Query Optimization

Shuigeng Zhou

May 28, 2014

School of Computer Science
Fudan University

Outline

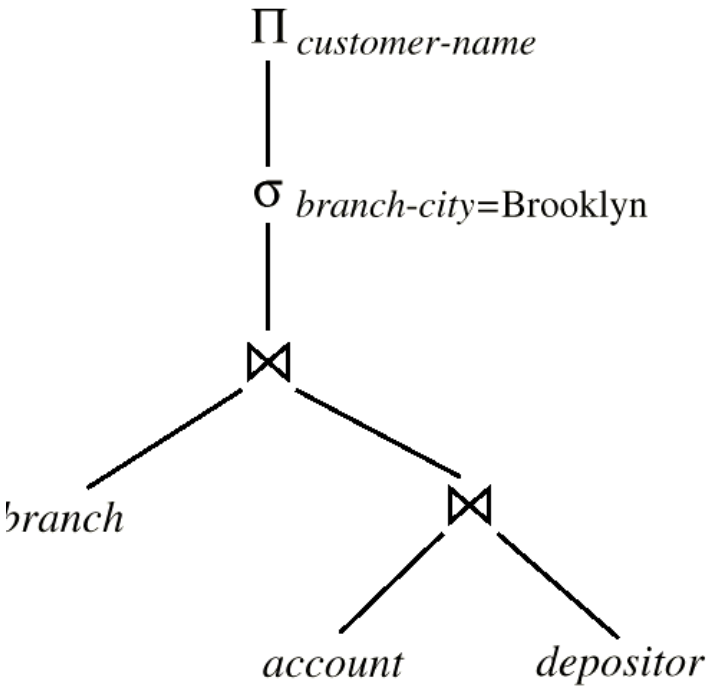
- Introduction
- Catalog Information for Cost Estimation
- Estimation of Statistics
- Transformation of Relational Expressions
- Dynamic Programming for Choosing Evaluation Plans

Introduction

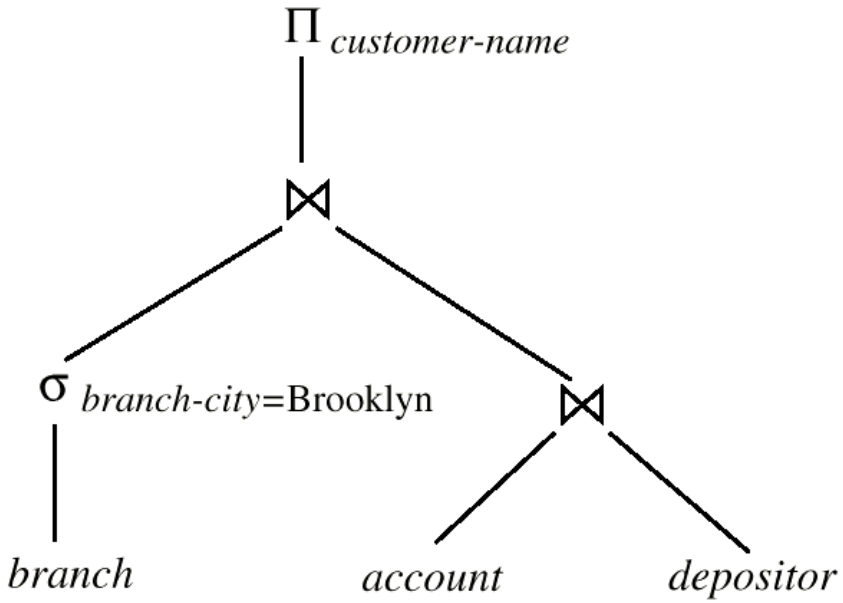
- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
  - 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.



(a) Initial Expression Tree



(b) Transformed Expression Tree

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**

Estimation

- Size
- Distinct Values

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, $HT_i = \lceil \log_{Fi}(V(A,r)) \rceil$.
  - 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/to disk* is used as a measure of the actual cost of evaluation.
  - It is assumed that all transfers of blocks have the same cost
  - Do not include cost to writing final 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 (A2) 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)
- $n_{account}$ = 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:

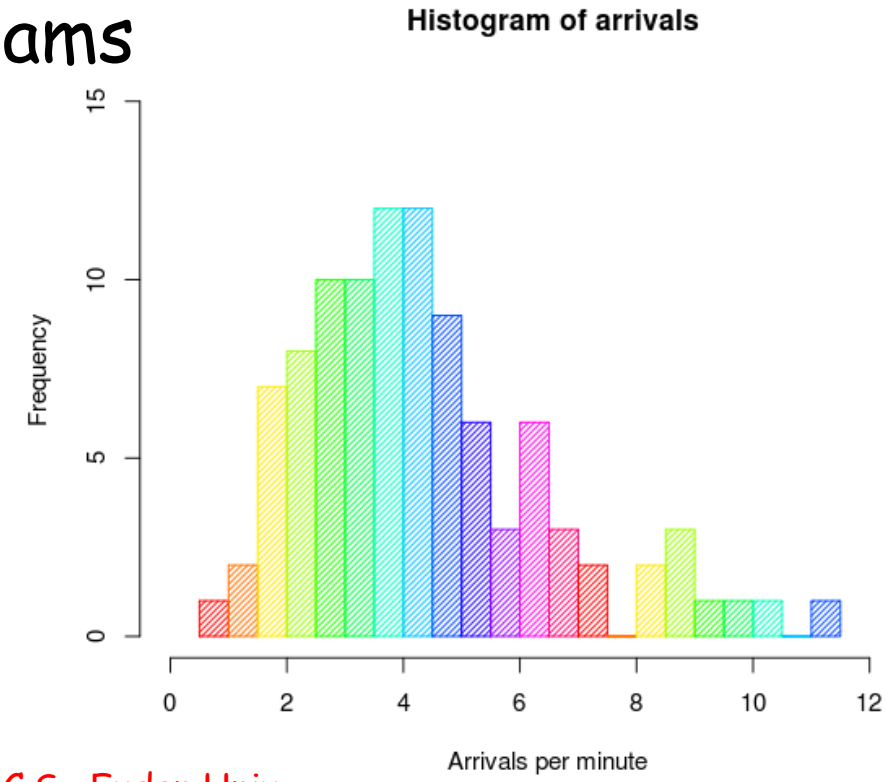
$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

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

$$n_r - size(\sigma_\theta(r))$$

Histogram

- Database systems usually use histograms to keep the value distributions of attributes
- With histograms, selectivity can be estimated
- Two types of histograms
  - Equi-width
  - Equi-deepth



Join Operation: Running Example

Running example: *depositor* ⋈ *customer*

Catalog information for join examples:

- $n_{customer}$ = 10,000.
- $f_{customer}$ = 25 ($b_{customer}$ =10000/25 = 400)
- $n_{depositor}$ = 5000.
- $f_{depositor}$ = 50, ($b_{depositor}$ = 5000/50 = 100)
- *V(customer-name, depositor)* = 2500, which implies that , on average, each customer having saving 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 $r \times s$ contains $n_r n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as $r \times s$.
- If $R \cap S$ 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.

Estimation of the Size of Joins (Cont.)

- Compute the size estimate 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 key.

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 $\sigma_{\theta 1 \vee \theta 2}(r)$
  - 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

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., $(A = 1 \vee A = 3 \vee A = 4)$),
- If the selection condition θ is of the form *A op v* 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: $r \bowtie s$

- 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 $A1$ from r and $A2$ from s, then estimated
  $V(A, r \bowtie s) = \min(\underline{V(A1,r) * V(A2 - A1,s)}, \underline{V(A1 - A2,r) * V(A2,s)}, \underline{n_{r \bowtie s}})$
  - 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 Π_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*)

Equivalence Rules

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
- 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.

$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$

if t1 $\subseteq$ t2...$\subseteq$ tn

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

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.

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 θ involves only attributes from $L_1 \cup L_2$:

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

   if L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.

(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):

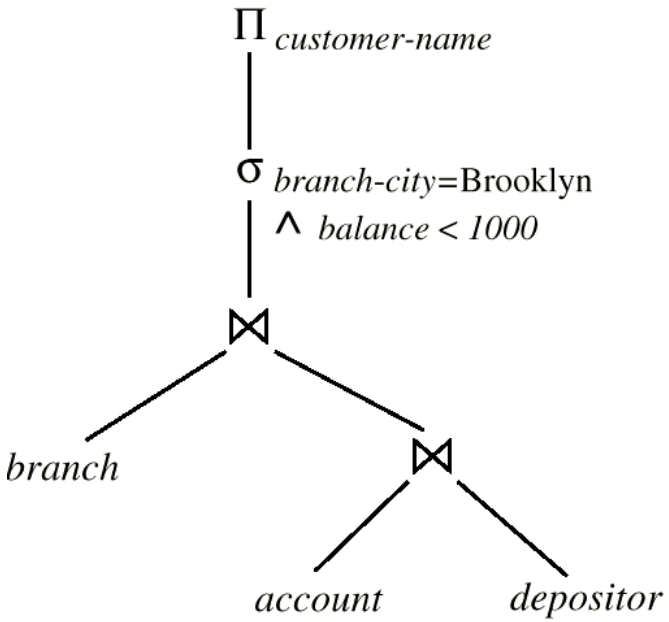
$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}((branch \bowtie account) \bowtie depositor))$$

- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the sub-expression:

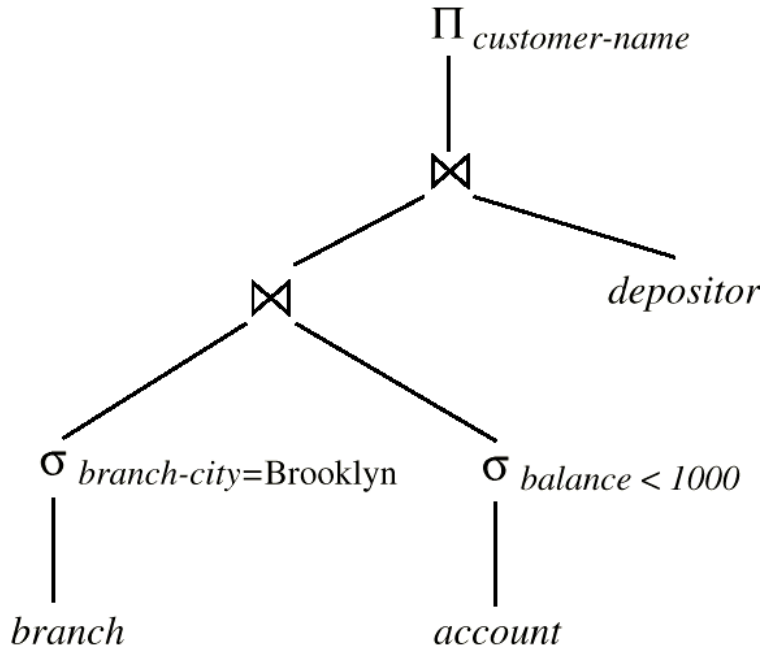
$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch) \bowtie \sigma_{balance > 1000}(account)$$

- 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

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

  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:

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

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

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

- 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

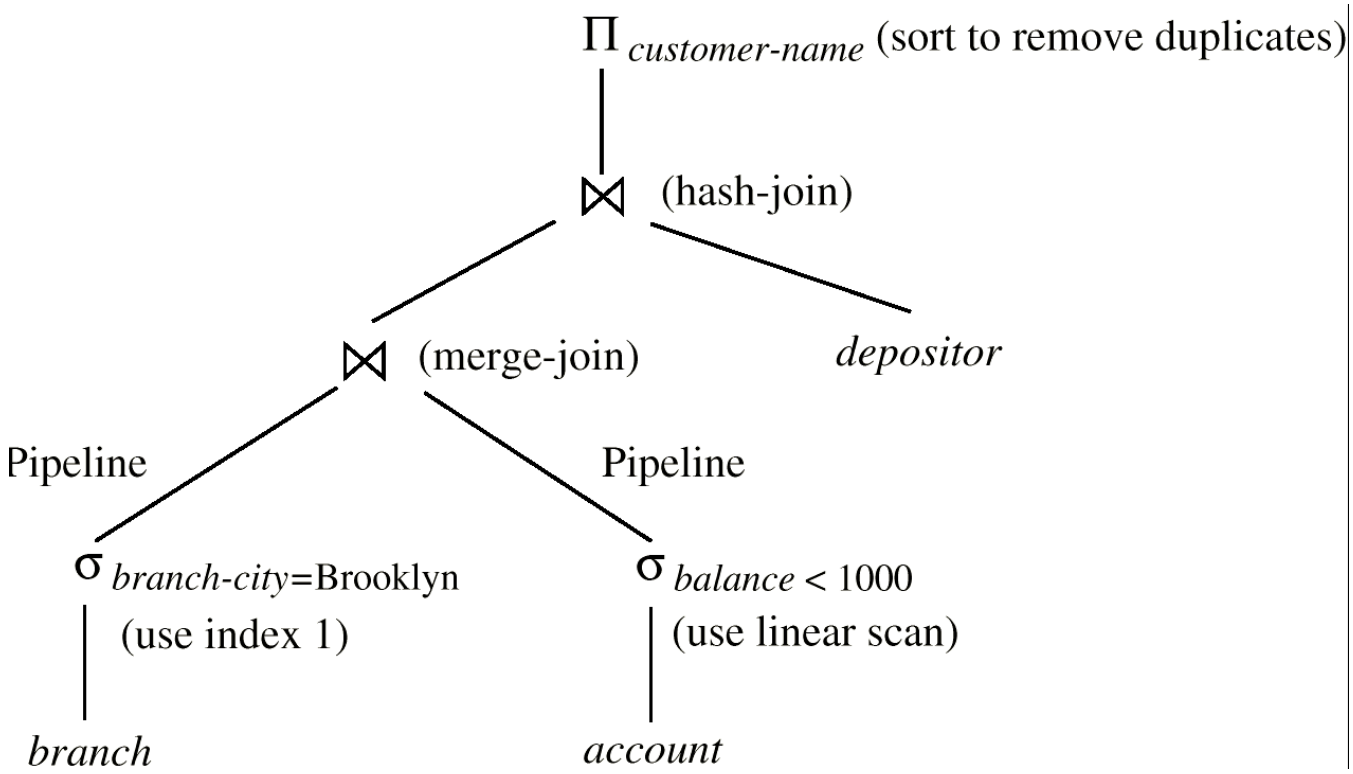
Evaluation Plan

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
  - Given an expression E, if any sub-expression Es of E matches one side of an equivalence rule, the optimizer generates a new expression where Es is transformed to match the other side of the rule
- The above approach is very expensive in space and time.
  - Space requirements reduced by sharing common sub-expressions for equivalent expressions
  - Time requirements are reduced by not generating all expressions

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.
    - 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 17.6 billion!
  - See Problem 14.7 in Practice Exercises

- 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
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of re-computing it
    - Dynamic programming

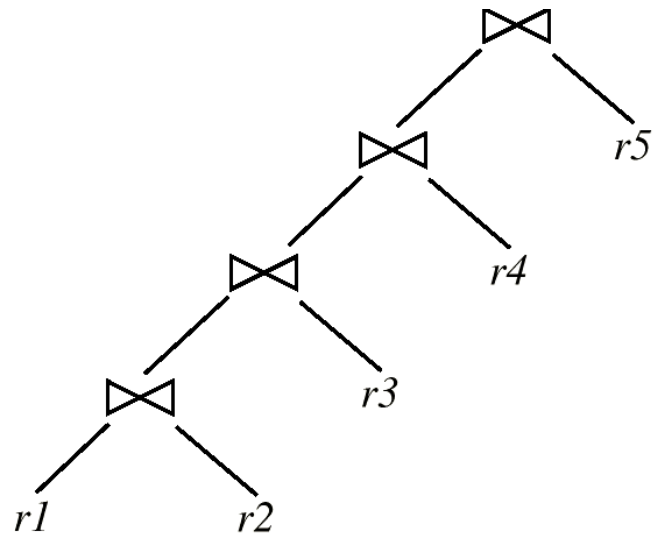
Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)  /* cost is initially set to ∞ */
          return bestplan[S]
    if (S contains only 1 relation)
          set bestplan[S].plan and bestplan[S].cost
                              based on best way of accessing S
    else for each non-empty subset S1 of S such that S1 ≠ S
          P1= findbestplan(S1)
          P2= findbestplan(S - S1)
          A = best algorithm for joining results of P1 and P2
          cost = P1.cost + P2.cost + cost of A
          if cost < bestplan[S].cost
                    bestplan[S].cost = cost
                    bestplan[S].plan = "execute P1.plan;
                                        execute P2.plan;
                                        join results of P1 and P2 using A"
    return bestplan[S]
```

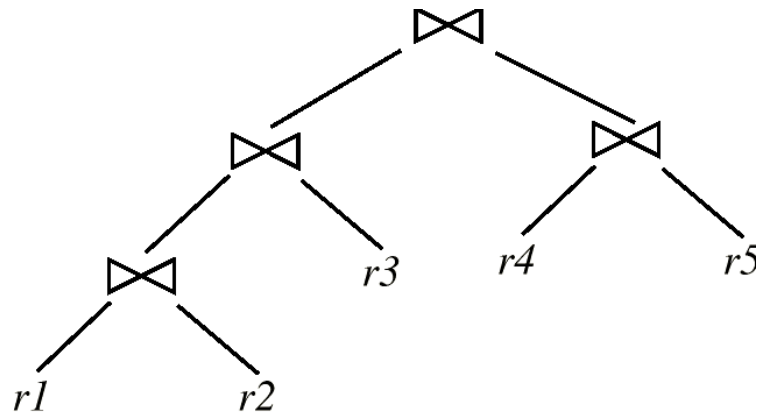
Dynamic-programming algorithm

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 17.6 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.
  - 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 of the join result for that subset
  - 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).

Assignments

- Practice Exercises: 14.4, 14.5, 14.6
- Exercises: 14.11, 14.12, 14.18, 14.19