# Database Development and Design (CPT201)

## Lecture 5a: Introduction to Query Optimisation 1

Dr. Wei Wang

Department of Computing

# Learning Outcomes

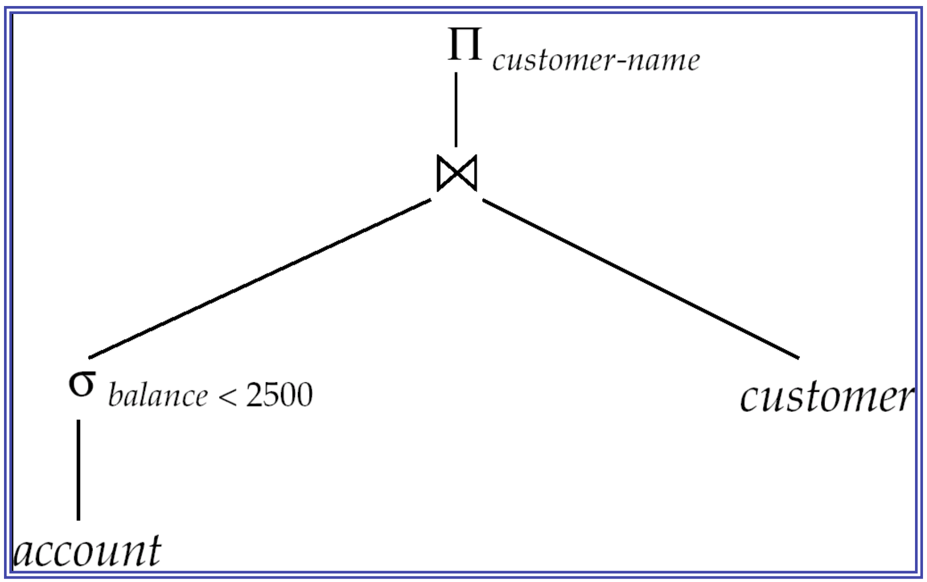
- Introduction to Query Optimisation
  - Transformation of Relational Expressions

# Evaluation of Expressions

- So far we have seen algorithms for individual operations
  - These have then to be combined to evaluate complex expressions, with multiple operations.
- Alternatives for evaluating an entire expression tree
  - **Materialisation**: generate results of an expression whose inputs are relations or relations that are already computed. Temporary relations must be **materialised** (stored) on disk.
  - **Pipelining**: pass on tuples to parent operations even as the operation is being executed.

# Materialisation

- Materialised evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialised into temporary relations to evaluate next-level operations.
  - e.g., in figure below, compute and store the selection, then compute its join with *customer and* and store the result, and finally compute the projections on *customer-name.*



# Materialisation cont'd

- Materialised evaluation is always applicable
- It may require considerable storage space. Moreover, cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing final results to disk, so:
  - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full, write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

# Pipelining

- Pipelined evaluation: evaluate several operations simultaneously, passing the results of one operation on to the next.
  - e.g., in previous expression tree, don't store result of the selection
    - instead, pass tuples directly to the join.
    - Similarly, don't store result of join, pass tuples directly to projection.
- It is much cheaper than materialisation: there is no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort and hash-join where a preliminary phase is required over the whole relations.
- Pipelines can be executed in two ways: demand driven and producer driven.

# Producer-Driven Pipelining

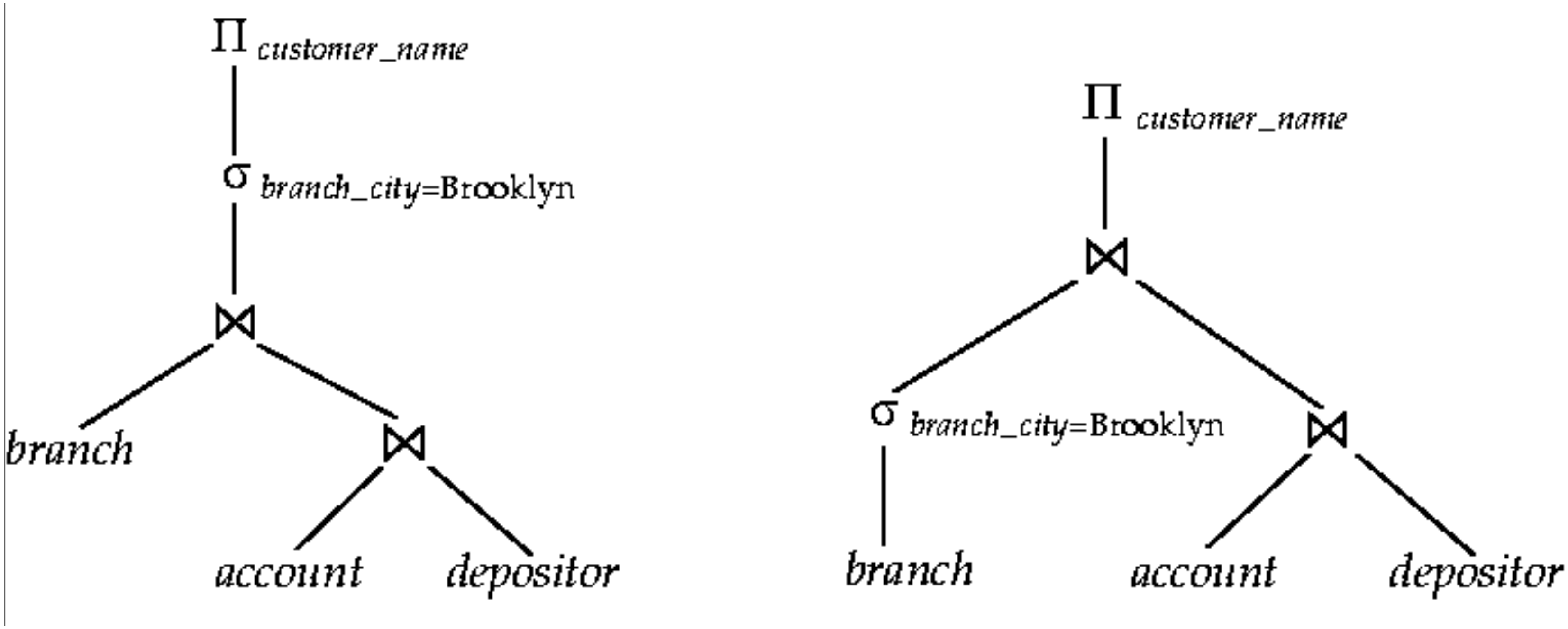
- In producer-driven (or eager or push) pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples.

# Demand-Driven Pipelining

- In demand driven (or lazy, or pull) evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from child operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "state" so it knows what to return next.

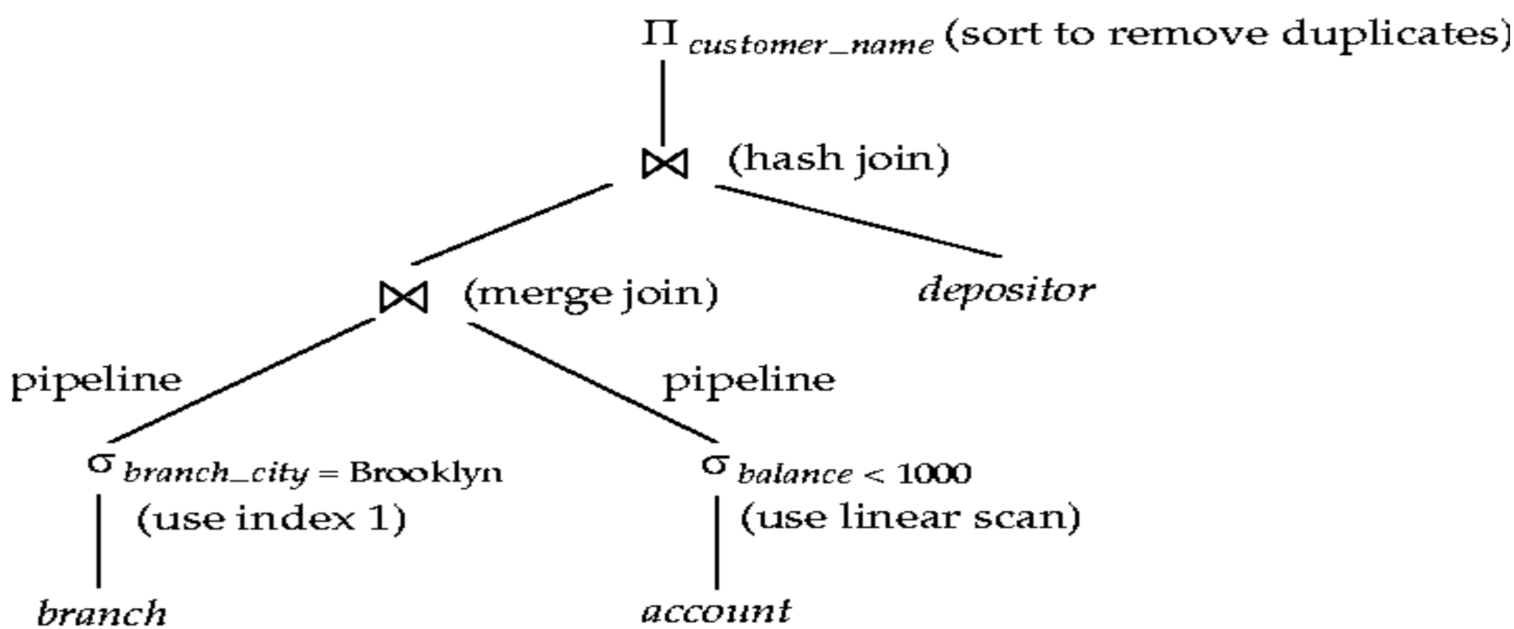
# Equivalent expressions

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



# Evaluation Plan

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



# Cost-based query Optimisation

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Cost-based query optimisation
  - Find logically equivalent expressions of the given expression (but more efficient to execute)
  - Select a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation or choosing the specific indices to use
- Estimation of plan cost based on:
  - Statistical information about relations, e.g., number of tuples, number of distinct values for an attribute
  - Statistical estimation for intermediate results to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics
- It should be noted that since the cost is an estimate, the selected plan is **not** necessarily the least-costly plan; however, as long as the estimates are good, the plan will not be much more costly than it.

# 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 if
  - Can replace expression of first form by second, or vice versa

# Equivalence Rules

- Rule 1: Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

- Rule 2: Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

- Rule 3: Only the last one in a sequence of projection operations is needed, the others can be omitted.

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

- Rule 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'd

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

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

- Rule 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 $\theta_2$ involves attributes from only $E_2$ and $E_3$.

# Equivalence Rules cont'd

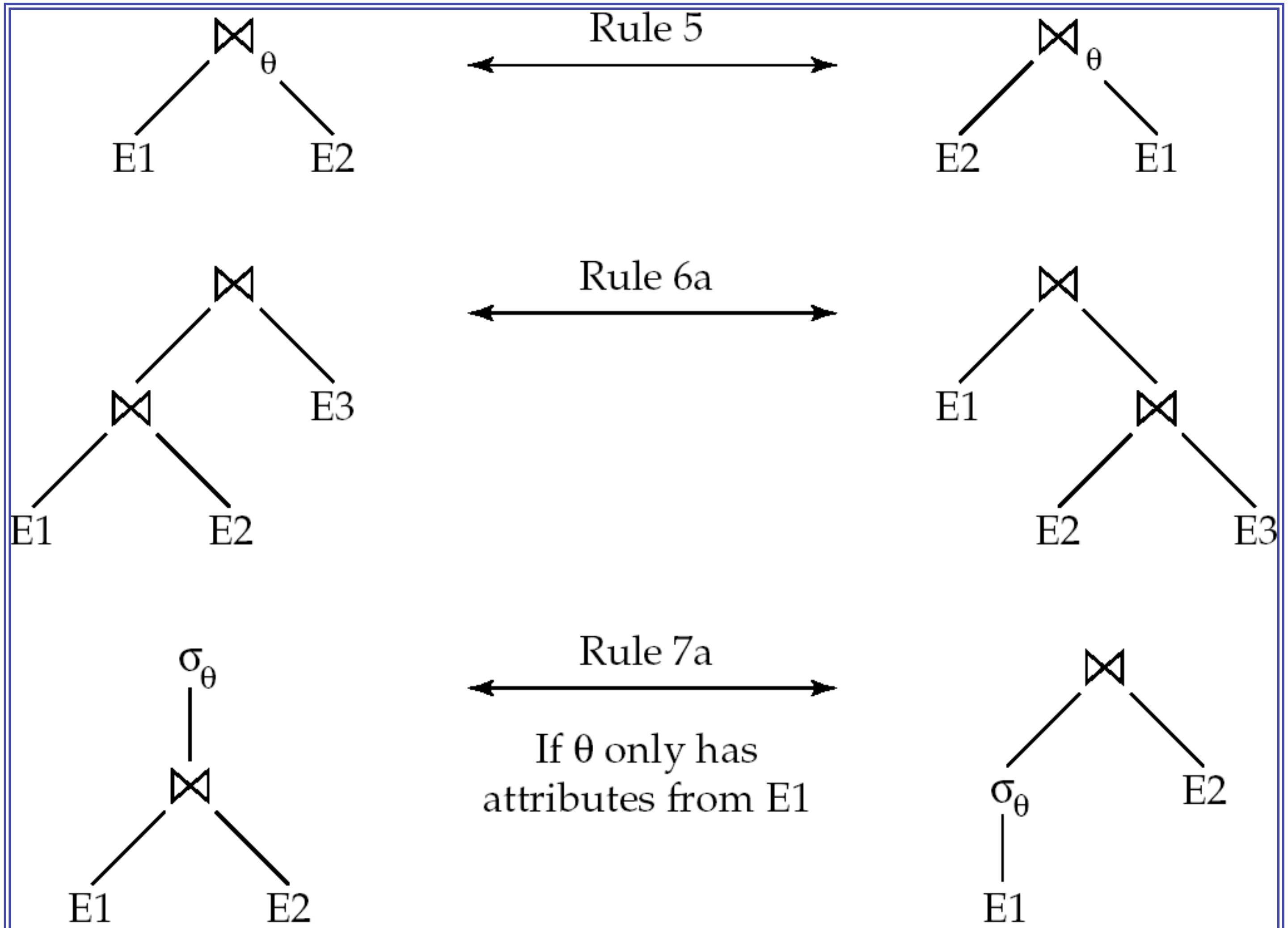
- Rule 7. The selection operation distributes over the theta join operation under the following two conditions:
  - (a) When $\theta_0$ involves only the attributes of one of the expressions ($E_1$) being joined.

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

  - (b) When $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$.

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

# Pictorial Depiction of Equivalence Rules



# Equivalence Rules cont'd

- Rule 8. The projection operation distributes over the theta join operation as follows:
  - (a) Let *L1* and *L2* be attributes from *E1* and *E2*, if $\theta$ 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 $\theta$, but are not in $L_1 \cup L_2$, and
    - let $L_4$ be attributes of $E_2$ that are involved in join condition $\theta$, 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'd

- Rule 9. The set operations union and intersection are commutative (set difference is not commutative)

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

- Rule 10. Set union and intersection are associative.

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

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

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

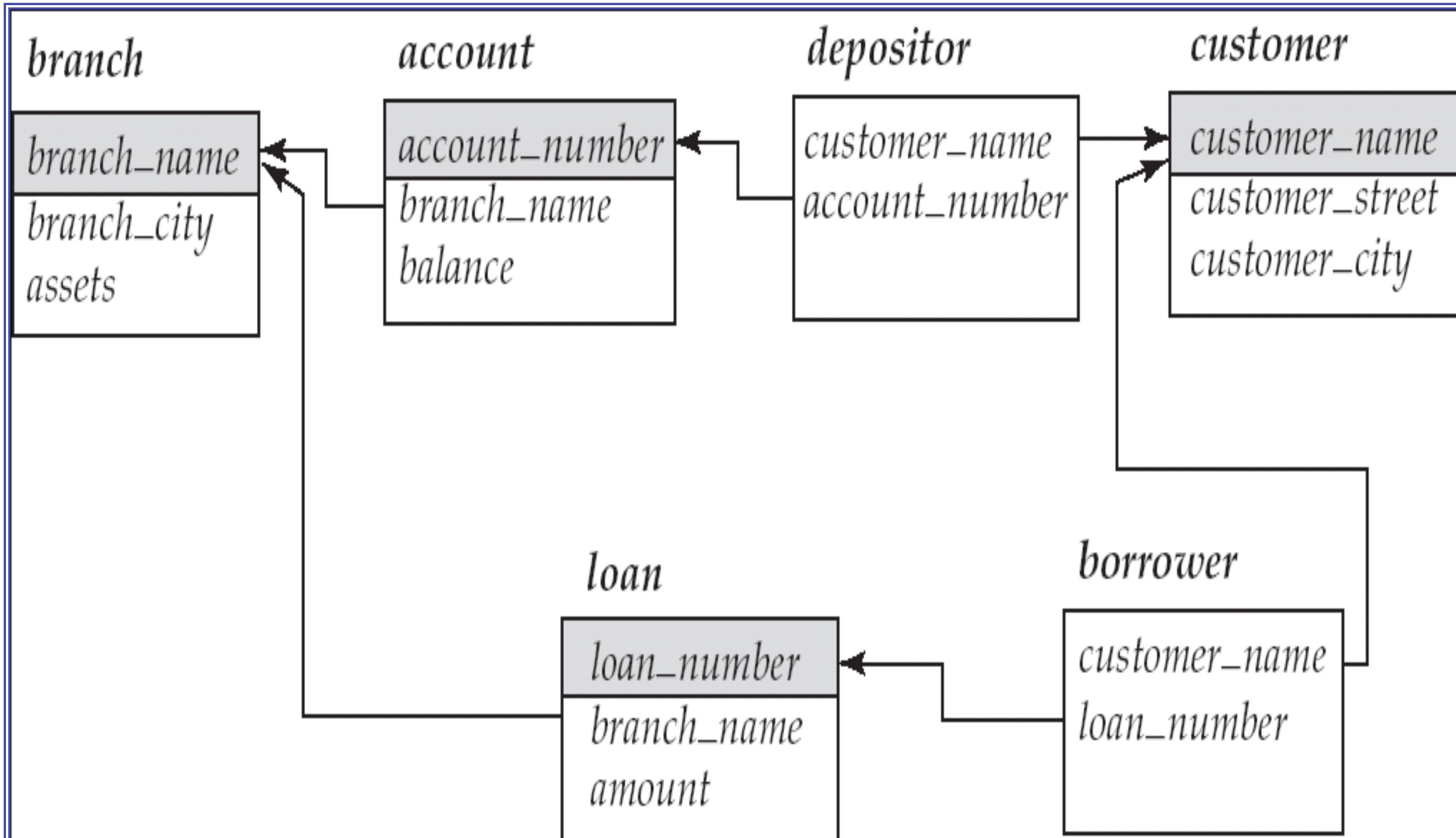
Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

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

- Rule 12. The projection operation distributes over union

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

# Banking Example



# 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 (distribute the selection).
  $\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"}}(branch)) \bowtie (account \bowtie depositor))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

# Example: 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 and 7a):

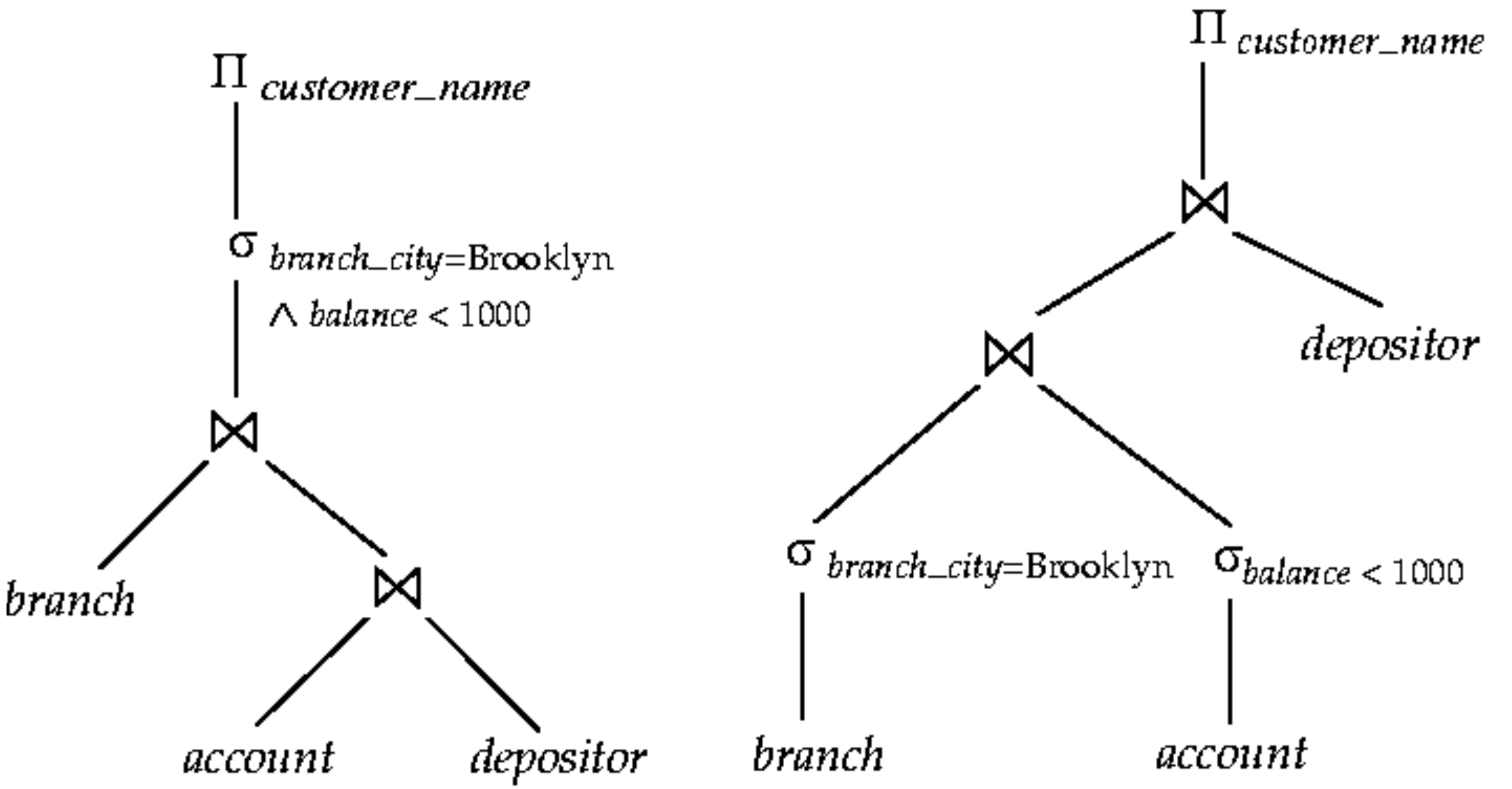
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression 7b

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

- Thus a sequence of transformations can be useful

# Multiple Transformations cont'd




(a) Initial expression tree (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

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

  we obtain a relation whose schema is:
  (*branch_name, branch_city, assets, account_number, balance*)
- Push projections using equivalence rules 8b; eliminate unneeded attributes from intermediate results to get:

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

  *(HINT: L1 is null, L2 is customer_name; L3=L4=account_number)*
- Performing projection as early as possible reduces the size of the tuples 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'd

- 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 first

    $$\sigma_{branch\_city\,=\,\text{"Brooklyn"}}\ (branch) \bowtie account$$

# Enumeration of Equivalent Expressions

- Query optimisers use equivalence rules to systematically generate expressions equivalent to the given expression
- The approach is very expensive in space and time

**procedure** genAllEquivalent($E$)
$EQ = \{E\}$
**repeat**
    Match each expression $E_i$ in $EQ$ with each equivalence rule $R_j$
    **if** any subexpression $e_i$ of $E_i$ matches one side of $R_j$
        Create a new expression $E'$ which is identical to $E_i$, except that
            $e_i$ is transformed to match the other side of $R_j$
        Add $E'$ to $EQ$ if it is not already present in $EQ$
**until** no new expression can be added to $EQ$

# End of Lecture

- Summary
  - Transformation of Relational Expressions

- Reading
  - Textbook chapter 13.1, 13.2, 13.3, and 13.4