CMSC 424 – Database design Lecture 18 Query optimization

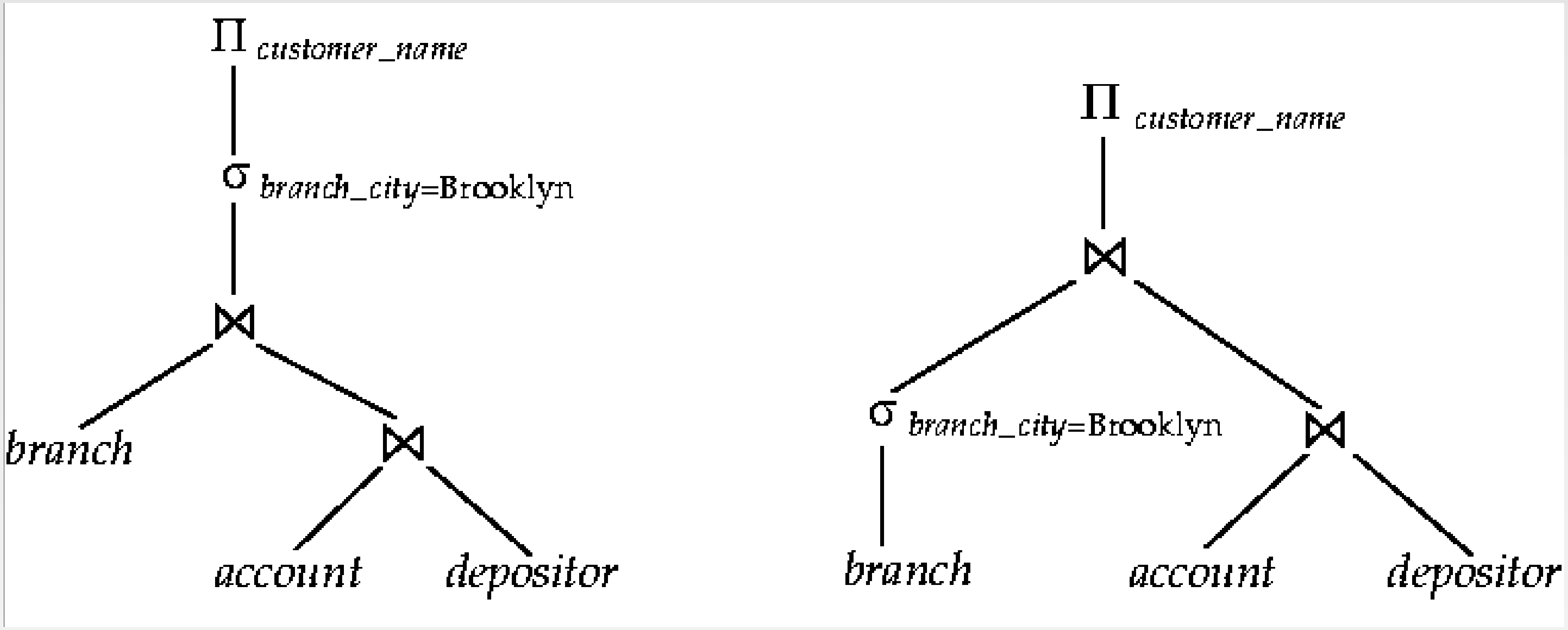
Mihai Pop

Admin

- More midterm solutions
- Projects – do not be late!

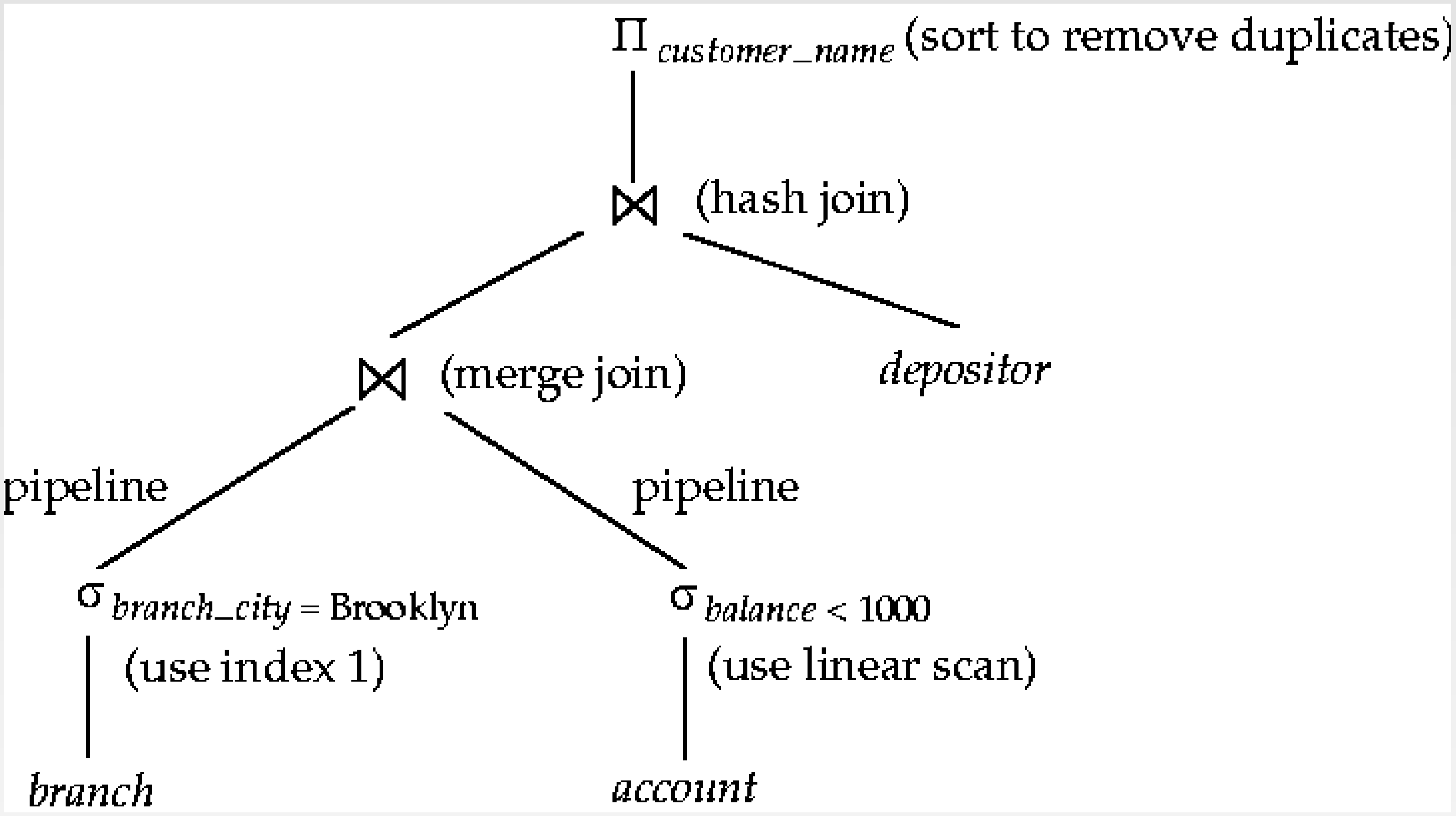
Introduction

- 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.



$\Pi_{customer\_name}$ (sort to remove duplicates)
$\bowtie$ (hash join)
$\bowtie$ (merge join)
depositor
pipeline
pipeline
$\sigma_{branch\_city = \text{Brooklyn}}$
(use index 1)
$\sigma_{balance < 1000}$
(use linear scan)
branch
account

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**
  - Generate logically equivalent expressions using **equivalence rules**
  - Annotate resultant expressions to get alternative query plans
  - 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

Generating Equivalent Expressions

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.

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

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

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

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:

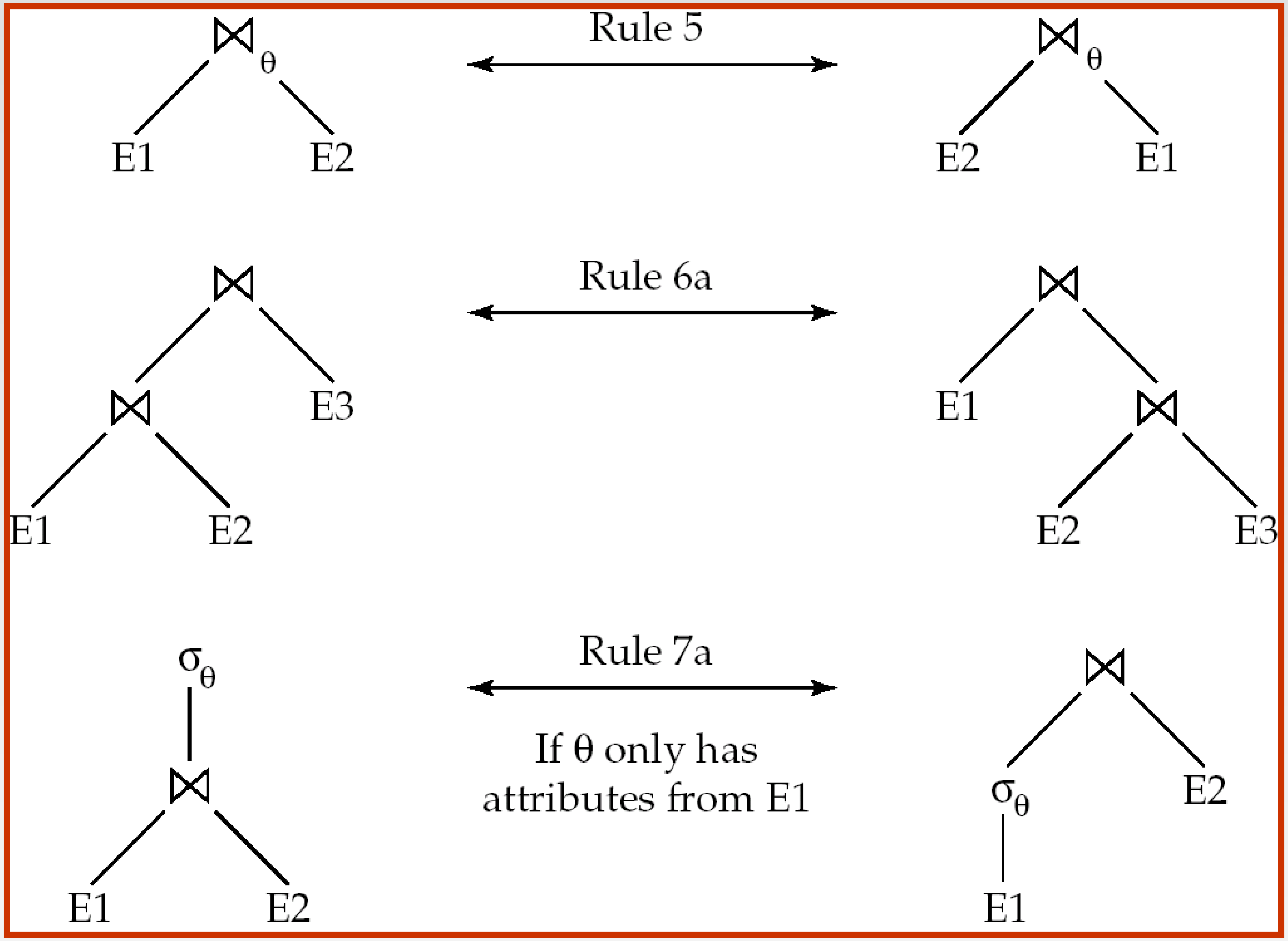
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(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



Transformation rules

- Many more...
- Read chapter 14!!!!

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):

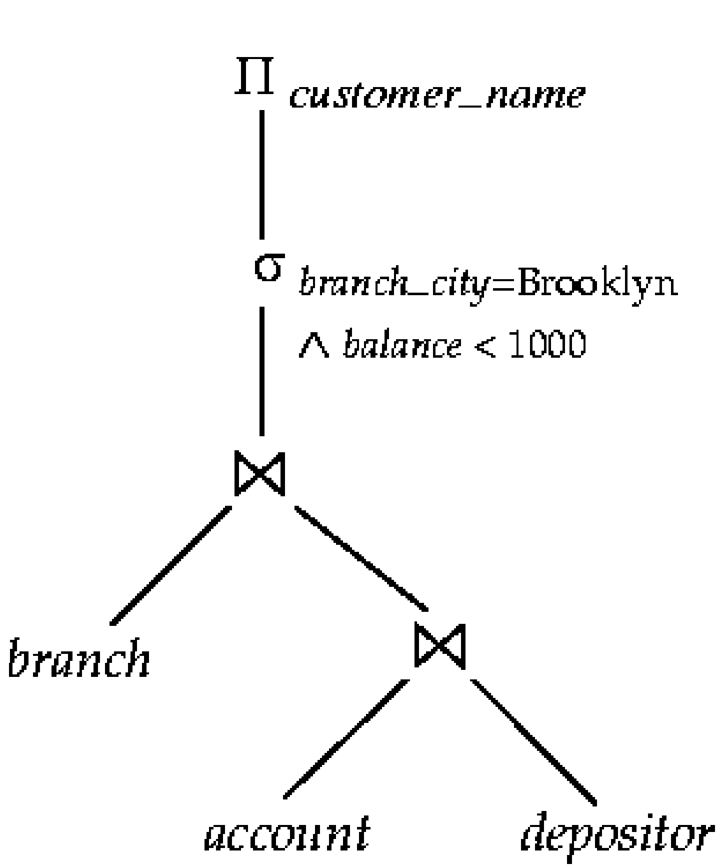
  $\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}$
  $(branch \bowtie account)) \bowtie depositor)$

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

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

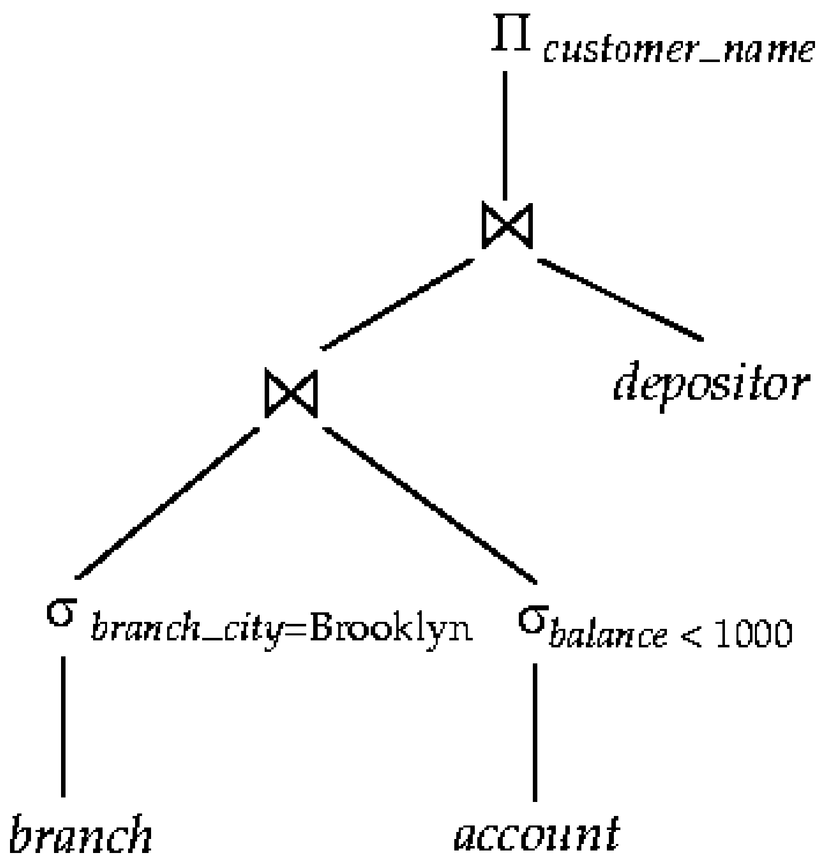
- Thus a sequence of transformations can be useful

Multiple Transformations (Cont.)



$\Pi_{customer\_name}$
$\sigma_{branch\_city = \text{Brooklyn} \wedge balance < 1000}$
⋈
branch
⋈
account
depositor

(a) Initial expression tree



$\Pi_{customer\_name}$
⋈
⋈
depositor
$\sigma_{branch\_city = \text{Brooklyn}}$
$\sigma_{balance < 1000}$
branch
account

(b) Tree after multiple transformations

Transformation Example: Pushing Projections

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

- 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:

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

- 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

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

- 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.

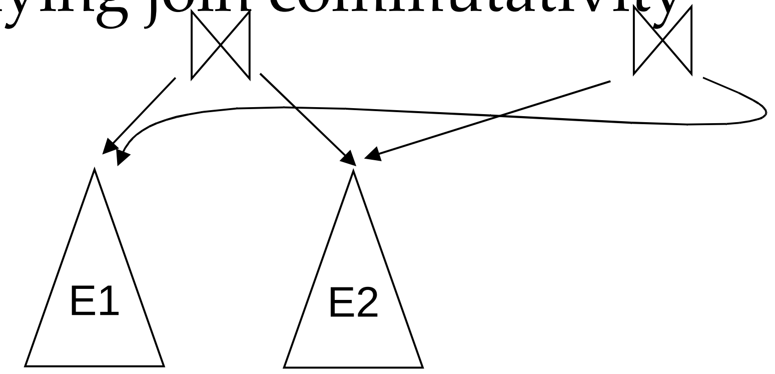
Enumeration of Equivalent Expressions

- 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