Advanced DB

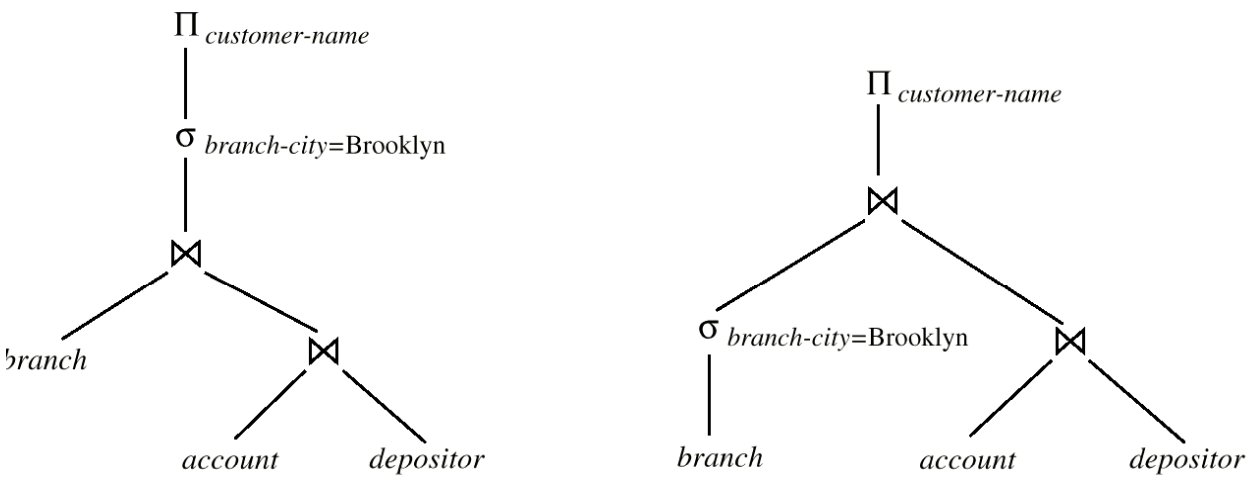
CHAPTER 14
QUERY OPTIMIZATION

Chapter 14: Query Optimization

- Overview
- Estimating Statistics of Expression Results
- Transformation of Relational Expressions
- Choice of Evaluation Plans
- Materialized Views

Introduction

- 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 resulting expressions to get alternative query plans
  3. Choosing the cheapest plan based on *estimated cost*
- The overall process is called cost based optimization.



(a) Initial Expression Tree (b) Transformed Expression Tree

Query Optimization

- Equivalence of Expressions

  Given a DB schema *S*, a query *Q* on *S* is ***equivalent*** to another query *Q*ʹ on *S*, if the answer sets of *Q* and *Q*ʹ are the same in *any* instances of the DB.

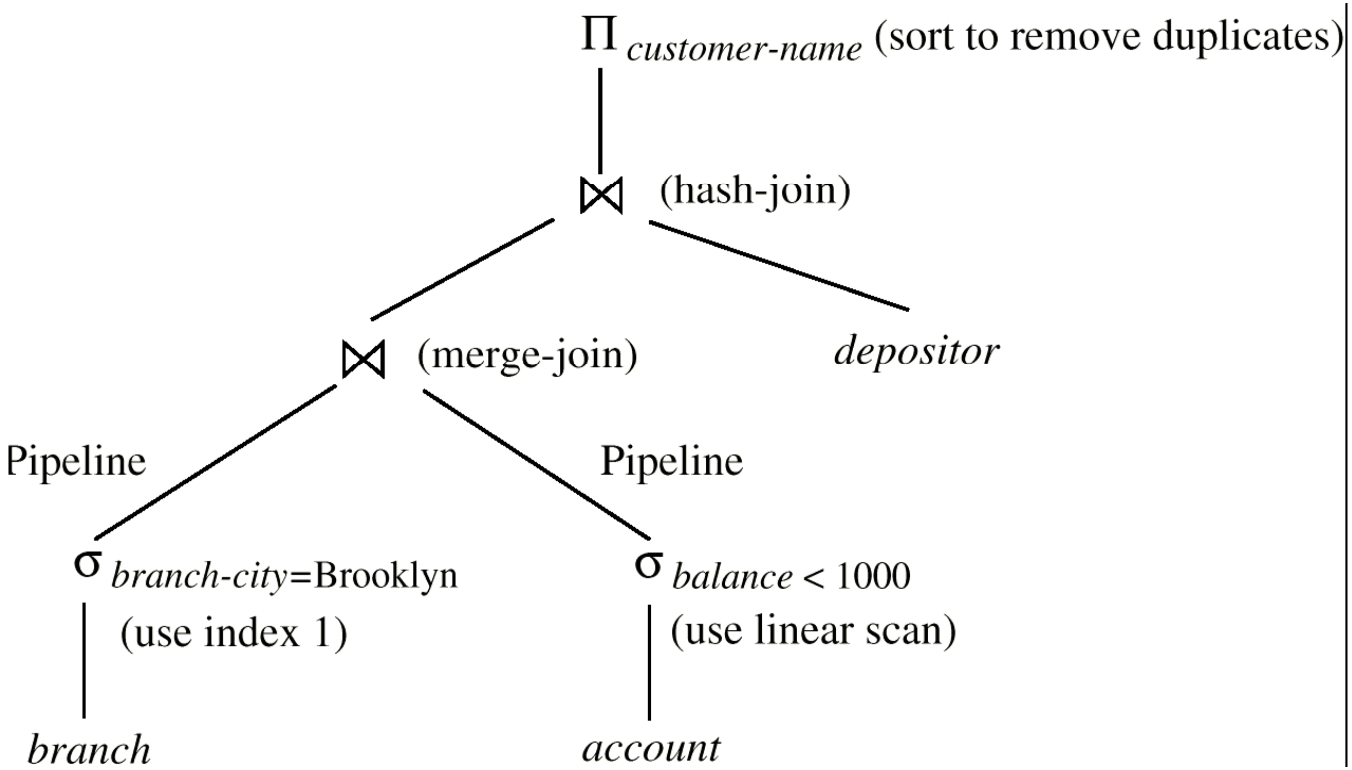
$$\Pi_{b\_name,\ asset}(\sigma_{c\_city=\text{"PC"}}(customer \bowtie depositor \bowtie branch)) \text{ vs}$$

$$\Pi_{b\_name,\ asset}((\sigma_{c\_city=\text{"PC"}}(customer)) \bowtie depositor \bowtie branch)$$

- Query optimization is the process of *selecting* the *most efficient query evaluation plan* for a given query

Evaluation Plan

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



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.

$$\Pi_{t1}(\Pi_{t2}(\ldots \Pi_{tn}(E) \ldots)) = \Pi_{t1}(E)$$

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

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:

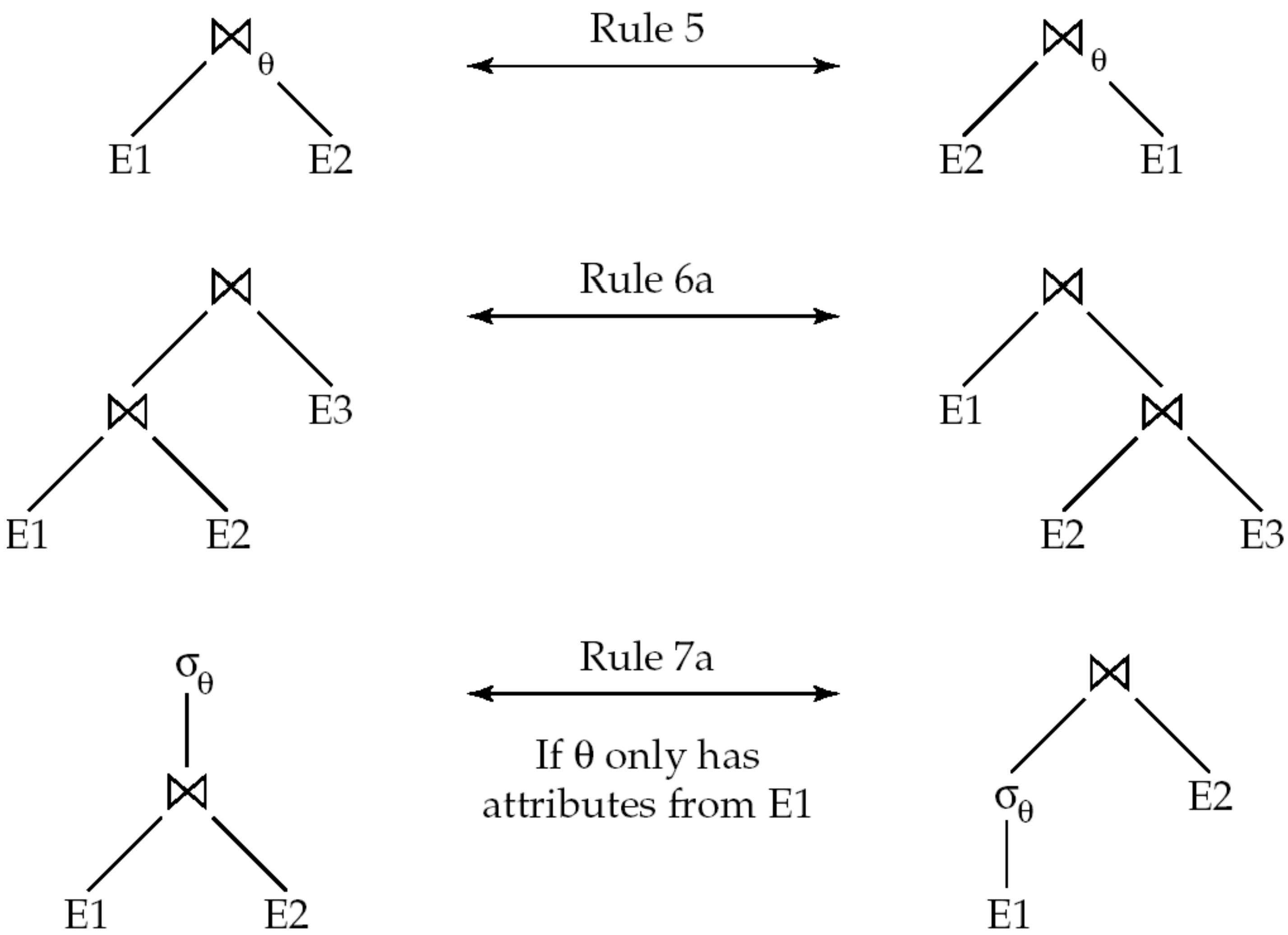
   $$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

   where θ_2 involves attributes from only E_2 and E_3.

7. Selection operation distributes over theta join when all the attributes in θ_1 involve only the attributes of one of the expressions (E_1) being joined.

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

Equivalence Rules (cont.)



Transformation – Example 1

- 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.

Transformation – Example 2

- Query:

  Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

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

- Using join associativity (Rule 6a):

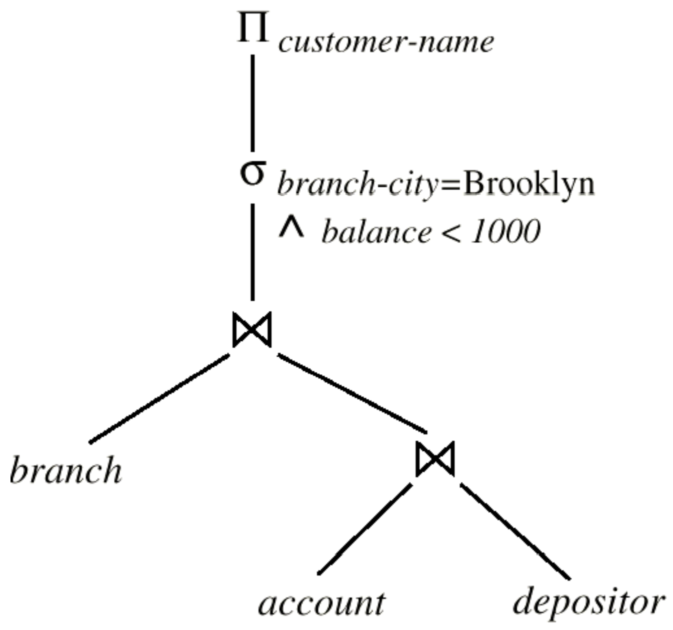
  $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}$
  $((branch \bowtie account) \bowtie depositor))$

- Push selection in (Rules 7a & 7b):

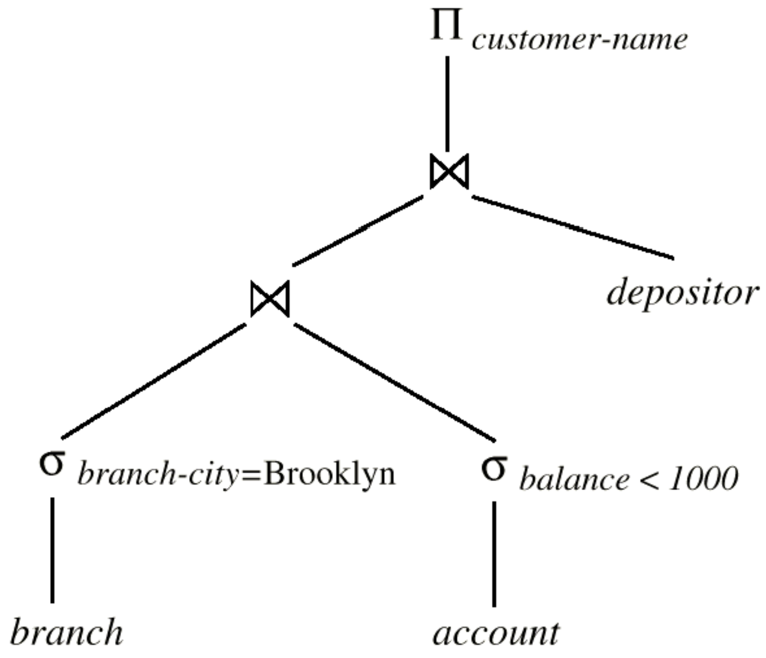
  $\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}$
  $(branch \bowtie account)) \bowtie depositor)$

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

Transformation – Example 2 (cont.)



(a) Initial Expression Tree



(b) Tree After Multiple Transformations

Join Ordering

- $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$ (Rule 6a)
- Choose the expression that will yield smaller temporary result
- Example

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

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

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 size of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute

Statistics 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 $\prod_A(r)$
- $SC(A, r)$: selection cardinality of attribute A of relation r; average number of records that satisfy equality on A
- $b_r = \lceil n_r/f_r \rceil$ if tuples of r are stored together physically in a file

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/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
- We do not include cost of 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

$$E_A = \lceil \log_2(b_r) \rceil + \lceil SC(A, r)/f_r \rceil - 1$$

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

Join Size Estimation

- $r \bowtie s = r \times s$ if $R \cap S = \emptyset$
  - $r \times s$ contains $n_r * n_s$ tuples
  - each tuple occupies $s_r + s_s$ bytes

- 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.*
  - 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

Join Size Estimation (cont.)

- If $R \cap S = \{A\}$ is not a key for R nor S
  - If every tuple t in r produces tuples in $r \bowtie s$: $(n_r * n_s) / V(A, s)$
  - If the reverse is true: $(n_r * n_s) / V(A, r)$
  - The lower of these two estimates is probably the more accurate one.
  - Compute the size estimates for *depositor* $\bowtie$ *customer* without using information about foreign keys:
    - *V(customer-name, depositor)* = 2,500
      V(customer-name, customer) = 10,000
    - The two estimates are 5,000 * 10,000/2,500 = 20,000 and
      5,000 * 10,000/10,000 = 5,000

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 > 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.

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming
  - Search space grows exponentially!
- Heuristic optimization
  - make transformations based on 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.

END OF CHAPTER 14