14.3 Transformation of Relational Expressions

So far, we have studied algorithms to evaluate extended relational-algebra operations, and have estimated their costs. As mentioned at the start of this chapter, a query can be expressed in several different ways, with different costs of evaluation. In this section, rather than take the relational expression as given, we consider alternative, equivalent 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. (Recall that a legal database instance is one that satisfies all the integrity constraints specified in the database schema.) Note that the order of the tuples is irrelevant; the two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

In SQL, the inputs and outputs are multisets of tuples, and a multiset version of the relational algebra is used for evaluating SQL queries. Two expressions in the *multiset* version of the relational algebra are said to be equivalent if on every legal database the two expressions generate the same multiset of tuples. The discussion in this chapter is based on the relational algebra. We leave extensions to the multiset version of the relational algebra to you as exercises.

14.3.1 Equivalence Rules

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa—that is we can replace an expression of the second form by an expression of the first form—since the two expressions would generate the same result on any valid database. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

We now list a number of general equivalence rules on relational-algebra expressions. Some of the equivalences listed appear in Figure 14.2. We use $\theta, \theta_1, \theta_2$, and so on to denote predicates, L_1, L_2, L_3, and so on to denote lists of attributes, and E, E_1, E_2, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

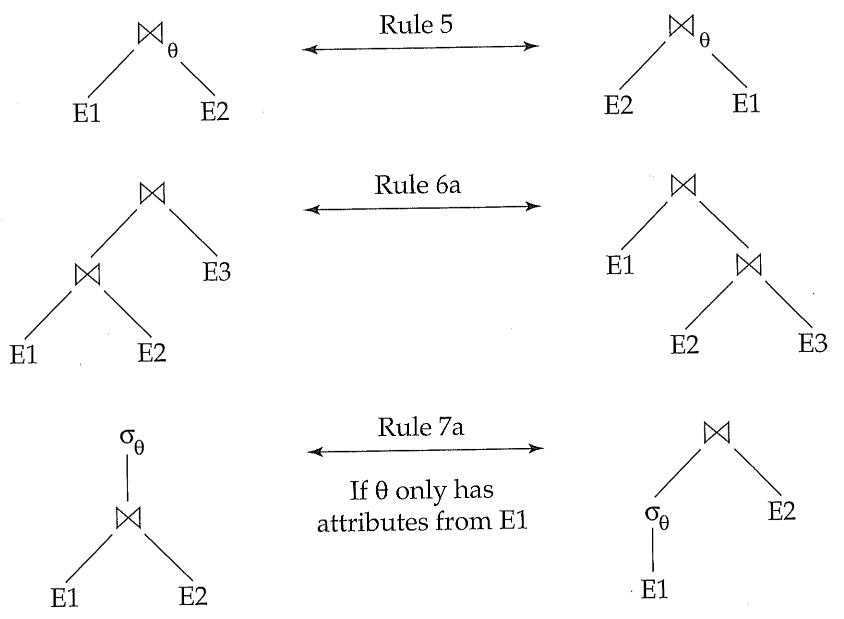

Figure 14.2 Pictorial representation of equivalences.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

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

2. Selection operations are **commutative**.

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

3. Only the final operations in a sequence of projection operations are needed, the others can be omitted. This transformation can also be referred to as a cascade of Π.

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

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   This expression is just the definition of the theta join.
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \;=\; E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations are commutative.

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

Actually, the order of attributes differs between the left-hand side and right-hand side, so the equivalence does not hold if the order of attributes is taken into account. A projection operation can be added to one of the sides of the equivalence to appropriately reorder attributes, but for simplicity we omit the projection and ignore the attribute order in most of our examples.

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

6. a. Natural-join operations are **associative**.

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

b. Theta joins are associative in the following manner:

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

where θ_2 involves attributes from only E_2 and E_3. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

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

a. It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the expressions (say, E_1) being joined.

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

b. It distributes when selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

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

8. The projection operation distributes over the theta-join operation under the following conditions.

a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

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

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 $L_1 \cup L_2$. Then,

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

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 the union, intersection, and set–difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

Similarly, the preceding equivalence, with $-$ replaced with either $\cup$ or $\cap$, also holds. Further,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

The preceding equivalence, with $-$ replaced by $\cap$, also holds, but does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation.

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

This is only a partial list of equivalences. More equivalences involving extended relational operators, such as the outer join and aggregation, are discussed in the exercises.

14.3.2 Examples of Transformations

We now illustrate the use of the equivalence rules. We use our bank example with the relation schemas:

Branch-schema = (*branch-name, branch-city, assets*)
Account-schema = (*account-number, branch-name, balance*)
Depositor-schema = (*customer-name, account-number*)

The relations *branch*, *account*, and *depositor* are instances of these schemas.

In our example in Section 14.1, the expression

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

was transformed into the following 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 generates smaller intermediate relations. We can carry out this transformation by using rule 7.a. Remember that the rule merely says that the two expressions are equivalent; it does not say that one is better than the other.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to customers who have a balance over \$1000. The new relational-algebra query is

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

We cannot apply the selection predicate directly to the *branch* relation, since the predicate involves attributes of both the *branch* and *account* relation. However, we can first apply rule 6.a (associativity of natural join) to transform the join *branch* ⋈ (*account* ⋈ *depositor*) into (*branch* ⋈ *account*) ⋈ *depositor*:

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city\,=\,\text{“Brooklyn”}\ \wedge\ balance\ >1000}\ ((branch \bowtie account) \bowtie depositor))$$

Then, using rule 7.a, we can rewrite our query as

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city\,=\,\text{“Brooklyn”}\wedge\ balance>1000}\ (branch \bowtie account)) \bowtie depositor)$$

Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

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

Both of the preceding expressions select tuples with *branch-city* = “Brooklyn” and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the “perform selections early” rule, resulting in the subexpression

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

Figure 14.3 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a

A set of equivalence rules is said to be **minimal** if no rule can be derived from any combination of the others. The preceding example illustrates that the set of equivalence rules in Section 14.3.1 is not minimal. An expression equivalent to the original expression may be generated in different ways; the number of different ways of generating an expression increases when we use a nonminimal set of equivalence rules. Query optimizers therefore use minimal sets of equivalence rules.

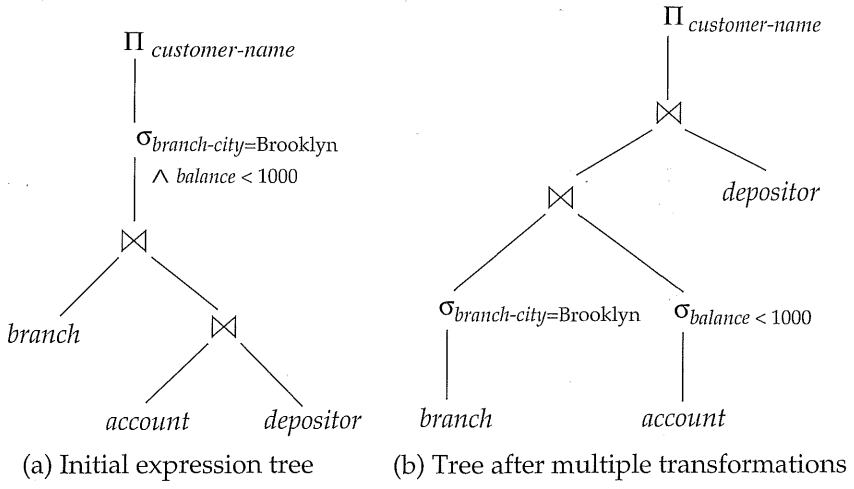

Figure 14.3 Multiple transformations.

Now consider the following form of our example query:

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

When we compute the subexpression

$$(\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie account)$$

we obtain a relation whose schema is

$$(branch\text{-}name, branch\text{-}city, assets, account\text{-}number, balance)$$

We can eliminate several attributes from the schema, by pushing projections based on equivalence rules 8.a and 8.b. The only attributes that we must retain are those that either appear in the result of the query or are needed to process subsequent operations. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, we reduce the size of the intermediate result. In our example, the only attribute we need from the join of *branch* and *account* is *account-number*. Therefore, we can modify the expression to

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

The projection $\Pi_{account\text{-}number}$ reduces the size of the intermediate join results.

14.3.3 Join Ordering

A good ordering of join operations is important for reducing the size of temporary results; hence, most query optimizers pay a lot of attention to the join order. As mentioned in Chapter 3 and in equivalence rule 6.a, the natural-join operation is associative. Thus, for all relations r_1, r_2, and r_3,

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

Although these expressions are equivalent, the costs of computing them may differ. Consider again the expression

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

We could choose to compute *account* ⋈ *depositor* first, and then to join the result with

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)$$

However, *account* ⋈ *depositor* is likely to be a large relation, since it contains one tuple for every account. In contrast,

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie account$$

is probably a small relation. To see that it is, we note that, since the bank has a large number of widely distributed branches, it is likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn. Thus, the preceding expression results in one tuple for each account held by a resident of Brooklyn. Therefore, the temporary relation that we must store is smaller than it would have been had we computed *account* ⋈ *depositor* first.

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations r_1 and r_2,

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

That is, natural join is commutative (equivalence rule 5).

Using the associativity and commutativity of the natural join (rules 5 and 6), we can consider rewriting our relational-algebra expression as

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

That is, we could compute

$$\left(\sigma_{branch\text{-}city = \text{"Brooklyn"}} (branch)\right) \bowtie depositor$$

first, and, after that, join the result with *account*. Note, however, that there are no attributes in common between *Branch-schema* and *Depositor-schema*, so the join is just a Cartesian product. If there are b branches in Brooklyn and d tuples in the *depositor* relation, this Cartesian product generates $b * d$ tuples, one for every possible pair of depositor tuple and branches (without regard for whether the account in *depositor* is maintained at the branch). Thus, it appears that this Cartesian product will produce a large temporary relation. As a result, we would reject this strategy. However, if the user had entered the preceding expression, we could use the associativity and commutativity of the natural join to transform this expression to the more efficient expression that we used earlier.

14.3.4 Enumeration of Equivalent Expressions

Query optimizers use equivalence rules to systematically generate expressions equivalent to the given query expression. Conceptually, the process proceeds as follows. Given an expression, if any subexpression matches one side of an equivalence rule, the optimizer generates a new expression where the subexpression is transformed to match the other side of the rule. This process continues until no more new expressions can be generated.

The preceding process is costly both in space and in time. Here is how the space requirement can be reduced: If we generate an expression E_1 from an expression E_2 by using an equivalence rule, then E_1 and E_2 are similar in structure, and have subexpressions that are identical. Expression-representation techniques that allow both expressions to point to shared subexpressions can reduce the space requirement significantly, and many query optimizers use them.

Moreover, it is not always necessary to generate every expression that can be generated with the equivalence rules. If an optimizer takes cost estimates of evaluation into account, it may be able to avoid examining some of the expressions, as we shall see in Section 14.4. We can reduce the time required for optimization by using techniques such as these.

14.4 Choice of Evaluation Plans

Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms. An evaluation plan is therefore needed to define exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated. Figure 14.4 illustrates one possible evaluation plan for the expression from Figure 14.3. As we have seen, several different algorithms can be used for each relational operation, giving rise to alternative evaluation plans. Further, decisions about pipelining have to be made. In the figure, the edges from the selection operations to the merge join operation are marked as pipelined; pipelining is feasible if the selection operations generate their output sorted on the join attributes. They would do so if the indices on *branch* and *account* store records with equal values for the index attributes sorted by *branch-name*.

14.4.1 Interaction of Evaluation Techniques

One way to choose an evaluation plan for a query expression is simply to choose for each operation the cheapest algorithm for evaluating it. We can choose any ordering of the operations that ensures that operations lower in the tree are executed before operations higher in the tree.

However, choosing the cheapest algorithm for each operation independently is not necessarily a good idea. Although a merge join at a given level may be costlier than a hash join, it may provide a sorted output that makes evaluating a later operation (such as duplicate elimination, intersection, or another merge join) cheaper. Similarly, a nested-loop join with indexing may provide opportunities for pipelining the results to the next operation, and thus may be useful even if it is not the cheapest way of

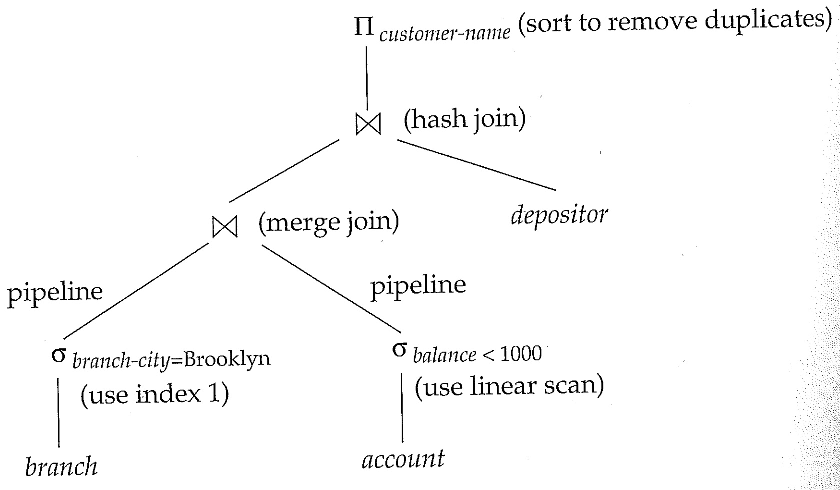
$\Pi_{customer\text{-}name}$ (sort to remove duplicates)
$\bowtie$ (hash join)
$\bowtie$ (merge join)
depositor
pipeline
pipeline
$\sigma_{branch\text{-}city=\text{Brooklyn}}$
(use index 1)
$\sigma_{balance < 1000}$
(use linear scan)
branch
account

Figure 14.4 An evaluation plan.

performing the join. To choose the best overall algorithm, we must consider even nonoptimal algorithms for individual operations.

Thus, in addition to considering alternative expressions for a query, we must also consider alternative algorithms for each operation in an expression. We can use rules much like the equivalence rules to define what algorithms can be used for each operation, and whether its result can be pipelined or must be materialized. We can use these rules to generate all the query-evaluation plans for a given expression.

Given an evaluation plan, we can estimate its cost using statistics estimated by the techniques in Section 14.2 coupled with cost estimates for various algorithms and evaluation methods described in Chapter 13. That still leaves the problem of choosing the best evaluation plan for a query. There are two broad approaches: The first searches all the plans, and chooses the best plan in a cost-based fashion. The second uses heuristics to choose a plan. We discuss these approaches next. Practical query optimizers incorporate elements of both approaches.