K.D.K COLLEGE OF ENGINEERING,NAGPUR

Database Management Systems

# Unit 4

**Unit 4**

## 4.1 Query processing

**Query processing** refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

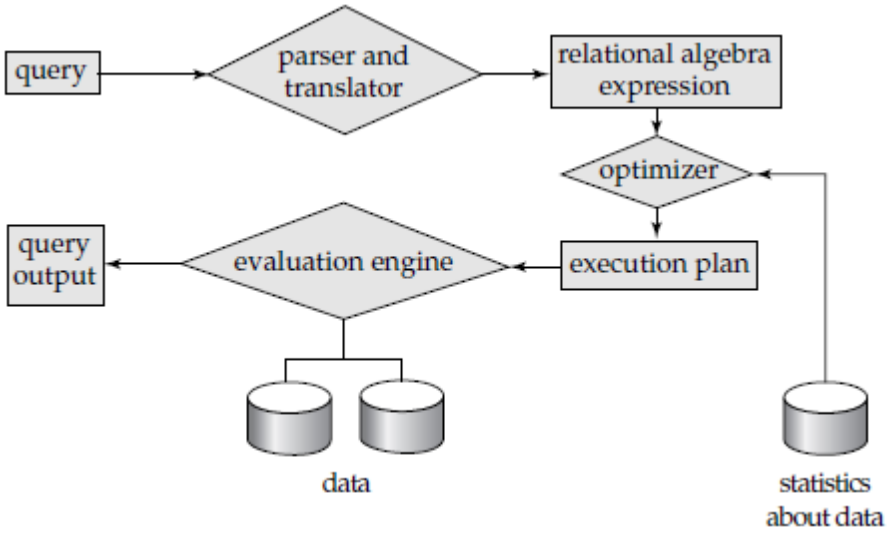



**Figure1:**Steps in query processing.

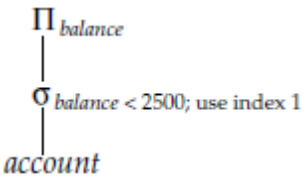



**Figure2:** A query-evaluation plan

The steps involved in processing a query appear in Figure1. The basic steps are

**1.** Parsing and translation
**2.** Optimization
**3.** Evaluation

Before query processing can begin, the system must translate the query into a usable form. Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

**select** *balance*
**from** *account*
**where** *balance* < 2500

This query can be translated into either of the following relational-algebra expressions:
• $\sigma_{balance<2500}(\Pi_{balance}(account))$
• $\Pi_{balance}(\sigma_{balance<2500}(account))$
A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations that can be used to evaluate a query is a **query execution plan** or **query-evaluation plan**. Figure 2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query. The different evaluation plans for a given query can have different costs. Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

### 4.2 Measures of Query Cost

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication. The response time for a query-evaluation plan assuming no other activity is going on on the computer, would account for all these costs, and could be used as a good measure of the cost of the plan.
In large database systems, however, disk accesses (which we measure as the number of transfers of blocks from disk) are usually themost important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. Finally, estimating the CPU time is relatively hard, compared to estimating the disk-access cost. Therefore, most people consider the disk-access cost a reasonable measure of the cost of a query-evaluation plan. We use the *number of block transfers* from disk as a measure of the actual cost.

#### 4.2.1 Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. **Basic Algorithms**
Consider a selection operation on a relation whose tuples are stored together in one file. Two scan algorithms to implement the selection operation are:

• **A1** (**linear search**). In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation. The cost of linear search, in terms of number of I/O operations, is *br*, where *br* denotes the number of blocks in the file. Selections on key attributes have an average cost of *br*/2, but still have a worst-case cost of *br*. Although it may be slower than other algorithms for implementing selection, the linear search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation.

• **A2** (**binary search**). If the file is ordered on an attribute, and the selection condition is an equality comparison on the attribute, we can use a binary search to locate records that satisfy the selection. The system performs the binary search on the blocks of the file. The number of blocks that need to be examined to find a block containing the required records is $\lceil \log_2(b_r) \rceil$ where *br*

denotes the number of blocks in the file. If the selection is on a nonkey attribute, more than one block may contain required records, and the cost of reading the extra blocks has to be added to the cost estimate.

### 4.2.2 Selections Using Indices

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. Search algorithms that use an index are referred to as **index scans**.

Search algorithms that use an index are:

• **A3** (**primary index, equality on key**). For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition. If a B+-tree is used, the cost of the operation, in terms of I/O operations, is equal to the height of the tree plus one I/O to fetch the record.

• **A4** (**primary index, equality on nonkey**). We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, *A*. The only difference from the previous case is that multiple records may need to be fetched. The cost of the operation is proportional to the height of the tree, plus the number of blocks containing records with the specified search key.

• **A5** (**secondary index, equality**). Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may get retrieved if the indexing field is not a key. In the first case, only one record is retrieved, and the cost is equal to the height of the tree plus one I/O operation to fetch the record. In the second case each record may be resident on a different block, which may result in one I/O operation per retrieved record. The cost could become even worse than that of linear search if a large number of records are retrieved.

.

### 4.2.3 Selections Involving Comparisons

We can implement the selection either by using a linear or binary search or by using indices in one of the following ways:

• **A6** (**primary index, comparison**). A primary ordered index (for example, a primary B+-tree index) can be used when the selection condition is a comparison. For comparison conditions of the form $A > v$ or $A \geq v$, a primary index on *A* can be used to direct the retrieval of tuples, as follows. For $A \geq v$, we look up the value *v* in the index to find the first tuple in the file that has a value of $A = v$. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For$A > v$, the file scan starts with the first tuple such that $A > v$.

• **A7** (**secondary index, comparison**). We can use a secondary ordered index to guide retrieval for comparison conditions involving $<, \leq, \geq$, or $>$. The lowestlevel index blocks are scanned, either from the smallest value up to *v* (for $<$ and $\leq$), or from *v* up to the maximum value (for $>$ and $\geq$).

**Join Operation**
We use the term **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$ where *A* and *B* are attributes or sets of attributes of relations *r* and *s* respectively.

We assume the following information about the two relations:
• Number of records of *customer*: *ncustomer* = 10, 000.
• Number of blocks of *customer*: *bcustomer* = 400.
• Number of records of *depositor*: *ndepositor* = 5000.
• Number of blocks of *depositor*: *bdepositor* = 100.

**4.3 Nested-Loop Join**

**for each** tuple *tr* **in** *r* **do begin**
**for each** tuple *ts* **in** *s* **do begin**

test pair (*tr*, *ts*) to see if they satisfy the join condition $\theta$ if they do, add $tr \cdot ts$ to the result.
**end**
**end**
**Figure:** Nested-loop join.

Figure shows a simple algorithm to compute the theta join $r \bowtie_\theta s$ of two relations *r* and *s*. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation *r* is called the **outer relation** and relation *s* the **inner relation** of the join, since the loop for *r* encloses the loop for *s*. The algorithm uses the notation $tr \cdot ts$, where *tr* and *ts* are tuples; $tr \cdot ts$ denotes the tuple constructed by concatenating the attribute values of tuples *tr* and *ts*. The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $nr * ns$, where *nr* denotes the number of tuples in *r*, and *ns* denotes the number of tuples in *s*. For each record in *r*, we have to perform a complete scan on *s*. In the worst case, the buffer can hold only one block of each relation, and a total of $nr * bs + br$ block accesses would be required, where *br* and *bs* denote the number of blocks containing tuples of *r* and *s* respectively. In the best case, there is enough space for both relations to fit simultaneously inmemory, so each block would have to be read only once; hence, only $br + bs$ block accesses would be required.

### 4.3.1 Block Nested-Loop Join

Figure shows **block nested-loop join**, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation.Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples .

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result.
            end
        end
    end
end
```

**Figure** Block nested-loop join

The performance of the nested-loop and block nested-loop procedures can be further improved:
• If the join attributes in a natural join or an equi-join form a key on the inner relation, then for each outer relation tuple the inner loop can terminate as soon as the first match is found.

• In the block nested-loop algorithm, instead of using disk blocks as the blocking unit for the outer relation, we can use the biggest size that can fit in memory, while leaving enough space for the buffers of the inner relation and the output. In other words, if memory has *M* blocks, we read in $M - 2$ blocks of the outer relation at a time, and when we read each block of the inner relation we join it with all the $M - 2$ blocks of the outer relation. This change reduces the number of scans of the inner relation from $\lceil b_r/(M-2) \rceil$ where *br* is the number of blocks of the outer relation. The total cost is then $\lceil b_r/(M-2) \rceil * b_s + b_r$

• We can scan the inner loop alternately forward and backward. This scanning method orders the requests for disk blocks so that the data remaining in the buffer fromthe previous scan can be reused, thus reducing the number of disk accesses needed.

• If an index is available on the inner loop's join attribute, we can replace file scans with more efficient index lookups.

### 4.3.2 Indexed Nested-Loop Join

In a nested-loop join , if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple *tr* in the outer relation *r*, the index is used to look up tuples in *s* that will satisfy the join condition with tuple *tr*.
This join method is called an **indexed nested-loop join**; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join. Looking up

tuples in *s* that will satisfy the join conditions with a given tuple *tr* is essentially a selection on *s*. For example, consider *depositor* □ *customer*. Suppose that we have a *depositor* tuple with *customer-name* "John". Then, the relevant tuples in *s* are those that satisfy the selection "*customer-name* = John". The cost of an indexed nested-loop join can be computed as follows. For each tuple in the outer relation *r*, a lookup is performed on the index for *s*, and the relevant tuples are retrieved. In the worst case, there is space in the buffer for only one page of *r* and one page of the index. Then, *br* disk accesses are needed to read relation *r*, where *br* denotes the number of blocks containing records of *r*. For each tuple in *r*, we perform an index lookup on *s*. Then, the cost of the join can be computed as *br* +*nr* ∗ *c*, where *nr* is the number of records in relation *r*, and *c* is the cost of a single selection on *s* using the join condition.

### 4.3.3 Merge Join

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                         end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ⋈ t_r to result ;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

**Figure 13** Merge join.

The **merge join** algorithm (also called the **sort–merge join** algorithm) can be used to compute natural joins and equi-joins. Let *r*(*R*) and *s*(*S*) be the relations whose natural join is to be computed, and let *R*∩*S* denote their common attributes. Figure  shows the merge join algorithm. In the algorithm, *JoinAttrs* refers to the attributes in *R* ∩ *S*, and *tr* □ *ts*, where *tr* and *ts* are tuples that have the same values for *JoinAttrs*, denotes the concatenation of the attributes of the tuples, followed by projecting out repeated attributes. The merge join algorithm associates one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read into *Ss*.

### 4.3.4 Hash Join

Like the merge join algorithm, the hash join algorithm can be used to implement natural joins and equi-joins. In the hash join algorithm, a hash function *h* is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have the same hash value on the join attributes.

We assume that
• *h* is a hash functionmapping *JoinAttrs* values to *{*0, 1, . . . , *nh}*, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
• *Hr*0 , *Hr*1, . . .,*Hrnh*
denote partitions of *r* tuples, each initially empty. Each tuple
*tr* ∈ *r* is put in partition *Hri*, where *i* = *h*(*tr*[*JoinAttrs*]).
• *Hs*0 ,*Hs*1 , ...,*Hsnh*
denote partitions of *s* tuples, each initially empty. Each tuple
*ts* ∈ *s* is put in partition *Hsi*, where *i* = *h*(*ts*[*JoinAttrs*]).
The idea behind the hash join algorithm is this: Suppose that an *r* tuple and an *s* tuple satisfy the join condition; then, they will have the same value for the join attributes. If that value is hashed to some value *i*, the *r* tuple has to be in *Hri* and the *s* tuple in *Hsi* . Therefore, *r* tuples in *Hri* need only to be compared with *s* tuples in *Hsi* ; they do not need to be compared with *s* tuples in any other partition.

```
/* Partition s */
for each tuple t_s in s do begin
   i := h(t_s[JoinAttrs]);
   H_si := H_si ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
   i := h(t_r[JoinAttrs]);
   H_ri := H_ri ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
   read H_si and build an in-memory hash index on it
   for each tuple t_r in H_ri do begin
      probe the hash index on H_si to locate all tuples t_s
         such that t_s[JoinAttrs] = t_r[JoinAttrs]
      for each matching tuple t_s in H_si do begin
         add t_r ⋈ t_s to the result
      end
   end
end
```

**Figure** Hash join

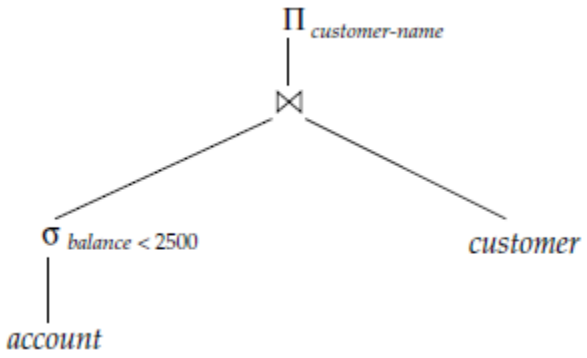
**Evaluation of Expressions**

So far, we have studied how individual relational operations are carried out. Now we consider how to evaluate an expression containing multiple operations. The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with the results of one operation passed on to the next, without the need to store a temporary relation.

### 4.4 Materialization

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**. Consider the expression

$$\Pi_{customer\text{-}name}\ (\sigma_{balance<2500}\ (account) \bowtie\ customer)$$




**Figure**Pictorial representation of an expression

If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation; the selection operation on *account*. The inputs to the lowest-level operations are relations in the database. We execute these operations by the algorithms and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our example, the inputs to the join are the *customer* relation and the temporary relation created by the selection on *account*. The join can now be evaluated, creating another temporary relation. By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join. Evaluation as just described is called **materialized evaluation**, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

### 4.5 Pipelining

We can improve query-evaluation efficiency by reducing the number of temporary files that are produced.We achieve this reduction by combining several relational operations into a *pipeline* of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called **pipelined evaluation**. Combining operations into a pipeline eliminates the cost of reading and writing temporary relations.

For example, consider the expression (Π*a*1,*a*2(*r* □ *s*)). If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection. These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result, and instead create the final result directly.

**Implementation of Pipelining**

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. In the example of Figure, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

**1.** Demand driven

**2.** Producer driven

In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent. In a **producer-driven pipeline**, operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process until all the output tuples have been generated.

### 4.6 Query optimization

**Query optimization** is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

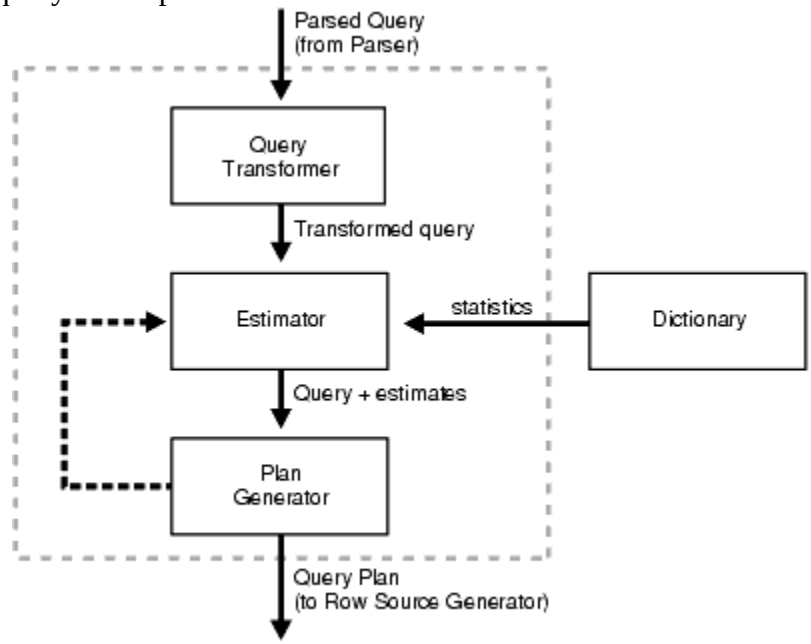



Figure:Query Optimizer

***Optimizer Operations***

Evaluation of expressions and conditions: The optimizer first evaluates expressions and conditions containing constants as fully as possible.

Statement transformation: For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement.

Choice of optimizer goals: The optimizer determines the goal of optimization.

Choice of access paths: The optimizer determines the goal of optimization.

Choice of access paths: For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain table data.

Choice of join orders: For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, and so on.

Consider the relational-algebra expression for the query "Find the names of all customers who have an account at any branch located in Brooklyn."

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

This expression constructs a large intermediate relation, *branch* □ *account* □ *depositor*. However, we are interested in only a few tuples of this relation (those pertaining to branches located in Brooklyn), and in only one of the six attributes of this relation. Since we are concerned with only those tuples in the *branch* relation that pertain to branches located in Brooklyn, we do not need to consider those tuples that do not which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure depicts the initial and transformed expressions.

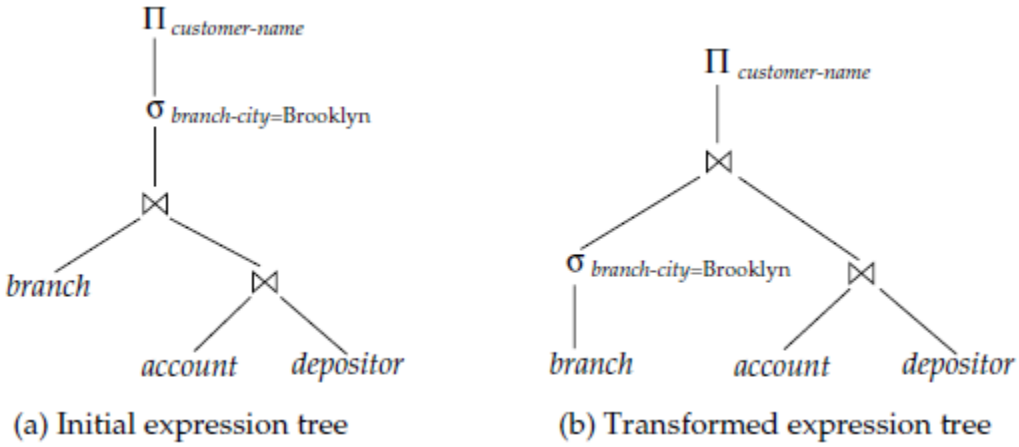



**Figure** Equivalent expressions.

have *branch-city* = "Brooklyn". By reducing the number of tuples of the *branch* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression

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

which is equivalent to our original algebra expression, but which generates smaller intermediate relations.

### 4.6.2 Cost-Based Optimization

A **cost-based optimizer** generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost. For a complex query, the number of different query plans that are equivalent to a given plan can be large.

As an illustration, consider the expression $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$ where the joins are expressed without any ordering.With $n$ = 3, there are 12 different join orderings:

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

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

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

In general, with $n$ relations, there are $(2(n-1))!/(n-1)!$ different join orders. Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic programming algorithms store results of computations and reuse them, a procedure that can reduce execution time greatly. A recursive procedure implementing the dynamic programming algorithm appears in Figure The procedure stores the evaluation plans it computes in an associative array *bestplan*, which is indexed by sets of relations. Each element of the associative array contains two components: the cost of the best plan of $S$, and the plan itself. The value of *bestplan*[$S$].*cost* is assumed to be initialized to $\infty$ if *bestplan*[$S$] has not yet been computed.

The procedure first checks if the best plan for computing the join of the given set of relations $S$ has been computed already (and stored in the associative array *bestplan*); if so it returns the already computed plan. Otherwise, the procedure tries every way of dividing $S$ into two disjoint subsets. For each division, the procedure recursively finds the best plans for each of the two subsets, and then computes the cost of the overall plan by using that division. The procedure picks the cheapest plan from among all the alternatives for dividing $S$ into two sets. The cheapest plan and its cost are stored in the array *bestplan*, and returned by the procedure. The time complexity of the procedure can be shown to be $O(3n)$.

### 3.6.4 Heuristic Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query processing can be reduced by clever optimizations, cost-based optimization is still expensive. Hence, many systems use **heuristics** to reduce the number of choices that must be made in a cost-based fashion. Some systems even choose to use only heuristics, and do not use cost-based optimization at all. An example of a heuristic rule is the following rule for transforming relationalalgebra queries:

• Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\sigma_\theta(r \bowtie s)$where the condition $\theta$ refers to only attributes in $s$. The selection can certainly be performed before the join. However, if $r$ is extremely small compared to $s$, and if there is an index on the join attributes of $s$, but no index on the attributes used by $\theta$, then it is probably a bad idea to perform the selection where the condition $\theta$ refers to only attributes in $s$.
The selection can certainly be performed before the join. However, if $r$ is extremely small compared to $s$, and if there is an index on the join attributes of $s$, but no index on the attributes used by $\theta$, then it is probably a bad idea to perform the selection.

• Perform projections early.

It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples. An example similar to the one used for the selection heuristic should convince you that this heuristic does not always reduce the cost. We now present an overview of the steps in a typical heuristic optimization algorithm.

**1.** Deconstruct conjunctive selections into a sequence of single selection operations. This step, based on equivalence rule 1, facilitates moving selection operations down the query tree.
Move selection operations down the query tree for the earliest possible execution. This step uses the commutativity and distributivity properties of the selection operation.
**2.** Determine which selection operations and join operations will produce the smallest relations—that is, will produce the relations with the least number of tuples. Using associativity of the  operation, rearrange the tree so that the leaf-node relations with these restrictive selections are executed first. This step considers the selectivity of a selection or join condition.
This step relies on the associativity of binary operations
**3 .** Replace with join operations those Cartesian product operations that are followed by a selection condition.
**4.** Replace with join operations those Cartesian product operations that are followed by a selection condition.
**5.** Identify those subtrees whose operations can be pipelined, and execute them using pipelining.
In summary, the heuristics listed here reorder an initial query-tree representation in such a way that the operations that reduce the size of intermediate results are applied first; early selection reduces the number of tuples, and early projection reduces the number of attributes. The **access-plan –selection** phase of a heuristic optimizer chooses the most efficient strategy for each operation.