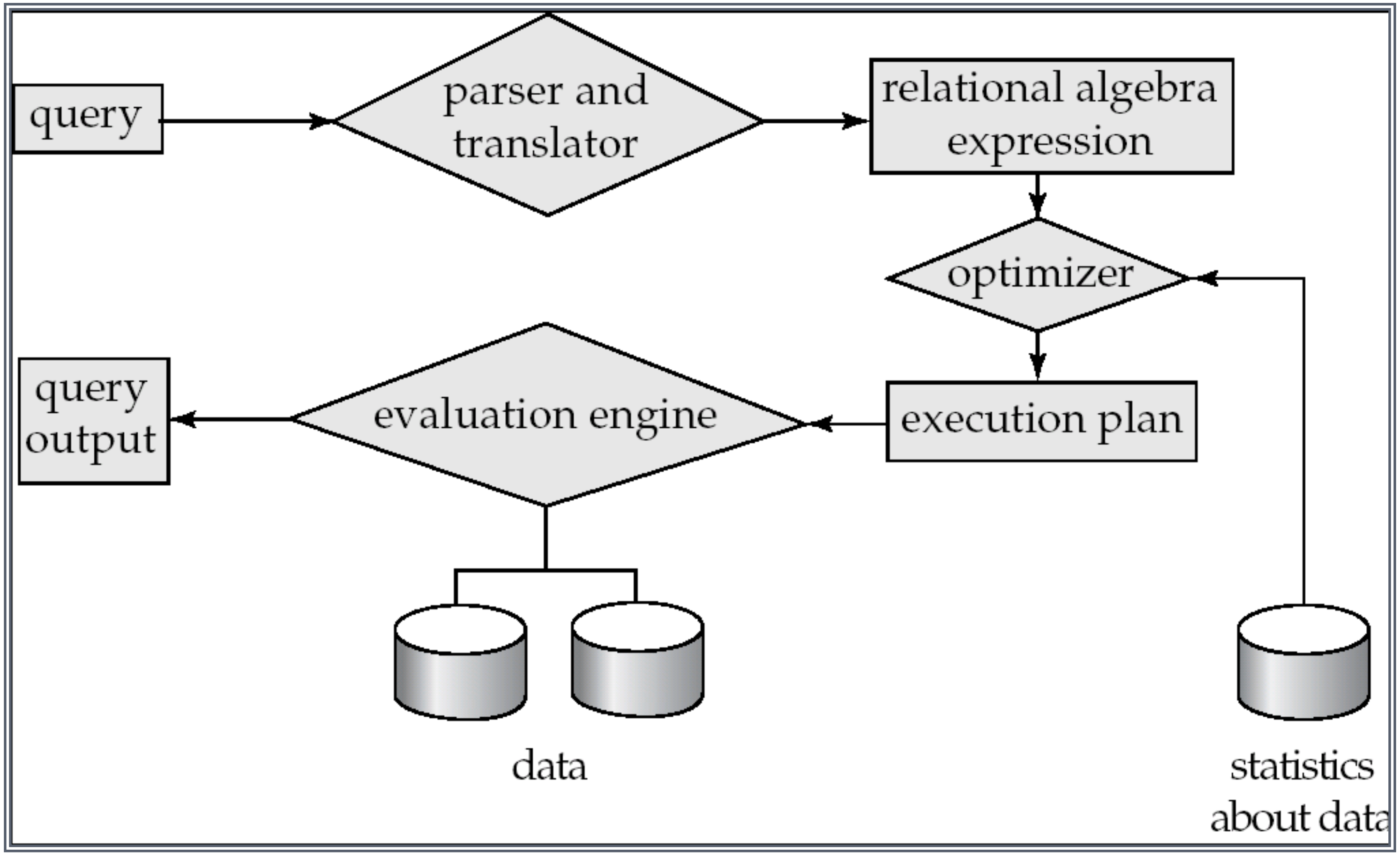
CS354: Query Processing & Optimization

Database

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- **Parsing and translation**
  - Parser checks syntax, verifies relations
  - This is then translated into parse tree representation and then into relational algebra expression
- **Optimization**
  - A query can be evaluated in several ways
  - Even relational algebra expression specifies partially how to evaluate a query
  - A sequence of primitive operations that can be used to evaluate a query is known as *query evaluation plan*
- **Evaluation**
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing

- Consider the following query:

```
SELECT balance
FROM account
WHERE balance < 2500
```

- The query can be expressed in relational algebra expressions:
  a. $\sigma_{balance<2500}(\Pi_{balance}(account))$
  b. $\Pi_{\mathrm{balance}}(\sigma_{\mathrm{balance<2500}}(account))$

Basic Steps in Query Processing : Optimization

- Each relational algebra operation can be evaluated using one of several different algorithms
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance ≥ 2500

$\Pi_{balance}$

$\sigma_{balance<2500;}$ use index 1

account

Query-evaluation plan

A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive

A sequence of primitive operations that can be used to evaluate a query is a query execution plan or query evaluation plan

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
- Next we will see
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
    + Number of blocks read * average-block-read-cost
    + Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful
  - Assumption: single disk
    - Can modify formulae for multiple disks/RAID arrays
    - Or just use single-disk formulae, but interpret them as measuring **resource consumption** instead of time

Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for ***b*** block transfers plus ***S*** seeks

    $$b * t_T + S * t_S$$

- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Measures of Query Cost (Cont.)

- Several algorithms can reduce disk I/O by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - **Cost estimate = b_r block transfers + 1 seek**
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - **Cost for avg. cases = (b_r/2) block transfers + 1 seek**
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Selection Operation (Cont.)

- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks = $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary B+ tree index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - $\boldsymbol{Cost = (h_i + 1) * (t_T + t_S)}$
- **A4** (*primary B+ tree index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $\boldsymbol{Cost = h_i * (t_T + t_S) + t_S + t_T * b}$
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - $\boldsymbol{Cost = (h_i + 1) * (t_T + t_S)}$
  - Retrieve multiple records if search-key is not a candidate key
    - each of n matching records may be on a different block
    - $\boldsymbol{Cost = (h_i + n) * (t_T + t_S)}$
      - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- **A6** *(primary index, comparison).* (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A7** *(secondary index, comparison).*
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - may require an I/O for each record
    - Linear file scan may be cheaper

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A8** *(conjunctive selection using one index).*
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A9** (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- **A10** *(conjunctive selection by intersection of identifiers).*
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **A11** *(disjunctive selection by union of identifiers).*
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

JOIN OPERATION

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Example: use the following information
  - Number of records-
    - *customer*: 10,000 and *depositor*: 5000
  - Number of blocks-
    - *customer*: 400 and *depositor*: 100

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \cdot t_s$ to the result.
    end
  end
- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  - Block transfers $\boldsymbol{n_r * b_s + b_r}$ and
  - Seeks $\boldsymbol{n_r + b_r}$
- Example: Assuming worst case memory availability
- Cost estimate is
  - with *depositor* as outer relation:
    - **5000 $*$ 400 + 100 = 2,000,100** block transfers,
  - with *customer* as the outer relation
    - **10000 $*$ 100 + 400 = 1,000,400** block transfers

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation.
- What would be the cost?
- The cost becomes
  - block transfers $\boldsymbol{b_r + b_s}$ and
  - seeks **2**
- Example: If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be **(100+400)=500** block transfers.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

for each block B_r **of** r **do begin**
  for each block B_s **of** $\boldsymbol{s}$ **do begin**
    for each tuple t_r **in** B_r **do begin**
      for each tuple t_s **in** B_s **do begin**
        Check if (t_r, t_s) satisfy the join condition
        if they do, add $t_r \cdot t_s$ to the result.
      end
    end
  end
end

Block Nested-Loop Join (Cont.)

- Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation
- Worst case estimate:
  - block transfers $\boldsymbol{b_r * b_s + b_r}$ and
  - Seeks $\boldsymbol{2b_r}$
- Clearly it is efficient to use the smaller relation as the outer relation
- Best case: $\boldsymbol{b_r + b_s}$ block transfers and **2** seeks
- If we use depositor as the outer relation
  - The worst case: **100*400+100=40,100** block accesses required
  - The best case: **100+400=500** remains same

Performance Improvement Strategies of Nested and Block Nested Loop Join

- If equi-join attribute forms a key on inner relation,
  - stop inner loop on first match
- Scan inner loop forward and backward alternately, to reduce the no. of disk accesses
- In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
  - The number of scans of inner relation reduces from b_r to $\lceil b_r / (M-2) \rceil$
  - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers and $2 \lceil b_r / (M-2) \rceil$ seeks
- Use index on inner relation if available

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r*, use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one block of *r*, and one block of index
- For each tuple in *r*, we perform an index lookup on *s*.
- Cost of the join: $\boldsymbol{b_r\,(t_T + t_S) + n_r * c}$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *depositor* $\bowtie$ *customer,* with *depositor* as the outer relation.
- Let *customer* relation has a primary B^+-tree index on the join attribute *customer-name,* which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5000 tuples
- Cost of block nested loops join
  - **400*100 + 100 = 40,100** block transfers
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - **100 + 5000 * 5 = 25,100** block transfers
  - CPU cost likely to be less than that for block nested loops join

Query Optimization

- Process of selecting most efficient query evaluation plan
- Users may not write the query efficiently
- However, the system has to construct a query evaluation plan that minimizes the cost of query evaluation
- Different aspects of query optimizations
  - Equivalent expressions at the relational algebra level
  - Different algorithms for each operation

- 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
- 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.
   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:

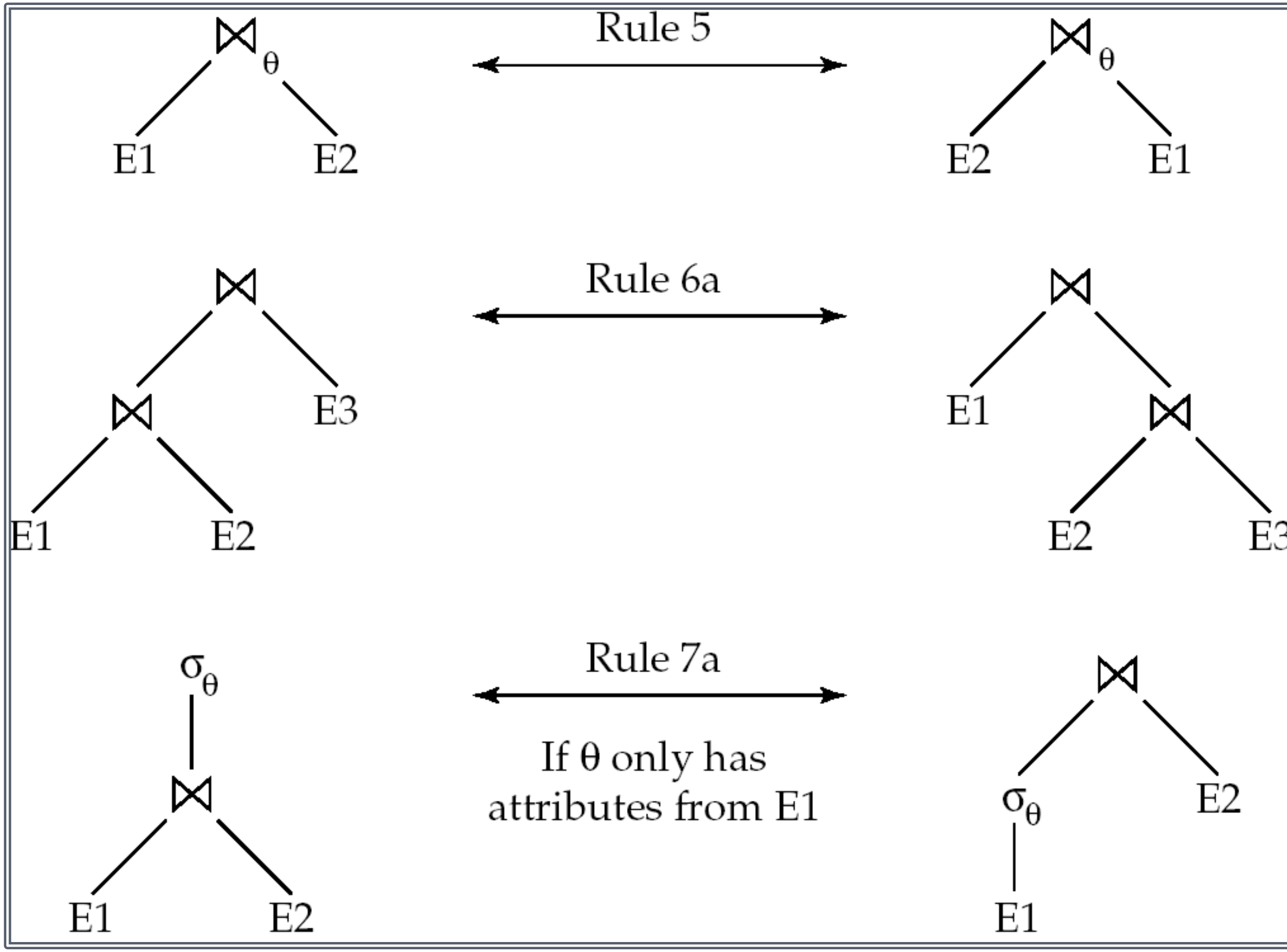
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(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 $L_1 \cup L_2$; where L_1 and L_2 be attributes of E1 and E2 respectively

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

   (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 $-$

Can we write $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$?

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):

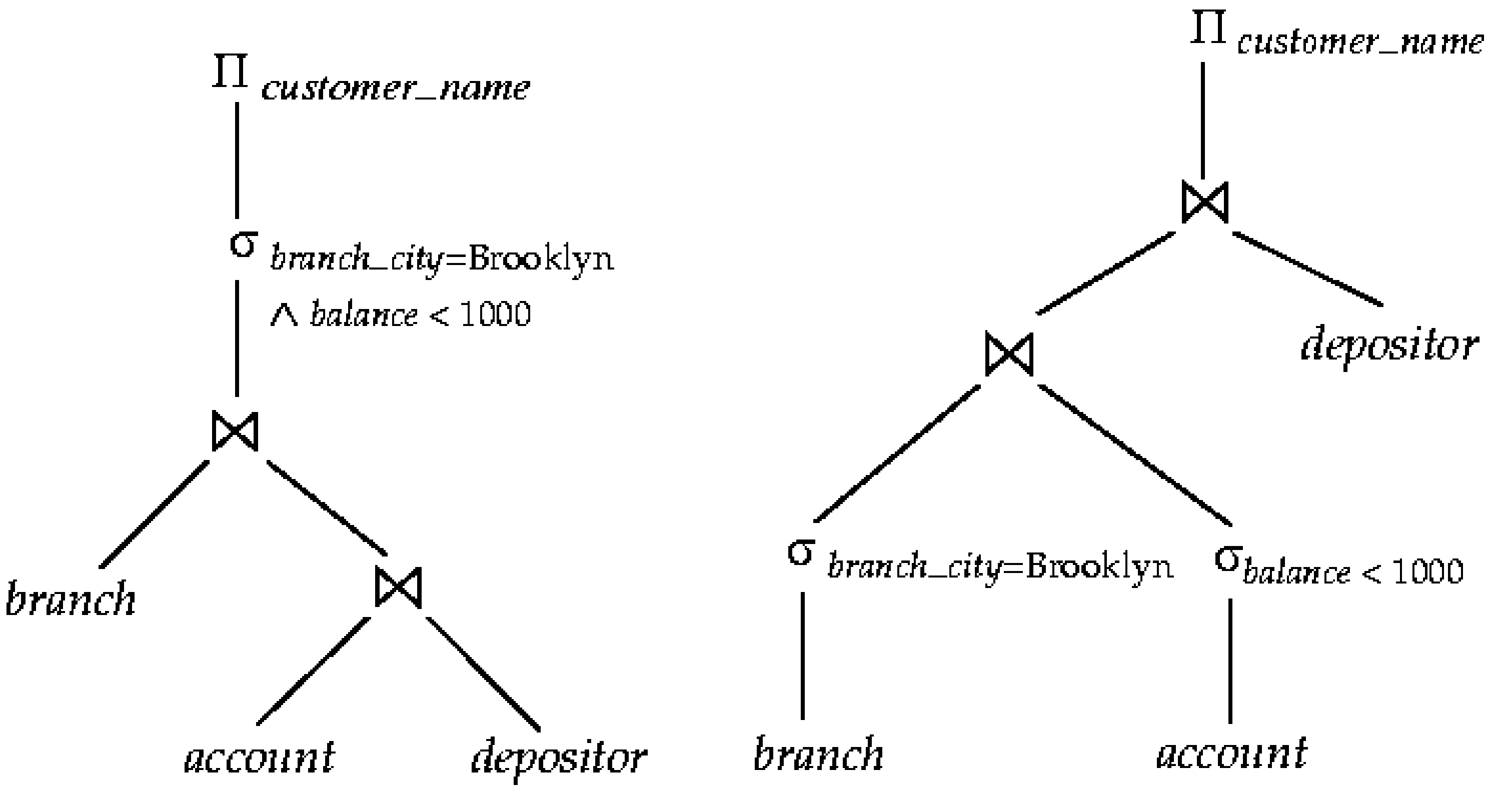
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge \; balance > 1000} \\ (branch \bowtie account)) \bowtie depositor)$$

- 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.)



(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

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

  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.