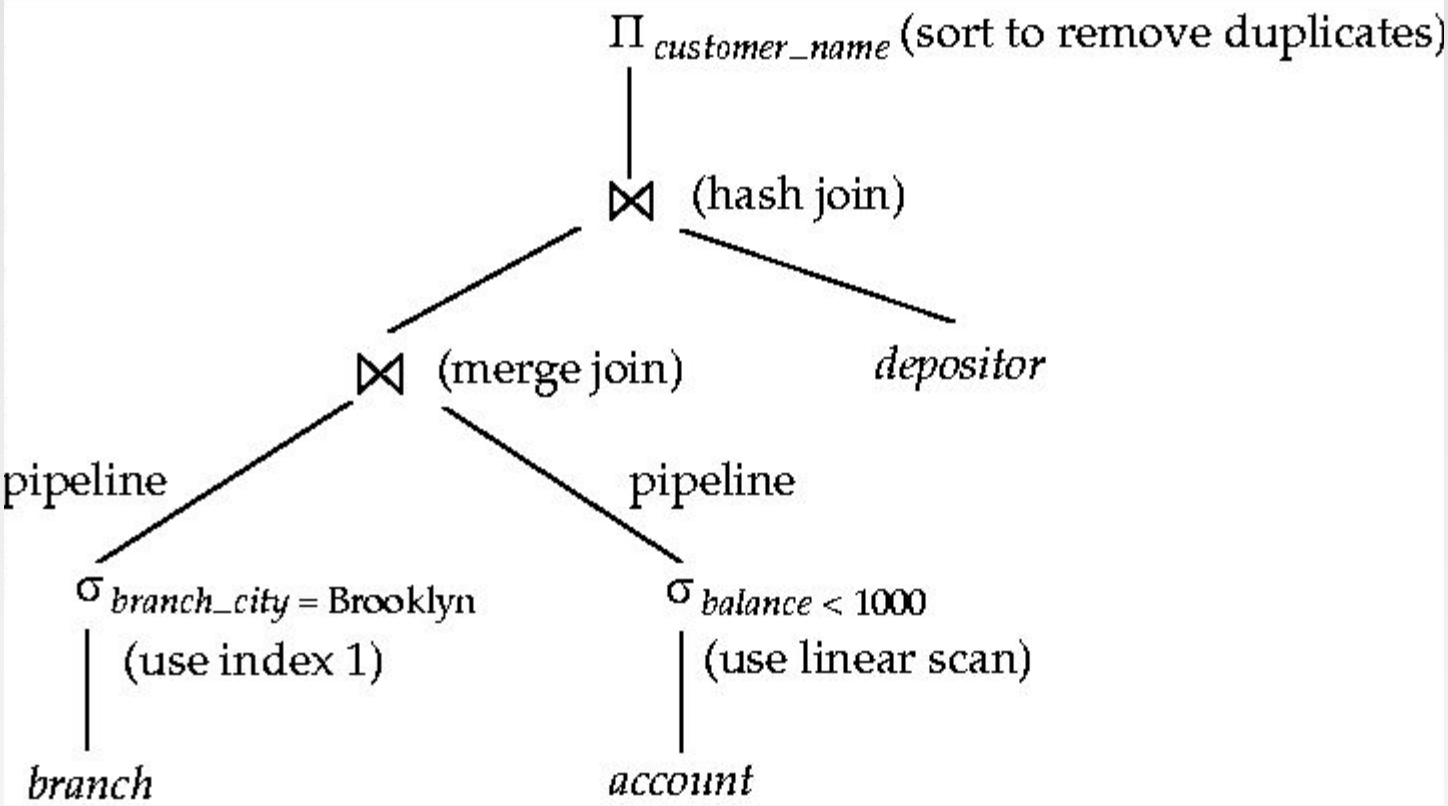
Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



Introduction (Cont.)

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



Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance
  - 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 the two expressions generate the same multiset of tuples on every legal database instance.
- 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.

$$\Pi_{L_1}(\Pi_{L_2}(\square\ (\Pi_{Ln}(E))\square\ )) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_{\theta}(E_1 \text{ X } E_2) = E_1 \bowtie_{\theta} E_2$

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

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:

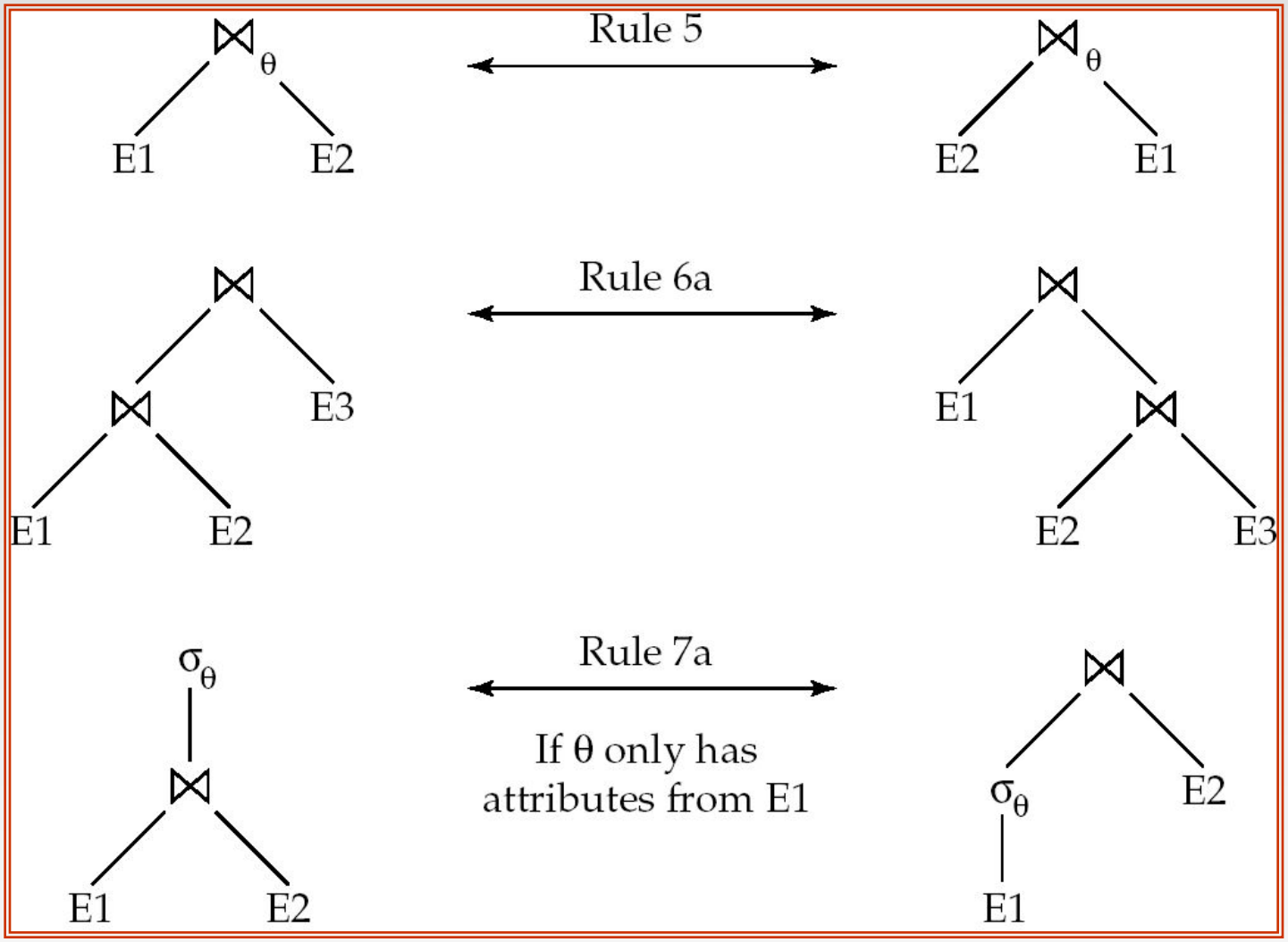
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

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

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

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



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 projection operation distributes over the theta join operation as follows:

   (a) if θ 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: Pushing Selections

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

$$\Pi_{customer\_name}(\sigma_{branch\_city = \text{“Brooklyn”}}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using rule 7a.

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch)) \bowtie (account \bowtie depositor))$$

- 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.

$$\Pi_{customer\_name}(\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account \bowtie depositor))$$

- Transformation using join associatively (Rule 6a):

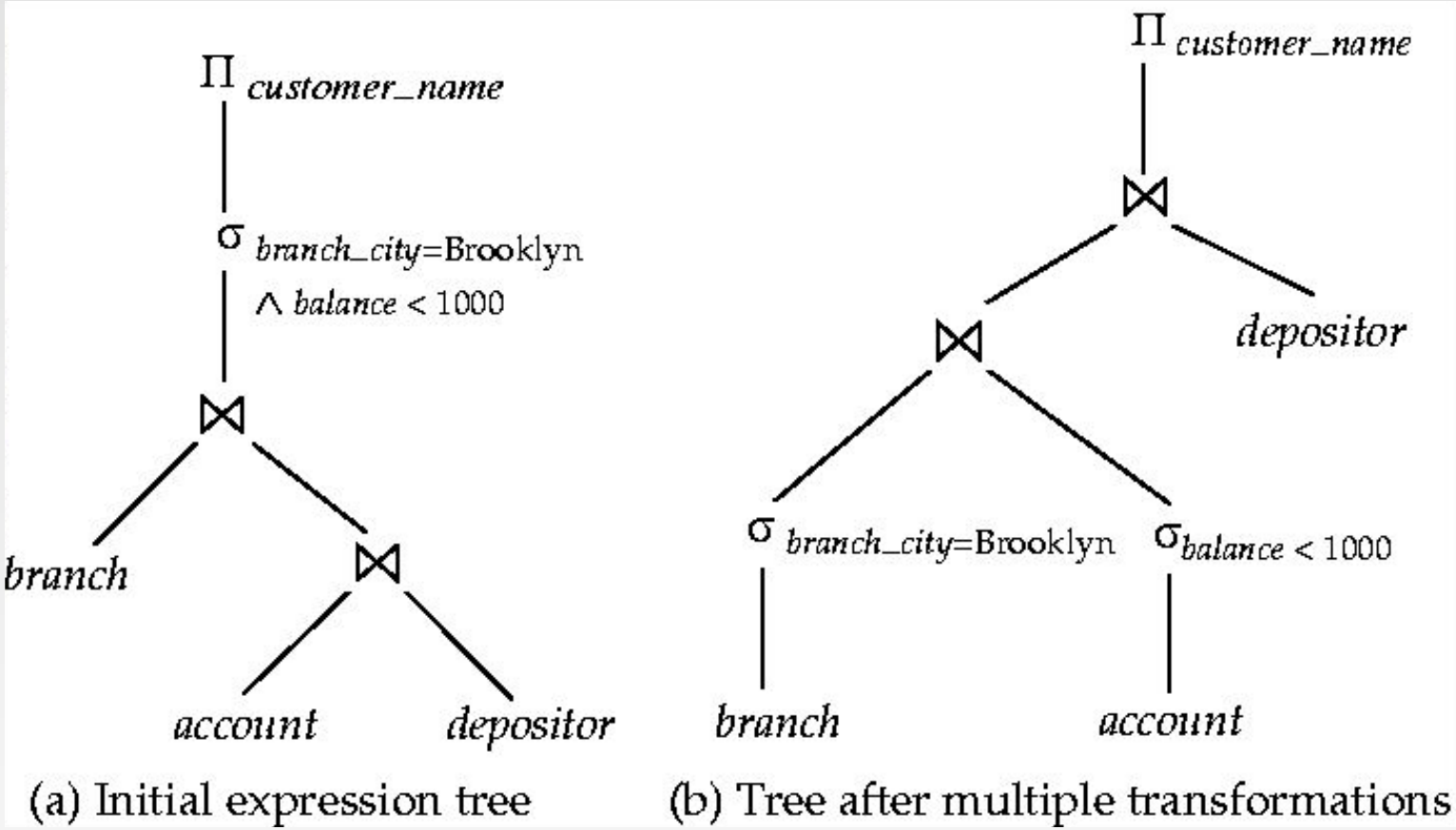
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie \sigma_{balance > 1000}(account)$$

- Thus a sequence of transformations can be useful

Multiple Transformations (Cont.)



Transformation Example: Pushing Projections

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account) \bowtie depositor)$$

- When we compute

  $$(\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account) \bowtie depositor$$

  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:

  $$\Pi_{customer\_name}((\Pi_{account\_number}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account)) \bowtie depositor)$$

- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,

  $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$

  (Join Associativity)
- 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

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch)) \bowtie account \bowtie depositor))$$

- Could compute $account \bowtie depositor$ first, and join result with

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

but $account \bowtie depositor$ is likely to be a large relation.

- Only a small fraction of the bank's customers are likely to have accounts in branches located in Brooklyn
  - it is better to compute

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

first.

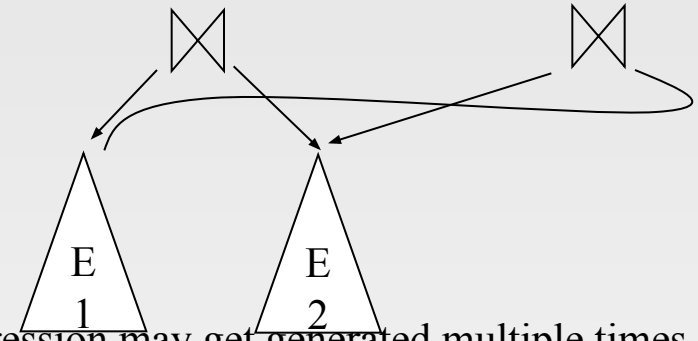
Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - 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 using pointers
    - E.g. when applying join commutativity



  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computer as described in Chapter 13
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

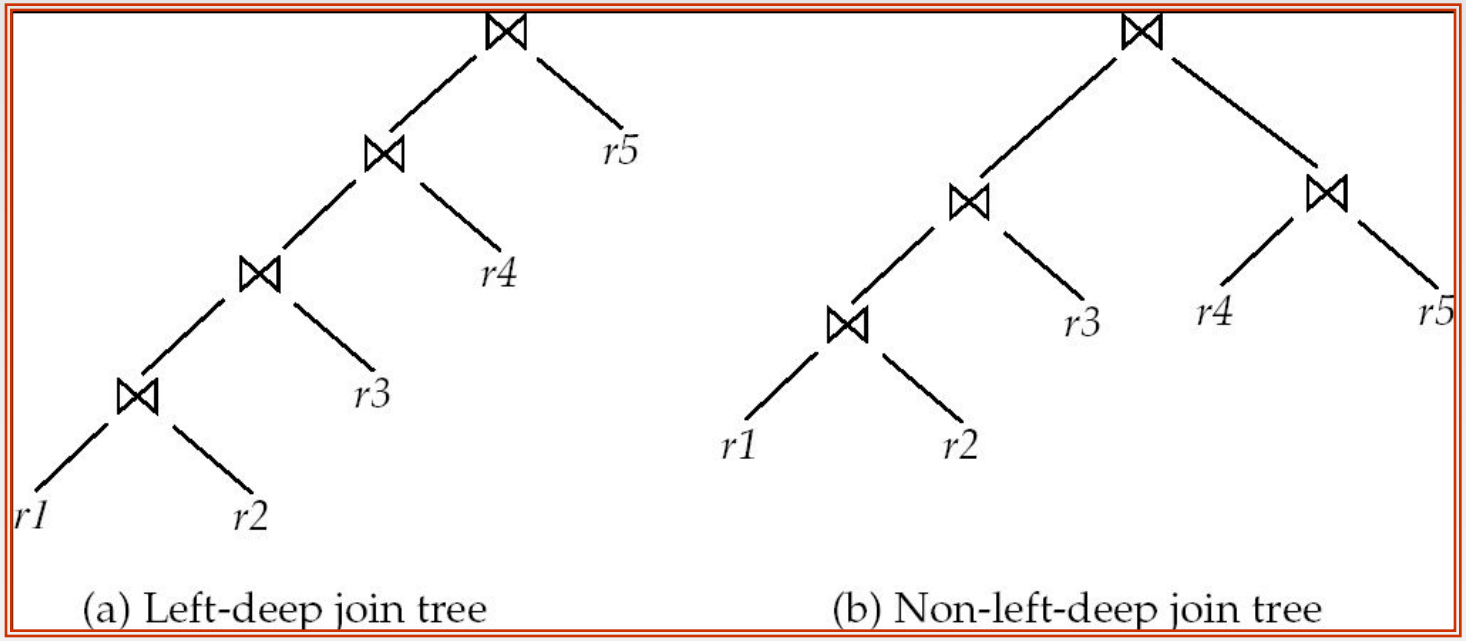
- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost 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 $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- 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.

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.



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 (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- 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
- 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.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

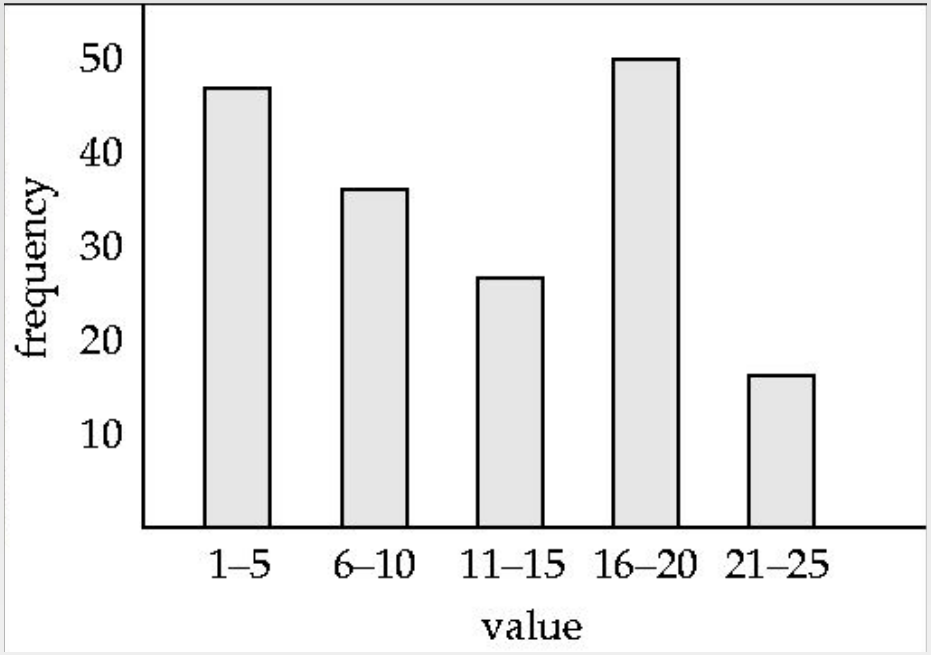
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_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 $\prod_A(r)$.
- If tuples of r are stored together physically in a file, then:

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

Histograms

- Histogram on attribute *age* of relation *person*



- Equi-width histograms
- Equi-depth histograms

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \le V}(r)$ (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 = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation 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:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming indepdence,* estimate 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:

$depositor \bowtie customer$

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.*
  - *V(customer_name, customer) = 10000 (primary key!)*

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 $R \cap S = \varnothing$, then $r \bowtie s$ 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 $r \bowtie s$ 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 $r \bowtie s$ 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* $\bowtie$ *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 $R \bowtie S$, the number of tuples in $R \bowtie S$ 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.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

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 $\prod_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_A\boldsymbol{g}_F(r)$ = $V(A,r)$
- 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 $\sigma_{\theta 1 \vee \theta 2}(r)$
  - 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 $r \mathbin{⟕} s$ = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of $r \mathbin{⟗} s$ = size of $r \bowtie s$ + size of r + size of s

Additional Optimization Techniques

- Nested Subqueries
- Materialized Views

Optimizing Nested Subqueries**

- Nested query example:
  select *customer_name*
  from *borrower*
  where exists (**select** *
  from *depositor*
  where *depositor.customer_name* = *borrower.customer_name*)
- 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**
- 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: the two queries generate different numbers of duplicates (why?)
    - Borrower can have duplicate customer-names
    - Can be modified to handle duplicates correctly 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

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*
  group by *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 the 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
  - Periodic recomputation (e.g. nightly)
  - Above methods are directly supported by many database systems
    - Avoids manual effort/correctness issues

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

THANK YOU