Database Management Systems 2010/11

– Chapter 5: Query Optimization –

J. Gamper

- Introduction
- Statistical information for cost estimation
- Transformation of relational expressions (equiv. rules)
- Rule-based and cost-based optimization
- Optimizing nested subqueries
- Materialized views and view maintenance

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Represented as **query evaluation plan** (query plan)
  - Annotated RA-expression that specifies for each operator detailed instructions on how to evaluate it.
- Cost difference between a good and a bad query evaluation plan can be enormous
  - e.g., performing a $r \times s$ followed by a selection $r.A = s.B$ is much slower than performing a join on the same condition
- Query optimizer needs to estimate cost of operations
  - Depends critically on statistical information about relations
  - Estimates statistics for intermediate results to compute cost of complex expressions

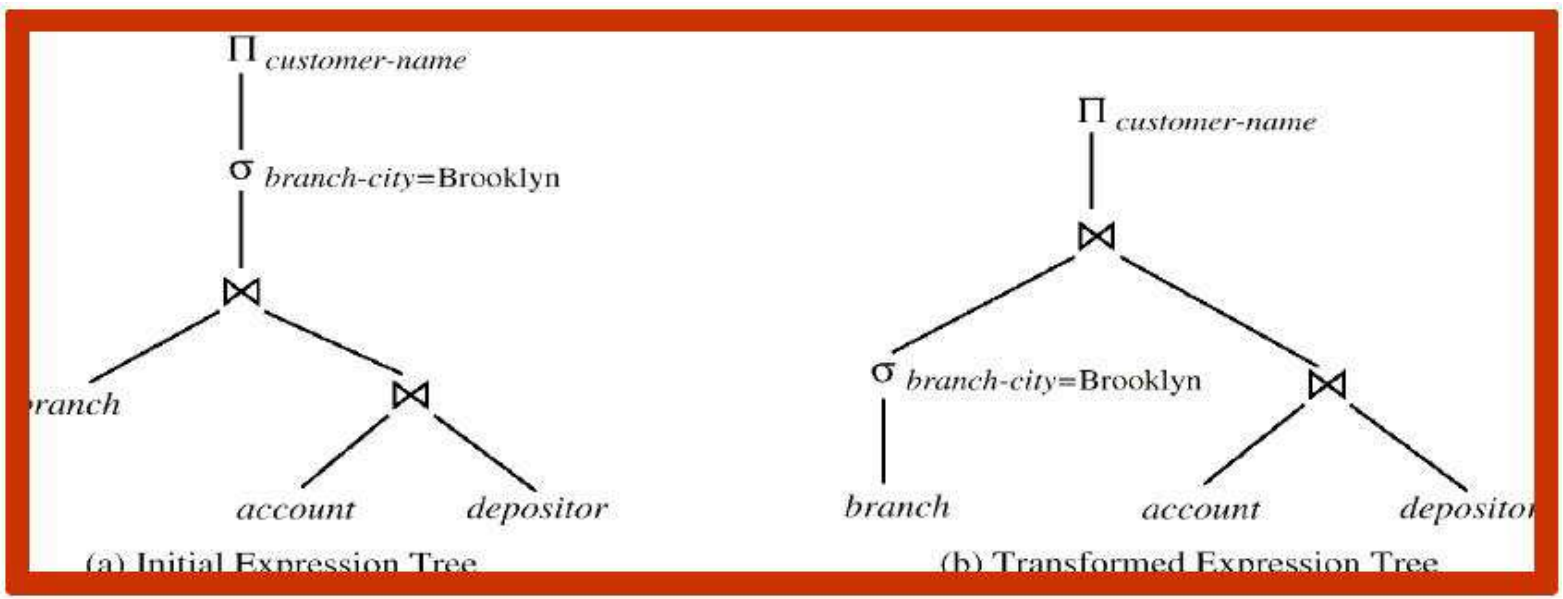
Introduction ...

- **Example:** Find the names of all customers who have an account at any branch located in Brooklyn.
  - The following expression produces a large intermediate relation

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

  - Transformation into a more efficient expression

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



Introduction ...

- **Goal of query optimizer:** Find the most efficient query evaluation plan for a given query.
- Two different paradigms for query optimization:
  - **Cost-based** optimization:
    1. Generate logically equivalent expressions (using equivalence rules to transform an expression into an equivalent one)
    2. Annotate resulting expressions with information about algorithms/indexes for each operator
    3. Choose the cheapest plan based on **estimated cost**
  - **Rule-based/heuristic** optimization:
    1. Generate logically equivalent expressions (using equivalence rules), but **controlled** by a set of heuristic query optimization rules
- In general, it is not possible to identify the optimal query tree. Instead, a reasonably efficient one is chosen.

Statistical Information

- The cost of an operation depends on the size and other statistics of its inputs, which is partially stored in the database catalog and can be used to estimate statistics on the results of various operations.
  - 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 $\pi_A(r)$
  - $SC(A, r)$: selection cardinality of attribute A of relation r; average number of records that satisfy equality on A

Catalog Information about Indices

- f_i: average fan-out of internal nodes of index i for tree-structured indexes such as B^+-trees
- HT_i: number of levels in index i, i.e., the height of i
  - For a B^+-tree on attribute A of relation r, $HT_i = \lceil \log_{f_i}(V(A, r)) \rceil$
  - For a hash index, $HT_i = 1$
  - LB_i: number of lowest-level index blocks in i, i.e., the number of blocks at the leaf level of the index
- For accurate statistics, the catalog information has to be updated every time a relation is modified
  - Many systems update statistics only during periods of light system load, thus statistics is not completely accurate.
  - Plan with lowest estimated cost might not be the cheapest!

Measures of Query Cost

- Recall that
  - typically disk access is the predominant cost, and it is relatively easy to estimate;
  - the number of block transfers from disk is used as a measure of the actual cost of evaluation;
  - it is assumed that all transfers of blocks have the same cost;
    - Real life optimizers do not make this assumption, and distinguish between sequential and random disk access
  - we do not include cost to writing output to disk.

Transformation of RA Expressions

- Two relational algebra expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - 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 on every legal database instance the two expressions generate the same multiset of tuples
- An **equivalence rule** says that expressions of two different forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

Let $E, E_1, \ldots$ be RA expressions and $\theta, \theta_1, \ldots$ be predicates/conditions

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 projections is needed, the others can be omitted (L_i are lists of attributes).

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(E))\ldots)) = \pi_{L_1}(E)$$

4. Selections can be combined with CP and theta joins

$$a. \quad \sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$$
$$b. \quad \sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

Equivalence Rules . . .

5. Theta-joins (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 way:

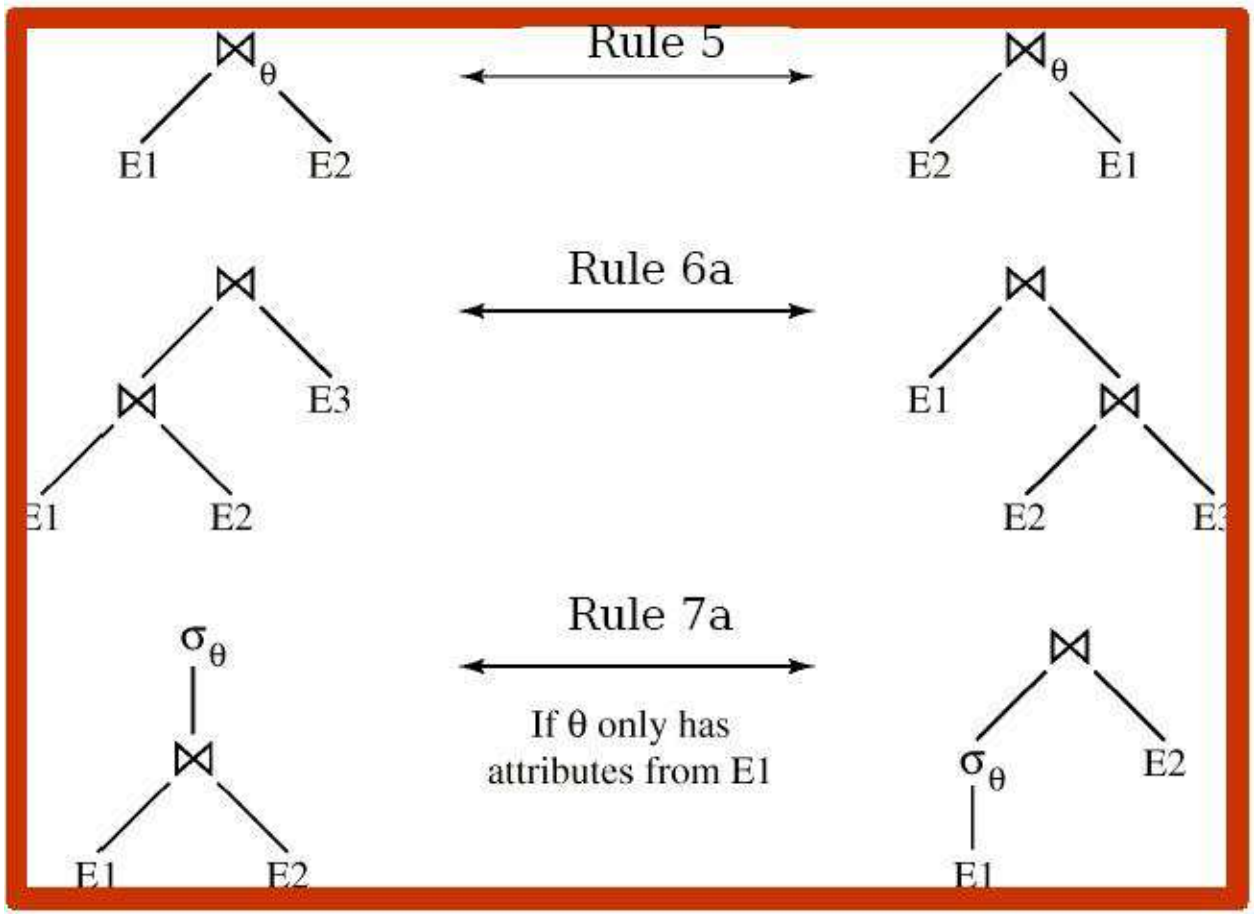
$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

- Any of the conditions might be empty, hence the Cartesian product operation is also associative and commutative
- Commutativity and associativity of join operations are important for join reordering.

Equivalence Rules . . .

- Graphical representation of equivalence rules



Equivalence Rules . . .

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 $(E1)$ 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:

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

Equivalence Rules . . .

8. The projection operation distributes over the theta join operation as follows:

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.

(a) if θ 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_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$
- 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 ...

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**.

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

Equivalence Rules . . .

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

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

- Also $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$
  and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\pi_L(E_1 \cup E_2) = (\pi_L(E)) \cup (\pi_L(E_2))$$

Transformation Examples

- **Example 1:** Bank database
  - *Branch(branch-name, branch-city, assets)*
  - *Account(account-number, branch-name, balance)*
  - *Depositor(customer-name, account-number)*
- **Query:** Find the names of all customers who have an account at some branch located in Brooklyn.

$$\pi_{customer-name}(\sigma_{branch-city='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using rule 7a:

$$\pi_{customer-name}(\sigma_{branch-city='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$$

- Performing the selection as early as possible reduces the size of the intermediate relation to be joined.

Transformation Examples

- **Example 2:** Multiple transformations are often needed
- **Query:** Find the names of all customers with an account at Brooklyn whose balance is below $1000.

$$\pi_{customer-name}(\sigma_{branch-city='Brooklyn' \wedge balance<1000}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using join associativity (Rule 6a):

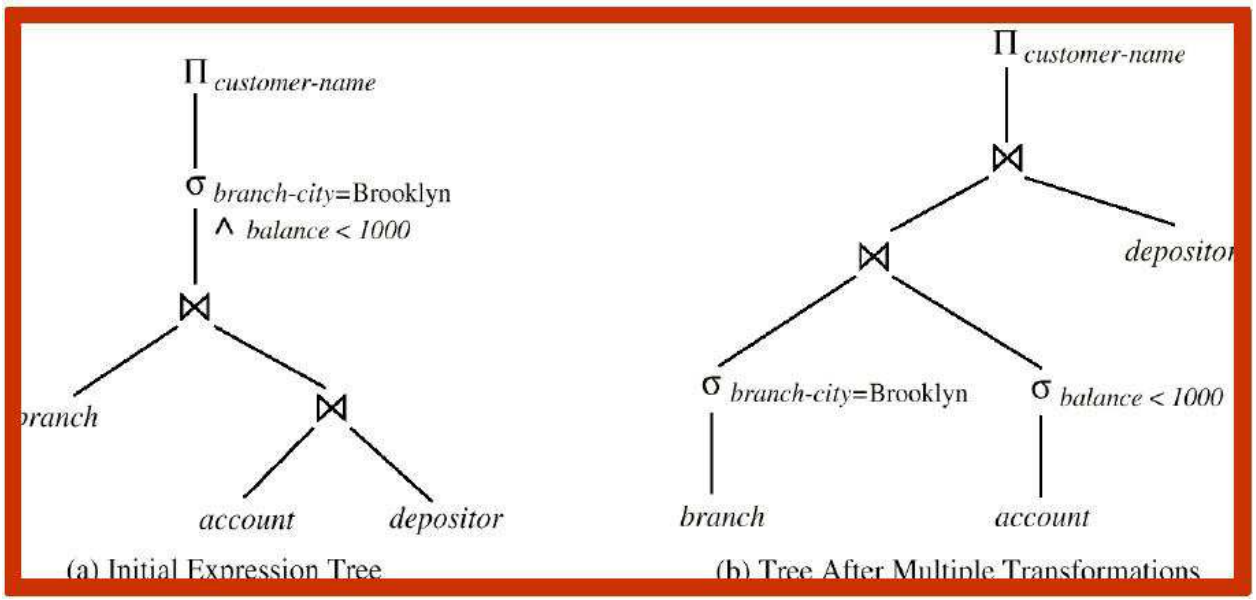
$$\pi_{customer-name}(\sigma_{branch-city='Brooklyn' \wedge balance<1000}(branch \bowtie account) \bowtie depositor)$$

- Second, apply the "perform selections early" rule (7b), resulting in the subexpression

$$\sigma_{branch-city='Brooklyn'}(branch) \bowtie \sigma_{balance<1000}(account)$$

Transformation Exampless . . .

- **Example 2** (continued)
  - Tree representation after multiple transformations



Transformation Exampless ...

- **Example 3:** Projection operation in the following query:

$$\pi_{customer-name}((\sigma_{branch-city='Brooklyn'}(branch) \bowtie account) \bowtie depositor)$$

- When we compute

$$(\sigma_{branch-city='Brooklyn'}(branch) \bowtie account)$$

  we obtain an intermediate relation with schema
  (branch-name, branch-city, assets, account-number, balance)
- Push projections using equiv. rules 8a and 8b; thus, eliminate unneeded attributes from intermediate results:

$$\pi_{customer-name}($$
$$\pi_{account-number}(\sigma_{branch-city='Brooklyn'}(branch) \bowtie account) \bowtie depositor)$$

Transformation Exampless ...

- **Example 4:** Join ordering
- For relations r_1, r_2, and r_3 we have

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

- 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.

Transformation Exampless ...

- **Example 5:** Join ordering
- Consider the expression

$$\pi_{customer-name}((\sigma_{branch-city='Brooklyn'}(branch)) \bowtie account \bowtie depositor)$$

- Could compute $account \bowtie depositor$ first, and join result with

$$branch - city =' Brooklyn'(branch)$$

  but $account \bowtie depositor$ is likely to be a large relation.
- Since it is more likely that only a small fraction of the banks customers have accounts in branches located in Brooklyn, it is better to compute first

$$branch - city =' Brooklyn'(branch) \bowtie account$$

Enumeration of Equivalence Expressions

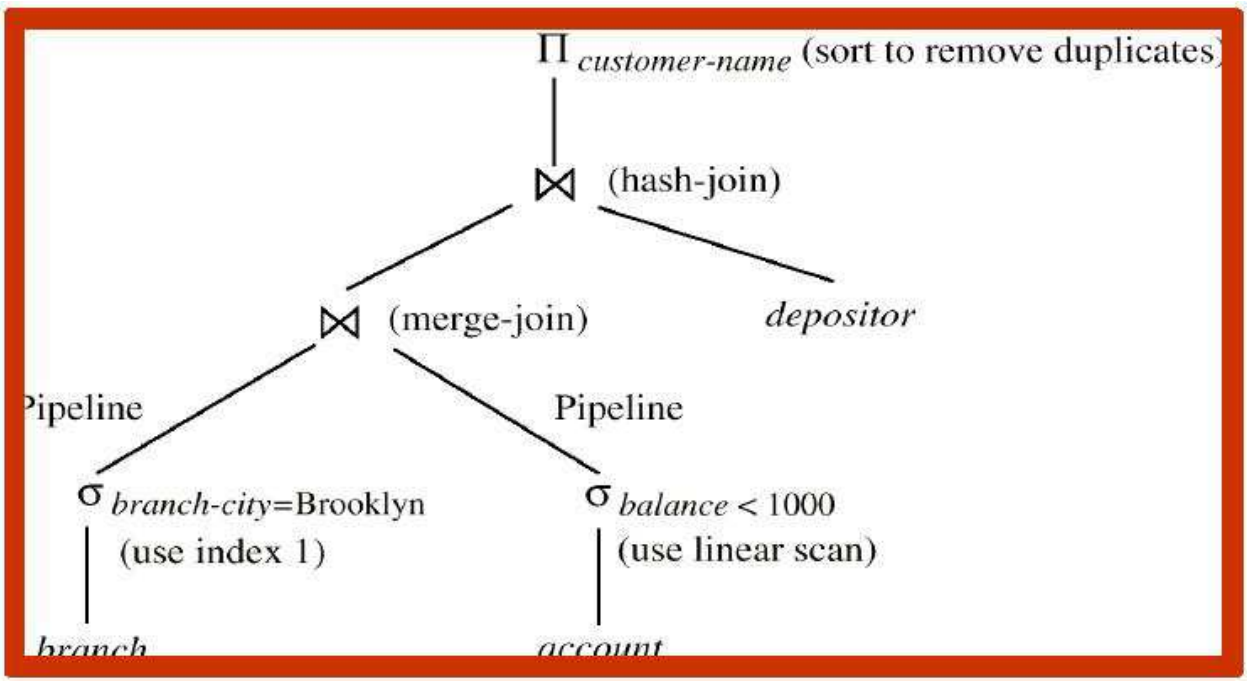
- **Query optimizers** use the equivalence rules to systematically generate expressions that are equivalent to the given expression
- Naive algorithm

  repeat
  foreach *expression found so far* **do**
  use all applicable equivalence rules, and add newly generated expression to the set of expressions found so far
  end
  until *no more expressions can be found*;

- This approach is very expensive in space and time
- Reduce space requirements by sharing common subexpressions:
  - 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 (e.g. when applying join associativity)
- Time requirements are reduced by not generating all expressions (e.g., take cost estimates into account)

Evaluation Plan

- **Evaluation plan (query plan/query tree):** Defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Choice of Evaluation Plans

- When choosing the "best" evaluation plan, the query optimizer must consider the interaction of evaluation techniques
- 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
  - **Cost-based optimization:** Search all the plans and choose the best plan in a cost-based fashion.
    - Recently, they are becoming more and more popular
  - **Rule-based optimization:** Uses heuristics to choose a plan.

Cost-Based Optimization

- **Algorithm**
  1. Use transformations (equivalence rules) to generate multiple candidate evaluation plans from the original evaluation plan.
  2. Cost formulas estimate the cost of executing each operation in each candidate evaluation plan. Cost formulas are parameterized by
     - statistics of the input relations;
     - dependent on the specific algorithm used by the operator;
     - CPU time, I/O time, communication time, main memory usage, or a combination.
  3. The candidate evaluation plan with the **least total cost** is selected for execution.

Cost-Based Optimization ...

- A good ordering of joins is important for reducing the size of temporary results.
- **Example:** 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 = 3, the number is 12
    - With n = 7, the number is 665,280
    - With n = 10, the number is greater than 17.6 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.
- **Example:** Find the best join order for $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
  - 12 different join orders for the first part, 12 different orders for the rest
  - Instead of considering 12 x 12 join orders we first find the best order for $\{r_1, r_2, r_3\}$ and then use that order for the second part (i.e., joins with r_3, r_4)
    - 12 + 12 orders instead of 144!

Join Order Optimization ...

- To find the best evaluation plan for a set S 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 costs for joining subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives.
  - When a plan for any subset is computed, store it and reuse itwhen it is required again, instead of recomputing it from scratch

$\Rightarrow$ **dynamic programming**

Join Order Optimization ...

- Dynamic programming algorithm for join order optimization

Algorithm: *findbestplan*(S)

if *bestplan*[S].*cost* $\neq \infty$ **then**
    return *bestplan*[S];
else
    //*bestplan*[S] has not been computed earlier, compute it now
    foreach *non-empty subset* $S1$ *of* S *such that* $S1 \neq S$ **do**
        $P1$ = *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* **then**
            bestplan[S].cost = cost;
            bestplan[S].*plan* = "execute P1.plan;
                execute P2.plan;
                join results of P1 and P2 using A";
        end
    end
end
return *bestplan*[S];

Cost of Optimization

- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small number of operations, n; generally $n < 10$)
- With dynamic programming, time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 17.6 billion!
- Space complexity is $O(2^n)$

Interesting Orders

- Consider the expression $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
- An **interesting sort order** is a particular sort order of tuples that could be **useful for a later operation**.
  - Generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common to only r_1 and r_2 is not useful.
  - Using merge-join to compute $r_1 \bowtie r_2 \bowtie r_3$ may be costlier, but may provide an output sorted in an interesting order.
- Not sufficient to find the best join order for each subset of the n relations; must find the best join order for each subset, for each interesting sort order
  - Simple extension of algorithm above; usually, the number of interesting orders is small and doesn't affect time/space complexity significantly

Cost-Based Optimization Example

- **Example:** $\sigma_{CPR=0810643773}(Emp)$
- Statistics:
  - $|Emp| = 10,000$ tuples
  - 5 tuples per block
  - Secondary B^+-tree index of depth 4 on CPR
  - CPR is primary key
- Plan p1: full table scan
  - $cost(p1) = (10,000/5)/2 = 1,000$ blocks
- Plan p2: B^+-tree lookup
  - $cost(p2) = 4 + 1 = 5$ blocks

Cost-Based Optimization Example ...

- **Example:** $\sigma_{DNo>15}(Emp)$
- Statistics:
  - $|Emp| = 10{,}000$ tuples
  - 5 tuples per block
  - Cluster index (primary index) on DNo of depth 2
  - 50 different departments
- Plan p1: full table scan
  - $cost(p1) = 10,000/5 = 2,000$ blocks
- Plan p2: cluster index search
  - $cost(p2) = 2 + (50 - 15)/50 * (10,000/5) = 1,400$ blocks

Cost-Based Optimization Example ...

- **Example:** $Emp \bowtie_{DNo=DNum} Dept$
- Statistics:
  - $|Emp| = 10{,}000$ tuples ; 5 *Emp*-tuples per block
  - $|Dept| = 125$; 10 *Dept*-tuples per block
  - Hash index on $Emp(DNo)$
  - 4 EmpDept result tuples per block
- Plan p1: Block nested loop join with *Emp* as outer loop
  - $cost(p1) = (10.000/5) + (10.000/5) * (125/10) + (10.000/4) = 30.500$ IOs
    - (10.000/4 is cost of writing final output)
- Plan p2: Indexed nested loop with *Dept* as outer loop and hashed lookup in *Emp*
  - $cost(p2) = (125/10) + 125 * (10.000/125/5) + (10.000/4) = 4.513$ IOs
    - 10.000/125/5 is the avg. number of blocks/department

Heuristic Optimization

- Cost-based optimization 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 optimization transforms the query-tree by using a set of heuristic rules that typically (but not in all cases) improve execution performance.
- Overall goal of heuristic rules:
  - Try to **reduce the size of (intermediate) relations** as early as possible!

Heuristic Optimization ...

- Heuristic rules
  - 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.

Heuristic Optimization ...

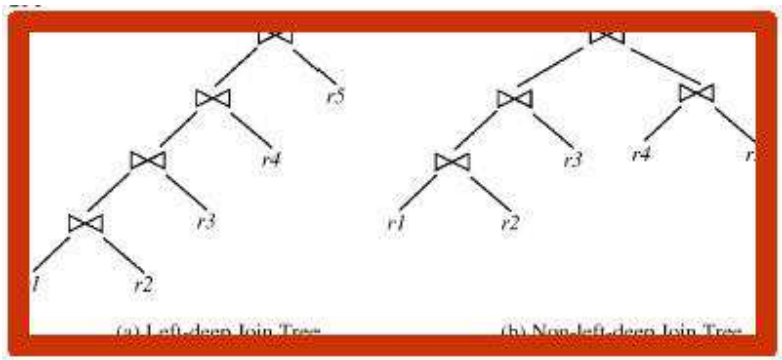
- **Example:** Consider the expression $\sigma_\theta(r \bowtie s)$, where θ is on attributes in s only.
- "Selection early" rule would push down the selection operator, producing $r \bowtie \sigma_\theta(s)$.
- This is not necessarily the best plan if
  - relation r is extremely small compared to s
  - and there is an index on the join attributes of s,
  - but there is no index on the attributes used by θ.
- The early select would require a scan of all tuples in s, which is probably more expensive than the join!

Heuristic Optimization ...

- Steps in typical heuristic optimization
  1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
  2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
  3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
  4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
  5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
  6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

Structure of Query Optimizers

- The System R/Starburst optimizer considers only **left-deep join orders**.
  - Reduces optimization complexity ($O(n!)$ to consider all plans)
  - Generates plans amenable to pipelined evaluation; only the left input to each join is pipelined.



- System R/Starburst also uses heuristics to push selections and projections down the query tree.
- Heuristic optimization used in some versions of Oracle.
- For scans using secondary indexes, some optimizers consider the probability that the page containing the tuple is in the buffer.

Structure of Query Optimizers ...

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - System R/Starburst uses a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.
  - Some Oracle versions work like this:
    - For an n-way join, consider n evaluation plans, each plan using a left-deep join order starting with a different one of the n relations.
    - Heuristic: Construct the join order for each of these plans by repeatedly selecting the best relation to join next, on the basis of the available access paths (nested loop or sort-merge join, depending on indexes).
    - Heuristic: Choose one of the n evaluation plans, based on minimizing the number of nested-loop joins that do not have an index on the inner relation and on the number of sort-merge joins.

Structure of Query Optimizers ...

- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
- This expense is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.
- Intricacies of SQL complicate query optimization further
  - e.g. nested subqueries

Optimizing Nested Subqueries

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- **Example:**

```
SELECT customer-name
FROM   borrower
WHERE EXISTS (SELECT *
              FROM   depositor
              WHERE  depositor.customer-name
                     = borrower.customer-name )
```

- Conceptually, nested subquery is executed once for each tuple in the outer level query

Optimizing Nested Subqueries ...

- Evaluation of nested subqueries may be inefficient since
  - a large number of calls may be made to the nested query (once for each tuple in the out level query)
  - there may be unnecessary random I/O as a result
- SQL optimizers try to transform nested subqueries to joins, enabling to use efficient join techniques
  - e.g., earlier nested query can be rewritten as

```
SELECT customer-name
FROM   borrower, depositor
WHERE  depositor.customer-name = borrower.customer-name
```

- In general, it is not possible to move the entire nested subquery FROM-clause to the outer level FROM-clause
  - Temporary relation is created instead and used in the outer level query

Materialized Views

- A **materialized view** is a view whose contents are computed and stored / cached on disk.
- Consider the view

```
CREATE VIEW branch-total-loan(branch-name, total-loan) AS
SELECT branch-name, SUM(amount)
FROM   loan
GROUP BY branch-name
```

- Materializing the above view would be very useful if the total loan amount is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts
- Materialization is a typical **space/time tradeoff**: We trade space for time.

Materialized View Maintenance

- **Materialized view maintenance**: Keeping a materialized view up-to-date with the underlying data
- Two different methods to maintain materialized views:
  - **Re-computation** on every update.
  - A better option is **incremental view maintenance**: Changes to database relations are used to compute changes to materialized view, which is then updated.
- Incremental view maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition.
  - Manually written code to update the view whenever database relations are updated.
  - Supported directly by the database.

Incremental View Maintenance

- The changes to a relation that can cause a materialized view to become out-of-date are: **insert, delete**, and **update**.
  - To simplify our description, we only consider inserts and deletes and replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple.
- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**.
- Set of tuples inserted to and deleted from a relation r are denoted i_r and d_r, respectively.
- In the following we describe for each RA operation how to compute the change/differential to the view, given changes to its inputs.

Incremental View Maintenance ...

- **Join:** Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Insert set of tuples i_r to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - and rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
  - Therefore, $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Delete set of tuples d_r from r:
  - We get in a similar way : $v^{new} = v^{old} - (d_r \bowtie s)$

Incremental View Maintenance ...

- **Selection**: Consider a view $v = \sigma_\theta(r)$
  - Insert set of tuples i_r to r: $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - Delete set of tuples d_r from r: $v^{new} = v^{old} - \sigma_\theta(d_r)$

- **Projection** is a more difficult operation
  - Assume a relation r: $R = (A, B)$ and $r(R) = \{(a, 2), (a, 3)\}$
  - Consider view $v = \pi_A(r)$ which has a single tuple $v = \{(a)\}$.
  - If we delete tuple $(a, 2)$ from r, we should not delete tuple (a) from v; but if we then delete $(a, 3)$ as well, we should delete the tuple.
  - Therefore, for each tuple in a projection $\pi_A(r)$, we keep a count of how many times it was derived
    - Insertion of a tuple to r: If the resultant tuple is already in $\pi_A(r)$, increment its count; else add a new tuple with $count = 1$.
    - Deletion of a tuple from r: Decrement the count of the corresponding tuple in $\pi_A(r)$; if the count becomes 0, delete the tuple from $\pi_A(r)$.

Incremental View Maintenance ...

- **Aggregation – count**: $v =_A \mathcal{G}_{count(B)}(r)$
- Insertion of a set of tuples i_r to r
  - For each tuple t in i_r: If the corresponding group is already present in v, increment its count; else add a new tuple $(t.A, 1)$ to v.
- Deletion of a set of tuples d_r from r
  - For each tuple t in d_r: Look for the group $t.A$ in v and subtract 1 from the count for the group. If the count becomes 0, delete from v the tuple for the group $t.A$

- **Aggregation – sum**: $v =_A \mathcal{G}_{sum(B)}(r)$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v.
    - Cannot simply test for $sum = 0$ (why?)

Incremental View Maintenance ...

- **Aggregation – avg**
  - To handle the case of avg, we maintain the sum and count aggregate values separately, and divide at the end

- **Aggregation – min, max**: $v =_A \mathcal{G}_{min(B)}(r)$
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of r that are in the same group to find the new minimum.

Incremental View Maintenance . . .

- **Set intersection:** $v = r \cap s$
  - Insertion of a tuple t into r: Check if t is present in s, and if so add tuple t to v.
  - Deletion of a tuple t from r: Delete tuple t from the intersection if it is present.
  - Updates to s are symmetric
  - The other set operations, **union** and **set difference** are handled in a similar fashion.

Query Optimization and Materialized Views

- Query optimization can be performed by treating materialized views just like regular relations
  - However, materialized views offer further opportunities for optimization.
- **Rewriting** queries to use materialized views:
  - Suppose a materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Might provide a more efficient query plan
    - Whether to do so depends on cost estimates for the two alternatives

Query Optimization and Materialized Views ...

- **Replacing** a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that s has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace the view v by its definition $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan.

Materialized View Selection

- **Materialized view selection:** "What is the best set of views to materialize?".
  - This decision must be made on the basis of the system **workload**, i.e., a sequence of queries and updates that reflect the typical load on the system.
- Indexes are just like materialized views; problem of **index selection** is closely related to that of materialized view selection, although it is simpler.
- Some database systems, provide tools to help the database administrator with index and materialized view selection