Database Development and Design (CPT201)

Lecture 5a: Introduction to Query Optimisation 1

Dr. Wei Wang

Department of Computing

Learning Outcomes

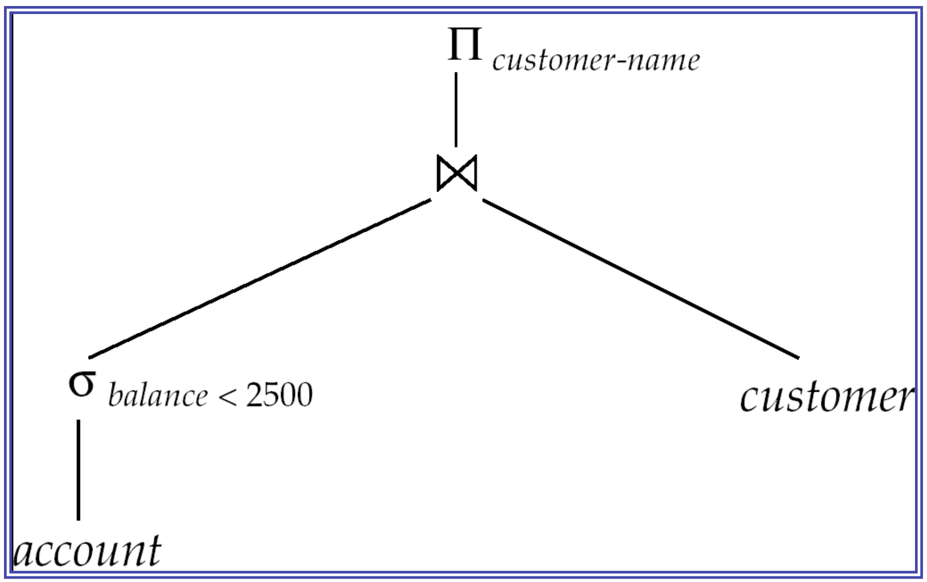
- Introduction to Query Optimisation
    - Transformation of Relational Expressions

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

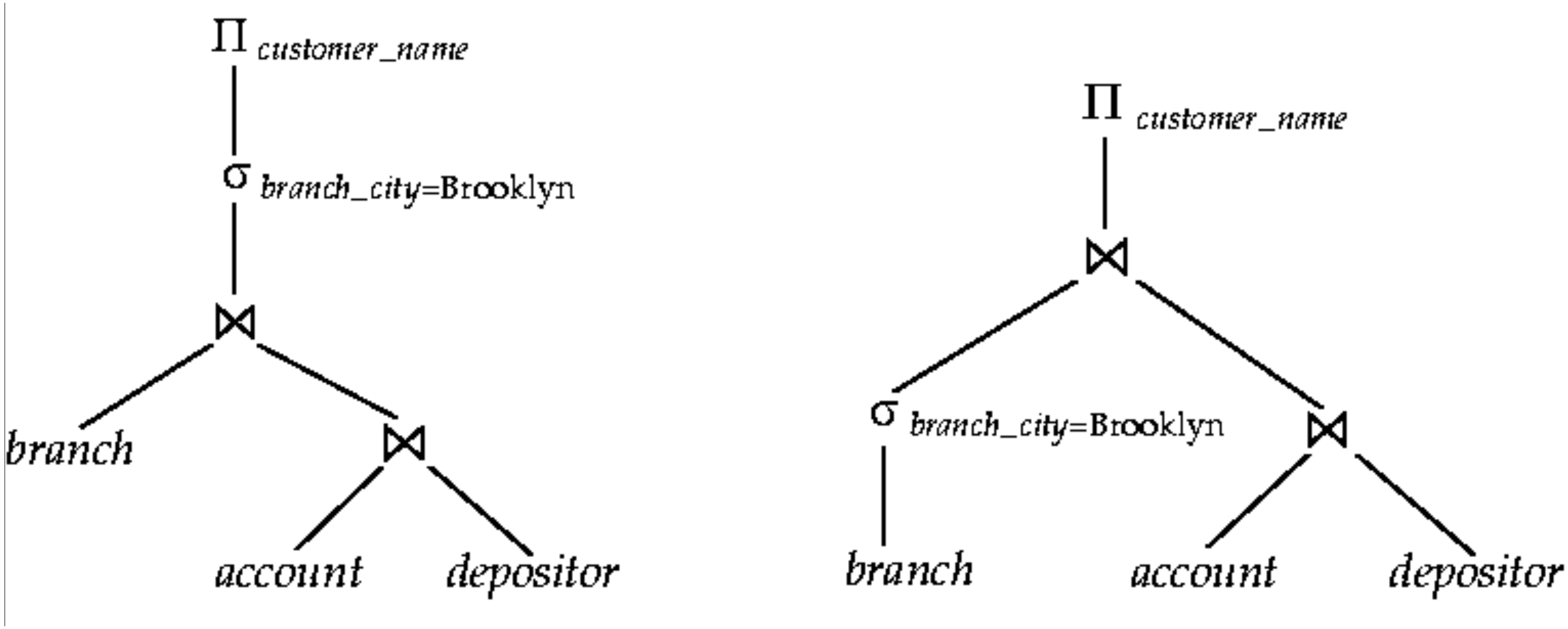
- In producer-driven (or eager or push) pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and 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.

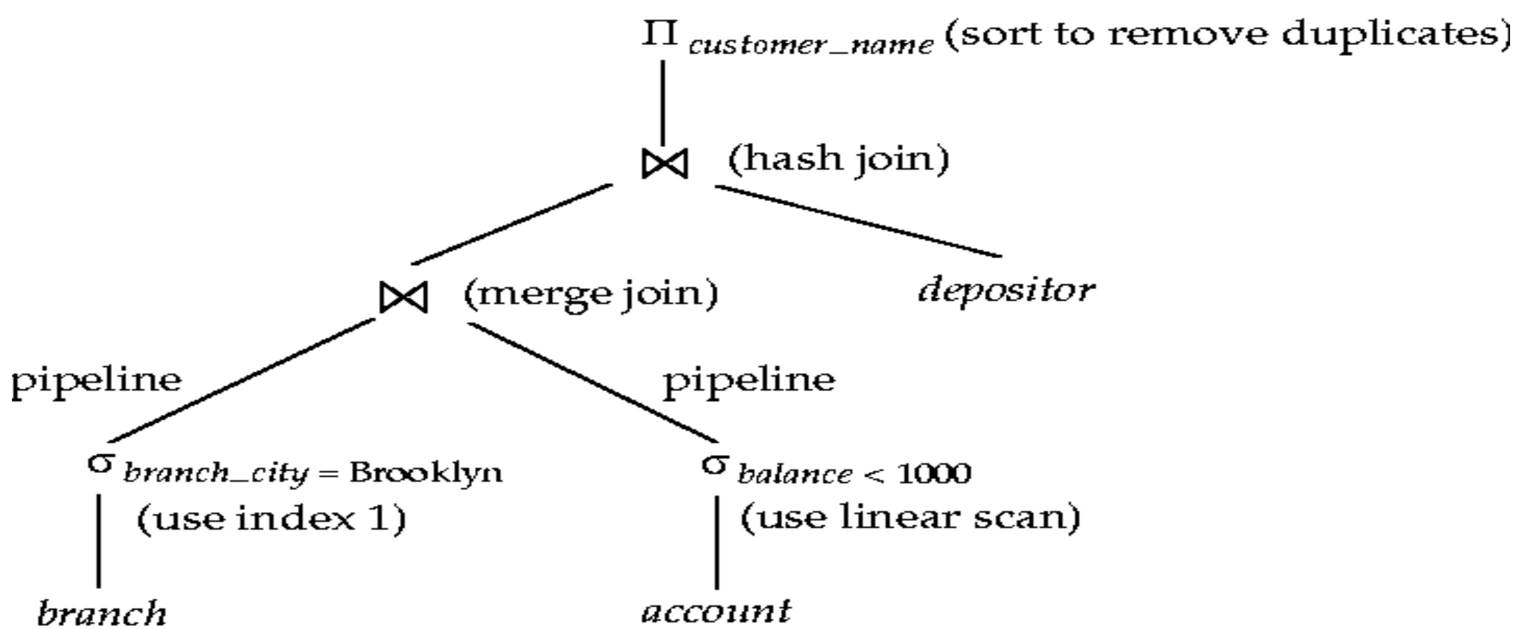
Equivalent expressions

- 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 θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules cont'd

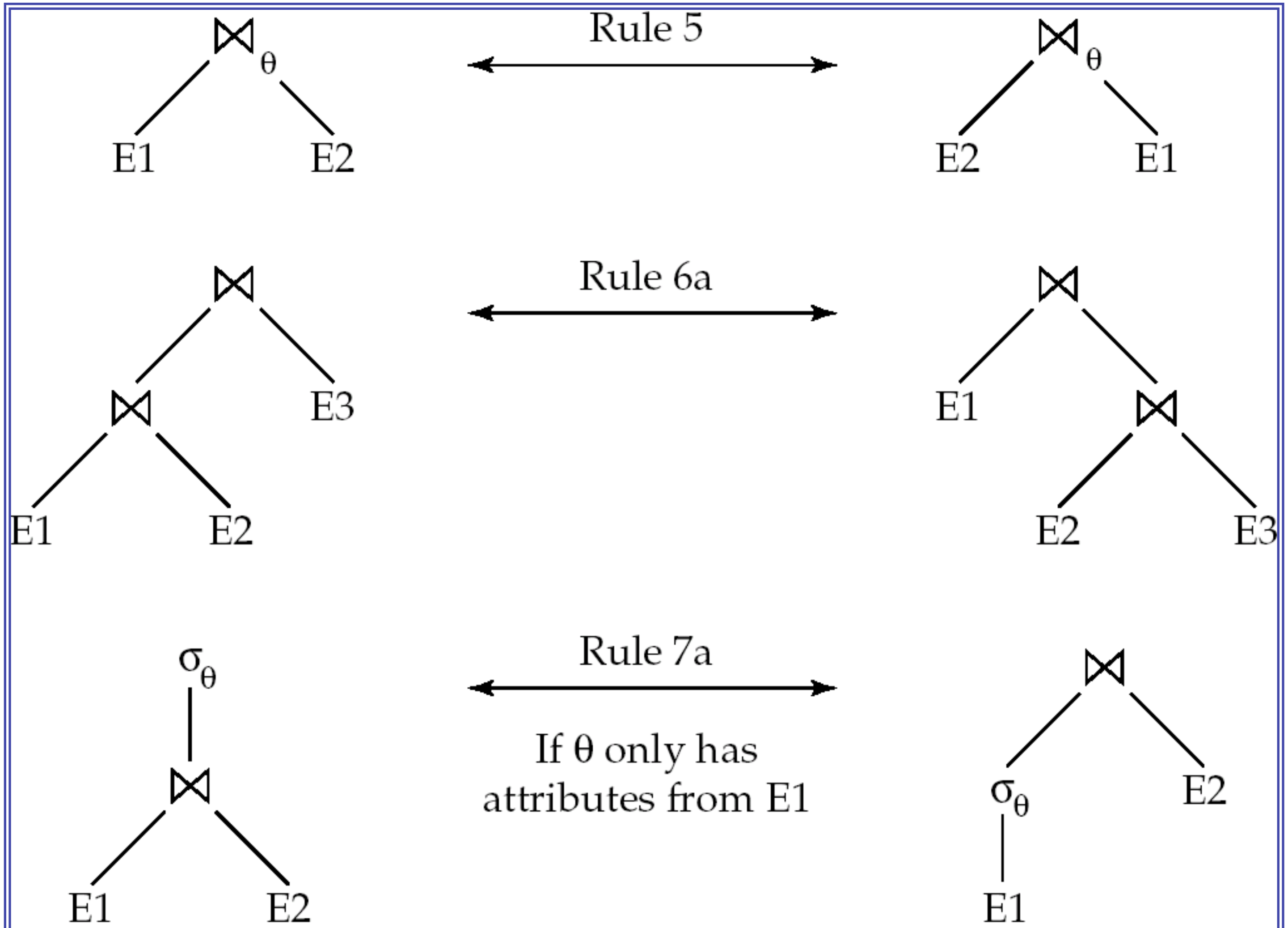
- Rule 7. The selection operation distributes over the theta join operation under the following two conditions:
  - (a) When θ_0 involves only the attributes of one of the expressions (E_1) being joined.

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

  - (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

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

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

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

  - (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 $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \prod_{L_1 \cup L_2}((\prod_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\prod_{L_2 \cup L_4}(E_2)))$$

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 $-$.

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

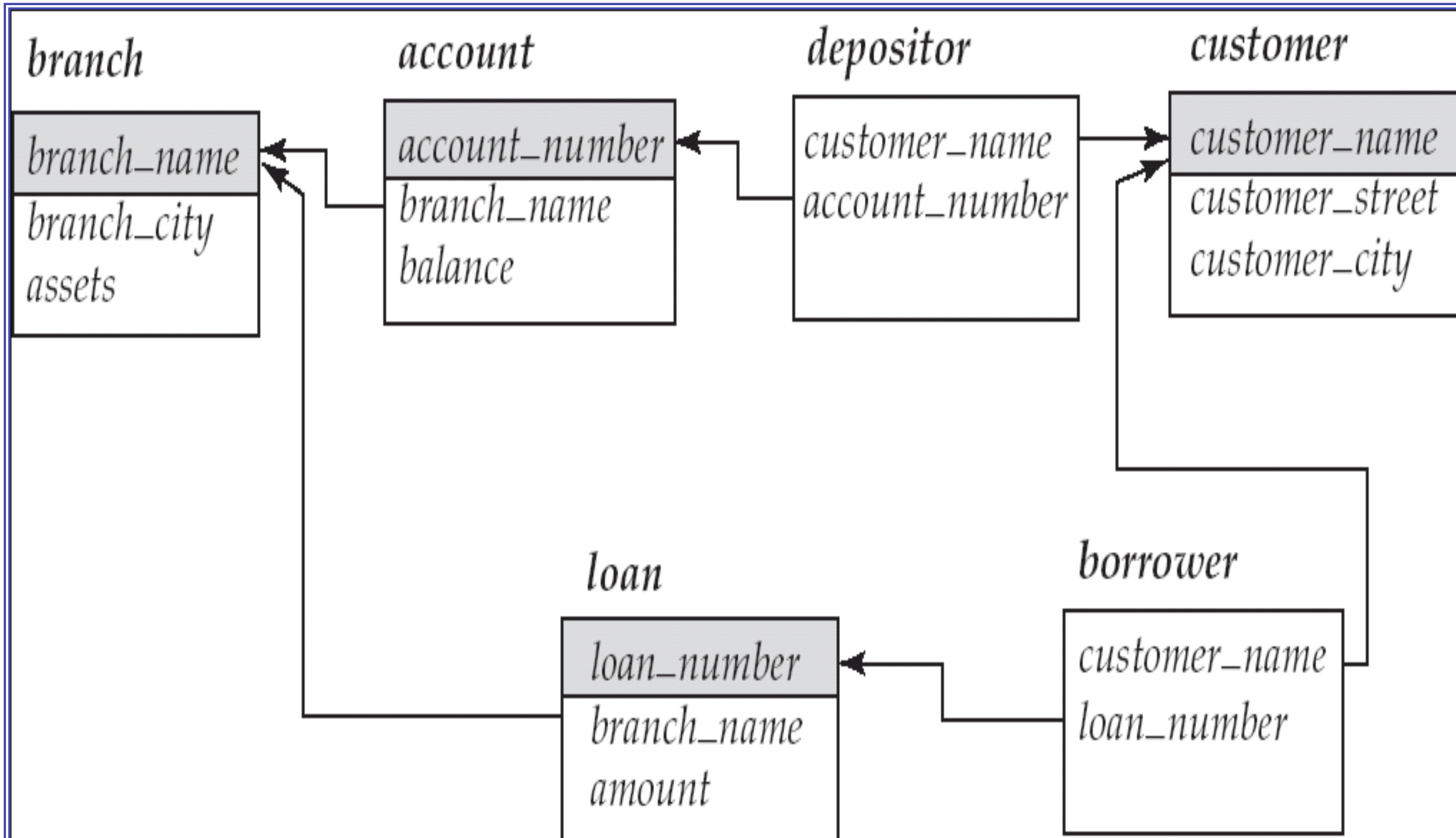
  Also: $$\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

  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):

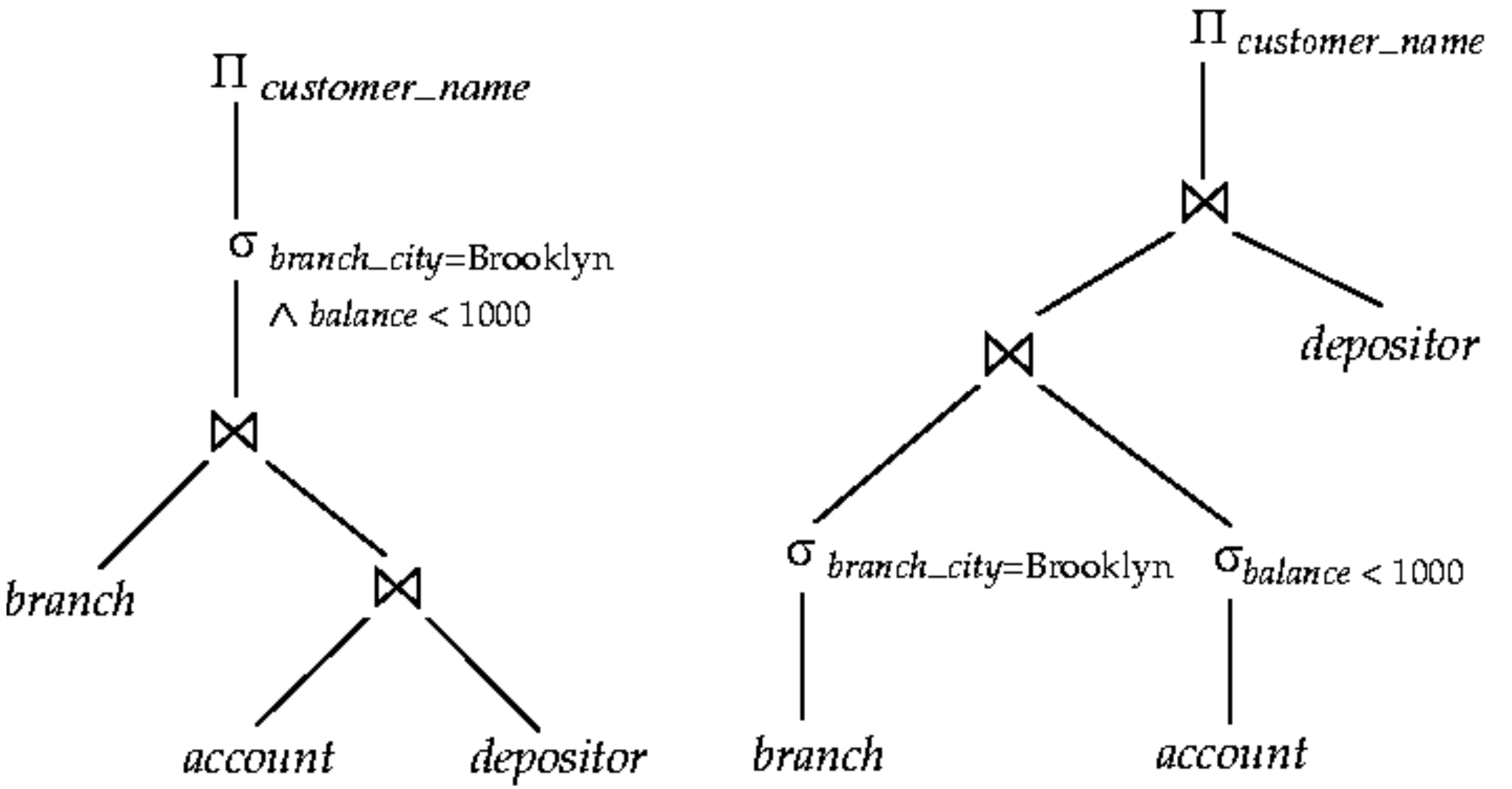
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

- 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

Database Development and Design (CPT201)

Lecture 5b: Introduction to Query Optimisation 2

Dr. Wei Wang

Department of Computing

Learning Outcomes

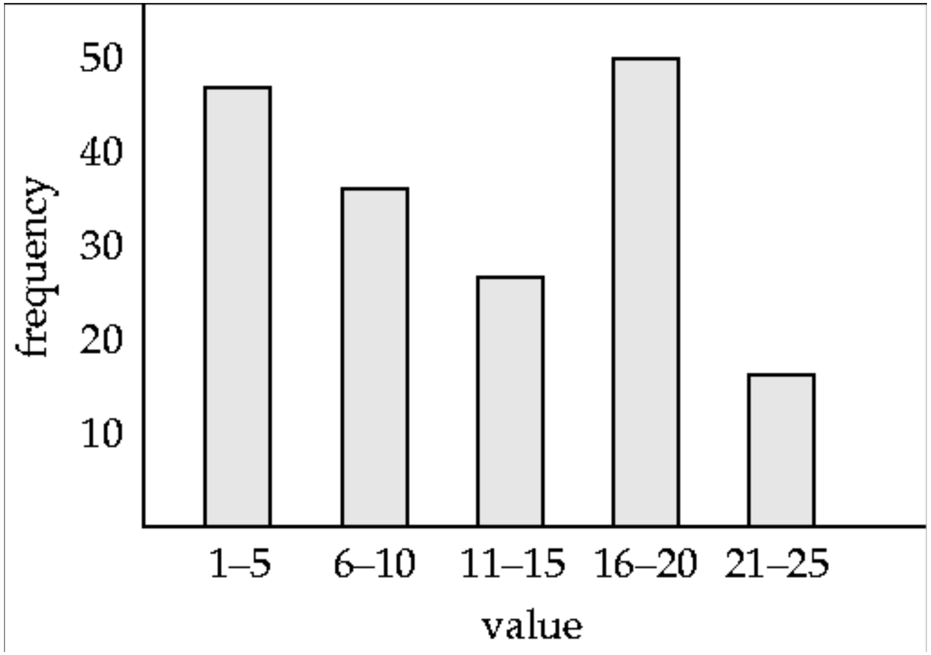
- Introduction to Query Optimisation
  - Catalog Information for Cost Estimation
  - Cost-based optimisation

Catalog Information for Cost Estimation

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_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 $\prod_A(r)$.
- If tuples of r are stored together physically in a file, then: $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$

Histograms

- Histogram on attribute *age* of relation *person*
- Equi-width histograms
- Equi-depth histograms



Estimation of the Size of Selection

- $\sigma_{A=v}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute (primary key): *size estimate = 1*
- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition. Let min(A,r) and max(A,r) denote the lowest and highest values for attribute A.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

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 $R \cap S = \varnothing$, then $r \bowtie s$ is the same as $r \times s$.
- 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.
- 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'd

- 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.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Join Operation: Running Example

- Running example: *depositor* $\bowtie$ *customer*
- Catalog information for join examples:
  - $n_{customer}$ = 10,000.
  - $f_{customer}$ = 25, which implies that $b_{customer}$ =10,000/25 = 400.
  - $n_{depositor}$ = 5000.
  - $f_{depositor}$ = 50, which implies that $b_{depositor}$ = 5,000/50 = 100.
  - *V(customer_name, depositor)* = 2,500, which implies that, on average, each customer has two accounts.
    - Also assume that *customer_name* in *depositor* is a foreign key on *customer*.
    - *V(customer_name, customer) = 10,000 (primary key)*

Join Operation: Running Example cont'd

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(customer_name, depositor)* = 2,500, and *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
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r)$ = $V(A,r)$
- 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.

Estimation of Number of Distinct Values in Selection

- 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 (*op* is >, <, etc),

  $$V(A,\sigma_\theta(r)) = V(A,r) * s$$
  - where s is the selectivity of the selection.
- In all the other cases: use approximate estimate of min($V(A,r)$, $n_{\sigma\theta(r)}$)

Estimation of Distinct Values cont'd

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
  $V(A,r \bowtie s) =$
  $\min(V(A1,r)^* V(A2 - A1,s), V(A1 - A2,r)^* V(A2,s), n_{r \bowtie s})$
  - More accurate estimate can be got using probability theory, but this one works fine generally
- Projections: Estimation of distinct values are straightforward for projections.
  - They are the same in $\prod_{A}(r)$ as in r.

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 optimisers incorporate elements of the following two broad approaches:
  - Search all the plans and choose the best plan in a cost-based fashion.
  - Uses heuristics to choose a plan.

Cost-Based Join Order Optimisation

- 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 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.

Dynamic Programming in Optimisation

- To find best plan (join tree) for a set 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 cost for joining subsets of S to find the cost of each plan. Choose the cheapest of the alternatives.
  - Base case for recursion: single relation access plan
    - Find the best selection strategy for a particular relation R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of re-computing it.

Join Order Optimisation Algorithm

```
// initialise bestplan[S].cost to ∞
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S  /* Using selections on S and indices on 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]
```

Cost of Join Order Optimisation

- With dynamic programming time complexity of optimisation with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$ as the number of subsets of the S is $2^{n.}$
- Although both numbers still increase rapidly with n, commonly occurring joins usually have less than 10 relations, and can be handled easily.

Cost-Based Optimisation with Equivalence Rules

- Many optimisers follow an approach based on
  - Using heuristic transformations to handle constructs other than joins
  - applying the cost-based join order selection algorithm to subexpressions involving only joins and selections
- General-purpose cost-based optimiser based on equivalence rules
  - easy to extend the optimiser with new rules to handle different query constructs
  - but the procedure to enumerate all equivalent expressions is very expensive

Cost-Based Optimisation with Equivalence Rules cont'd

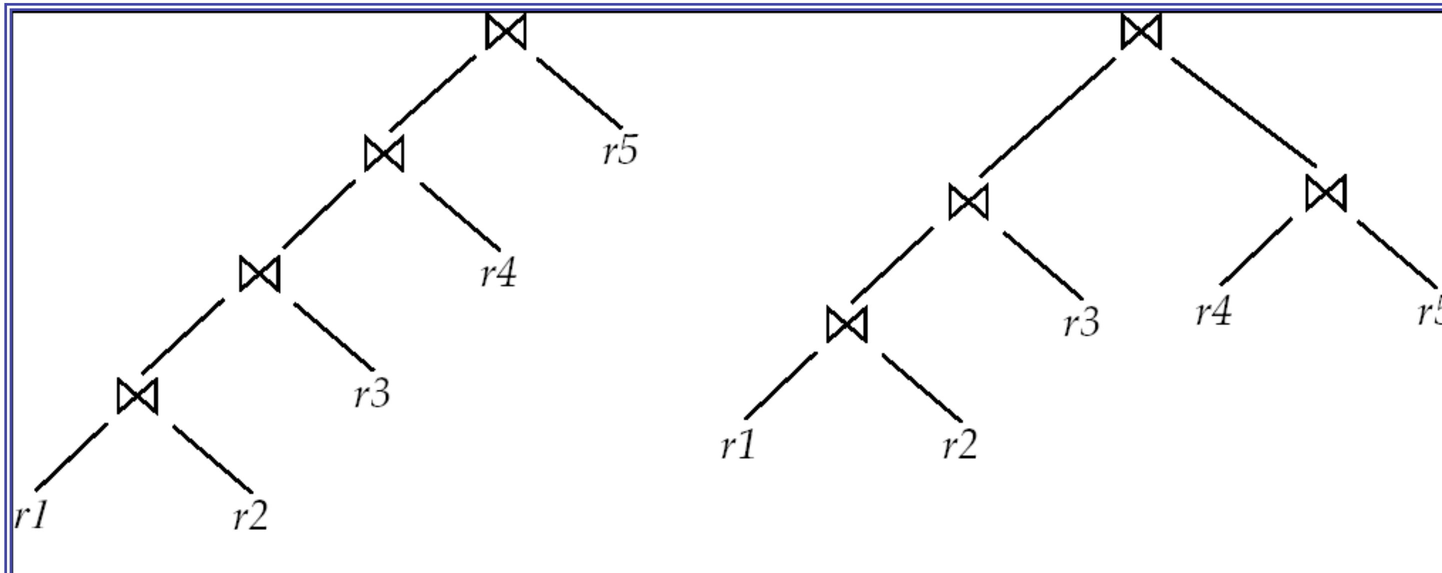
- To make the approach work efficiently requires the following:
  - A space-efficient representation of expressions
  - Efficient techniques for detecting duplicate derivations of the same expression
  - dynamic programming based on memoisation
  - avoid generating all possible equivalent plans

Heuristic Optimisation

- Cost-based optimisation 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 optimisation 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 the most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimisation.

Other heuristics: 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 left-deep join Optimisation

- 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.
  - Modify optimisation algorithm:
    - Replace "**for each** non-empty subset *S1* of *S* such that $S1 \neq S$"
    - By: **for each** relation r in S, let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is O(n!), with dynamic programming this can be reduced to $O(n\,2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimisation is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Structure of Query Optimisers

- Many optimisers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimisation complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimisation used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these

Structure of Query Optimisers cont'd

- Some query optimisers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimisation for each block
  - Some optimisers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
- Even with the use of heuristics, cost-based query optimisation imposes a substantial overhead.
  - But is worth for expensive queries
  - Optimisers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

End of Lecture

- Summary
  - Transformation of Relational Expressions
  - Catalog Information for Cost Estimation
  - Cost-based optimisation
  - Dynamic Programming for Choosing Evaluation Plans

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