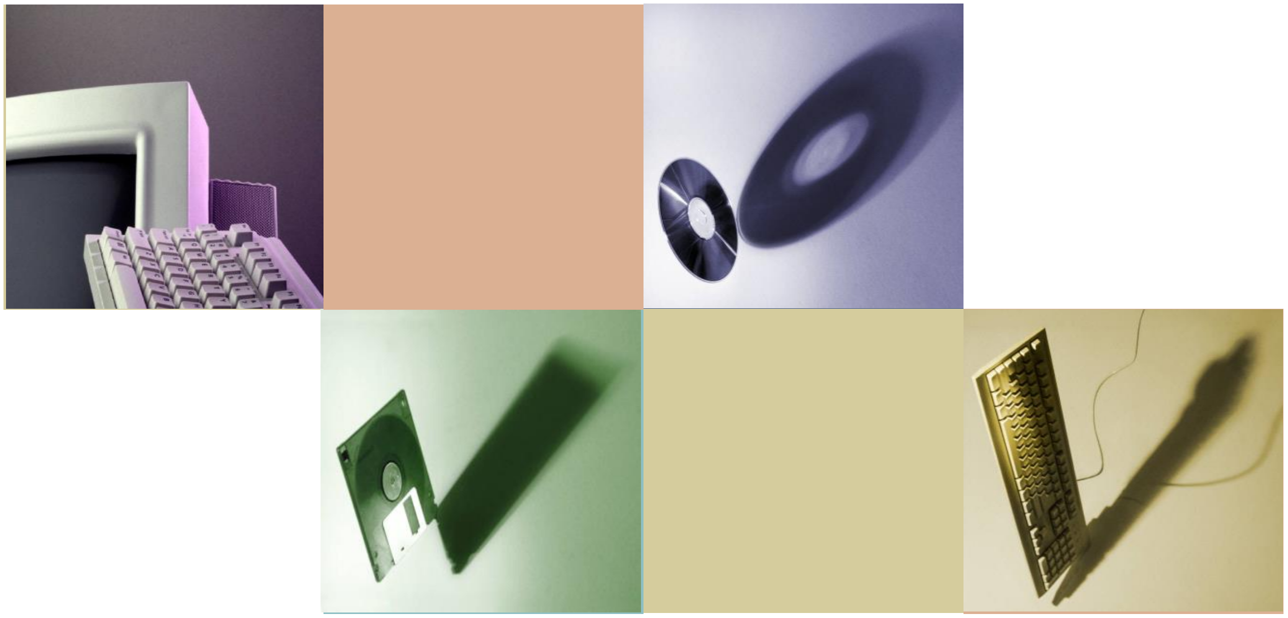
Query Processing & Optimization



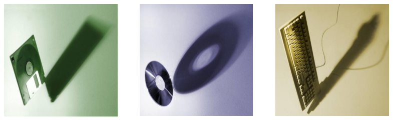
Database

Faculty of Computer Science

Universitas Indonesia

Objectives

- ♦ To understand the techniques used by a DBMS to process, optimize, and execute high level queries.
- ♦ To enable students representing SQL query in relational algebra expressions and query tree

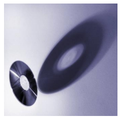
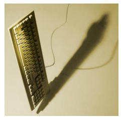


Outline (1)

- Introduction
- Translating SQL Queries into Relational Algebra
- Algorithms for External Sorting
- Algorithms for SELECT and JOIN Operations
  - Implementing SELECT Operation
  - Implementing JOIN Operation

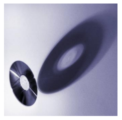
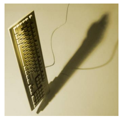
Outline (2)

- ♦ Algorithms for PROJECT and Set Operations
  - Implementing Project operation
  - Implementing Set operations
  - Implementing Aggregate Functions operations
  - Implementing Outer Join operation

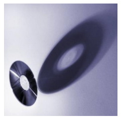
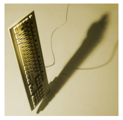
Introduction (1)

- A query expressed in high level query language such as SQL must first be **scanned**, **parsed**, and **validated**.
  - **Scanner** identifies the language tokens (SQL keywords, attribute names, and relation names).
  - **Parser** checks the correctness in query syntax.
  - **Validate** the query by checking that all attribute and relation names are valid and semantically meaningful names in the schema

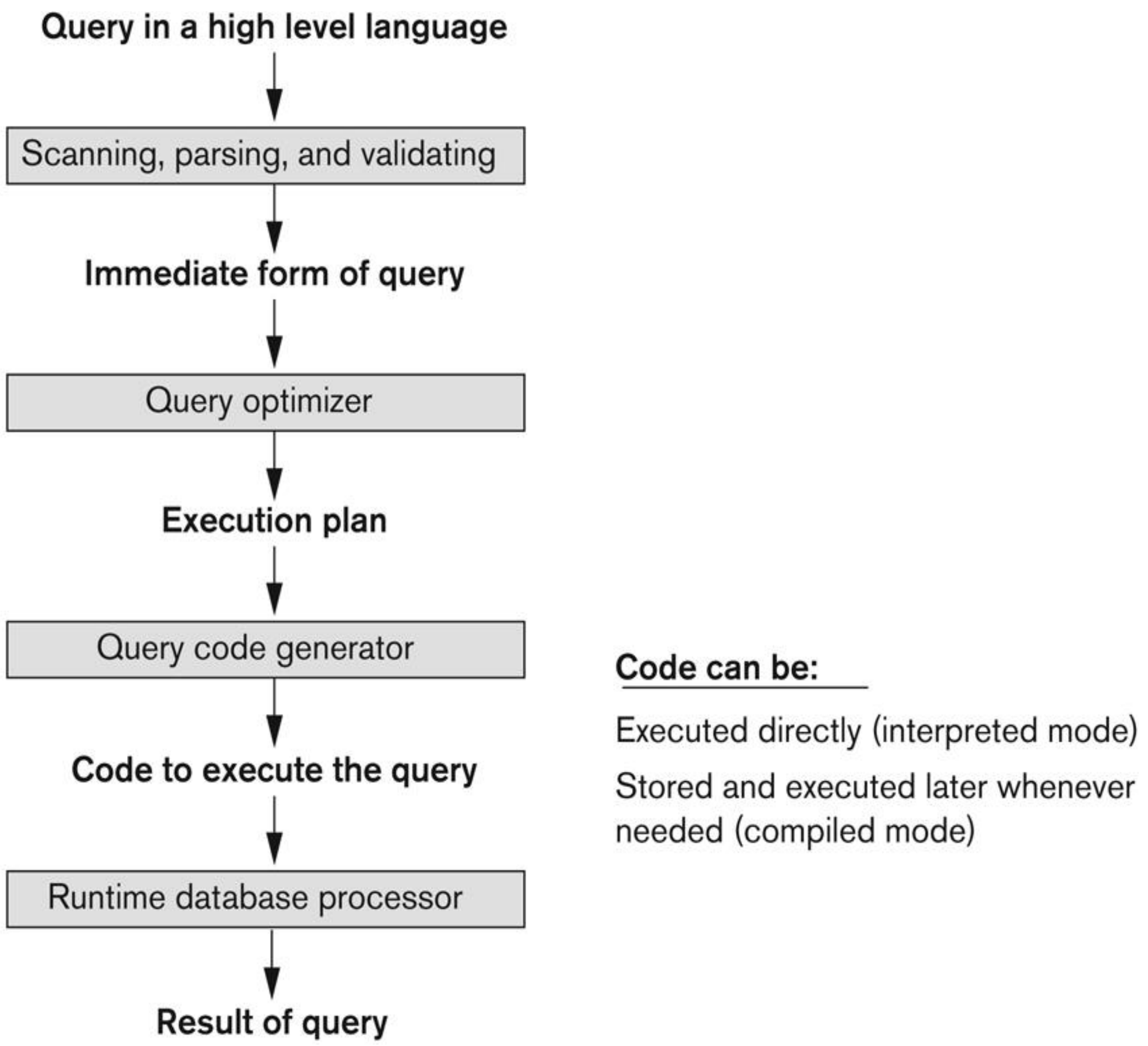



Introduction (2)

- After those three steps, an internal representation of the query is created using:
  - A tree data structure called **query tree**, or
  - A graph data structure called **query graph**
- Next, the DBMS must devise an **execution strategy** for retrieving the result. A query usually has many possible execution strategies.
- Process of choosing a suitable one for processing a query is known as **query optimization**.

Introduction (3)

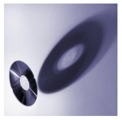
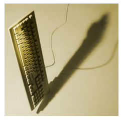


Query in a high level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query
Code can be:
Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)

Figure 15.1
Typical steps when processing a high-level query.

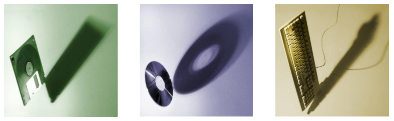
Introduction (4)

- **Query optimizer** module has the task of producing an execution plan.
- **Code generator** generates the code to execute the plan.
- **Runtime database processor** has the task of running the code (compiled or interpreted mode) to produce the query result.

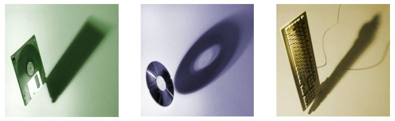
Introduction (5)

- The term **optimization** is misnomer because in some cases the chosen execution plan is not optimal, just **reasonably efficient strategy** for executing the query.
- Finding the optimal strategy is too time-consuming
- **Planning of an execution strategy** is a more accurate description.



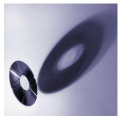
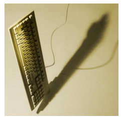
Introduction (6)

- RDBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or optimal strategy.
- Each DBMS typically has a number of general database access algorithms that implement relational operations such as SELECT or JOIN or combination of both.
- Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query and particular physical database design, can be considered by the query optimization module.



Measures of Query Cost (1)

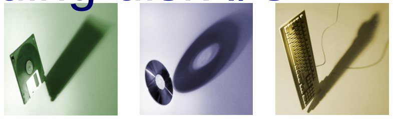
- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost and is also relatively easy to estimate. Measured by taking into account
  - Number 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

Measures of Query Cost (2)

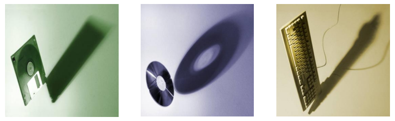
- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $b * t_T + S * t_S$
- 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
- Several algorithms can reduce disk IO 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
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation



Translating SQL into Relational Algebra (1)

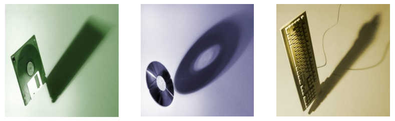
- An SQL query translated into an equivalent extended relational algebra expression (represented as a query tree) and then optimized it.
- Typically SQL queries are decomposed into **query block** (which form the basic units that can be translated into the algebraic operators and optimized).



Translating SQL into Relational Algebra (2)

- A Single “SELECT-FROM-WHERE” expression, as well as GROUP BY and HAVING clause is part of a query block.
- Nested queries, on the other hand, are identifed as separate query blocks.
- Consider the following query:

```
SELECT  Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX(Salary)
                 FROM EMPLOYEE
                 WHERE Dno=5 );
```



Translating SQL into Relational Algebra (3)

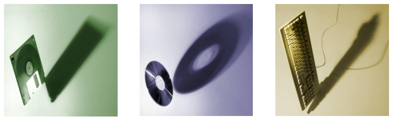
- From that example, the query will be split into two blocks:
  - The inner block:

```
( SELECT MAX (Salary)
  FROM EMPLOYEE
  WHERE Dno=5 )
```

  - The outer block:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > c
```

- The c in outer block represents returned result from the inner block.



Translating SQL into Relational Algebra (4)

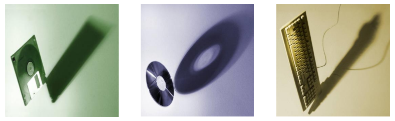
- The extended relational algebra expression of the example:
  - The inner block:

$$\Im_{MAX\,Salary}(\sigma_{Dno\,=\,5}(EMPLOYEE))$$

  - The outer block:

$$\pi_{Lname,\,Fname}(\sigma_{Salary\,>\,c}(EMPLOYEE))$$

- After translating this, then the query optimizer would choose an execution plan for each block.



Algorithms for SELECT Operations (1)

- There are many options for executing a SELECT operations (depend on the file having specific access paths and may apply only to certain types of selection condition).
- SELECT operations that we will discuss for implementing the algorithms:
  - OP1: $\sigma_{Ssn='123456789'}(EMPLOYEE)$
  - OP2: $\sigma_{Dnumber > 5}(DEPARTEMENT)$
  - OP3: $\sigma_{Dno = 5}(EMPLOYEE)$
  - OP4: $\sigma_{Dno = 5 AND Salary > 30000 AND Sex ='F'}(EMPLOYEE)$
  - OP5: $\sigma_{Essn='123456789' AND Pno = 10}(WORKS\_ON)$

Algorithms for SELECT Operations (2)

- Search Methods for Simple Selection:
  - S1 – Linear Search

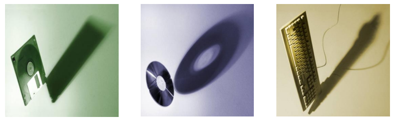
Retrieve **every record** and test whether its attribute values satisfy the selection condition. Cost = b_R block transfer + 1 seek or (b_R /2) block transfer + 1 seek (if on key attribute)

  - S2 – Binary Search

If selection condition involves an equality comparison on a key attribute on which the file is ordered then use binary search (e.g. OP1). Cost of locating the first tuple: $\lceil \log_2(b_r) \rceil * (t_T + t_S)$

  - S3 – Using a Primary Index (or Hash Key)

If the selection condition involves an equality comparison on a **key attribute** with a primary index (or hash key) (e.g. OP1). Cost = $(h_i + 1) * (t_T + t_S)$



Algorithms for SELECT Operations (3)

- Search Methods for Simple Selection:
  - S4 – Using a Primary Index to retrieve multiple records

  If the comparison operater is {=, <, >, ≤, ≥} on a key field with a primary index (e.g. OP2). Cost = $h_i * (t_T + t_S) + t_S + t_T * b$

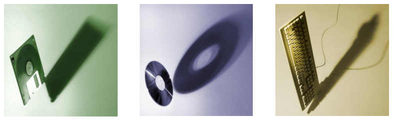
  - S5 – Using a Clustering Index to retrieve multiple records

  If the selection condition involves an equality comparison on a **nonkey attribute** with a clustering index then use the index to retrieve all the records satisfying the condition (e.g. OP3).

  - S6 – Using a secondary (B^+-Tree) Index on an equality comparison

  Retrieve a single record if the indexing field is a **key** (has unique value). Cost = $(h_i + 1) * (t_T + t_S)$

  Or to retrieve multiple records if the indexing field is a **not a key**. Cost = $(h_i + n) * (t_T + t_S)$



Algorithms for SELECT Operations (4)

- Search Methods for Simple Selection:
  - S1 applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition.
  - S4 and S6 can be used to retrieve records in a certain ranges known as **range queries**.

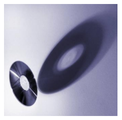
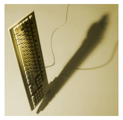
Algorithms for SELECT Operations (4)

- Search Methods for Complex Selection – If a condition of SELECT operation is a **conjunctive condition**:
  - S7 – Conjunctive selection using and individual index

  If an attribute involved in any **single simple condition** in the conjunctive condition has and access path that permits the use of one of the methods S2 to S6 to retrieve the records and then check whether each retrieved record **satisfies the remaining simple conditions**.

  - S8 – Conjunctive selection using a composite index

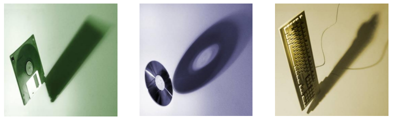
  If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields then use the index directly (e.g. OP5).

Algorithms for SELECT Operations (5)

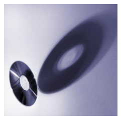
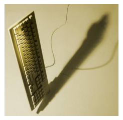
- Search Methods for Complex Selection – If a condition of SELECT operation is a **conjunctive condition**:
  - S9 – Conjunctive selection by intersection of record pointers

  If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers give the record pointers that satisfy the conjuctive condition.



Algorithms for SELECT Operations (6)

- If an access path exists on the attribute involved in the condition then the method corresponding to that access path is used other wise use the linear search approach (S1).
- Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever **more than one** of the attribute involved in the conditions have an access path.
- Query optimizer chooses the access path that retrieve **the fewer records**.
- In choosing between multiple simple conditions in a conjunctive select condition, the optimizer consider the **selectivity** of each condition.
- **Selectivity(s)**: the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation). No exact selectivites can be measure only **estimate of seletivities** kept in DBMS catalog.

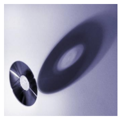
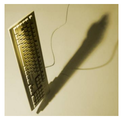



Algorithms for SELECT Operations (7)

- **Disjunctive condition** is where simple conditions are connected with OR logical connective instead of AND logical connective.
- Disjunctive condition is harder to process and optimize. Example:

OP4`: $\sigma_{Dno\ =\ 5 OR Salary\ >\ 30000 OR Sex\ =' F'}(EMPLOYEE)$

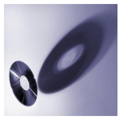
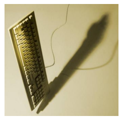
- From the example above, the result of disjunctive condition are the union of the records satisfying the individual condition, thus a little optimization can be done (only if any one of the conditions have an access path then it can be optimize, else we have to use linear search).

Algorithm for JOIN Operations (1)

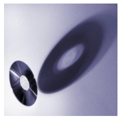
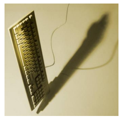
- ♦ JOIN operation is the most time-consuming operations in query processing.
- ♦ EQUIJOIN and varieties of NATURAL JOIN are the most commonly encountered in a query.
- ♦ We will explain algorithms of JOIN operation in the form of:

$$R >\!< _{A=B} S$$

Algorithm for JOIN Operations (2)

- JOIN operation that we will discuss for implementing the algorithms:
  - OP6: $CUSTOMER \bowtie_{CustLocation \,=\, BranchId} DEPOSITOR$
  - OP7: $DEPOSITOR \bowtie_{DepositorId \,=\, CustomerId} CUSTOMER$
  - Customer: 10,000 records ; 400 blocks
  - Depositor: 5000 records ; 100 blocks

Algorithm for JOIN Operations (3)

- Method for implementing joins:
  - J1 – Nested-loop join (brute force)

For each record *t* in R (outer loop), retrieve every record *s* from S (inner loop) and test whether the two records satisfy the join condition *t*[A] = *s*[B] (it's expensive).

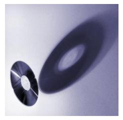
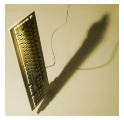
Worst case (memory only fits one block of each relation):

Estimated cost: $n_R * b_S + b_R$ block transfers

Number of seek: $n_R + b_R$

If the smaller relation fits in entirely in memory, then use that as the inner loop

Reduces cost to $b_R + b_S$ block transfers plus 2 seeks

Algorithm for JOIN Operations (4)

- Method for implementing joins:
  - J1 – Nested-loop join (brute force) (cont..)

  Cost estimate on worst case:

  Depositor as the outer loop:

  Block transfer: 5000 * 400 + 100 = 20001000

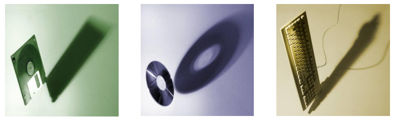
  Number of seeks: 5000 + 100 = 5100

  Customer as the outer loop:

  Block transfer: 10000 * 100 + 400 = 1000400

  Number of seeks: 10000 + 400 = 10400

  If smaller relation (Depositor) fits entirely in memory then the cost estimate: 500 block transfer



Algorithm for JOIN Operations (5)

- Method for implementing joins:
  - J2 – Single-loop join (using an access structure to retrieve the matching records)

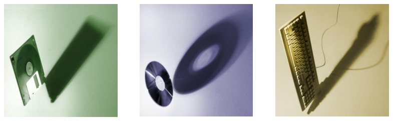
If an index (or hash key) exists for one of the two join attributes (e.g. B of S) then retrieve each record *t* in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records *s* from S that satisfy *s*[B] = *t*[A].

Worst case: buffer has space only for one page of R, and, for each tuple in R, we perform an index lookup on S

Cost of the join: $(b_R + n_R * c)\ (t_T + t_S)$

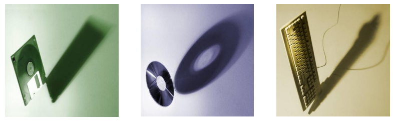
c is the cost of traversing index and fetching all matching S tuples for one tuple or R

c can be estimated as cost of a single selection on S using the join condition



Algorithm for JOIN Operations (6)

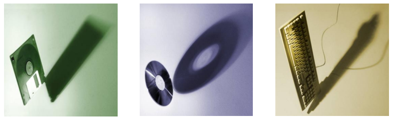
- Compute *depositor* $\bowtie$ *customer,* with *depositor* as the outer relation.
- Let *customer* have a primary B$^+$-tree index on the join attribute *customer-name,* which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5000 tuples
- Cost of single-loops join
- 100 + 5000 * 5 = 25,100 block transfers and seeks.



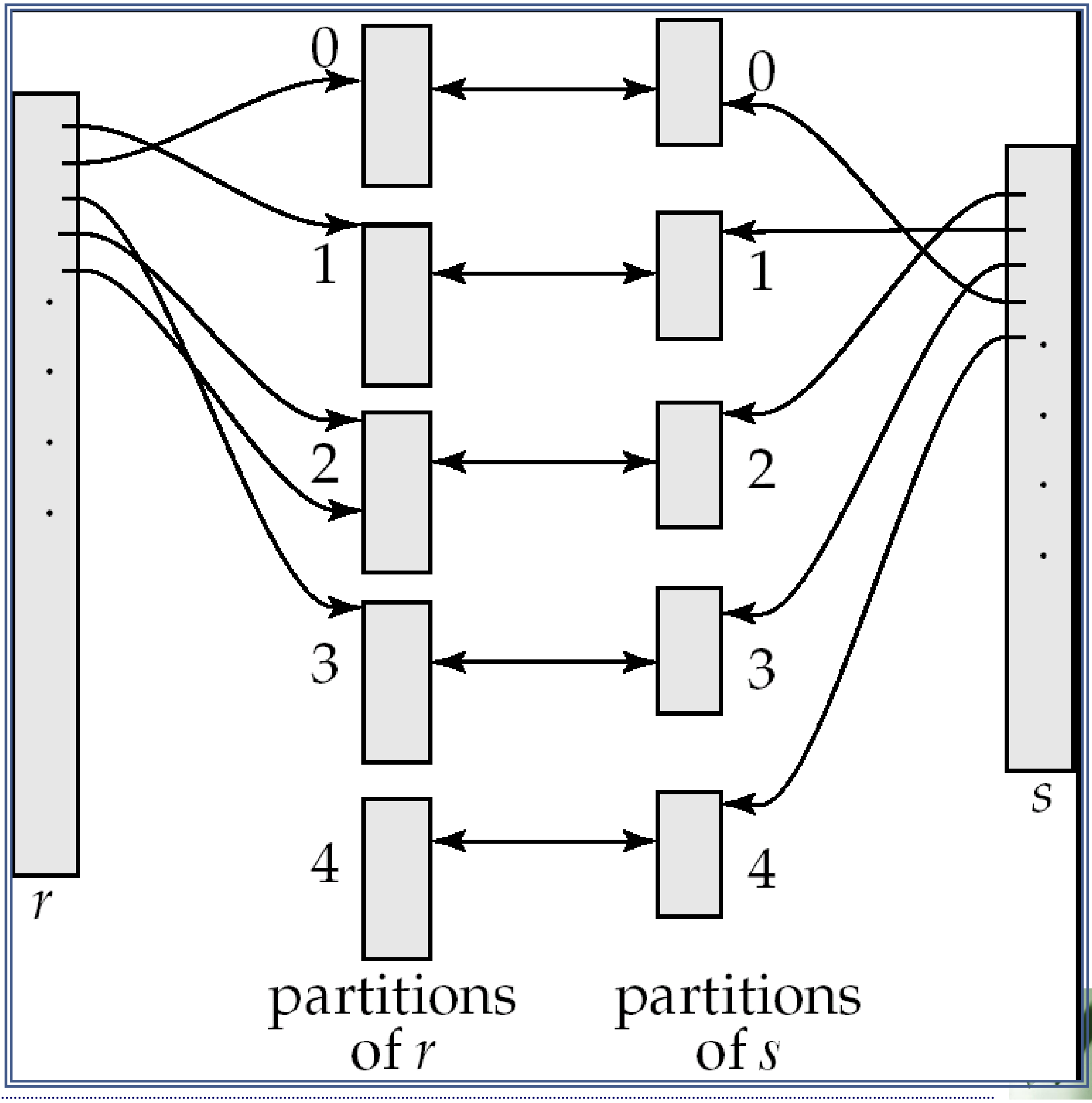
Algorithm for JOIN Operations (8)

- Method for implementing joins:
  - J4 – Hash join

  The records of file R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. First, single pass through the file with fewer records (e.g. R) then hashes its records to the hash file buckets (**partitioning phase**). Second, single pass through the other file (e.g. S) then hashes each of its records to probe the appropriate bucket and that record is combined with all matching records from R in that bucket (**probing phase**).



Algorithm for JOIN Operations (9)

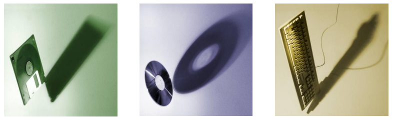


Algorithm for JOIN Operations (6)

- Those methods, in practice, are implemented by accessing whole disk blocks of a file, rather than individual records.
- Depending on the available buffer space in memory, the number of blocks read in from the file can be adjusted.

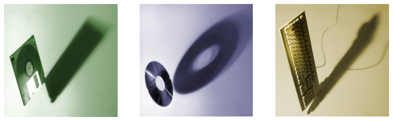
Effects of Available Buffer Space and Join Selection Factor on Join Performance (4)

- Another factor that affects performance of a join is the percentage of records in a file that will be joined with the records in the other file (this occurs particularly in single-loop method).
- That factor is called **join selection factor** of a file with respect to an equijoin condition with another file.
- Example: $DEPARTMENT \bowtie_{Mgr\_ssn = Ssn} EMPLOYEE$
  - From the previous example we knew that DEPARTMENT has 50 records.
  - We assume that 5950 records of EMPLOYEE are not managing any department, thus these will not be joined.



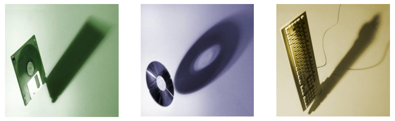
Algorithm for PROJECT and Set Operations (1)

- For PROJECT operation $\pi_{<attribute\_list>}(R)$:
  - If <attribute_list> includes a key of relation R then the result will have the same number of tuples as R but only retrieve attributes in the <attribute_list>.
  - If <attribute_list> doesn’t include a key of R, **duplicate elimination** is needed by sorting the result and eliminate duplicate tuple which appear consecutively after sorting.
    - Hashing can also be use in eliminating duplicate tuples $\rightarrow$ if a tuple were about to be insert into a bucket a file then it will be check if it already in the bucket, if it’s true (there’s a duplicate in the bucket) then do not insert.



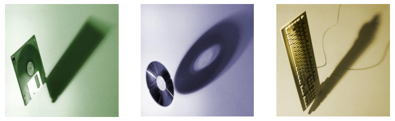
Algorithm for PROJECT and Set Operations (2)

- CARTESIAN PRODUCT (R x S) is expensive because the result of this operation is a combination of records from R and S.
  - If R has n records with j attributes and S has m records with k attributes then the result would be n*m records with j+k attributes.
- UNION, INTERSECTION, and SET DIFFERENCE are also expensive and apply only to union-compatible relations, which have the same number of attributes and the same attribute domains.
  - A customary way to implement those operation is using variations fo the **sort-merge technique**



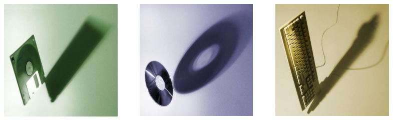
Implementing Aggregate Operations (1)

- Agregate functions (MIN, MAX, COUNT, AVERAGE, SUM) can be computed by a table scan or using an appropriate index.
- Example: SELECT MAX (Salary) FROM EMPLOYEE
  - If an (ascending) index on Salary exists for EMPLOYEE then the query optimizer can decide on using the index to search for the largest value on the **rightmost** pointer in each index node (rightmost leaf). This is more efficient than a full table scan since no actual record is retrieve. For MIN, is the same as MAX only the other way around.



Implementing Aggregate Operations (2)

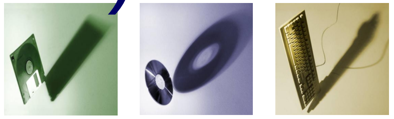
- Index could also be used on COUNT, AVERAGE, and SUM but only if it’s a **dense index**.
- The use of GROUP BY gives effect of applying aggregate operator separately to each group of tuples.
  - SELECT Dno, AVG(Salary) FROM EMPLOYEE GROUP BY Dno;
  - Partitioned into subset of tuples where each partition hash the same value for the grouping attribute.
  - The usual technique is to first use **sorting** or **hashing** on grouping attributes to partition, then algorithm computes the aggregate function.



Implementing OUTER JOIN (1)

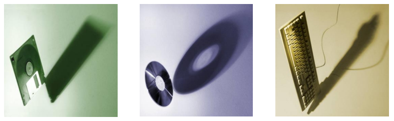
- Variation: left outer join, right outer join, full outer join.
- Example:

  -
    ```
    SELECT Lname, Fname, Dname FROM
    (EMPLOYEE LEFT OUTER JOIN
    DEPARTMENT ON Dno = Dnumber);
    ```
- The result is a table of employee with their associated department even though no associated department exist in employee (this marked as NULL)



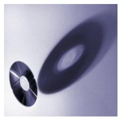
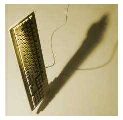
Implementing OUTER JOIN (2)

- Outer join can be computed by modifying one of the join algorithm (nested-loop or single-loop).
- To compute left outer join, use the left relation as the outer-loop or single-loop (because every tuple in the left must appear in the result). If matching tuples were found the saved in result and no matching will also save in result but padded with NULL.



Evaluation of Expression

- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed

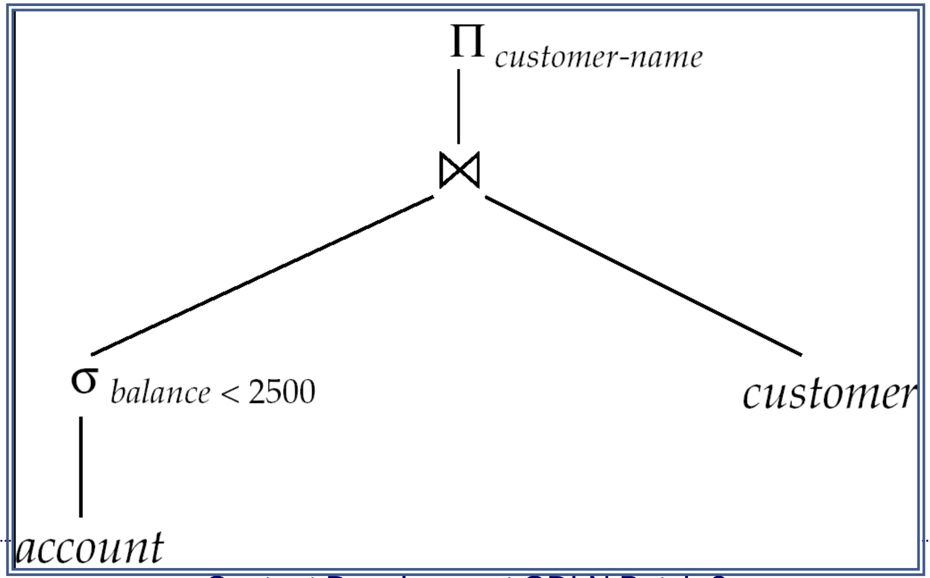



Materialization (1)

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{balance<2500}(account)$$

then compute the store its join with *customer,* and finally compute the projections on *customer-name.*



Pipelining (1)

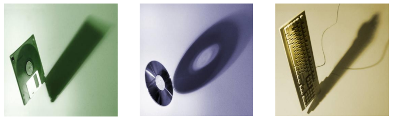
- **Pipelined evaluation :** evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.

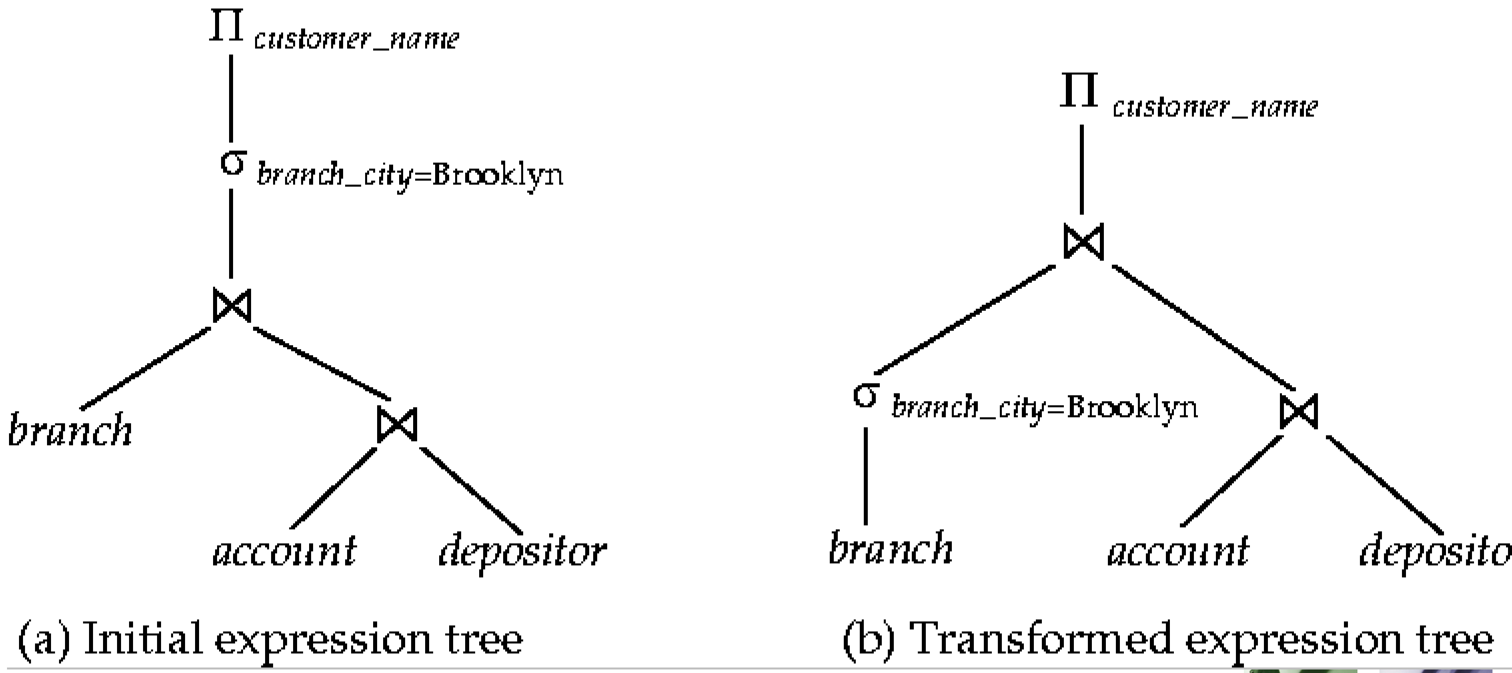
Query Optimization (Intro 1)

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
  - Statistical information about relations. Examples:
    - number of tuples,
    - number of distinct values for an attributes,
    - Etc.
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions



Query Optimization (Intro 2)

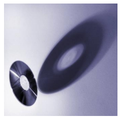
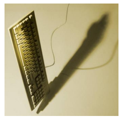
- Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples
  - although their tuples/attributes may be ordered differently.



(a) Initial expression tree (b) Transformed expression tree

Query Optimization (Intro 3)

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions using **equivalence rules**.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on **estimated cost**
- The overall process is called **cost based optimization.**

Equivalence Rules (1)

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

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

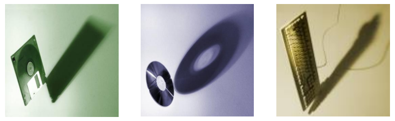
2. Selection operations are commutative.

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

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 (2)

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

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

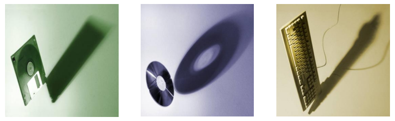
6. (a) Natural join operations are associative:

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

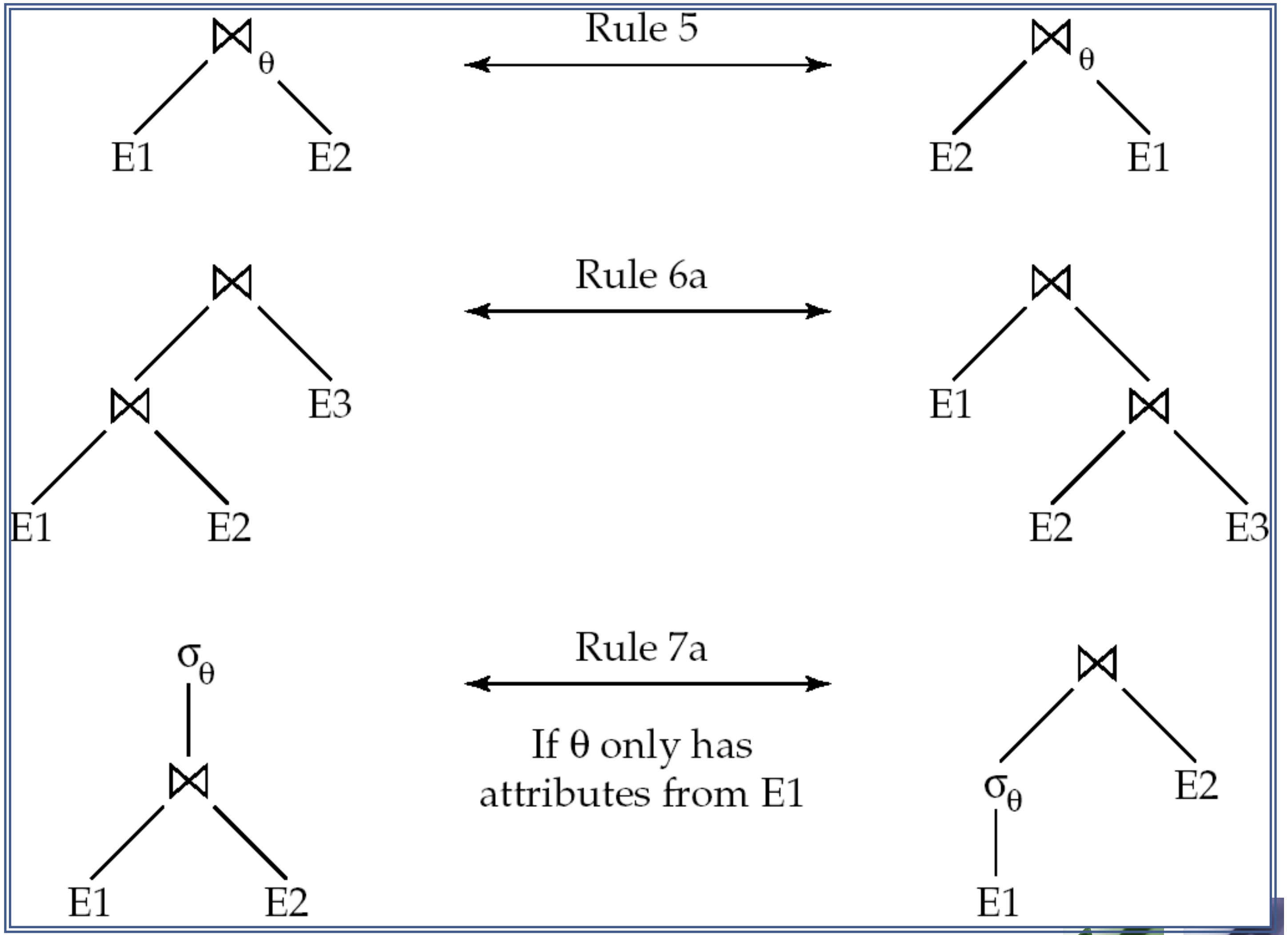
   (b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

   where θ_2 involves attributes from only E_2 and E_3.



Pictorial Depiction of Equivalence Rules



Equivalence Rules (3)

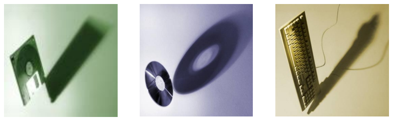
7. The selection operation distributes over the theta join operation under the following two conditions:

   (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 (4)

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

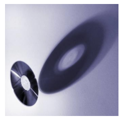
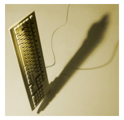
   (a) if θ involves only attributes from $L_1 \cup L_2$:

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

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_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 (5)

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 –.

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

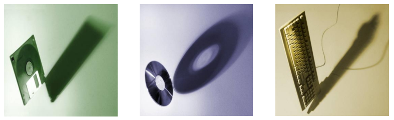
and similarly for $\cup$ and $\cap$ in place of –

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

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



Transformation Example

- 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):

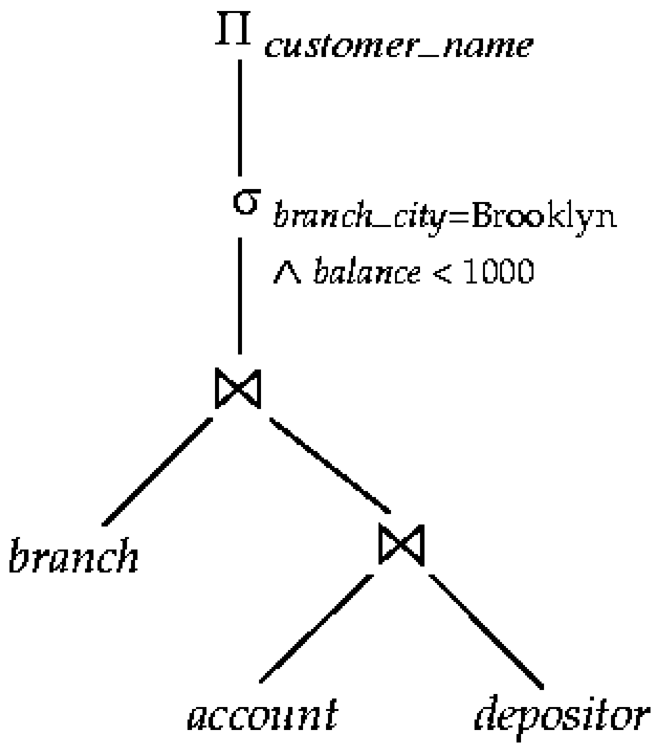
  $\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$

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

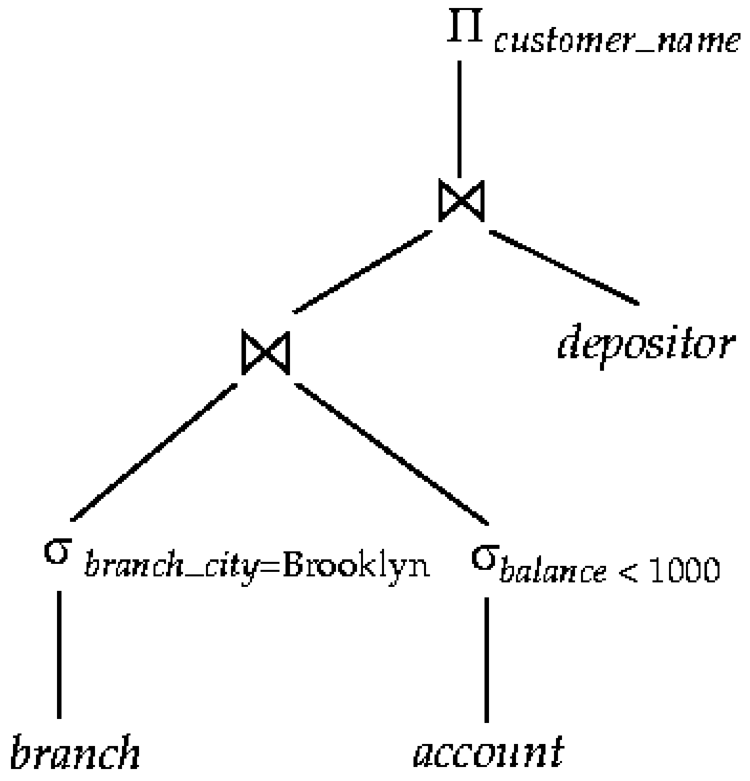
  $\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie \sigma_{balance > 1000}(account)$

- Thus a sequence of transformations can be useful

Example of Multiple Transformations



(a) Initial expression tree


$\Pi_{customer\_name}$
$\bowtie$
$\bowtie$
depositor
$\sigma_{branch\_city=\text{Brooklyn}}$
$\sigma_{balance < 1000}$
branch
account
(b) Tree after multiple transformations

Projection Operation Example

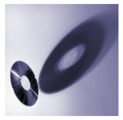
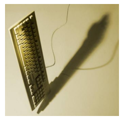
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}\ (branch) \bowtie account) \bowtie depositor)$$

- 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 (1)

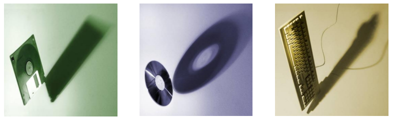
- For all relations $r_1, r_2,$ and r_3,

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

- 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.



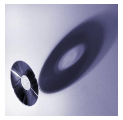
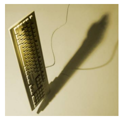
Join Ordering Example (2)

- Consider the expression

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

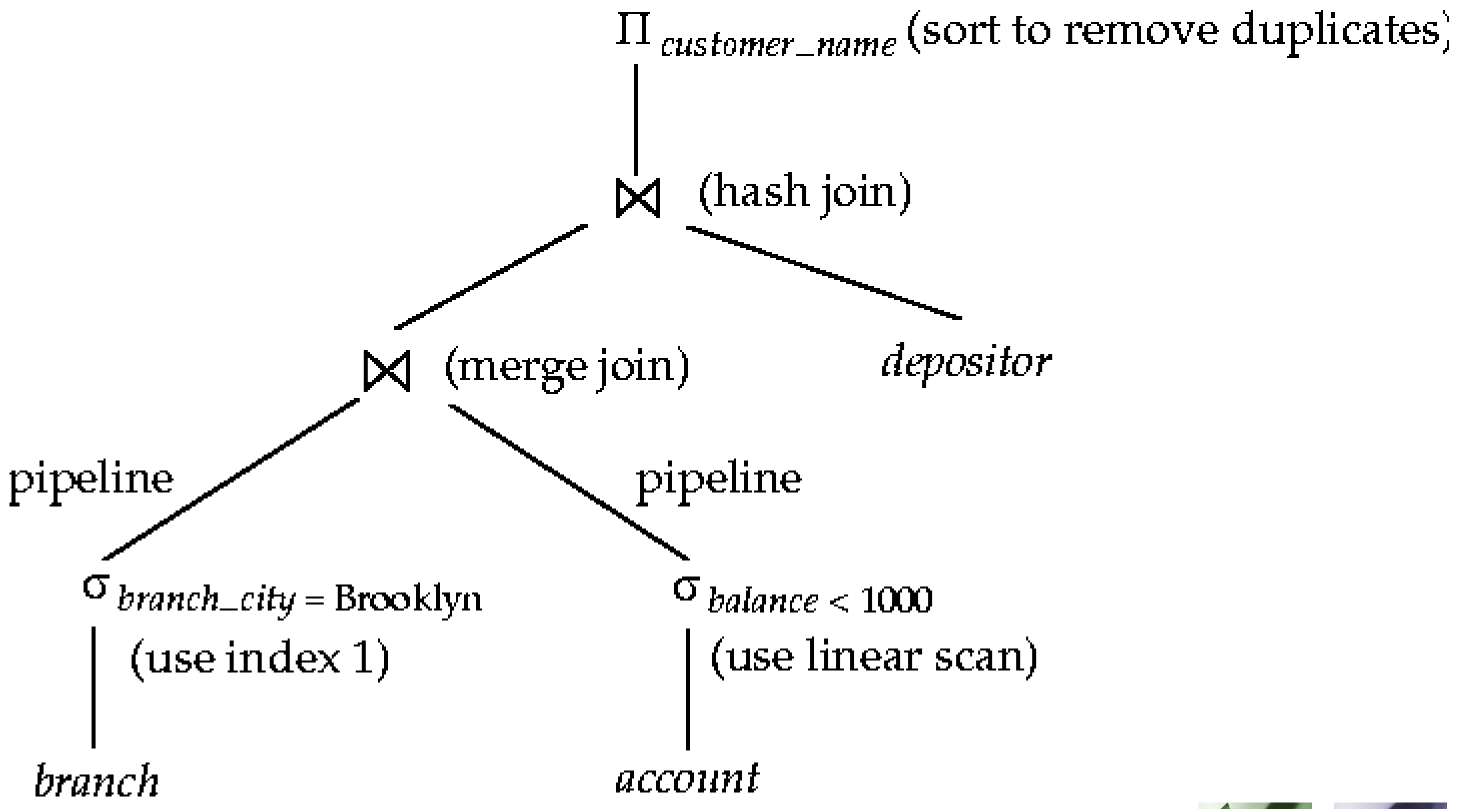
- Could compute *account* $\bowtie$ *depositor* first, and join result with
$\sigma_{branch\_city\ =\ \text{“Brooklyn”}}\ (branch)$
but *account* $\bowtie$ *depositor* is likely to be a large relation.
- Only a small fraction of the bank’s customers are likely to have accounts in branches located in Brooklyn
  - it is better to compute

$$\sigma_{branch\_city\ =\ \text{“Brooklyn”}}\ (branch) \bowtie account \text{ first.}$$

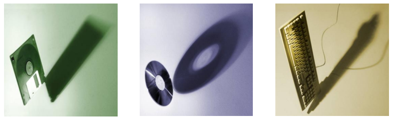
Evaluation Plan

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


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

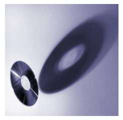
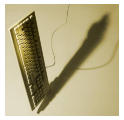
Choice of Evaluation Plans

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



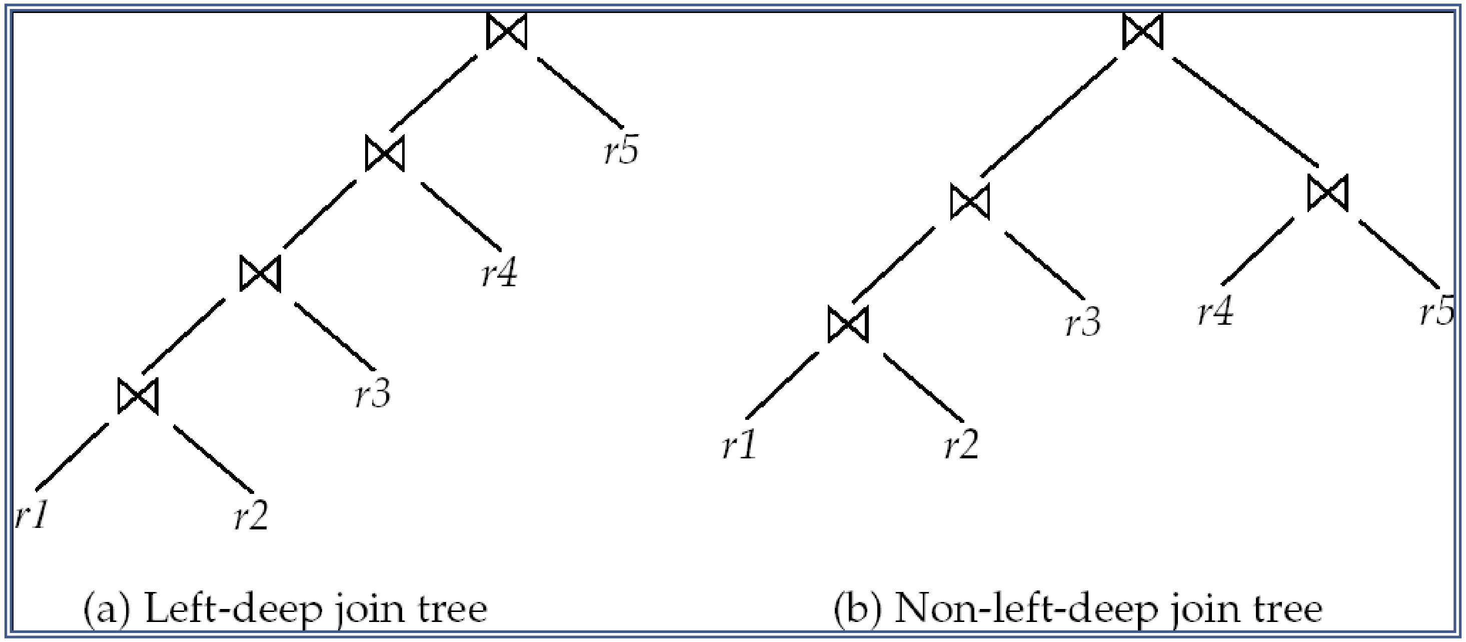
Cost-Based Optimization

- 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.

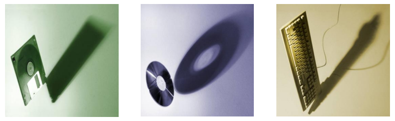
Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



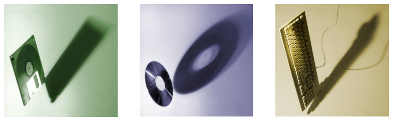
Heuristic Optimization

- 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 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 most restrictive selection and join operations before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based 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).



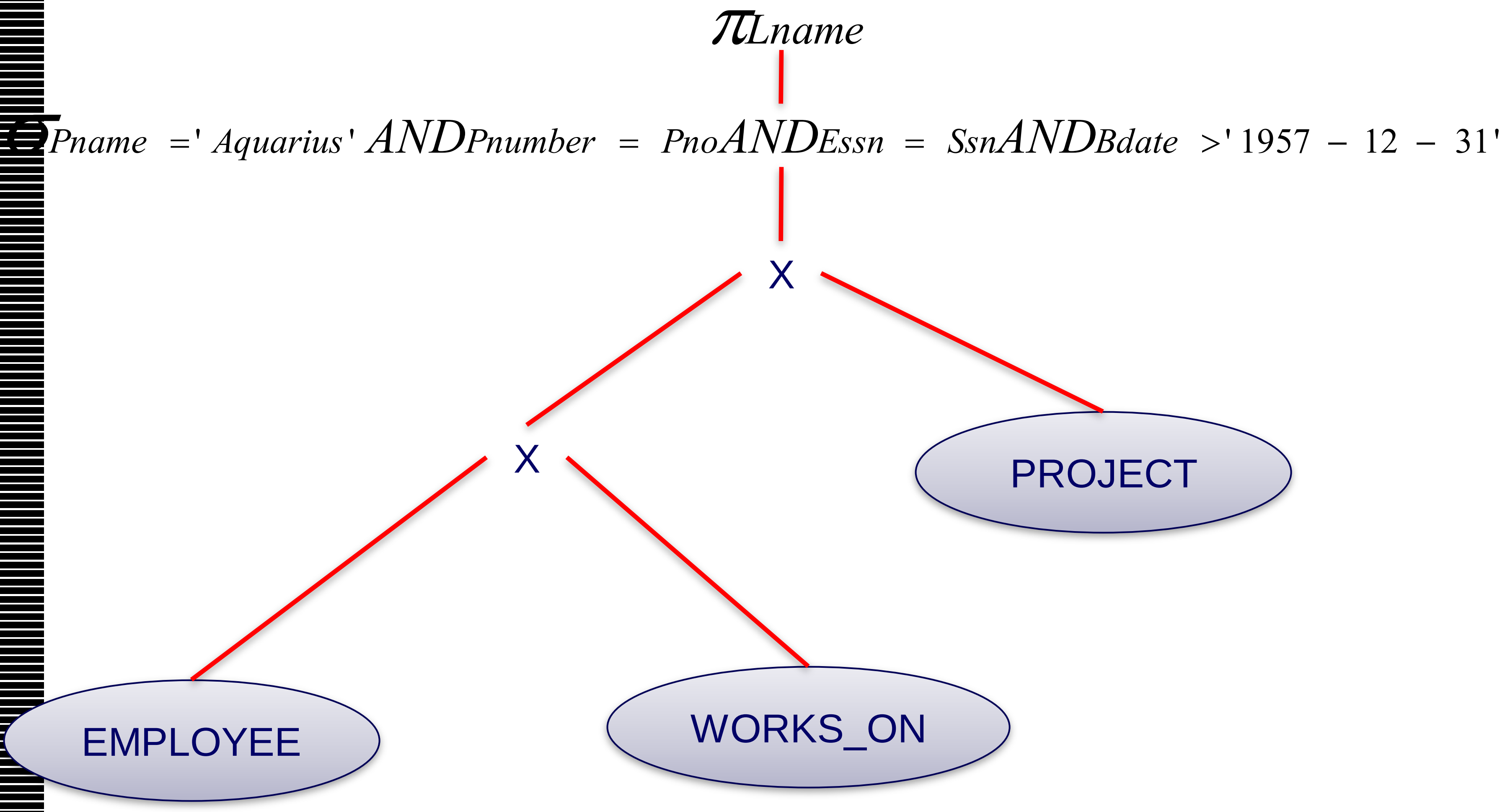
Example of Transforming Query (1)

- Q: SELECT Lname

```
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE Pname='Aquarius' AND Pnumber=Pno
      AND Essn=Ssn AND Bdate >
          '1975-12-31';
```

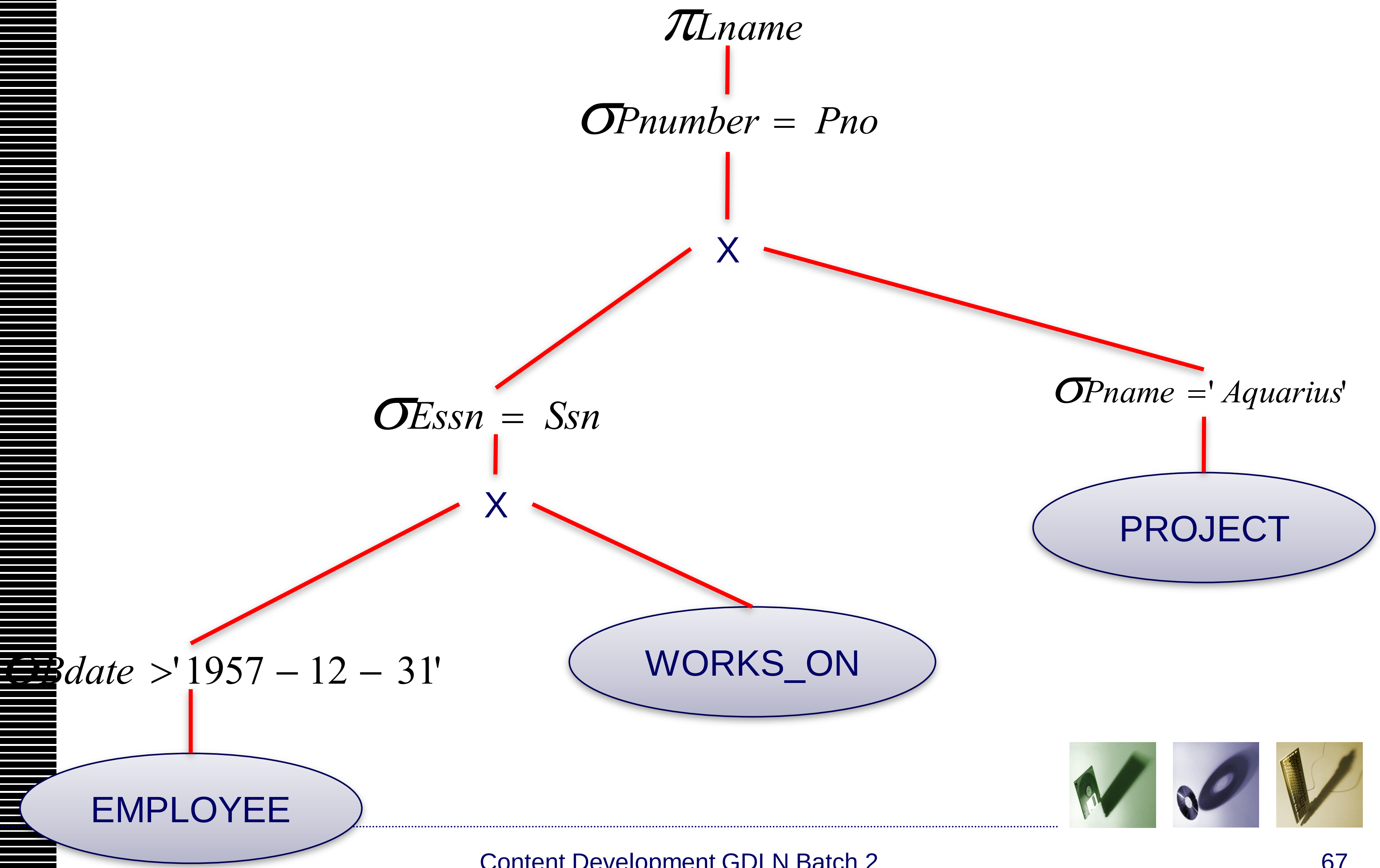
Example of Transforming Query (2)

Initial Tree of Q


π_{Lname}
$\sigma_{Pname='Aquarius' \text{ AND } Pnumber = Pno \text{ AND } Essn = Ssn \text{ AND } Bdate > '1957-12-31'}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

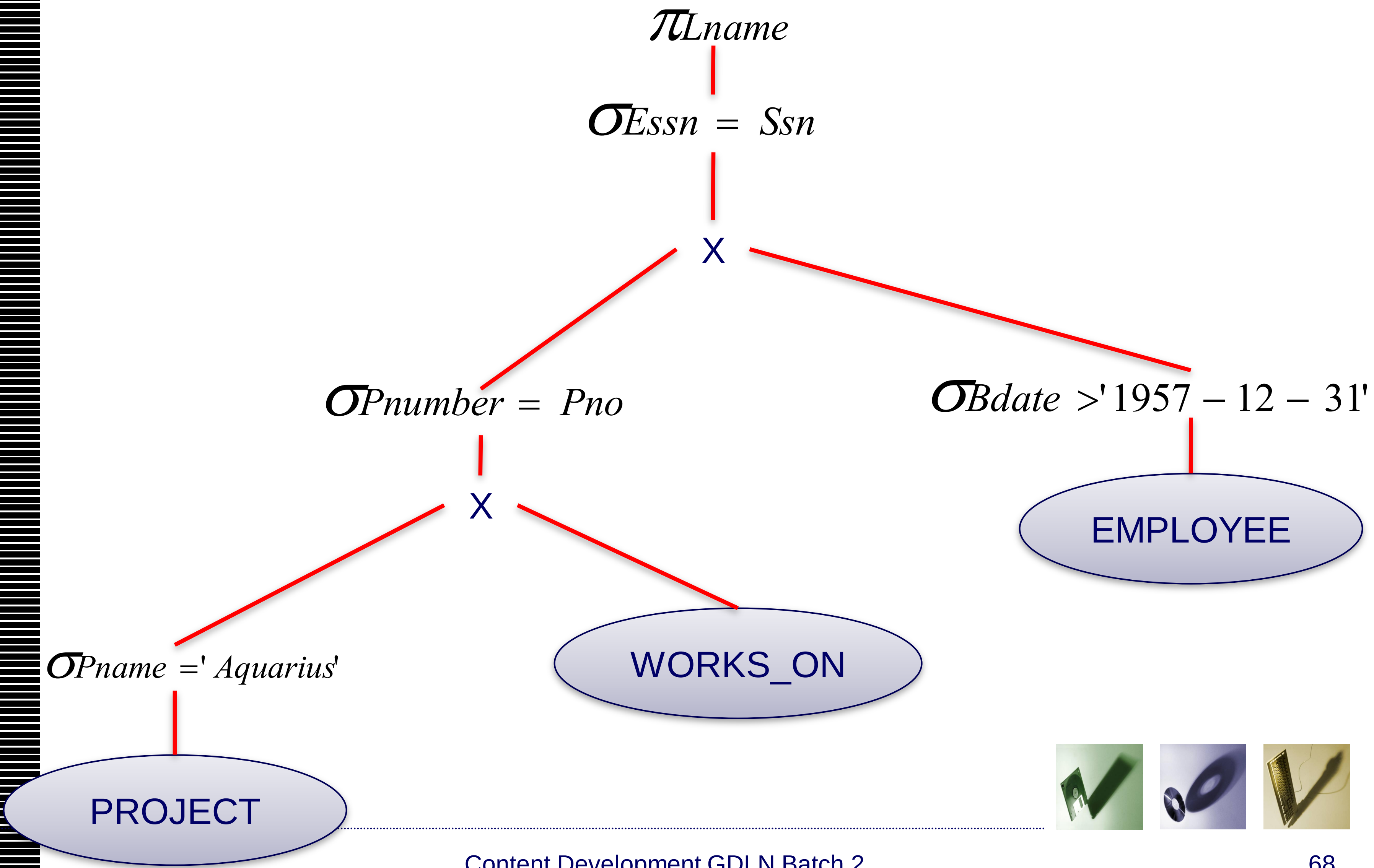
Example of Transforming Query (3)

Moving SELECT operations down the query tree



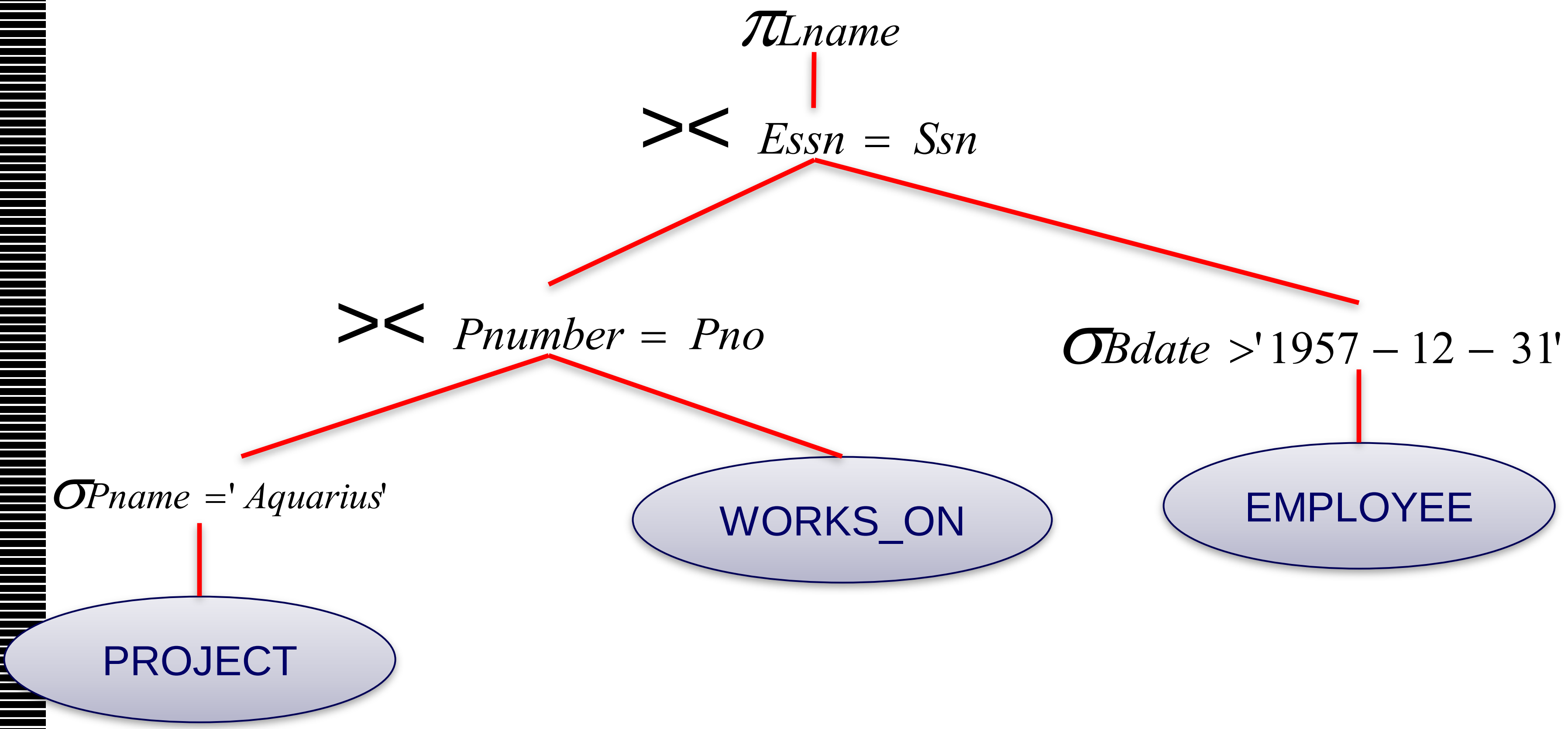
Example of Transforming Query (3)

Applying the more restrictive SELECT operation first



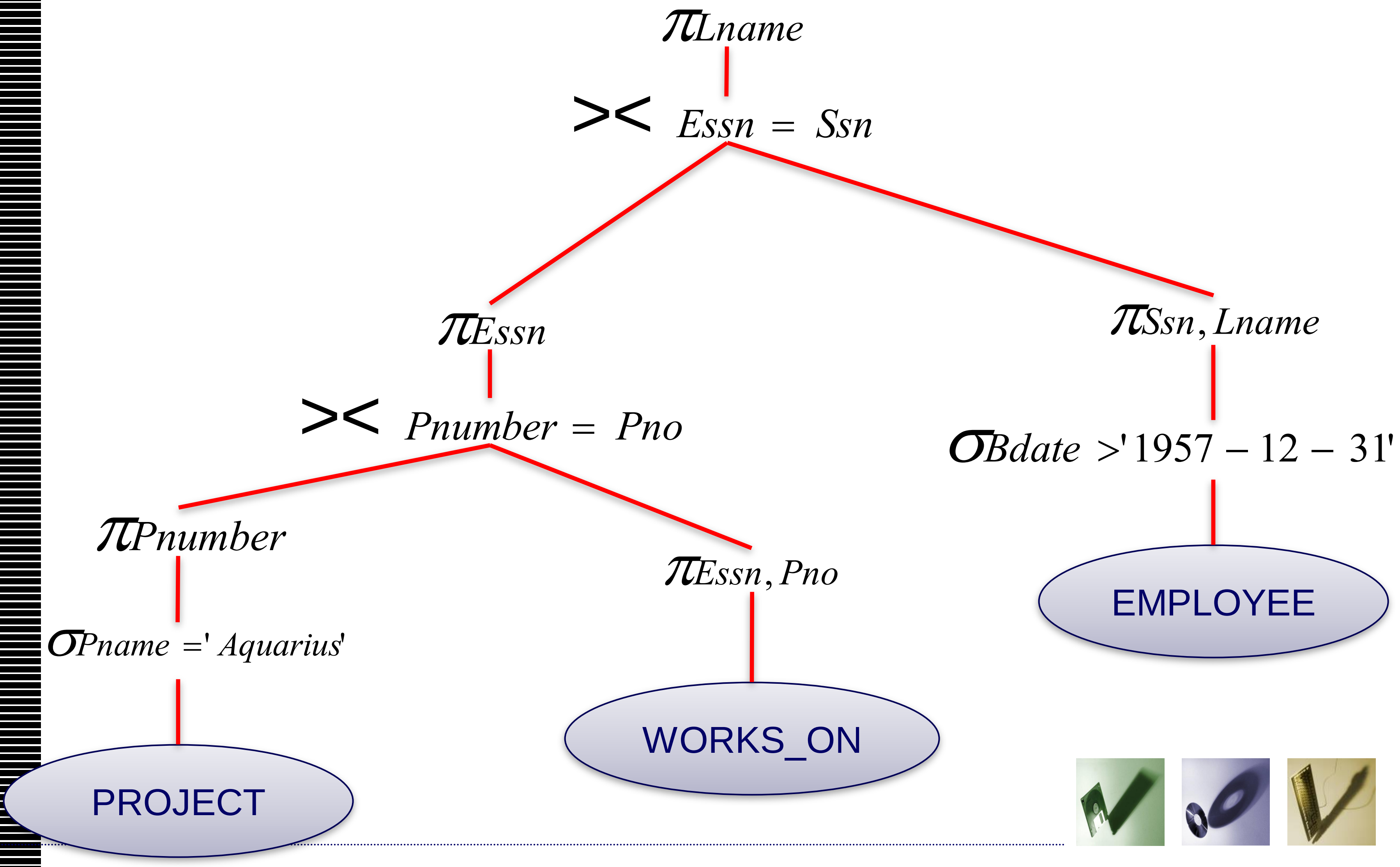
Example of Transforming Query (4)

Replacing CARTESIAN PRODUCT and SELECT with JOIN operations


π_{Lname}
$\bowtie_{Essn = Ssn}$
$\bowtie_{Pnumber = Pno}$
$\sigma_{Bdate > '1957-12-31'}$
$\sigma_{Pname = 'Aquarius'}$
WORKS_ON
EMPLOYEE
PROJECT

Example of Transforming Query (5)

Moving PROJECT operations down the query tree



π_{Lname}
$\bowtie_{Essn = Ssn}$
π_{Essn}
$\pi_{Ssn, Lname}$
$\bowtie_{Pnumber = Pno}$
$\sigma_{Bdate > '1957-12-31'}$
$\pi_{Pnumber}$
$\pi_{Essn, Pno}$
EMPLOYEE
$\sigma_{Pname = 'Aquarius'}$
WORKS_ON
PROJECT