Chapter 11

QUERY MODIFICATION

The query processor is a central component in any database system. The job of the query processor is to take a query expressed in the system's query language, analyze it, and perform the file accesses and computations necessary to evaluate the query. Alternatively, it may generate code to perform those accesses and computations, rather than evaluating the query itself. There is generally a package of file management routines available to the query processor for performing the actual file accesses. As part of its analysis, the query processor might modify the query, usually for reasons of computational efficiency.

If the query language is based on tuple or domain calculus, the first modification that generally takes place is the translation of the query into relational algebra (although the algebraic expression may be present only implicitly in some internal representation of the query). The tuple and domain calculi are basically non-procedural query systems: they express *what* the value of a query should be, but do not express *how* to compute the value. We do have methods to evaluate safe tuple and domain calculus expressions directly, but if the formulas involved contain quantifiers, those methods are computationally prohibitive. Algebraic expressions, on the other hand, can be evaluated directly if procedures exist for each of the operators involved. Thus, translation of a query to an algebraic expression is a means to specify how the value of the query should be computed. Note that the translations used in the theorems of Chapter 10 are not the ones used in query processors. Query languages based on tuple or domain calculus usually represent only some restricted subset of the possible tuple or domain calculus expressions. The restricted form of the expressions usually allows more direct translation schemes that produce fairly succinct algebraic expressions.

The next modification the query processor is likely to make is substitution for virtual relations. Relational systems often support two types of relations: *base relations*, which are physically stored, and *virtual relations*, which are defined in terms of the base relations. Virtual relations, singly or collectively, are sometimes called *views*. The virtual relations are not stored, but the users

of the system may use them to formulate queries, along with the base relations. During query processing, occurrences of virtual relations within expressions must be replaced by expressions for those relations in terms of the base relations. The result is an equivalent expression involving only base relations.

Example 11.1 In our little airline, pilots are periodically tested by examiners and rated. The information on these tests is kept in three relations,

rp(P# PN BD)
re(E# EN)
rt(P# E# DT RG),

where P#, PN, BD, E#, EN, DT, and RG stand for pilot number, pilot name, based, examiner number, examiner name, date, and rating. We might have a virtual relation *low*, which gives information on pilots with low ratings, defined as

$$low = rp \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG} \leq 6.5}(rt)).$$

An expression such as

$$\sigma_{\text{PN} = \text{Jacobs}}(low)$$

would be modified to

$$\sigma_{\text{PN} = \text{Jacobs}}(rp \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG} \leq 6.5}(rt))$$

before evaluation.

Some relational systems, such as INGRES, enforce integrity constraints and security through query modification.

Example 11.2 For the database of Example 11.1, suppose we wanted to grant only limited access to the rp relation to someone. Say we only allowed him to access tuples for pilots based at JFK or Atlanta. In INGRES, it is possible to stipulate that each time this person uses rp, the selection

$$\sigma_{\text{BD} = \text{JFK} \vee \text{BD} = \text{Atlanta}}$$

is applied to it.

Modifications are also made because not every algebraic operator may be supported, or some may be supported only in limited form. Some systems support only joins involving a single attribute in each relation. Joins involving multiple attributes in each relation must be converted to a single-attribute join followed by a series of selections. Some systems support only selections involving the conjunction and negation of comparisons. Selections with disjunctions are converted to the union of selections of the restricted type for evaluation.

A large class of modifications are lumped together under the heading of *query optimization*. The purpose of such modifications is to find an expression that is equivalent to a given expression, but whose evaluation will be more time- or space-efficient. "Improvement" might be a better term than "optimization," for it is seldom possible to efficiently find an equivalent expression that minimizes the time or space required for evaluation.

Some optimizations are merely simplifications to remove redundant operations, or to combine two operations into one.

Example 11.3 Using the relations in Example 11.1, if we start with the expression

$$\pi_{\text{PN}}(rp \bowtie low),$$

substitution gives us

$$\pi_{\text{PN}}(rp \bowtie rp \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG} \leq 6.5}(rt))),$$

which simplifies to

$$\pi_{\text{PN}}(rp \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG} \leq 6.5}(rt))),$$

since $rp \bowtie rp = rp$.
The expression

$$\pi_{\text{P\#}}(\pi_{\text{P\# DT DG}}(rt))$$

can have its projections combined to get

$$\pi_{\text{P\#}}(rt).$$

Other optimizations are made to take advantage of the form of the operations that are supported by the query processor.

Example 11.4 Suppose there is a "restrict" operation available that performs a selection followed by a projection. The expression

$$\sigma_{\text{DT}=7\ \text{Jun}}(\pi_{\text{P\# E\# DT}}(rt)),$$

for relation rt of Example 11.1, requires two invocations of restrict, since the selection follows the projection. However, the equivalent expression

$$\pi_{\text{P\# E\# DT}}(\sigma_{\text{DT}=7\ \text{Jun}}(rt))$$

requires only one invocation, since the projection follows the selection.

The goal of all optimization is efficiency of execution. Even if the original query is seemingly written to execute efficiently, translation to another query system, virtual relation substitution and other modifications can yield a result that is not so efficient. Joins are a critical operation for efficiency, since the time required for a join is often proportional to the product of the sizes of the operands. Query modifications that reduce the size of join operands will improve time-efficiency.

Example 11.5 Consider the expression

$$\sigma_{\text{PN}=\text{Jacobs}}(rp \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG}\leq 6.5}(rt))$$

from Example 11.1, which was obtained by virtual relation substitution. Presumably, $\sigma_{\text{PN}=\text{Jacobs}}(rp)$ has only a single tuple, hence is much smaller than rp. To reduce the time needed for the join, the equivalent expression,

$$\sigma_{\text{PN}=\text{Jacobs}}(rp) \bowtie \pi_{\text{P\# DT RG}}(\sigma_{\text{RG}\leq 6.5}(rt))$$

should be used.

Sometimes constraints on a database can be helpful in optimizing queries.

Example 11.6 Suppose that for the relations rp and rt in Example 11.1 every pilot number that appears in rp must also appear in rt. The expression

$$\pi_{\text{PN}}(rp \bowtie rt)$$

can be replaced by simply $\pi_{\text{PN}}(rp)$. Without the constraint, the two expressions are not equivalent.

Picking the order of evaluation for an expression also falls under the heading of optimization. The query processor must decide which intermediate results to compute first and how to associate and commute operations. If an expression has a repeated subexpression, it may save time to save the value of that subexpression rather than computing it twice. The order that joins are taken in a string of joins is also important.

Example 11.7 Consider the expression

$$\pi_{P\#}(\sigma_{DT=7\ Jun}(rt \bowtie \sigma_{EN=Lewin}(re))) - \pi_{P\#}(\sigma_{DT=8\ Jun}(rt \bowtie \sigma_{EN=Lewin}(re)))$$

for the relations of Example 11.1. It may prove profitable to first compute

$$r = rt \bowtie \sigma_{EN=Lewin}(re)$$

and then compute

$$\pi_{P\#}(\sigma_{DT=7\ Jun}(r)) - \pi_{P\#}(\sigma_{DT=8\ Jun}(r)).$$

Example 11.8 Consider the join

$$rp \bowtie re \bowtie rt$$

for the relations of Example 11.1. If this join is evaluated as

$$(rp \bowtie re) \bowtie rt,$$

an intermediate result larger than the final result may be obtained, since $rp \bowtie re$ is a Cartesian product. A better way to evaluate it is

$$rp \bowtie (re \bowtie rt),$$

so that no joins are Cartesian products.

Finally, query optimization includes choosing the method of evaluation. The query processor may have several options as to how to perform a given operation. The option selected may depend on how a relation is stored, and what auxiliary structures exist. A selection can be computed by scanning the entire relation a tuple at a time. However, if an index exists for the attribute on which the selection is performed, the selection can be computed through index look-up. Projection may be implemented with and without duplicate

removal. If one projection is nested within another, duplicate removal need only be done at the outermost, if that choice is more efficient. Pointers from tuples in one relation to tuples in another are sometimes maintained to help compute joins.

Although the order of tuples in a relation is immaterial in the formal model, in query evaluation, order is important. Almost every query language allows the user to specify that a result be sorted by one or more attributes. Ordering an intermediate result can make a subsequent operation easier. Union, intersection, and difference can be computed on a single pass through each operand if both operands are sorted in the same order. The ordering of operands also makes a difference as to the best method to compute a join.

Example 11.9 Suppose we want to compute the join $r \bowtie s$ where the scheme of r is AB and the scheme of s is BC. Assume that each relation is stored on disk, with 20 tuples per disk block, and that input from the disk comes only in entire blocks. Assume that r requires 30 disk blocks and that s requires 40 blocks, and that there is room in memory for 5 disk blocks at a time. We shall count complexity by the number of disk accesses made.

If neither r nor s is sorted on B, we cannot do much better than the following method. Read the blocks of s into memory, 4 at a time. For each four blocks of s, we read in the blocks of r, one at a time. joining all possible tuples from the block of r with those in the four blocks of s. Each block of s is accessed once, and each block r is accessed 10 times, for a total of 340 accesses. This method of computing the join can be viewed as one loop nested within another. The outside loop steps through blocks of r, and the inside loop steps through blocks of s.

If r and s are both sorted on B, a more efficient method is available. Assume tuples with a given B-value do not span more than two blocks of r or s. We compute $r \bowtie s$ by considering each B-value in turn, and joining all the tuples from r and s with that B-value. At most, two blocks of r and two blocks of s need be in memory at a time. If we consider B-values in sort order, each block of r and s is accessed once, for a total of 70 accesses. This method of computing the join can be viewed as merging r and s by simultaneously making a single pass through each.

The query modifications we shall cover will not depend a lot on implementation details, for to do so we would have to delve into file management techniques and trade-offs between secondary storage access and computation time. We shall try to use general principles that are valid in most implementations, such as reducing the number of joins or the size of intermediate

results improves efficiency. However, the reader should be familiar with file structures and indexing techniques used for secondary storage, to have an intuition for the general principles.

11.1 LEVELS OF INFORMATION IN QUERY MODIFICATION

This section is a short aside to indicate how the range of modifications available depends on the amount of information available on a particular database and the system that supports it. Even if a lot of information is available, it may not be worthwhile to take the time to consult it for every query. For a query that must be evaluated only once, it is counter-productive to spend more time to obtain an optimized version than it takes to evaluate the original query. For a query that is expected to be evaluated many times, a large amount of time spent on optimization can be made up over the life of the query.

At the lowest level of information, nothing is known about the relations in a database but their names.

Example 11.10 The expressions

$$r \cap r,$$
$$\pi_X(r \cup s), \text{ and}$$
$$\sigma_{A=B \wedge B=C \wedge A=C}(r)$$

can be modified to the equivalent expressions

$$r,$$
$$\pi_X(r) \cup \pi_X(s), \text{ and}$$
$$\sigma_{A=B \wedge B=C}(r)$$

knowing nothing about the schemes of r and s except what is implicit in the expressions. Note, however, if we are given

$$\pi_X(r) \cup \pi_X(s)$$

originally, we cannot be sure that

$$\pi_X(r \cup s)$$

is a well-formed expression, so we do not allow that modification.

With information about schemes of relations, a wider class of modifications is available.

Example 11.11 Given relations $r(A\ B)$ and $s(B\ C)$, the expression

$$\sigma_{A=a}(r \bowtie s)$$

is equivalent to

$$\sigma_{A=a}(r) \bowtie s.$$

The equivalence does not hold if the schemes of the two relations are reversed.

Constraints on relations allow more modifications.

Example 11.12 If relation r satisfies the FD $B \rightarrow C$, then

$$\pi_{AB}(r) \bowtie \pi_{BC}(r)$$

can be replaced with

$$\pi_{ABC}(r).$$

The usefulness or necessity of certain modifications depends upon the operations supported by a specific database system and what retrieval methods are available for secondary storage. Multiple-relation joins might be implemented as a single operation. If selection is supported only where the condition is the conjunction of comparisons, selections with conditions involving disjunction then must be converted to a union of selections. It is possible in some systems to apply a selection and projection during the retrieval of a relation from secondary storage.

Some modifications depend upon the particular implementation of a database on a given system. The presence or absence of an index can dictate how to evaluate a selection or join. At the highest level of information, the state of the database comes into play. In the join $q \bowtie r \bowtie s$, the relative sizes of q, r, and s can dictate which pair of relations to join first. Even if $q \bowtie r$ is a Cartesian product, it is probably best to evaluate it first if $|q| \cdot |r| < |s|$. The method used to evaluate an expression might depend on which intermediate results are likely to fit in main memory. The decision

whether to store the value of a common subexpression or recompute it can also hinge on the size of the relations involved.

11.2 SIMPLIFICATIONS AND COMMON SUBEXPRESSIONS IN ALGEBRAIC EXPRESSIONS

The following equivalences can be used to eliminate useless operations from an expression:

$$r \cup r \equiv r$$
$$r \cap r \equiv r$$
$$r \bowtie r \equiv r$$
$$r - r \equiv \emptyset$$

where $\emptyset$ is the empty relation with the appropriate scheme. Once the empty relation can appear in expressions, other simplifications present themselves:

$$r \cup \emptyset \equiv r \qquad \pi_X(\emptyset) \equiv \emptyset$$
$$r \cap \emptyset \equiv \emptyset \qquad \sigma_C(\emptyset) \equiv \emptyset$$
$$r \bowtie \emptyset \equiv \emptyset \qquad r[C]\emptyset \equiv \emptyset$$
$$r - \emptyset \equiv r \qquad \emptyset[C]r \equiv \emptyset$$
$$\emptyset - r \equiv \emptyset \qquad \delta_N(\emptyset) \equiv \emptyset$$

where C is an arbitrary condition for selection or theta-join, and N is an arbitrary renaming.

Example 11.13 The expression

$$(r - r) \bowtie (r \cup r)$$

simplifies to

$$\emptyset \bowtie r$$

and thence to $\emptyset$.

The equivalences above also apply for an arbitrary expression E in place of r, such as $(r \bowtie s) \cup (r \bowtie s) \equiv r \bowtie s$. In order to apply such simplifica-

tions, we must locate common subexpressions within an expression. Ideally, we would like to locate equivalent subexpressions, such as $r \bowtie s$ and $s \bowtie r$. We shall describe a method that detects common subexpressions as well as some equivalent subexpressions that come from associativity and commutativity. As we have noted, detecting such subexpressions is useful for deciding how to evaluate an expression, as well as for applying simplifications.

We shall represent algebraic expressions by *expression trees*. An expression tree is a directed tree with interior nodes labeled by operators and leaves labeled by relations and constants. The outgoing edges from an interior node point to the operands for its operator.

Example 11.14 An expression tree for

$$\pi_X(s \bowtie r) - \pi_X(q \bowtie r \bowtie s)$$

is shown in Figure 11.1.

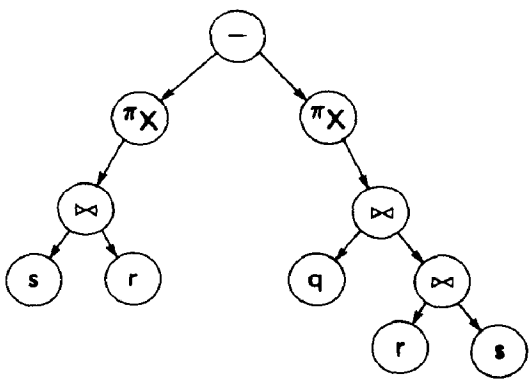

Figure 11.1

The idea is to convert an expression tree into a DAG by first merging identical leaves, and then merging interior nodes labeled with the same operator and having the same operands. For interior nodes with two operands, the order of the operands only matters for difference and theta-join.

Example 11.15 Starting with the expression tree in Figure 11.1, we can merge leaves to obtain the DAG in Figure 11.2. Two join nodes can then be merged to get the DAG in Figure 11.3.

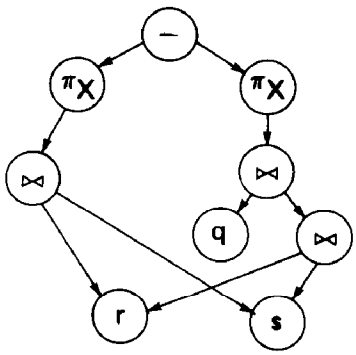

Figure 11.2

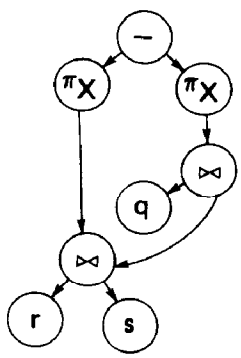

Figure 11.3

If associative operations in an expression are not completely parenthesized, there is a choice as to how to form the expression tree. The choice might preclude finding equivalent subexpressions.

Example 11.16 The subtree for $q \bowtie r \bowtie s$ in the expression tree of Figure 11.1 might have been formed as shown in Figure 11.4. In that case, no common subexpression would be detected.

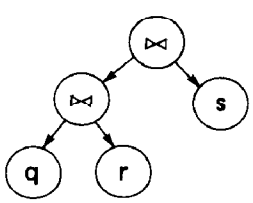

Figure 11.4

To avoid missing subexpressions because of the wrong choice of expression tree, we allow any number of out edges from an interior node representing an associative and commutative operation. If two interior nodes with the same label have sets of operands that overlap by two or more, the overlap can be brought out as a subexpression.

Example 11.17 The expression tree for

$$(r_1 \bowtie r_2 \bowtie r_3) \cup (r_2 \bowtie r_3 \bowtie r_4)$$

is shown in Figure 11.5. A merged version is shown in Figure 11.6.

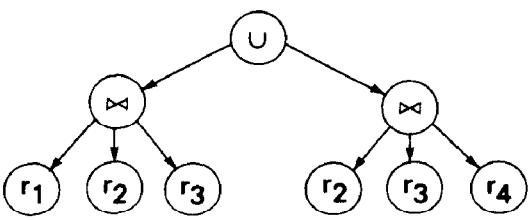

Figure 11.5

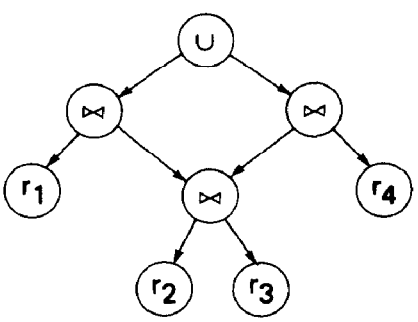

Figure 11.6

Even allowing more than two operands for interior nodes does not guarantee that every common subexpression will be found. Certain choices for merging will preclude others. If the expression of Example 11.17 is actually part of a larger expression that also includes $r_1 \bowtie r_2$, the merging done in Figure 11.6 means $r_1 \bowtie r_2$ will not be identified as a common subexpression. For small expressions, it might be possible to try all mergings in order to detect all subexpressions.

As common subexpressions are detected, we can check to see if any simplifications can be made.

Example 11.18 The expression tree for

$$q \bowtie \pi_X(r \bowtie s) \bowtie \pi_X(r \bowtie s)$$

can have its nodes merged to obtain the DAG in Figure 11.7, which can then be simplified to the DAG in Figure 11.8.

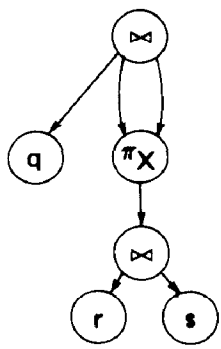

Figure 11.7

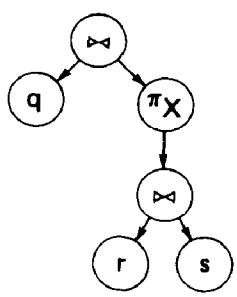

Figure 11.8

There are also simplifications based upon containment, some of which are covered in Exercise 11.1. While containment of relations might be unusual in a database, containment can readily arise at the expression level.

Example 11.19 A DAG obtained from the expression

$$((r - (r \cup s)) \bowtie q) \cup \sigma_{A=a}(s)$$

is given in Figure 11.9. If we can recognize that $r \subseteq r \cup s$, then we can make the sequence of simplifications shown in Figure 11.10.

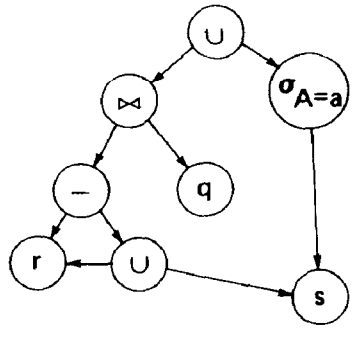

Figure 11.9

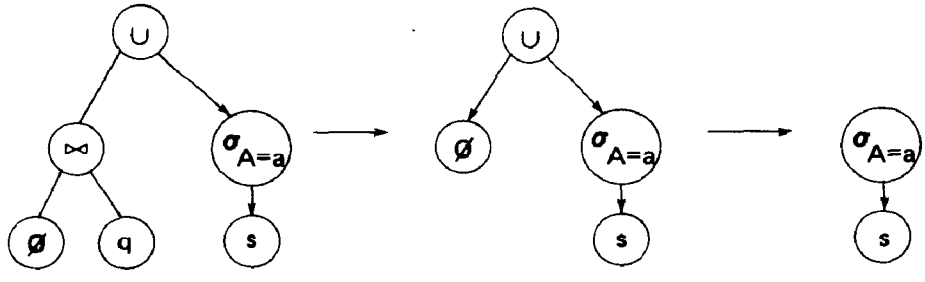

Figure 11.10

Unary operators can be combined to simplify expressions, using the following equivalences:

$$\sigma_{C_1}(\sigma_{C_2}(r)) \equiv \sigma_{C_1 \wedge C_2}(r)$$
$$\pi_X(\pi_Y(r)) \equiv \pi_X(r)$$
$$\delta_{N_1}(\delta_{N_2}(r)) \equiv \delta_{N_3}(r),$$

for the appropriate choice of N_3. Of course, such simplifications may mask common subexpressions. It is possible to apply these equivalences to split an operator and then get a common subexpression.

Example 11.20 In the expression

$$(r \bowtie \sigma_{A=a}(q \bowtie s)) - (r \bowtie \sigma_{A=a \wedge B=b}(q \bowtie s)),$$

it might prove fruitful to split $\sigma_{A=a \wedge B=b}(q \bowtie s)$ into $\sigma_{B=b}(\sigma_{A=a}(q \bowtie s))$ to exploit the subexpression $\sigma_{A=a}(q \bowtie s)$.

11.3 OPTIMIZING ALGEBRAIC EXPRESSIONS

For the optimizations presented in this chapter, the general principle is that the time and space required to perform a binary operation grows with the number of tuples in each operand and the size of the scheme of each operand. The strategy we employ here is to push selections and projections down the expression tree. Pushing selections down reduces the number of tuples in the operands of binary operations. Pushing projections down also reduces the number of tuples and it decreases the size of schemes. Also, as a general principle, we want to perform selections before projections, because selection requires at most a single pass through a relation, while projection can require sorting to remove duplicates.

Our goal is to start with an expression tree, push selections and projections as far down the tree as possible, and combine projections and selections such that in any path down the tree, between any two nodes for binary operations there is at most one projection and one selection node. The intuition here comes from the way an expression tree (or DAG) might be evaluated. For any interior node whose operands are all relations, we evaluate the operation at the node, store the result in a temporary relation, and replace the interior node by a leaf labeled with that relation. The process is repeated until only a single leaf is left in the tree.

Example 11.21 Consider the expression tree in Figure 11.11. We compute $s_1 = r_1 \bowtie r_2$ and modify the tree as shown in Figure 11.12. We next compute $s_2 = \pi_X(s_1)$ and modify the tree as shown in Figure 11.13. The evaluation is completed by computing $s_3 = s_2 \bowtie r_3$.

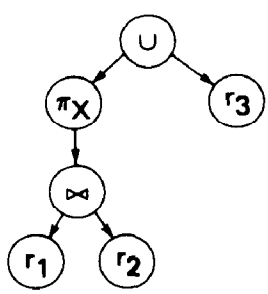

Figure 11.11

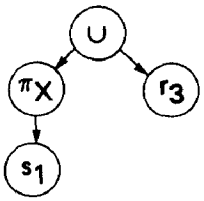

Figure 11.12

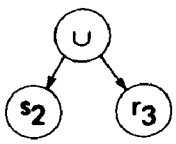

Figure 11.13

The reason for wanting a single projection and selection between nodes for binary operations is that the unary operations can easily be performed at the same time as the preceding or following binary operation. When retrieving tuples from a relation for binary operations, it is a simple matter to drop attributes or screen out tuples to perform projection or selection. The same can be done when storing tuples that are the result of the binary operation. In addition, the file manager might be able to remove duplicates during storage.

Example 11.22 In Example 11.21, the projection can be combined with the subsequent join, and the temporary relation s_2 need not be stored.

Below we give the equivalences that can be used to push selection down an expression tree. We assume that theta-join $r[C]s$ is transformed to $\sigma_C(r \bowtie s)$, where the natural join is necessarily a Cartesian product. The equivalences are

$$\sigma_C(\pi_X(r)) \equiv \pi_X(\sigma_C(r))$$
$$\sigma_C(\delta_N(r)) \equiv \delta_N(\sigma_{C'}(r))$$
$$\sigma_C(r \cup s) \equiv \sigma_C(r) \cup \sigma_C(s)$$
$$\sigma_C(r \cap s) \equiv \sigma_C(r) \cap \sigma_C(s)$$
$$\sigma_C(r - s) \equiv \sigma_C(r) - \sigma_C(s),$$

where condition C' is condition C with appropriate renamings. Joins present a problem, since C might contain comparisons that involve attributes in both relations. Given relations $r(R)$ and $s(S)$, suppose C can be written as $C_1 \wedge C_2 \wedge C_3$, where C_2 applies only to attributes in R and C_3 applies only to attributes in S. We then have the equivalence

$$\sigma_{C_1 \wedge C_2 \wedge C_3}(r \bowtie s) \equiv \sigma_{C_1}(\sigma_{C_2}(r) \bowtie \sigma_{C_3}(s)).$$

Example 11.23 Let q, r, and s be relations with schemes AB, BC, and CD. Then the expression

$$\sigma_{B \leq C \wedge C=4 \wedge D<A}(q \bowtie r \bowtie s)$$

can be modified to

$$\sigma_{D<A}(q \bowtie \sigma_{B \leq C \wedge C=4}(r) \bowtie \sigma_{C=4}(s)).$$

Note that

$$\sigma_{D<A}(q \bowtie \sigma_{B \leq C}(r) \bowtie \sigma_{C=4}(s))$$

is also an equivalent expression, but we always choose to push selection down as many branches of the expression tree as possible.

All of the equivalences above for selection, of course, work when relations r and s are replaced by arbitrary algebraic expressions E_1 and E_2. The following equivalences are used to push projection down the expression tree:

$$\pi_X(\delta_N(r)) \equiv \delta_N(\pi_{X'}(r))$$
$$\pi_X(\sigma_C(r)) \equiv \pi_X(\sigma_C(\pi_{XY}(r))$$
$$\pi_X(r \cup s) \equiv \pi_X(r) \cup \pi_X(s),$$

where X' is the appropriate renaming of X, and Y is the smallest set of attributes such that XY contains all the attributes mentioned in C. Note that projection cannot be pushed past an intersection or difference. Join, once

again, is a little tricky. Let R and S be the relation schemes of r and s. Let $R' = R \cap XS$ and $S' = S \cap XR$. We can then use the equivalence

$$\pi_X(r \bowtie s) \equiv \pi_X(\pi_{R'}(r) \bowtie \pi_{S'}(s)).$$

We must retain all the attributes in X plus all the attributes in $R \cap S$. As before, the equivalences hold with expressions substituted for relations.

Algebraic optimization proceeds by using the equivalences above to push selections and projections down the expression tree as far as possible. Where there is a conflict between pushing a selection or pushing a projection, the selection goes lower than the projection, for reasons already discussed. A project-select-project sequence is simplified to a project-select sequence. Any projections onto the entire scheme of an operand are removed.

Example 11.24 Consider the following expression using the relations of Example 11.1, which gives all the pilots that have been given a low rating by Randolph and who are based in the same place as some other pilot who has been given a low rating since 1 June.

$$\pi_{\text{PN}}(\sigma_{\text{PN} \neq \text{PN}'}(\sigma_{\text{EN}=\text{Randolph} \wedge \text{RG} \leq 6.5}(rp \bowtie re \bowtie rt) \bowtie \pi_{\text{PN}', \text{BD}}(\delta_{\text{PN} \leftarrow \text{PN}'}(\sigma_{\text{DT} \geq 1 \text{ June} \wedge \text{RG} \leq 6.5}(rp \bowtie rt))))).$$

The expression tree for this expression is shown in Figure 11.14. The topmost selection cannot be pushed through the join, but the other two selections can be pushed downward through joins, as shown in Figure 11.15. Figure 11.16 shows the topmost projection pushed down through a selection and join. Figure 11.17 shows projections pushed further down the tree, and the next-to-topmost projection removed. If we do not worry about removing duplicates in the two projections that lie below all binary operations, and perform the three-way join as two binary joins, then the tree in Figure 11.17 can be evaluated in four stages. If duplicates are eliminated for those projections, then the tree can be evaluated in six stages:

$$\begin{aligned}
r_1 &\leftarrow \pi_{\text{P\# E\# RG}}(\sigma_{\text{RG} \geq 6.5}(rt)) \\
r_2 &\leftarrow \sigma_{\text{EN}=\text{Randolph}}(re) \bowtie r_1 \\
r_3 &\leftarrow \pi_{\text{PN BD}}(rp \bowtie r_2) \\
r_4 &\leftarrow \sigma_{\text{P\#}}(\sigma_{\text{DT} \geq 1 \text{ June} \wedge RG \leq 6.5}(rt)) \\
r_5 &\leftarrow \delta_{\text{PN} \leftarrow \text{PN}'}(\pi_{\text{PN BD}}(rp \bowtie r_4)) \\
r_6 &\leftarrow \pi_{\text{PN}}(\sigma_{\text{PN} \neq \text{PN}'}(r_3 \bowtie r_5)).
\end{aligned}$$

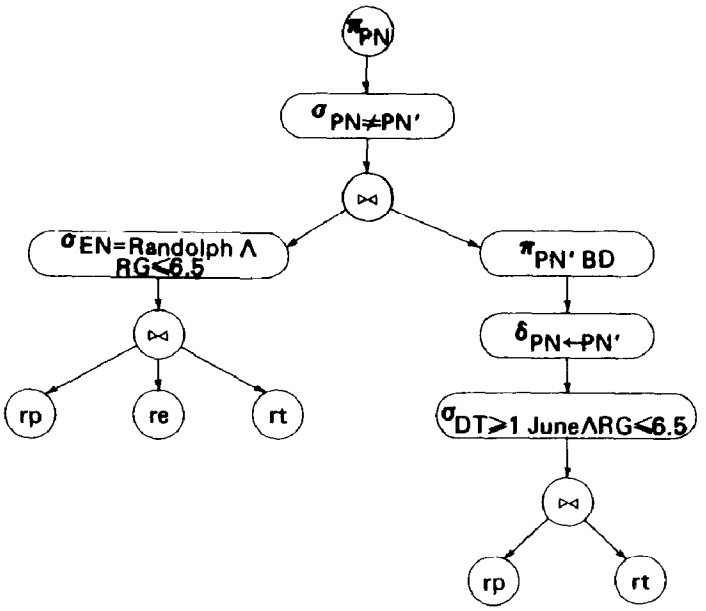

Figure 11.14

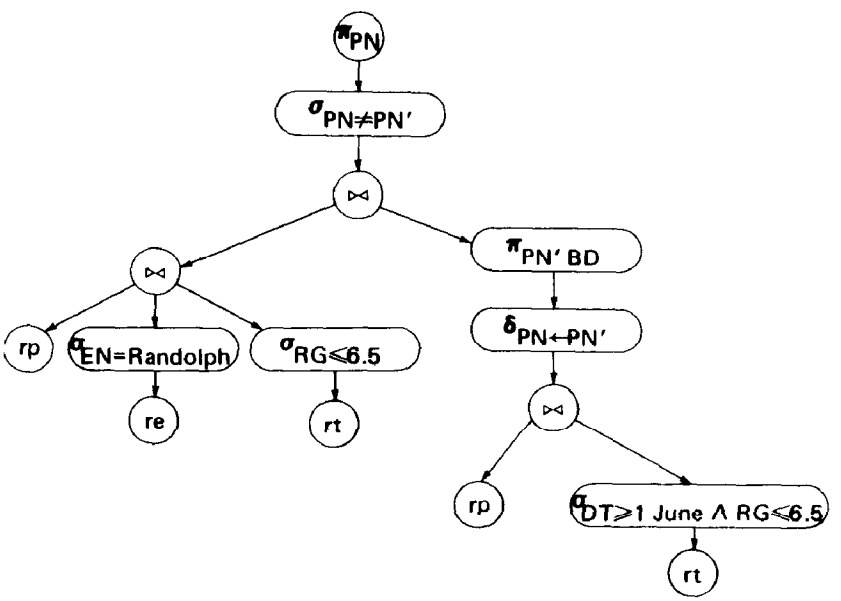

Figure 11.15

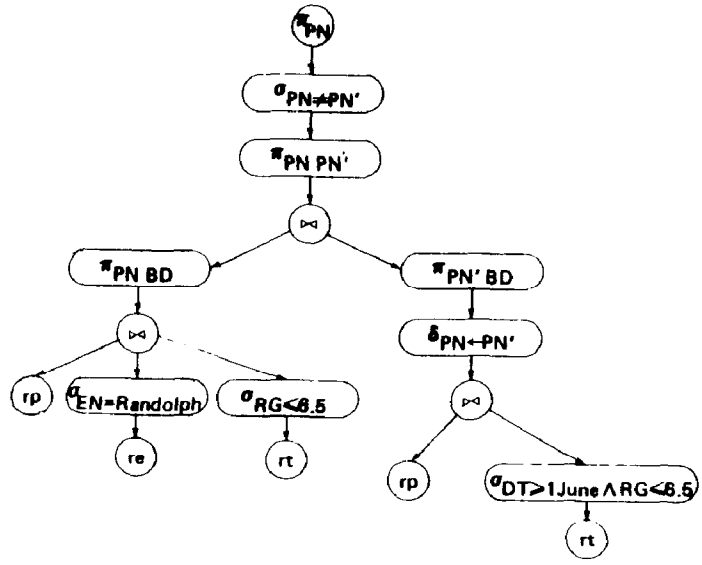

Figure 11.16

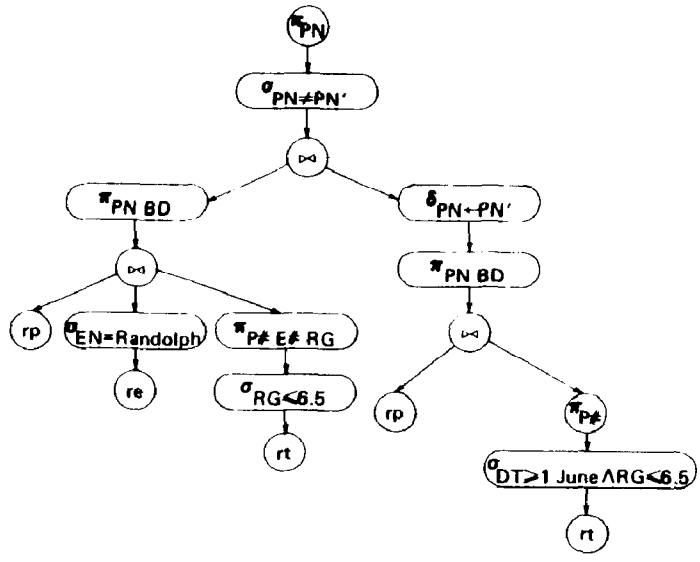

Figure 11.17

The interaction between finding common subexpressions and algebraic optimization is complex. Certainly, performing algebraic optimizations can mask common subexpressions. In Example 11.24, the common subexpression $rp \bowtie rt$ is present in the original tree but not in the optimized tree. However, it is not obvious that computing $rp \bowtie rt$ and using the result in two subtrees saves any time over the optimized version. There is also a problem in pushing selections and projections down a DAG, since one node can represent the operand of multiple operations. It would seem best to optimize first and then look for common subexpressions. The optimized expression may even contain common subexpressions that were not present in the original expression.

Example 11.25 Let q, r and s be relations on schemes AB, BC, and CD. The expression

$$(q \bowtie r) - \pi_{ABC}(q \bowtie \sigma_{D=d}(r \bowtie s))$$

has no common subexpressions. The optimized version of the expression,

$$(q \bowtie r) - \pi_{ABC}(q \bowtie r \bowtie \pi_C(\sigma_{D=d}(s)))$$

has the common subexpression $q \bowtie r$.

We state again that "optimization" is somewhat of a misnomer for what we are doing. Pushing selections and projections down a tree is a heuristic; there is no guarantee that the modified expression will actually save any time or space for a particular state of the database.

11.4 QUERY DECOMPOSITION

This section deals with a method for evaluating queries that is used by the QUEL query processor in the INGRES relational database system. The goal is to reduce each query to a program involving only assignment, selection, projection, and **for**-loops. The INGRES algorithm also uses recursive calls, but we shall use assignments in their place for simplicity.

The decomposition method works on a class of queries described by tuple calculus expressions of the form

$$\{x(A_1 A_2 \cdots A_n) \mid \exists y_1(R_1) \in r_1 \; \exists y_2(R_2) \in r_2 \cdots \exists y_m(R_m) \in r_m \\ (f(x, y_1, y_2, \ldots, y_m) \wedge g(y_1, y_2, \ldots, y_m))\}.$$

The formula f is a conjunction of atoms. For each i, $1 \leq i \leq n$, it contains the atoms $x(A_i) = y_j(B)$ for some y_j and some attribute B. The formula g contains no quantifiers, nor atoms of the form $r(z)$. Such a tuple calculus expression can be easily translated to an equivalent algebraic expression. Let s_i stand for $\delta_{N_i}(r_i)$, where N_i renames each attribute B in R_i to "$y_i.B$". The equivalent algebraic expression has the form

$$\delta_N(\pi_X(\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m))).$$

For each atom $x(A_i) = y_j(B)$ in f, N includes $A_i \leftarrow y_j.B$ and X contains B. C is the selection condition obtained from g by converting each atom $y_i(A)\ \theta\ y_j(B)$ to the comparison $y_1.A\ \theta\ y_j.B$. Note that by the way attributes are named in the s_i's, all the joins are Cartesian products.

Example 11.26 Consider the expression

$$\{x(\text{PN RG}) \mid \exists y_1(\text{P\# PN BD}) \in rp\ \exists y_2(\text{P\# E\# DT RG}) \in rt$$
$$((x(\text{PN}) = y_1(\text{PN}) \wedge x(\text{RG}) = y_2(\text{RG})) \wedge$$
$$(y_1(\text{BD}) = \text{"JFK"} \wedge y_1(\text{P\#}) = y_2(\text{P\#}) \wedge y_2(\text{RG}) > 9))\},$$

using the relations rp and rt from Example 11.1. An equivalent algebraic expression is

$$\delta_{\text{PN}\leftarrow y_1.\text{PN},\text{RG}\leftarrow y_2.\text{RG}}(\pi_{y_1.\text{PN}\, y_2.\text{RG}}$$
$$(\sigma_{y_1.\text{BD}=\text{"JFK"} \wedge y_1.\text{P\#}=y_2.\text{P\#} \wedge y_2.\text{RG}>9}\ (rp' \bowtie rt'))).$$

where rp' and rt' are rp and rt with appropriate renamings.

We shall present a graphical representation of algebraic expressions of the form

$$\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m),$$

but first we need to massage C into a certain form.

Definition 11.1 A selection condition C is in *conjunctive normal form* (CNF) if it has the form $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ where no C_i contains $\wedge$ and negation applies only to individual comparisons.

An arbitrary selection condition C can easily be put in CNF. First, apply the following two identities (De Morgan's Laws) to move negations inward until they apply to individual comparisons:

1. $\neg(C_1 \wedge C_2) = \neg C_1 \vee \neg C_2$
2. $\neg(C_1 \vee C_2) = \neg C_1 \wedge \neg C_2$.

Next, apply the following equalities to distribute $\vee$ over $\wedge$:

3. $(C_1 \wedge C_2) \vee C_3 = (C_1 \vee C_3) \wedge (C_2 \vee C_3)$
4. $(C_1 \vee (C_2 \wedge C_3) = (C_1 \vee C_2) \wedge (C_1 \vee C_3)$.

Example 11.27 Starting with the selection condition

$$\neg((A_1 = A_2 \vee A_2 < a) \vee \neg(A_1 = A_3 \vee A_3 = a)),$$

we move the first negation inward to get

$$\neg(A_1 = A_2 \vee A_2 < a) \wedge (A_1 = A_3 \vee A_3 = a).$$

Moving negation inward once more, we get

$$(\neg A_1 = A_2 \wedge \neg A_2 < a) \wedge (A_1 = A_3 \vee A_3 = a)),$$

which has negation applying only to individual comparisons. We can then distribute to get

$$(\neg A_1 = A_2 \vee (A_1 = A_3 \vee A_3 = a)) \\ \wedge (\neg A_2 < a \vee (A_1 = A_3 \vee A_3 = a)),$$

which is in CNF.

Let E be the algebraic expression

$$\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m),$$

where $C = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is in CNF and the joins are all Cartesian products. We represent E by a labeled hypergraph H_E, called the *connection graph* for E. In a hypergraph, edges may contain one or more nodes, rather than just two as in regular graphs. H_E has a node for each of $s_1, s_2, \ldots, s_m$. H_E contains an edge e_i for each conjunct C_i. Edge e_i contains node s_j if $y_j.B$ appears in C_i for some attribute B. Edge e_i is labeled by C_i.

Example 11.28 Let E be the algebraic expression

$$\sigma_C(s_1 \bowtie s_2 \bowtie s_3 \bowtie s_4 \bowtie s_5)$$

where C is the selection condition

$$y_1.A = y_2.A \wedge y_1.B = y_2.B \wedge y_3.D = d \wedge$$
$$(y_2.G = y_3.G \vee y_2.F < y_4.F) \wedge y_4.A < y_5.A.$$

The connection graph H_E for E is shown in Figure 11.18. Edges are depicted with dashed lines. Note that two edges can have the same nodes, but the edges will have different labels.

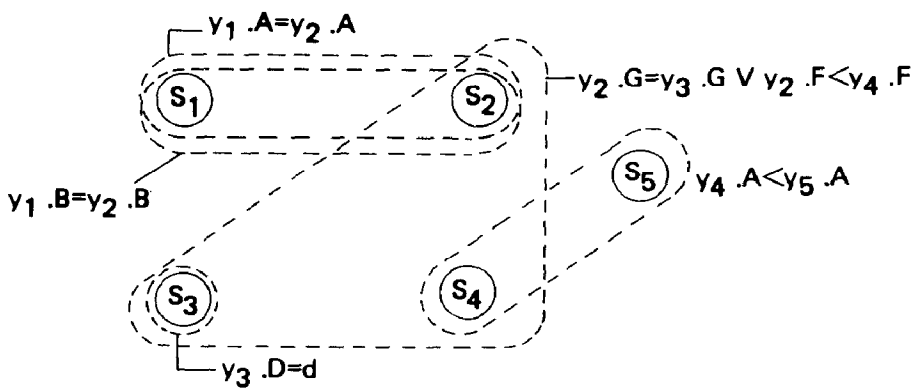

Figure 11.18

In all subsequent examples, edges will have at most two nodes, for simplicity. We shall draw the connection graphs as regular graphs. An edge containing only one node is represented as a loop from the node to itself.

Query decomposition starts with a one-statement program of the form

$$r \leftarrow \pi_X \sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m),$$

where the schemes of the s_i's are disjoint. The end result is a multiple-statement program that contains assignment, selection, projection, and **for**-loops, but no joins. The joins are performed via **for**-loops instead. Two transformations on programs, called *instantiation* and *iteration*, are used to achieve that goal. The transformations correspond to edge removal and node removal in connection graphs, and the goal is to transform the connection graph for

$\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m)$ to a graph with no edges. We examine each transformation in detail.

11.4.1 Instantiation

Instantiation is analogous to pushing selections down an expression tree. We start with the statement

$$r \leftarrow \pi_X \sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m),$$

where $C = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is a CNF selection condition. Let E denote the expression.

$$\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m).$$

Instantiation starts with the choice of some subset of the relations $\{s_1, s_2, \ldots, s_m\}$ to instantiate. Suppose we choose $\{s_1, s_2, \ldots, s_p\}$. Let $e_1, e_2, \ldots, e_q$ be the edges in H_E that consist only of nodes in $\{s_1, s_2, \ldots, s_p\}$. Recall that edge e_i is labeled with conjunct C_i. The transformed program for this instantiation has two statements:

$$r' \leftarrow \pi_Y(\sigma_{C_1 \wedge C_2 \wedge \cdots \wedge C_q}(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_p));$$
$$r \leftarrow \pi_X(\sigma_{C_{q+1} \wedge C_{q+2} \wedge \cdots \wedge C_k}(r' \bowtie s_{p+1} \bowtie s_{p+2} \bowtie \cdots \bowtie s_m)).$$

Y is the set of attributes in relations $s_1, s_2, \ldots, s_p$ that are mentioned in $C_{q+1}, C_{q+2}, \ldots, C_k$ or contained in X. Relation r' is a temporary relation to hold the intermediate result.

The corresponding change in the connection graph substitutes r' for $s_1, s_2, \ldots, s_k$ in the edges $e_{q+1}, e_{q+2}, \ldots, e_k$. The result is a graph with two components, one with nodes $s_1, s_2, \ldots, s_p$ and edges $e_1, e_2, \ldots, e_q$, and the other with nodes $r', s_{p+1}, s_{p+2}, \ldots, s_m$ and modified edges $e_{q+1}, e_{q+2}, \ldots, e_k$.

Example 11.29 Consider the statement

$$r \leftarrow \pi_{y_1.A\, y_3.B\, y_4.G}(\sigma_C(s_1 \bowtie s_2 \bowtie s_3 \bowtie s_4))$$

where

$$C \text{ is } y_1.A = y_2.A \wedge y_2.A = y_3.A \wedge y_2.B \leq y_3.B$$
$$\wedge\, y_3.D \leq y_4.D \wedge y_4.F = 6.$$

The connection graph for the selection is shown in Figure 11.19. Instantiating on $\{s_2, s_3\}$ gives the statements

$$r' \leftarrow \pi_{y_2.A\, y_3.B\, y_3.D}(\sigma_{C'}(s_2 \bowtie s_3));$$
$$r \leftarrow \pi_{y_1.A\, y_3.B\, y_4.G}(\sigma_{C''}(r' \bowtie s_1 \bowtie s_4)),$$

where

C' is $y_2.A = y_3.A \wedge y_2.B \leq y_3.B$ and
C'' is $y_1.A = y_2.A \wedge y_3.D \leq y_4.D \wedge y_4.F = 6$.

The modified connection graph is shown in Figure 11.20.

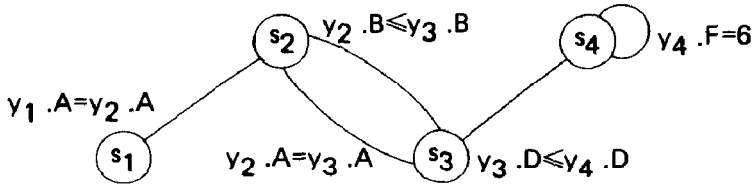

Figure 11.19

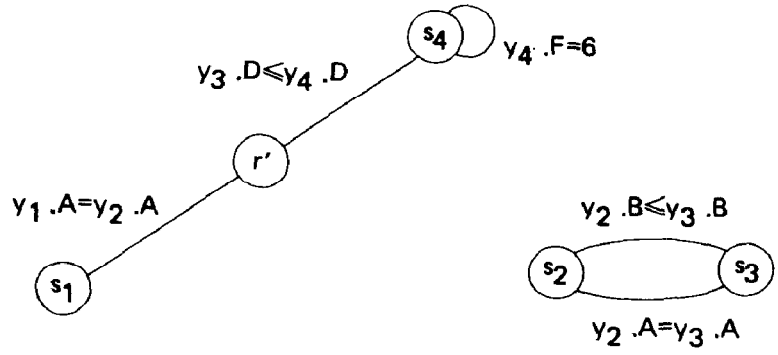

Figure 11.20

A useful special case is where a single relation, say s_1, is chosen for instantiation. In that case the two statements in the transformed program are

$$r' \leftarrow \pi_Y(\sigma_{C_1 \wedge C_2 \wedge \cdots \wedge C_q}(s_1));$$
$$r \leftarrow \pi_X(\sigma_{C_{q+1} \wedge C_{q+2} \wedge \cdots \wedge C_k}(r' \bowtie s_2 \bowtie s_3 \bowtie \cdots \bowtie s_m)).$$

Example 11.30 Suppose we start with the program in Example 11.29 and instantiate $\{s_4\}$ instead. The resulting program is

$$r' \leftarrow \pi_{y_4.D\, y_4.G}(\sigma_{y_4.F=6}(s_4));$$
$$r \leftarrow \pi_{y_1.A\, y_3.B\, y_4.G}(\sigma_{C'}(r' \bowtie s_1 \bowtie s_2 \bowtie s_3)),$$

where C' is C without the comparison $y_4.F = 6$. The modified connection graph is in Figure 11.21.

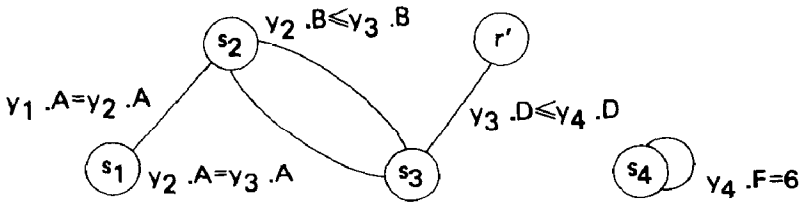

Figure 11.21

11.4.2 Iteration

This transformation is also called tuple substitution. It starts with a single statement

$$r \leftarrow \pi_X \sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m)$$

as before. One of the relations, say s_1, is then chosen for iteration. The transformed program is

$r \leftarrow \emptyset$
for each tuple t in s_1 **do**
  begin
  $r' \leftarrow \pi_Y \sigma_{C(t)}(s_2 \bowtie s_3 \bowtie \cdots \bowtie s_m)$;
  add r' to r with appropriate padding
  end.

Here $C(t)$ means C with every occurrence of an attribute $y_1.A$ replaced by the value $t(y_1.A)$. Y contains those attributes in X that are not in the scheme of s_1. The appropriate padding for r' is $t(X-Y)$. That is, r' is extended by the portion of t that is included in X.

The corresponding change in the connection graph is to remove the node s_1. Any edges that were incident upon s_1 become loops.

Example 11.31 Consider the statement

$$r \leftarrow \pi_{y_1.A\, y_2.B\, y_3.G}(\sigma_C(s_1 \bowtie s_2 \bowtie s_3)),$$
$$C \text{ is } y_1.A = y_2.A \wedge y_2.D \leq y_3.D \wedge y_3.F = 6.$$

This statement is the same as one of the statements in Example 11.29 with some renaming. The connection graph for this statement is shown in Figure 11.22. Iterating on s_2 gives the transformed program

$r \leftarrow \emptyset$
for each tuple t in s_2 **do**
  begin
  $r' \leftarrow \pi_{y_1.A\, y_3.G}(\sigma_{C(t)}(s_1 \bowtie s_3))$;
  add $r' \bowtie \langle t(y_2.B) \rangle$ to r
  end.

$$C(t) \text{ is } y_1.A = t(y_2.A) \wedge t(y_2.D) \leq y_3.D \wedge y_3.F = 6.$$

The modified connection graph is shown in Figure 11.23.

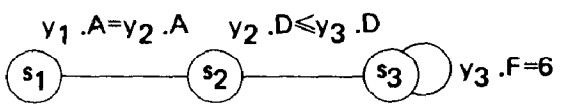

Figure 11.22

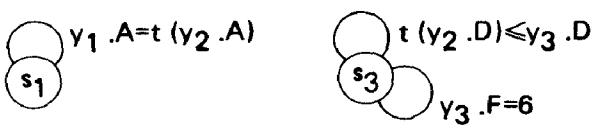

Figure 11.23

11.4.3 The Query Decomposition Algorithm

The query decomposition algorithm is simply stated as: start with a single-statement program, and repeatedly apply the instantiation and iteration transformations until all assignment statements have the form

$r \leftarrow \pi_X \sigma_C(s)$,
$r \leftarrow \pi_X(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_k)$ or
$r \leftarrow \emptyset$.

The second form can be transformed further by more applications of iteration. However, since no selection in involved, we use that form as shorthand for the k-1 **for**-loops necessary to compute the Cartesian product. Also, depending on the attributes in the projection, a **for**-loop might not be necessary for every join (see Exercise 11.6). In terms of the connection graph, the goal of the algorithm is essentially to isolate every node. That is, no edge connects two different nodes. Actually, once all the nodes are isolated, some applications of instantiation might still be necessary.

Example 11.32 In the connection graph (Figure 11.23) for the transformed program of Example 11.31, all nodes are isolated. The second assignment statement has a selection applied to a join. However, since the selection condition has no comparisons between attributes of s_1 and s_3, iteration is not necessary, only instantiation. A fully transformed version of the program is

$r \leftarrow \emptyset$
for each tuple t in s_2 **do**
  begin
  $r_1' \leftarrow \pi_{y_1.A}(\sigma_{C_1(t)}(s_1))$;
  $r_3' \leftarrow \pi_{y_3.G}(\sigma_{C_2(t)}(S_3))$;
  $r' \leftarrow r_1' \bowtie r_3'$;
  add $r' \bowtie \langle t(y_2.B) \rangle$ to r
  end.

$C_1(t)$ is $y_1.A = t(y_2.A)$ and $C_2(t)$ is $t(y_2.D) \leq y_3.D \wedge y_3.F = 6$.

Example 11.33 For this example we shall use relations s_w, s_x, s_y, and s_z, and assume attributes from s_w are prefaced with "w.", attributes from s_x with "x.", and so forth. Assume that we start with the single statement

$$r \leftarrow \pi_{w.A\ z.G}(\sigma_{C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6}(s_w \bowtie s_x \bowtie s_y \bowtie s_z))$$

where

C_1 is $w.B \leq 7$ $\quad$ C_4 is $x.D = y.D$
C_2 is $w.A = x.A$ $\quad$ C_5 is $y.F = z.F$
C_3 is $w.A \leq y.A$ $\quad$ C_6 is $x.F \leq z.F$.

The connection graph for this statement is shown in Figure 11.24.
We start by iterating s_z to get

1. $r \leftarrow \emptyset$
   for each tuple t in s_z **do**
   begin
2. $r_1 \leftarrow \pi_{w.A}(\sigma_{C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5(t) \wedge C_6(t)}(s_w \bowtie s_x \bowtie s_y))$
   add $r_1 \bowtie \langle t(z.G) \rangle$ to r
   end.

where

$C_5(t)$ is $y.F = t(z.F)$ and $C_6(t)$ is $x.F \leq t(z.F)$.

The connection graph for this program is shown in Figure 11.25.
We now instantiate s_w, s_x, and s_y, in turn, in statement 2 to get

2.1. $r_2 \leftarrow \pi_{w.A}(\sigma_{C_1}(s_w))$;
2.2. $r_3 \leftarrow \pi_{x.A\ x.D}(\sigma_{C_6(t)}(s_x))$;
2.3. $r_4 \leftarrow \pi_{y.A\ y.D}(\sigma_{C_5(t)}(s_y))$;
2.4. $r_1 \leftarrow \pi_{w.A}(\sigma_{C_2 \wedge C_3 \wedge C_4}(r_2 \bowtie r_3 \bowtie r_4))$.

The connection graph is now as shown in Figure 11.26.
We next iterate r_3 in statement 2.4 to get

2.4.1. $r_1 \leftarrow \emptyset$;
   for each tuple u in r_3 **do**
   begin
2.4.2. $r_5 \leftarrow \pi_{w.A}(\sigma_{C_2(u) \wedge C_3 \wedge C_4(u)}(r_2 \bowtie r_4))$;
   add r_5 to r_1
   end.

where

$C_2(u)$ is $w.A = u(x.A)$ and $C_4(u)$ is $u(x.D) = y.D$.

The connection graph is now as shown in Figure 11.27. Finally, we instantiate r_4, then iterate the resulting relation, r_6, in statement 2.4.2 to get

2.4.2.1. $r_6 \leftarrow \pi_{y.A}(\sigma_{C_4(r)}(r_4))$;
2.4.2.2. $r_5 \leftarrow \emptyset$;
for each tuple v in r_6 **do**
begin
2.4.2.3. $r_7 \leftarrow \pi_{w.A}(\sigma_{C_2(u) \wedge C_3(v)}(r_2))$;
add r_7 to r_5
end.

where

$C_3(v)$ is $w.A \leq v(y.A)$.

The final connection graph is shown in Figure 11.28.

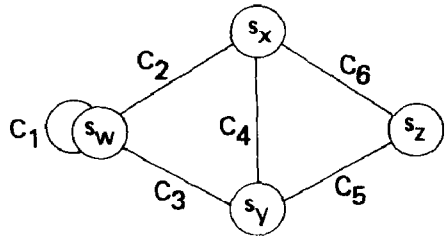

Figure 11.24

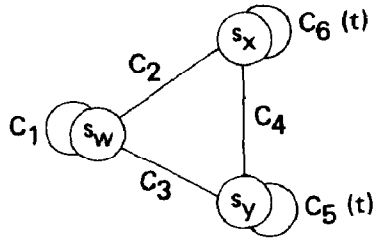

Figure 11.25

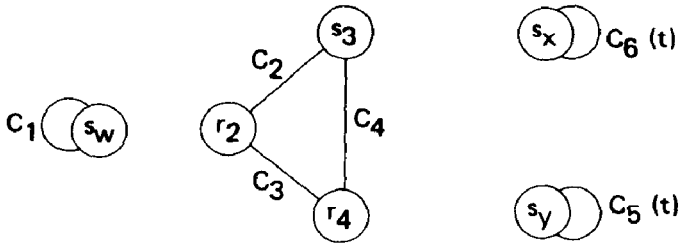

Figure 11.26

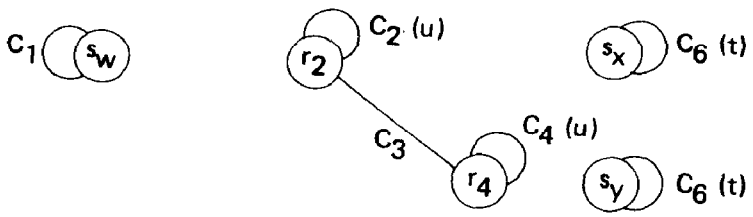

Figure 11.27

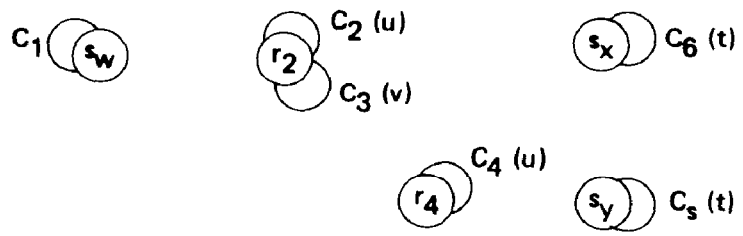

Figure 11.28

The query decomposition algorithm as described leaves a lot of leeway, since it does not say which instantiations or iterations to perform, nor in what order to perform them. Some final programs will be better than others. We present a heuristic method for applying the transformations in the decomposition algorithm to minimize the time complexity of the resulting program. We note here that the QUEL query processor does not use multiple-relation instantiation (see Exercise 11.8). While multiple-relation instantiation is not necessary to get a program in the desired form, it is useful for applying heuristics during decomposition.

The first principle is to minimize the number of repetitions for each **for**-loop. Hence, we want to instantiate a relation before iterating it whenever possible, to reduce the number of tuples that must be considered in the **for**-loop. The second principle is to minimize the number of times the iteration

transformation must be applied. In general, the number of iterations required will be one fewer than the number of relations involved. Sometimes, however, not all the iterations need be performed for a statement of the form $r \leftarrow \pi_X(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_k)$ (see Exercise 11.8). The third principle is to minimize the depth of the nesting of **for**-loops, since nesting has a multiplicative effect upon time complexity (see Exercise 11.10).

The second and third principles are served by always trying to choose a relation to iterate whose node, if removed, disconnects some portions of the connection graph.

Example 11.34 If we are working on a statement with the connection graph shown in Figure 11.29, picking s_1 to iterate means at least one more application of iteration. If s_2 is iterated first, no more applications of iteration are necessary.

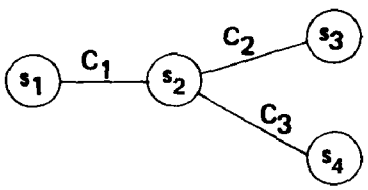

Figure 11.29

In the following method, we make crude estimates on the sizes of relations. A *simple* edge in the connection graph is a one- or two-node edge whose selection condition is the disjunction of equality comparisons. Let r be a new relation generated by instantiating a single relation s:

$$r \leftarrow \pi_X(\sigma_C(s)).$$

If any of the selection conditions in C comes from a simple edge, we label r "small," since presumably the equalities hold for only a few tuples in s.

The heuristic is described in terms of the connection graph by the following options. During the query decomposition algorithm, always choose the lowest numbered option possible.

1. Instantiate a relation that is contained in some one-node edge. If a simple edge is involved in the instantiation, label the newly generated relation "small."
2. Iterate a "small" relation. Prefer one in simple edges.
3. Iterate a relation whose removal disconnects some portion of the graph. Prefer one in simple edges.

4. Instantiate a set of relations whose removal disconnects some portion of the graph. Give preference to sets
   a. with few relations,
   b. that are connected in the graph, and
   c. that are in simple edges.
5. Iterate a relation. Prefer one in simple edges.

The idea behind option 4 is that the new relation generated by the instantiation is a candidate for option 3 during the next transformation.

Example 11.35 We repeat the query decomposition algorithm on the initial statement of Example 11.33, this time using the heuristic. In the connection graphs, we denote "small" relations by heavy circles. The graph for the initial statement is shown in Figure 11.24. Note that the edges for C_2, C_4, and C_5 are simple. Option 1 applies to relation s_w, so we instantiate s_w to get

1. $r_1 \leftarrow \pi_{w.A}(\sigma_{C_1}(s_w))$;
2. $r \leftarrow \pi_{w.A\ z.G}(\sigma_{C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6}(r_1 \bowtie s_x \bowtie s_y \bowtie s_z))$.

The edge labeled C_1 is not simple, so r_1 is not "small." The modified connection graph is shown in Figure 11.30. None of options 1-3 now apply, so we apply option 4. We instantiate $\{s_x, s_y\}$ in statement 2 to get

2.1. $r_2 \leftarrow \pi_{x.A\ x.F\ y.A\ y.F}(\sigma_{C_4}(s_x \bowtie s_y))$;
2.2. $r \leftarrow \pi_{x.A\ z.G}(\sigma_{C_2 \wedge C_3 \wedge C_5 \wedge C_6}(r_1 \bowtie r_2 \bowtie s_z))$.

The new connection graph is shown in Figure 11.31.

We now use option 3 and iterate r_2 in statement 2.2 to get

2.2.1. $r \leftarrow \emptyset$;
for each tuple t **in** r_2 **do**
begin
2.2.2. $r_3 \leftarrow \pi_{w.A\ z.G}(\sigma_{C_2(t) \wedge C_3(t) \wedge C_5(t) \wedge C_6(t)}(r_1 \bowtie s_z))$;
add r_3 to r
end.

where

$C_2(t)$ is $w.A = t(x.A)$ $C_5(t)$ is $t(y.F) = z.F$
$C_3(t)$ is $w.A \leq t(y.A)$ $C_6(t)$ is $t(x.F) \leq z.F$.

The new connection graph is shown in Figure 11.32.

Using option 1, we can instantiate r_1 and s_z in statement 2.2.2 to get

2.2.2.1. $r_4 \leftarrow \pi_{w.A}(\sigma_{C_2(t) \wedge C_3(t)}(r_1))$;
2.2.2.2. $r_5 \leftarrow \pi_{z.G}(\sigma_{C_5(t) \wedge C_6(t)}(s_z))$;
2.2.2.3. $r_3 \leftarrow r_4 \bowtie r_5$.

The new connection graph is given in Figure 11.33. Since the edges for $C_2(t)$ and $C_5(t)$ are simple, r_4 and r_5 are "small" relations. However, that fact does not affect the rest of the decomposition. Finally, we iterate s_x in statement 2.1 to get

2.1.1. $r_2 \leftarrow \emptyset$;
for each tuple u in s_x **do**
begin
2.1.2. $r_6 \leftarrow \pi_{y.A\, y.F}(\sigma_{C_4(u)}(s_y))$;
add $r_6 \bowtie \langle u(x.A)\, u(x.F)\rangle$ to r_2
end.

where $C_4(u)$ is $u(x.D) = y.D$. The final connection graph is shown in Figure 11.34.

Note that the order of instantiation and iteration would be completely different under the heuristic if C_1 were $w.B = 7$. After instantiating s_w, r_1 would be "small," and hence would be iterated at the next step (see Exercise 11.7 d).

Comparing the decomposition obtained in this example to the decomposition in Example 11.33, we see that there we had 3 explicit iterations, all nested, while here we have two explicit iterations, not nested.

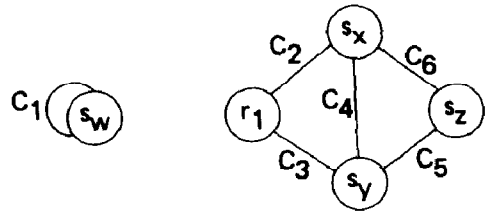

Figure 11.30

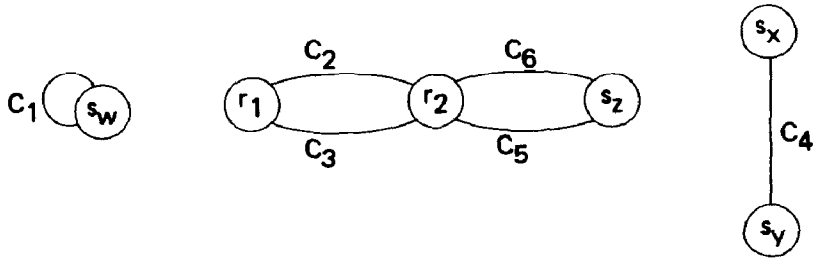

Figure 11.31

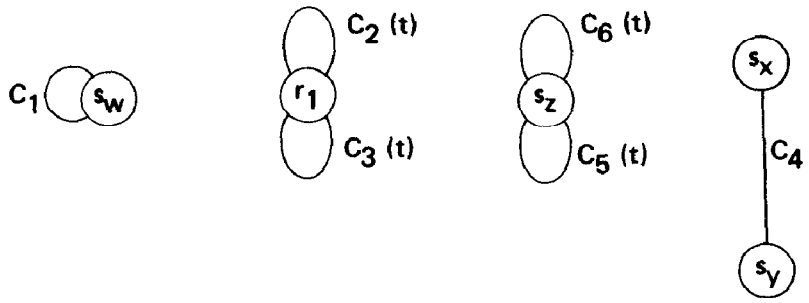

Figure 11.32

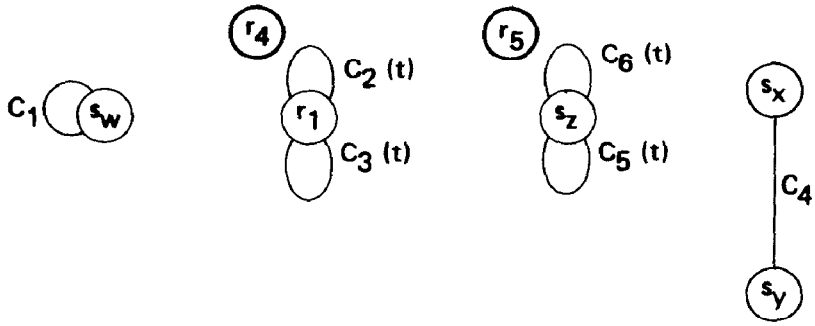

Figure 11.33

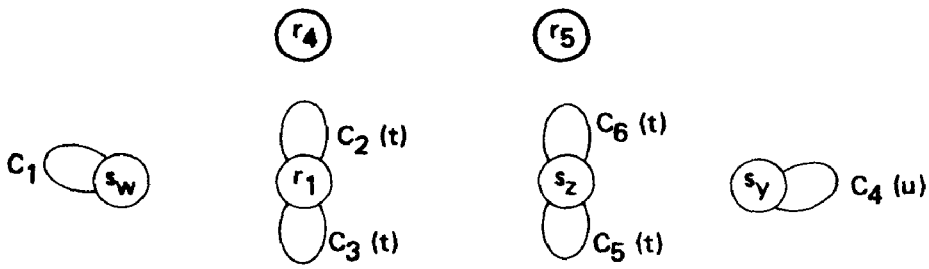

Figure 11.34

Our heuristic could be improved by several methods. One is to have different degrees of "small," depending upon the selection conditions involved. Another is to take into account the current sizes of relations in the database when making choices for instantiation and iteration. If the relative sizes of the relations change significantly, however, the decomposition may have to be redone.

11.5 TABLEAU QUERY OPTIMIZATION

The number of rows in a tableau query is a measure of the number of joins required in an equivalent algebraic expression. Given a tableau query, our goal will be to find an equivalent tableau query with the minimum number of rows. While in general such a search is NP-complete, the search is efficient for a subset of tableau queries: the *simple* tableau queries.

We shall also explore how to use data dependencies to reduce the number of rows in a tableau query. The method is a variant of the chase computation.

Many results in this section are similar to results in Chapter 8. We shall not spend much time on proofs that are similar to ones there. Our development will focus on untagged tableau queries. We shall discuss the modifications for tagged queries at the end of the section.

11.5.1 Tableau Query Equivalence

Definition 11.2 Let Q_1 and Q_2 be compatible tableau queries with scheme R. Q_1 *contains* Q_2, written $Q_1 \supseteq Q_2$, if for every relation $r(R)$, $Q_1(r) \supseteq Q_2(r)$. Q_1 and Q_2 are *equivalent*, written $Q_1 \equiv Q_2$, if for every relation $r(R)$, $Q_1(r) = Q_2(r)$. Evidently, $Q_1 \equiv Q_2$ if and only if $Q_1 \supseteq Q_2$ and $Q_2 \supseteq Q_1$.

In the following definition, a mapping of tableau symbols to tableau symbols extends to a mapping of rows to rows in the obvious way.

Definition 11.3 Let Q_1 be a tableau query on scheme $A_1\ A_2 \cdots A_n$ with summary w_0 and rows $w_1, w_2, \ldots, w_p$. Let Q_2 be a tableau query with summary v_0. Let a_i be the distinguished variable for the A_i-column in both tableaux. A mapping ψ from the symbols of Q_1 to the symbols of Q_2 is a *containment mapping* from Q_1 to Q_2 if

1. $\psi(c) = c$ for every constant c in Q_1,
2. $\psi(w_0) = v_0$, and
3. $\psi(w_i)$ is a row of Q_2 for $1 \leq i \leq p$.

Conditions 2 and 3 require that Q_1 and Q_2 are compatible. Condition 2 also requires that $\psi(a_i)$ be a_i or the constant in the A_i-column of v_0. We let $\psi(Q_1) = \{\psi(w_i) | 1 \leq i \leq p\}$.

Theorem 11.1 Let Q_1 and Q_2 be compatible tableau queries with scheme R. Let w_0 be the summary of Q_1 and v_0 the summary of Q_2. $Q_1 \supseteq Q_2$ if and only if there is a containment mapping ψ from Q_1 to Q_2.

Proof (if) Let r be a relation with scheme R. If ρ is a valuation of Q_2 such that $\rho(Q_2) \subseteq r$, then $\rho \circ \psi$ is a valuation of Q_1 where $\rho \circ \psi(Q_1) \subseteq r$ and $\rho \circ \psi(w_0) = \rho(v_0)$. Hence every tuple of $Q_2(r)$ is in $Q_1(r)$.

(only if) We treat Q_2 as both a relation and as a tableau query. When treating it as a relation, we ignore the summary. Since $Q_1 \supseteq Q_2$, $Q_1(Q_2) \supseteq Q_2(Q_2)$. Let ρ_I be the identity valuation for Q_2. Clearly $\rho_I(Q_2) \subseteq Q_2$, so $\rho_I(v_0) = v_0 \in Q_2(Q_2)$. Therefore, v_0 must be in $Q_1(Q_2)$. Let ψ be the valuation of Q_1 such that $\psi(Q_1) \subseteq Q_2$ and $\psi(w_0) = v_0$. Clearly, ψ is a containment mapping from Q_1 to Q_2.

Corollary Compatible tableau queries Q_1 and Q_2 are equivalent if and only if there is a containment mapping from Q_1 to Q_2 and another from Q_2 to Q_1.

Note that if Q_1 and Q_2 are equivalent, they necessarily have identical summaries, so the containment mappings in the corollary always map distinguished variables to distinguished variables.

$Q_1(A_1\ A_2\ A_3\ A_4)$

a_1		a_3	
a_1	b_1	b_2	b_3
b_4	b_1	a_3	4

$Q_2(A_1\ A_2\ A_3\ A_4)$

a_1		a_3	
a_1	b_1	b_2	b_3
b_4	b_1	a_3	4
a_1	b_1	b_5	b_6
b_7	b_1	b_8	4

$Q_3(A_1\ A_2\ A_3\ A_4)$

a_1		a_3	
a_1	b_1	b_2	b_3
6	b_1	a_3	4

Figure 11.35

Example 11.36 Consider tableau queries Q_1, Q_2, and Q_3 in Figure 11.35. Using the identity containment mapping from Q_1 to Q_2, we see $Q_1 \supseteq Q_2$. The containment mapping ψ_1 from Q_2 to Q_1 in Figure 11.36 shows that $Q_2 \supseteq Q_1$, hence $Q_1 \equiv Q_2$. The containment mapping ψ_2 from Q_1 to Q_3 in Figure 11.37 shows $Q_1 \supseteq Q_3$. However $Q_3 \not\supseteq Q_1$, since there is no containment mapping from Q_3 to Q_1. The second row of Q_3 contains a 6, and hence cannot be mapped to any row of Q_1.

$$\begin{array}{lll} \psi_1(a_1) = a_1 & & \psi_1(b_1) = b_1 \\ \psi_1(a_3) = a_3 & & \psi_1(b_2) = b_2 \\ \psi_1(4) = 4 & & \psi_1(b_3) = b_3 \\ & & \psi_1(b_4) = b_4 \\ & & \psi_1(b_5) = b_2 \\ & & \psi_1(b_6) = b_3 \\ & & \psi_1(b_7) = b_4 \\ & & \psi_1(b_8) = a_3 \end{array}$$

Figure 11.36

$$\begin{array}{lll} \psi_2(a_1) = a_1 & & \psi_2(b_1) = b_1 \\ \psi_2(a_3) = a_3 & & \psi_2(b_2) = b_2 \\ \psi_2(4) = 4 & & \psi_2(b_3) = b_3 \\ & & \psi_2(b_4) = 6 \end{array}$$

Figure 11.37

Corollary If tableau queries Q_1 and Q_2 are identical up to a one-to-one renaming of nondistinguished variables, then $Q_1 \equiv Q_2$.

The converse of this corollary is not true, as we can see from tableau queries Q_1 and Q_2 of Example 11.36.

The following definitions extend the definitions for subsumes (Section 8.3) and supersedes (Exercise 8.17).

Definition 11.4 Let w_1 and w_2 be rows over scheme R. Row w_1 *subsumes* row w_2 if for every attribute $A \in R$ such that $w_2(A)$ is a distinguished variable or constant, $w_1(A) = w_2(A)$.

Definition 11.5 Let T_1 and T_2 be sets of rows on scheme R. T_1 *covers* T_2 if for every row v in T_2 there exists a row w of T_1 that subsumes v. Tableau

query Q_1 *covers* tableau query Q_2 if the rows of Q_1 cover the rows of Q_2. We write $Q_1 \simeq Q_2$ if Q_1 covers Q_2 and Q_2 covers Q_1.

Definition 11.6 Let Q be a tableau query with scheme R and let w_1 and w_2 be rows of Q. Row w_1 *supersedes* row w_2 if $w_1(A) = w_2(A)$ for every attribute A in $match(w_2)$.

Example 11.37 Let Q be the tableau query of Figure 11.38. Row w_2 subsumes row w_4, but does not supersede it. Row w_3 supersedes row w_4. Q covers tableau query Q_3 in Figure 11.35.

Q	A_1	A_2	A_3	A_4
w_0	a_1	a_2	a_3	
w_1	a_1	a_2	b_1	b_2
w_2	6	a_2	a_3	4
w_3	b_3	a_2	a_3	b_2
w_4	b_5	a_2	b_6	b_2

Figure 11.38

Lemma 11.1 Let Q_1 be a tableau query. Let Q_2 be Q_1 with one or more superseded rows removed. $Q_1 \equiv Q_2$.

Proof Left to the reader (see Exercise 11.13).

Example 11.38 In tableau query Q_2 of Figure 11.35, the first row supersedes the third and the second supersedes the fourth. Q_1 is Q_2 with the third and fourth rows removed. We saw in Example 11.35 that $Q_1 \equiv Q_2$.

Definition 11.7 A tableau query Q is *minimum* if no tableau query equivalent to Q has fewer rows than Q.

Note that a minimum equivalent tableau query for a tableau query Q is the same as an equivalent minimum tableau query for Q.

Definition 11.8 Let Q_1 and Q_2 be compatible tableau queries. Q_2 is a *subtableau* of Q_1 if Q_2 is Q_1 with 0 or more rows removed.

Theorem 11.2 For any tableau query Q_1, there is a subtableau Q_2 of Q_1 that is a minimum equivalent tableau for Q_1.

Proof Let Q_3 be a minimum equivalent tableau for Q_1. Let ψ_{13} be a containment mapping from Q_1 to Q_3 and let ψ_{31} be a containment mapping from Q_3 to Q_1. Let Q_2 be the subtableau of Q_1 containing the rows in $\psi_{31}(\psi_{13}(Q_1))$. The composition $\psi_{31} \circ \psi_{13}$ is a containment mapping from Q_1 to Q_2; $\psi_{13} \circ \psi_{31}$ is a containment mapping from Q_2 to Q_1. Hence $Q_1 \equiv Q_2$. Since $\psi_{13}(Q_1)$ is a subset of the rows of Q_3, $\psi_{31}(\psi_{13}(Q_1))$ has no more rows than Q_3, hence Q_2 is also minimum.

The next result is a partial converse to the second corollary of Theorem 11.1.

Theorem 11.3 Let Q_1 and Q_2 be compatible minimum tableau queries. $Q_1 \equiv Q_2$ if and only if there is a one-to-one containment mapping from Q_1 to Q_2 whose inverse is a containment mapping from Q_2 to Q_1. That is, Q_1 and Q_2 are identical up to a one-to-one renaming of nondistinguished variables.

Proof Left to the reader (see Exercise 11.14).

In general, testing equivalence of tableau queries is hard. It is an NP-complete problem given tableau queries Q_1 and Q_2 to decide whether $Q_1 \equiv Q_2$. The problem is NP-complete even if Q_1 and Q_2 come from restricted algebraic expressions and Q_2 is a subtableau of Q_1.

11.5.2 Simple Tableau Queries

In this section we introduce the simple tableau queries and show that they can be minimized efficiently. We shall also show that equivalence of minimum simple queries can be decided efficiently, which means equivalence can be decided efficiently for arbitrary simple queries.

Definition 11.9 A tableau query Q is *simple* if in any column where a nondistinguished variable is matched, no other symbol is repeated.

Example 11.39 Consider tableau queries Q_1 and Q_2 in Figure 11.39. Q_1 is not simple, because b_4 is matched in the A_2-column and a_2 repeats, and b_3 is matched in the A_4-column and 4 repeats. Q_2 is the same as Q_1 except for the last row, but Q_2 is simple.

Q_1

A_1	A_2	A_3	A_4
a_1	a_2	a_3	
a_1	a_2	b_1	4
b_2	a_2	a_3	b_3
a_1	b_4	a_3	b_3
a_1	b_4	b_5	4

Q_2

A_1	A_2	A_3	A_4
a_1	a_2	a_3	
a_1	a_2	b_1	4
b_2	a_2	a_3	b_3
a_1	b_4	a_3	b_3
a_1	a_2	b_5	b_3

Figure 11.39

Our task is to show that simple tableau queries can be minimized efficiently and equivalence of minimum simple queries is easy to test, thus deriving an efficient algorithm for equivalence of simple queries.

Let Q_1 be a simple tableau query that is not minimum. By Theorem 11.2 there is an equivalent subtableau Q_2 of Q_1 that is equivalent to Q_1. Let w be a row of Q_1 that does not appear in Q_2. There is a containment mapping ψ from Q_1 to Q_2 and a row v in Q_2 such that $\psi(w) = \psi(v) = v$ (see Lemma 11.2 below).

Given a simple tableau query Q_1, in order to minimize it, we try to find rows w, v, and a containment mapping ψ from Q_1 to Q_1 such that $\psi(w) = \psi(v) = v$. To aid in the search, we compute the *companion set of* w *relative to* v, $COMP_v(w)$, that contains all the rows ψ must map to v if ψ maps w to v. If $\{v\}$ covers $COMP_v(w)$, then ψ can be chosen to map every row in $COMP_v(w)$ to v and every other row to itself. The subtableau Q_2 of Q_1 containing the rows in $\psi(Q_1)$ is equivalent to Q_1 but has fewer rows. If Q_2 is not minimum, we can repeat the process with a new w, v and ψ.

We now fill in the details and prove the correctness of the method above.

Lemma 11.2 Let Q_1 be a tableau query, and let Q_2 be a proper subtableau of Q_1. If ψ is a containment mapping from Q_1 to Q_2 and w is a row in Q_1 not in Q_2, then there is a containment mapping ψ_0 from Q_1 to Q_2 and a row v of Q_2 such that $\psi_0(w) = \psi_0(v) = v$.

Proof We first note that composition of containment mappings is a containment mapping. Let ψ^i be the containment mapping obtained by composing ψ with itself i times. Consider the sequence of rows $\psi^1(w)$, $\psi^2(w)$, $\psi^3(w)$, ... Since each row is contained in Q_1, the sequence must contain a duplicate. Select i and j such that $i \leq j$ and $\psi^i(w) = \psi^j(w)$. Let $\psi_0 = \psi^{i(j-i)}$ and let $v = \psi_0(w)$. We show that $\psi_0(v) = v$.
First,

$$\psi_0(v) = \psi^{i(j-i)}(v) = \psi^{2i(j-i)}(w).$$

We can rewrite $\psi^{2i(j-i)}(w)$ as

$$\psi^{i(j-i-1)}(\psi^{i(j-i)}(\psi^i(w))),$$

which simplifies to

$$\psi^{(j-i-1)}(\psi^i(w)),$$

since

$$\psi^{(j-i)}(\psi^i(w)) = \psi^j(w) = \psi^i(w).$$

Finally,

$$\psi^{i(j-i-1)}(\psi^i(w)) = \psi^{i(j-i)}(w) = \psi_0(w) = v.$$

Definition 11.10 Let Q be a simple tableau query with scheme R, and let w and v be rows of Q. The *companion set of* w *relative to* v, denoted $COMP_v(w)$, is the smallest set T containing v subject to the following closure condition:

> If w_1 is a row in T, w_2 is a row of Q, and A is an attribute in R such that $w_1(A)$ is a nondistinguished variable and $w_1(A) = w_2(A) \neq v(A)$, then w_2 is in T.

Example 11.40 Let Q be the tableau in Figure 11.40. Let us compute $COMP_{w_3}(w_4)$. We start with row w_4. For row w_5 we have $w_5(A_4) = w_4(A_4) = b_{11} \neq w_3(A_4)$, so w_5 is included. Also, $w_5(A_5) = w_6(A_5) = b_{16} \neq w_3(A_5)$, so w_6 is included. Hence, $COMP_{w_3}(w_4) = \{w_4, w_5, w_6\}$. Row w_2 is not included, even though $w_2(A_2) = w_6(A_2) = b_1$, since $w_3(A_2) = b_1$.

	$Q(A_1$	A_2	A_3	A_4	$A_5)$
w_0	a_1		a_3		
w_1	a_1	b_1	b_2	4	b_3
w_2	a_1	b_1	b_4	b_5	b_6
w_3	b_7	b_1	a_3	b_8	7
w_4	b_9	b_{10}	a_3	b_{11}	b_{12}
w_5	b_{13}	b_{14}	b_{15}	b_{11}	b_{16}
w_6	b_{17}	b_1	b_{18}	b_{19}	b_{16}

Figure 11.40

Lemma 11.3 Let Q be a simple tableau query and let w and v be rows of Q. If ψ is a containment mapping from Q to Q such that $\psi(w) = v$, then $\psi(u) = v$ for every row u in $COMP_v(w)$.

Proof Left to the reader (see Exercise 11.23).

Theorem 11.4 Let Q be a simple tableau query with scheme R and let w and v be distinct rows of Q. There is a containment mapping ψ from Q to Q such that $\psi(u) = v$ for all u in $COMP_v(w)$ if and only if $\{v\}$ covers $COMP_v(w)$.

Proof (if) Let ψ be the identity on all symbols except nondistinguished variables appearing in the rows of $COMP_v(w)$. For each attribute $A \in R$ and each row u in $COMP_v(w)$, let $\psi(u(A)) = \psi(v(A))$. (Since v and w are distinct, $COMP_v(w)$ does not contain v. Why?)

We claim ψ is well-defined. Let b be a nondistinguished variable in the A-column. If b appears both in a row u_1 in $COMP_v(w)$ and a row u_2 not in $COMP_v(w)$, then $u_1(A) = u_2(A) = v(A)$ or else u_2 would be in $COMP_v(w)$. Hence $\psi(b) = b$, which implies ψ is the identity on rows not in $COMP_v(w)$. Also, if some row u in $COMP_v(w)$ has a distinguished variable or a constant in the A-column, so does v, since v subsumes u. Hence ψ is the identity on distinguished symbols and constants.

It is not hard to verify that $\psi(u) = u$ if $u \notin COMP_v(w)$ and $\psi(u) = v$ if $u \in COMP_v(w)$. Hence ψ is the desired containment mapping.

(only if) Since ψ is a containment mapping, it maps constants to themselves. It must also be the identity on distinguished variables, since if Q contains a distinguished variable in the A-column, there is no constant in the

A-column of the summary. For any row $u \in COMP_v(w)$, since $\psi(u) = v$, every place u has a distinguished variable or constant, v has the same symbol. Thus v subsumes u and so $\{v\}$ covers $COMP_v(w)$.

Example 11.41 As we saw in Example 11.40, for the tableau query Q in Figure 11.40, $COMP_{w_3}(w_4) = \{w_4, w_5, w_6\}$. We see that $\{w_3\}$ covers $\{w_4, w_5, w_6\}$. Hence, there is a containment mapping ψ from Q to Q such that $\psi(w_1) = w_1$, $\psi(w_2) = w_2$, and $\psi(w_3) = \psi(w_4) = \psi(w_5) = \psi(w_6) = w_3$.

$COMP_{w_6}(w_3) = \{w_3, w_4, w_5\}$. However $\{w_6\}$ does not cover $\{w_3, w_4, w_5\}$. No containment mapping maps w_3 to w_6, since $w_3(A_5) = 7$ and $w_6(A_5) = b_{16}$.

We combine our results into the algorithm MINEQ in Figure 11.41. Given a simple tableau query Q, we search for distinct rows w and v such that $\{v\}$ covers $COMP_v(w)$. By Lemmas 11.2, 11.3, and Theorem 11.4, if Q is not minimum, v and w will exist. The rows of $COMP_v(w)$ can be removed from Q to get a smaller equivalent tableau. If no such w and v exist, Q must be minimum.

```
Input:  A simple tableau query Q.
Output: A minimum equivalent subtableau of Q.
MINEQ(Q)
  begin
  let T be the rows of Q;
  while changes to T occur do
    for each row v in T do
      for each row w ≠ v in T do
        if v covers COMP_v(w) in T
        then T := T − COMP_v(w);
  let Q' be T with the summary of Q;
  return (Q')
  end.
```

Figure 11.41

Example 11.42 Let Q_1 be the tableau query in Figure 11.42. $COMP_{w_1}(w_2) = \{w_2\}$, and $\{w_1\}$ covers $\{w_2\}$, so we remove row w_2. $COMP_{w_1}(w_3) = \{w_3\}$, and $\{w_1\}$ covers $\{w_3\}$, so we also remove w_3. We are left with rows w_1, w_4, w_5, w_6, w_7. $COMP_{w_5}(w_6) = \{w_6, w_7\}$ and is covered by $\{w_5\}$, so we remove w_6 and w_7. No more rows can be removed. The minimum equivalent subtableau Q_1' is shown in Figure 11.43.

Q_1	A_1	A_2	A_3	A_4	A_5
w_0	a_1			7	a_5
w_1	a_1	b_1	5	b_2	b_3
w_2	a_1	b_1	b_4	b_5	b_6
w_3	a_1	b_7	b_8	b_9	b_{10}
w_4	b_{11}	b_1	b_{12}	7	b_{14}
w_5	b_{15}	b_{16}	b_{17}	7	a_5
w_6	b_{18}	b_{19}	b_{20}	7	b_{21}
w_7	b_{22}	b_{23}	b_{20}	b_{24}	a_5

Figure 11.42

Q_1'	A_1	A_2	A_3	A_4	A_5
w_0	a_1			7	a_5
w_1	a_1	b_1	5	b_2	b_3
w_4	b_{11}	b_1	b_{12}	7	b_{14}
w_5	b_{15}	b_{16}	b_{17}	7	a_5

Figure 11.43

Let us determine the time complexity of MINEQ. Assume Q has k rows and n columns. $COMP_v(w)$ can be found in $O(k^2n)$ time (see Exercise 11.24). Testing whether v covers $COMP_v(w)$ can be done in $O(kn)$ time. Each **for**-loop iterates at most k times, so the body of the **while**-loop takes $O(k^4n)$ time. Each time a change occurs to T, at least one row is removed, so the **while**-loop iterates for at most k times. Hence the total time-complexity of *MINEQ* is $O(k^5n)$.

Actually, the time-complexity is no more than $O(k^4n)$. The **while**-loop need only execute once, for if a row is not removed on the first iteration, it will not be removed on any subsequent iteration, as the following lemma shows.

Definition 11.11 Let Q be a simple tableau query with scheme R and let v and w be rows of Q. Let $K = w_1, w_2, \ldots, w_m$ be a sequence of distinct rows for Q. K is a *construction sequence* for $COMP_v(w)$ if

1. $COMP_v(w) = \{w_1, w_2, \ldots, w_m\}$, and
2. For each j, $1 < j \leq m$, there is an $i < j$ and an attribute A in R such that $w_i(A)$ and $w_j(A)$ are the same nondistinguished variable and $w_j(A) \neq v(A)$.

Clearly, every companion set has a construction sequence. The first row in the sequence may be any row of the companion set.

Lemma 11.4 Let Q be a simple tableau query with scheme R. Assume Q contains distinct rows v_0 and w_0 such that $COMP_{v_0}(w_0)$ is covered by $\{v_0\}$. Let Q' be the subtableau obtained from Q by removing the rows in $COMP_{v_0}(w_0)$. (That is, Q' is Q after one step of MINEQ.) Let v and w be rows of Q' and let $COMP_v(w)$ and $COMP'_v(w)$ be the companion sets of w relative to v in Q and Q', respectively. If $\{v\}$ covers $COMP'_v(w)$, then $\{v\}$ covers $COMP_v(w)$.

Proof We focus on the identities of v_0 and w_0. Note that if w' is a row in $COMP_v(w)$, then $COMP_v(w) = COMP_v(w')$ (see Exercise 11.26).

$COMP_{v_0}(w_0)$ must contain some row u in $COMP_v(w)$, or else $COMP_v(w) = COMP'_v(w)$ and we are finished. If $v = v_0$, then $COMP_{v_0}(w_0) = COMP_{v_0}(u) = COMP_v(u) = COMP_v(w)$, so w would not be a row of Q'. Thus, the rows removed going from Q to Q' are $COMP_{v_0}(u)$ where $v_0 \neq v$.

We claim v_0 is in $COMP_v(w)$. Why? First, note that $COMP_{v_0}(u) \not\supseteq COMP_v(w)$, or else w would not be a row of Q'. Let $u_1, u_2, \ldots, u_m$ be a construction sequence for $COMP_v(w)$ such that $u_1 = u$. Let j be the smallest integer such that u_j is not in $COMP_{v_0}(u)$. There must be an $i < j$ and an attribute A in R such that $u_i(A)$ and $u_j(A)$ are the same nondistinguished variable and $u_i(A) \neq v(A)$. We must have $u_i(A) = u_j(A) = v_0(A)$, or else u_j would be in $COMP_{v_0}(u)$. We know $u_i(A)$ and $v_0(A)$ are the same nondistinguished symbol. Since $u_i(A)$ is in $COMP_v(w)$ and $v_0(A) = u_i(A) \neq v(A)$, v_0 is also in $COMP_v(w)$, as we claimed.

We next show that v_0 is in $COMP'_v(w)$. Let $w_1, w_2, \ldots, w_p$ be a prefix of a generating sequence for $COMP_v(w)$ such that $w = w_1$ and $v_0 = w_p$. If all of $w_1, w_2, \ldots, w_p$ are in Q', we have proved that v_0 is in $COMP'_v(w)$, so let j, $z < j \leq p$, be the least integer such that w_j is not a row of Q'. There must be an $i < j$ and an attribute B in R such that $w_i(B)$ and $w_j(B)$ are the same nondistinguished symbol, which is not $v(B)$. We know w_i was not in $COMP_{v_0}(u)$ but that w_j was. We conclude $w_i(B) = w_j(B) = v_0(B)$. All of $w_1, w_2, \ldots, w_i$ are in $COMP'_v(w)$ and $v_0(B) = w_i(B) \neq v(B)$, so v_0 is also in $COMP'_v(w)$, as desired.

To conclude, any row u_0 in $COMP'_v(w)$ must be in $COMP_{v_0}(u)$. Since v_0 is in $COMP'_v(w)$ and $\{v\}$ covers $COMP'_v(w)$, v subsumes u_0 and so $\{v\}$ covers $COMP_v(w)$.

We can use MINEQ to construct an efficient test for equivalence. Let Q_1 and Q_2 be simple tableau queries with scheme R that we wish to test for equivalence. We first compute $Q_1' = \text{MINEQ}(Q_1)$ and $Q_2' = \text{MINEQ}(Q_2)$, and then test Q_1' and Q_2' for equivalence. Q_1' and Q_2' are both minimum. If Q_1' and Q_2' are equivalent, by Theorem 11.3 there is a one-to-one containment mapping ψ from Q_1' to Q_2' that is the identity on distinguished variables and

whose inverse is a containment mapping from Q_1' to Q_2'. If Q_1' has a repeated nondistinguished variable b_1 in the A-column, for some attribute A in R, then $\psi(b_1) = b_2$ must be a repeated nondistinguished variable in the A-column of Q_2'. If we formed Q_1'' and Q_2'' by replacing b_1 and b_2 by a new constant c in Q_1' and Q_2', then Q_1'' and Q_2'' will also be equivalent.

To test Q_1' and Q_2' for equivalence, we proceed as follows. For each attribute A such that Q_1' has a repeated nondistinguished variable b_1 in the A-column, we check if Q_2' has a repeated nondistinguished variable, say b_2, in its A-column. If so, we replace b_1 and b_2 by some new constant c. If not, Q_1' and Q_2' are not equivalent. We attempt such a substitution for every repeated nondistinguished variable in Q_1'. Call the resulting tableau queries Q_1'' and Q_2''.

Q_1' and Q_2' are equivalent if and only if Q_1'' and Q_2'' are. If Q_2'' has any repeated, nondistinguished variables, then it is not equivalent to Q_1''. Otherwise, we can test $Q_1'' \equiv Q_2''$ by checking that the summaries are the same and that $Q_1'' \simeq Q_2''$. (Recall that $Q_1'' \simeq Q_2''$ means each tableau query covers the other.)

Example 11.43 Let Q_1 be the tableau query in Figure 11.42 and let Q_2 be the tableau query in Figure 11.44. $Q_1' = \text{MINEQ}(Q_1)$ is shown in Figure 11.43. $Q_2' = \text{MINEQ}(Q_2)$ is shown in Figure 11.45. Q_1' has repeated nondistinguished variable b_1 in the A_2-column, and Q_2' has b_2 repeated in the same column. We replace b_1 and b_2 by the constant 1 to get tableau queries Q_1'' and Q_2'' in Figure 11.46. Q_1'' and Q_2'' are equivalent, for we can map rows w_1, w_4, and w_5 to rows v_5, v_1, and v_4 by renaming only nondistinguished variables. Hence $Q_1 \equiv Q_2$.

For tableau query Q_3 in Figure 11.46, $Q_3 = \text{MINEQ}(Q_3)$. $Q_1 \not\equiv Q_3$, since Q_3 has a repeated nondistinguished variable in the A_4-column, where Q_1' has none.

Q_2	A_1	A_2	A_3	A_4	A_5
v_0	a_1			7	a_5
v_1	b_1	b_2	b_3	7	b_4
v_2	b_5	b_2	b_6	7	b_7
v_3	b_8	b_9	b_6	7	b_{10}
v_4	b_{11}	b_{12}	b_{13}	7	a_5
v_5	a_1	b_2	5	b_{14}	b_{15}

Figure 11.44

$Q_2'(A_1$	A_2	A_3	A_4	$A_5)$
v_0 a_1			7	a_5
v_1 b_1	b_2	b_3	7	b_4
v_4 b_{11}	b_{12}	b_{13}	7	a_5
v_5 a_1	b_2	5	b_{14}	b_{15}

Figure 11.45

$Q_1''(A_1$	A_2	A_3	A_4	$A_5)$
w_0 a_1			7	a_5
w_1 a_1	1	5	b_2	b_3
w_4 b_{11}	1	b_{12}	7	b_{14}
w_5 b_{15}	b_{16}	b_{17}	7	a_5

$Q_2''(A_1$	A_2	A_3	A_4	$A_5)$
v_0 a_1			7	a_5
v_1 b_1	1	b_3	7	b_4
v_4 b_{11}	b_{12}	b_{13}	7	a_5
v_5 a_1	1	5	b_{14}	b_{15}

Figure 11.46

$Q_3(A_1$	A_2	A_3	A_4	$A_5)$
a_1			7	a_5
b_1	b_2	b_3	b_4	b_5
b_6	b_7	b_8	b_4	a_5
a_1	b_2	5	b_9	b_{10}

Figure 11.47

11.5.3 Equivalence with Constraints

As we might expect, in the presence of FDs and JDs otherwise inequivalent tableau queries can be equivalent.

Example 11.44 Tableau query Q_1 in Figure 11.48 comes from Example 10.38 in the last chapter. Q_2 in Figure 11.49 is another tableau query on the

same scheme. $Q_1 \supseteq Q_2$, since there is a containment mapping that takes the first two rows of Q_1 to the first row of Q_2 and the last row of Q_1 to the last row of Q_2. However, $Q_2 \not\supseteq Q_1$. Both Q_1 and Q_2 are simple, and both are left unchanged by MINEQ hence both are minimum. Since Q_1 has more rows than Q_2, $Q_1 \not\equiv Q_2$.

Suppose we are only interested in relations that satisfy the FD OP $\rightarrow$ ME. No option is available for more than one meal. (No more pizza for breakfast.) Let r be a relation in SAT(OP $\rightarrow$ ME). Any valuation ρ for Q_1 such that $\rho(Q_1) \subseteq r$ must have $\rho(b_4) = \rho(b_5)$. Hence $\rho(Q_2) \subseteq r$ and so $Q_1(r) \subseteq Q_2(r)$. Q_1 and Q_2 define the same mapping on SAT(OP $\rightarrow$ ME).

Q_1(FL	DT	OP	NM	ME)
	a_2	a_3	a_4	
b_1	b_2	a_3	b_3	b_4
106	a_2	a_3	a_4	b_5
107	b_6	b_7	b_8	b_4

Figure 11.48

Q_2(FL	DT	OP	NM	ME)
	a_2	a_3	a_4	
106	a_2	a_3	a_4	b_4
107	b_6	b_7	b_8	b_4

Figure 11.49

If Q_1 and Q_2 are compatible tableau queries on scheme R and **C** is a set of FDs and JDs, $Q_1 \supseteq_{\mathbf{C}} Q_2$ means $Q_1(r) \supseteq Q_2(r)$ for every relation $r(R)$ in SAT(**C**). Similarly, $Q_1 \equiv_{\mathbf{C}} Q_2$ means $Q_1(r) = Q_2(r)$ for every relation in SAT(**C**).

We can extend the chase computation to a tableau query Q with two slight modifications. First, the F-rule gives priority to constants in renaming. Suppose we have an FD $X \rightarrow A$ and two rows w_1 and w_2 where $w_1(X) = w_2(X)$. If $w_1(A)$ and $w_2(A)$ are unequal constants, we replace the entire tableau query by the tableau query $\emptyset$ that maps every relation to the empty relation. Clearly, for any relation $r \in SAT(X \rightarrow A)$, there can be no valuation ρ such that $\rho(Q) \subseteq r$. If $w_1(A)$ is a constant and $w_2(A)$ is not, we set $w_2(A)$ to be that

constant. The second modification is that if ever a distinguished variable is changed to a constant, the change carries through to the summary.

We let $chase_{\mathbf{C}}(Q)$ be the result of applying F- and J-rules for **C** to Q until no more changes can be made. The proofs of Chapter 8 work with minor changes to show that $chase_{\mathbf{C}}(Q)$ is thereby well-defined. We also state the following two theorems without proof.

Theorem 11.5 Let Q be a tableau query and let **C** be a set of FDs and JDs. $Q \equiv_{\mathbf{C}} chase_{\mathbf{C}}(Q)$.

Theorem 11.6 Let Q_1 and Q_2 be tableau queries and let **C** be a set of FDs and JDs. $Q_1 \supseteq_{\mathbf{C}} Q_2$ if and only if $chase_{\mathbf{C}}(Q_1) \supseteq chase_{\mathbf{C}}(Q_2)$.

Corollary $Q_1 \equiv_{\mathbf{C}} Q_2$ if and only if $chase_{\mathbf{C}}(Q_1) \equiv chase_{\mathbf{C}}(Q_2)$.

Example 11.45 Returning to the last example, if $\mathbf{C} = \{\text{OP} \rightarrow \text{ME}\}$, then $Q_1^* = chase_{\mathbf{C}}(Q_1)$ is shown in Figure 11.50. $Q_2 = chase_{\mathbf{C}}(Q_2)$. Also, MINEQ $(Q_1^*) = Q_2$, so $Q_1 \equiv_{\mathbf{C}} Q_2$.

Q_1^*(FL	DT	OP	NM	ME)
	a_2	a_3	a_4	
b_1	b_2	a_3	b_3	b_4
106	a_2	a_3	a_4	b_4
107	b_6	b_7	b_8	b_4

Figure 11.50

Example 11.46 Tableau queries Q_1 and Q_2 in Figure 11.51 clearly are not equivalent, since their summaries are different. If $\mathbf{C} = \{A_1 \rightarrow A_2, A_4 \twoheadrightarrow A_1\}$, then $Q_1^* = chase_{\mathbf{C}}(Q_2)$ are shown in Figure 11.52. The third row of Q_1^* supersedes all the rest, so $Q_1^* \equiv Q_3$, where Q_3 is shown in Figure 11.53. Similarly, the first row of Q_2^* supersedes the last, so $Q_2^* \equiv Q_3 \equiv Q_1^*$. Hence $Q_1 \equiv_{\mathbf{C}} Q_2$.

Definition 11.12 Let **C** be a set of FDs and JDs. A tableau query Q_1 is **C**-*minimum* if there is no tableau query Q_2 with fewer rows than Q_1 such that $Q_1 \equiv_{\mathbf{C}} Q_2$.

Certainly, finding a **C**-minimum equivalent tableau query given a tableau query Q and a set **C** of FDs and JDs is no easier than finding a minimum

$Q_1(A_1$	A_2	A_3	$A_4)$
a_1	a_2	a_3	
a_1	a_2	b_1	b_2
b_3	7	a_3	b_2

$Q_2(A_1$	A_2	A_3	$A_4)$
a_1	7	a_3	
a_1	b_1	a_3	b_2
a_1	7	b_3	b_4

Figure 11.51

$Q_1^*(A_1$	A_2	A_3	$A_4)$
a_1	7	a_3	
a_1	7	b_1	b_2
b_3	7	a_3	b_2
a_1	7	a_3	b_2
b_3	7	b_1	b_2

$Q_2^*(A_1$	A_2	A_3	$A_4)$
a_1	7	a_3	
a_1	7	a_3	b_2
a_1	7	b_3	b_4

Figure 11.52

$Q_3(A_1$	A_2	A_3	$A_4)$
a_1	7	a_3	
a_1	7	a_3	b_2

Figure 11.53

equivalent tableau query. We might hope to combine MINEQ and the chase computation to get an algorithm for C-minimum equivalence of simple tableau queries. If Q is a simple tableau query, we cannot necessarily apply MINEQ to $chase_C(Q)$. Note that in Example 11.46, Q_1 was simple, but $chase_C(Q_1)$ was not. The problem can arise even with FDs alone, and remov-

ing superseded rows does not always restore simplicity (see Exercise 11.29). Applying the chase to MINEQ(Q) does not necessarily yield the desired result either (see Exercise 11.30).

11.5.4 Extensions for Multiple-Relation Databases

We first consider a database d on database scheme **R** over **U** where every relation is the projection of a common instance r(**U**). Our results on equivalence carry over easily, since if a tableau query Q applies to d, then so does any minimum equivalent query for Q.

Example 11.47 Let d be the database $\{q(AB), r(BC), s(AC)\}$ that is the projection of some common instance over ABC. Figure 11.54 shows a tableau query Q_1 for the algebraic expression

$$\pi_{BC}(\sigma_{B=c}(q) \bowtie r \bowtie s).$$

Q_1 is simple, so we may apply MINEQ to get tableau query Q_1' in Figure 11.55. The algebraic expression above is equivalent to $\sigma_{B=c}(r)$. However, the tableau query Q_2 in Figure 11.56 for the algebraic expression

$$\pi_{BC}(\sigma_{A=c}(q) \bowtie r \bowtie s)$$

is minimum.

$Q_1(A$	B	$C)$
	c	a_3
b_1	c	a_3
b_2	c	b_4
b_2	b_5	a_3

Figure 11.54

$Q_1'(A$	B	$C)$
	c	
b_1	c	a_3

Figure 11.55

$Q_2(A$	B	$C)$
	a_2	a_3
b_1	a_2	a_3
c	a_2	b_2
c	b_3	a_3

Figure 11.56

When we introduce dependencies, problems arise, for $chase_C(Q)$ might not apply to d, even if Q did. We might interpret our result $Q \equiv_C chase_C(Q)$ as a statement about equivalence of queries over different databases that are projections of the same instance.

Example 11.48 Let $\{q(AB), r(BC), s(CD)\}$ and $d' = \{q'(ABC), r'(BCD)\}$ both be databases that are the projections of the same instance over $A\ B\ C\ D$. Let $\mathbf{C} = \{C \rightarrow D\}$. Q in Figure 11.57 is the tableau query for the algebraic expression

$$\pi_{AD}(q \bowtie r \bowtie s)$$

on database d. $Q' = chase_C(Q)$ is shown in Figure 11.58 and $Q'' =$ MINEQ (Q') is shown in Figure 11.59. Neither Q' nor Q'' applies to d. However, both apply to d'. We can interpret $Q \equiv Q'$ as saying the algebraic expression

$$\pi_{AD}(\pi_{AB}(q') \bowtie r')$$

for database d' is equivalent to the expression above for database d whenever instance r is in $SAT(\mathbf{C})$.

$Q(A$	B	C	$D)$
a_1			a_4
a_1	b_1	b_2	b_3
b_4	b_1	b_5	b_6
b_7	b_8	b_5	a_4

Figure 11.57

$Q'(A$	B	C	$D)$
a_1			a_4
a_1	b_1	b_2	b_3
b_4	b_1	b_5	a_4
b_7	b_8	b_5	a_4

Figure 11.58

$Q''(A$	B	C	$D)$
a_1			a_4
a_1	b_1	b_2	b_3
b_4	b_1	b_5	a_4

Figure 11.59

When **C** consists of only FDs, there is another way to interpret $chase_{\mathbf{C}}(Q)$. The rows that do not correspond to any relation for database d can be regarded as corresponding to joins of relations, provided that F-rules were applied in a certain restricted manner. The joins are ones that can be computed efficiently, so any minimization that takes place in computing $chase_{\mathbf{C}}(Q)$ can be viewed as replacing arbitrary joins by efficient joins. We now take up this type of join.

Let d be a database over database scheme R that is the projection of a common instance $r(U)$. Let F be a set of FDs that r satisfies. If $r_1(XY)$ and $r_2(YZW)$ are two relations in d such that $XY \cap YZW = Y$, and $Y \rightarrow Z$ is an FD in F^+, then the r_2-*extension of* r_1 *by* $Y \rightarrow Z$ is $r_1 \bowtie \pi_{YZ}(r_2)$. The r_2-extension of r_1 by $Y \rightarrow Z$ has the same number of tuples as r_1 and can be computed by a single pass through r_1 and r_2 if both relations are sorted on Y. Note that $r_1 \bowtie \pi_{YZ}(r_2) = \pi_{XYZ}(r)$. Such a join, where the common attributes of the two relations functionally determine all the attributes of one of the relations, is called an *extension join*.

A subset R of **U** is an $\mathbf{R}_F$-extension if $\pi_R(r)$ can be computed from database d only using extension joins and projection. That is, R is a subscheme of some relation in d or there is a program P of the form

$$
\begin{aligned}
&q_1 \leftarrow s_1 \bowtie \pi_{Y_1Z_1}(s_1');\\
&q_2 \leftarrow s_2 \bowtie \pi_{Y_2Z_2}(s_2');\\
&\vdots\\
&q_k \leftarrow s_k \bowtie \pi_{Y_kZ_k}(s_k')
\end{aligned}
$$

where

1. s_i is either a relation in d or q_j for $j < i$; the same for s_i',
2. q_i is the s_i'-extension of s_i by $Y_i \rightarrow Z_i$ for some FD $Y_i \rightarrow Z_i$ in F^+, and
3. the scheme of q_k is $R' \supseteq R$.

(See Exercise 11.34.)

Example 11.49 Let $d = \{r_1(AB), r_2(BC), r_3(CD), r_4(BDEI)\}$ be a database over database scheme $\mathbf{R} = \{AB, BC, CD, BDEI\}$. Assume d is the projection of an instance $r(A\ B\ C\ D\ E\ I)$ in $SAT(F)$, where $F = \{C \rightarrow D, BD \rightarrow E\}$. CE is an $\mathbf{R}_F$-extension, for consider the program

$$q_1 \leftarrow r_2 \bowtie r_3;$$
$$q_2 \leftarrow q_1 \bowtie \pi_{BDE}(r_4).$$

Both joins are extension joins and the scheme of q_2 is $BCDE$, which contains CE. AE is not an $\mathbf{R}_F$-extension. A only appears in relation r_1, which can never participate in an extension join because none of A, B, and AB appear as the left side of a nontrivial FD in F^+.

Given a set of FDs F and a database scheme $\mathbf{R}$ over $\mathbf{U}$, we can modify the chase computation to decide whether some set $R \subseteq \mathbf{U}$ is an $\mathbf{R}_F$-extension. The modification is that an F-rule cannot be applied to equate two nondistinguished variables. However, any FD in F^+ may be used as the basis for an F-rule. We call this computation the *extension chase with respect to* F, denoted $echase_F$.

Theorem 11.7 Let F be a set of FDs and let $\mathbf{R}$ be a database scheme over attributes $\mathbf{U}$. Let $T_{\mathbf{R}}$ be the tableau for $\mathbf{R}$. Let $R \subseteq \mathbf{U}$. If $echase_F(T_{\mathbf{R}})$ has a row that is distinguished on all the attributes in R (and possibly more), then R is an $\mathbf{R}_F$-extension.

Proof Initially, if some row in $T_{\mathbf{R}}$ is distinguished on the attributes in R, there must be some relation scheme R' in $\mathbf{R}$ with $R' \supseteq R$, so R is in an $\mathbf{R}_F$-extension.

The inductive hypothesis is that at any point of the computation of $echase_F(T_{\mathbf{R}})$, if some row w is distinguished on the attributes in S, then S is an $\mathbf{R}_F$-extension. Assume at some point in the computation we have rows w_1 and w_2 that are distinguished on S_1 and S_2, and that we apply the F-rule for $X \rightarrow A$ to make $w_1(A)$ distinguished (hence $w_2(A)$ already is). Since no non-

distinguished variable is ever repeated, and $w_1(X) = w_2(X)$, we must have $X \subseteq S_1$ and $X \subseteq S_2$.

We need to show that S_1A is an $\mathbf{R}_F$-extension. Assume d is a database on $\mathbf{R}$ that is the projection of an instance $r(\mathbf{U})$. Since S_1 and S_2 are $\mathbf{R}_F$-extensions, we can construct relations $q_1(S_1) = \pi_{S_1}(r)$ and $q_2(S_2) = \pi_{S_2}(r)$ using only extension joins and projection. We can then construct $q_3(S_1A)$ as the q_2-extension of q_1 by $X \rightarrow A$. Hence S_1A is an $\mathbf{R}_F$-extension. Note that the only way $w_1(A)$ can become distinguished is through the direct application of an F-rule to w_1.

Example 11.50 Figure 11.60 shows $T_{\mathbf{R}}$ for the database scheme $\mathbf{R} = \{AB, BC, CD, BDEI\}$ of Example 11.49. Figure 11.61 shows $echase_F(T_{\mathbf{R}})$ for $F = \{C \rightarrow D, BD \rightarrow E\}$. We see that R is an $\mathbf{R}_F$-extension if $R \subseteq AB$, $R \subseteq BCDE$ or $R \subseteq BDEI$.

$T_{\mathbf{R}}$

A	B	C	D	E	I
a_1	a_2	b_1	b_2	b_3	b_1
b_5	a_2	a_3	b_6	b_7	b_8
b_9	b_{10}	a_3	a_4	b_{11}	b_{12}
b_{13}	a_2	b_{14}	a_4	a_5	a_6

Figure 11.60

$echase_F(T_{\mathbf{R}})$

A	B	C	D	E	I
a_1	a_2	b_1	b_2	b_3	b_4
b_5	a_2	a_3	a_4	a_5	b_8
b_9	b_{10}	a_3	a_4	b_{11}	b_{12}
b_{13}	a_2	b_{14}	a_4	a_5	a_6

Figure 11.61

It is not sufficient for R to be an $\mathbf{R}_F$-extension that $chase_F(T_{\mathbf{R}})$ be distinguished on R. In fact, the converse of Theorem 11.7 holds (see Exercise 11.37).

Example 11.51 Figure 11.62 shows the tableau $T_{\mathbf{R}}$ for database scheme $\mathbf{R} = \{AD, AB, BDE, CE\}$, which we last saw in Example 8.41. Let $F = \{A \rightarrow C, B \rightarrow C, CD \rightarrow E\}$. We see that $echase_F(T_{\mathbf{R}}) = T_{\mathbf{R}}$. However, $chase_F(T_{\mathbf{R}}) \neq T_{\mathbf{R}}$, as we see in Figure 11.63. There is a row of $chase_F(T_{\mathbf{R}})$ distinguished on CDE, but ADE is not an $\mathbf{R}_F$-extension (see Exercise 11.39).

T_R(A	B	C	D	E)
a_1	b_1	b_2	a_4	b_3
a_1	a_2	b_3	b_4	b_5
b_6	a_2	b_7	a_4	a_5
b_8	b_9	a_3	b_{10}	a_5

Figure 11.62

$chase_F(T_R)$(A	B	C	D	E)
a_1	b_2	b_2	a_4	a_5
a_1	a_2	b_2	b_4	b_5
b_6	a_2	b_2	a_4	a_5
b_8	b_9	a_3	b_{10}	a_5

Figure 11.63

Returning to tableau queries, let d be a database on database scheme **R** that is the projection of an instance r(**U**). Let Q be a tableau query that applies to d. For a set of FDs F, we can apply $echase_F$ to Q. However, rather than requiring that a symbol may only be replaced by a distinguished variable, we require that a symbol may only be replaced by a matched symbol. The matched symbols in Q correspond to distinguished variables in T_R. Since Q applies to d, for any row w in Q, there is always a relation scheme S in **R** such that $match(W) \subseteq S$.

Consider $Q' = echase_F(Q)$. There may be a row w' in Q' such that $S' = match(w')$ is not contained in any scheme in **R**. However, S' will always be an $\mathbf{R}_F$-extension. Thus, Q' applies to some database $d' = \{r_1(R_1), r_2(R_2), \ldots, r_p(R_p)\}$ where R_i is an $\mathbf{R}_F$-extension for $1 \leq i \leq p$. Every relation in d' can be computed from relations in d through extension joins and projections. If Q is the tableau query for some restricted algebraic expression for d, Q' will be a tableau query for some restricted algebraic expression for d' (see Exercise 11.40).

Example 11.52 Referring back to Example 11.47, Figure 11.57 shows the tableau query for the algebraic expression

$$\pi_{AD}(q \bowtie r \bowtie s)$$

for the database $d = \{q(AB), r(BC), s(CD)\}$. It turns out that $Q' = chase_F(Q) = echase_F(Q)$ for $F = \{C \rightarrow D\}$, so $Q \equiv_F Q'$ for the tableau

query Q' in Figure 11.58. Q'' does not apply to d, because the matched set of the second row is BD. However, BD is an R_F-extension for database scheme $\mathbf{R} = \{AB, BC, CD\}$. Thus, the algebraic expression above is equivalent under F to

$$\pi_{AD}(q \bowtie r'),$$

where r' can be computed by extension joins and projections from r and s.

We actually have slightly more leeway in computing $echase_F(Q)$. Suppose w is a row in Q and $match(w) = S$, and suppose there is a relation $s(S')$ in d such that $S' \supseteq S$. We can treat $w(A)$ as a matched symbol for any $A \in S'$, even if $A \notin S$. Consider: If we add a row w' to Q, where $w'(S') = q(S')$ and w' is new nondistinguished variables elsewhere, then we have not changed the mapping Q defines, since w supersedes w'. Furthermore, Q still applies to d and $match(w) = S'$.

Finally, in this section, we turn to tagged tableau queries and extensions of our equivalence results to databases that are not projections of a common instance. The fundamental change to validate the results is that a containment mapping between tagged tableau queries must preserve tags. That is, we require that a containment mapping always maps blanks to blanks, and never maps a symbol to a blank.

Example 11.53 Let d be a database $\{q(AD), r(BC), s(AC)\}$ that is not necessarily the projection of any common instance. Figure 11.64 shows the tagged tableau query Q_1 for the algebraic expression

$$\pi_{BC}(\sigma_{B=c}(q) \bowtie r \bowtie s).$$

Figure 11.65 shows the tagged tableau query Q_2 for the algebraic expression

$$\pi_{BC}(r \bowtie s).$$

The mapping ψ defined by

$$\psi(a_2) = c$$
$$\psi(a_3) = a_3$$
$$\psi(b_1) = b_1$$

is a containment mapping from Q_2 to Q_1, so $Q_2 \sqsupseteq Q_1$. We conclude

$$\pi_{BC}(\sigma_{B=c}(q) \bowtie r \bowtie s) \subseteq \pi_{BC}(r \bowtie s)$$

for any choices of q, r, and s.

Recall that in Example 11.46 we saw that

$$\pi_{BC}(\sigma_{B=c}(q) \bowtie r \bowtie s) = \sigma_{B=c}(r)$$

if q, r, and s are all projections of a common instance. Q_3 in Figure 11.66 is the tagged tableau query for $\sigma_{B=c}(r)$. Note that $Q_1 \not\equiv Q_3$, because no containment mapping is possible from Q_1 to Q_3.

Q_1

A	B	C	
	c	a_3	
b_1	c		(AB)
	c	a_3	(BC)
b_1		a_3	(AC)

Figure 11.64

Q_2

A	B	C	
	a_2	a_3	
	a_2	a_3	(BC)
b_1		a_3	(AC)

Figure 11.65

Q_3

A	B	C	
	c	a_3	
	c	a_3	(BC)

Figure 11.66

The results for data dependencies in the single relation case do not carry over entirely to tagged tableau queries. In general, in computing the chase of

a tagged tableau query, only F-rules may be applied, and then only to rows with the same tag.

Example 11.54 Figure 11.67 shows two tagged tableau queries, Q_1 and Q_2, for the database $d = \{r(A\ B\ C), s(B\ C\ D)\}$. Given that r and s both satisfy $F = \{B \rightarrow C\}$, we might conclude that Q_1 and Q_2 are equivalent under F. However, as the reader may verify, $Q_1(d) \not\equiv Q_2(d)$ for the states of r and s given in Figure 11.68.

Q_1

A	B	C	D	
a_1			a_4	
a_1	b_1	b_2		(ABC)
	b_1	b_3	a_4	

Q_2

A	B	C	D	
a_1			a_4	
a_1	b_1	b_2		(ABC)
	b_1	b_2	a_4	(BCD)

Figure 11.67

r

A	B	C
1	3	5
2	4	5

s

B	C	D
3	5	7
4	6	8

Figure 11.68

The problem in Example 11.54 is that while r and s both satisfy $B \rightarrow C$, the function from B to C defined by r is not consistent with the function from B to C defined by s. F-rules can only be applied to rows with different tags if consistency of FDs is required. Note that if a database d has a minimal BCNF database scheme with respect to the FDs in F, then the issue is moot. No FD in F^+ applies to two rows in a tagged tableau query for d if the rows have different tags.

11.5.5 Tableau Set Query Equivalence

Here we return to untagged tableau queries. Recall that a tableau set query $\mathbf{Q}$ with scheme R is a set of compatible tableau queries $\{Q_1, Q_2, \ldots, Q_p\}$, all with scheme R (necessarily). Recall also that $\mathbf{Q}(r)$ is

$$Q_1(r) \cup Q_2(r) \cup \cdots \cup Q_p(r).$$

The notions of containment and equivalence extend directly to tableau set queries.

Definition 11.13 Let $\mathbf{Q}_1$ and $\mathbf{Q}_2$ be tableau set queries with scheme R. Then $\mathbf{Q}_1 \sqsubseteq \mathbf{Q}_2$ if $\mathbf{Q}_1(r) \subseteq \mathbf{Q}_2(r)$ for every relation $r(R)$. $\mathbf{Q}_1 \equiv \mathbf{Q}_2$ if $\mathbf{Q}_1(r) = \mathbf{Q}_2(r)$ for every relation $r(R)$.

The following theorem characterizes containment of tableau set queries.

Theorem 11.8 Let $\mathbf{Q}_1$ and $\mathbf{Q}_2$ be compatible tableau set queries with scheme R. $\mathbf{Q}_1 \sqsupseteq \mathbf{Q}_2$ if and only if for each tableau query Q_2 in $\mathbf{Q}_2$ there is a tableau query Q_1 in $\mathbf{Q}_1$ such that $Q_1 \sqsupseteq Q_2$.

Proof The if direction follows directly from the definition of $\mathbf{Q}_1(r)$ and $\mathbf{Q}_2(r)$. For the only if direction, we again use the technique of regarding the rows of a tableau query as a relation. Let Q_2 be any tableau query in $\mathbf{Q}_2$ and consider $\mathbf{Q}_2(Q_2)$. If w_0 is the summary of Q_2, then $w_0 \in \mathbf{Q}_2(Q_2)$, since $w_0 \in Q_2(Q_2)$ (by the identity valuation). Since $\mathbf{Q}_1 \sqsupseteq \mathbf{Q}_2$, $w_0 \in \mathbf{Q}_1(Q_2)$. There must be a tableau query Q_1 in S_1 such that $w_0 \in Q_1(Q_2)$. Let v_0 be the summary of Q_1. The valuation ρ such that $\rho(Q_1) \subseteq Q_2$ and $\rho(v_0) = w_0$ is a containment mapping from Q_1 to Q_2. Hence $Q_1 \sqsupseteq Q_2$.

Example 11.55 Using the tableau queries $Q_1 = Q_4$ in Figure 11.69,\we define tableau set queries $\mathbf{Q}_1 = \{Q_1, Q_2, Q_3\}$ and $\mathbf{Q}_2 = \{Q_3, Q_4\}$. $\mathbf{Q}_2 \sqsupseteq \mathbf{Q}_1$, since $Q_4 \sqsupseteq Q_1$, $Q_4 \sqsupseteq Q_2$, and $Q_3 \sqsupseteq Q_3$. $\mathbf{Q}_1 \not\sqsupseteq \mathbf{Q}_2$, since none of Q_1, Q_2, or Q_3 contain Q_4.

Definition 11.14 A tableau set query $\mathbf{Q}$ is *nonredundant* if it does not contain distinct tableau queries Q_1 and Q_2 such that $Q_1 \sqsupseteq Q_2$.

Theorem 11.9 Let $\mathbf{Q}_1$ and $\mathbf{Q}_2$ be nonredundant tableau set queries. If $\mathbf{Q}_1 \equiv \mathbf{Q}_2$, then for every tableau query $Q_1 \in \mathbf{Q}_1$, there is one and only one tableau query $Q_2 \in \mathbf{Q}_2$ such that $Q_1 \equiv Q_2$.

Q_1

A	B	C
a_1		4
a_1	b_1	b_2
3	b_1	4

Q_2

A	B	C
a_1		4
a_1	b_1	b_2
b_3	b_1	4

Q_3

A	B	C
3		a_3
3	b_1	a_3
3	5	b_2

Q_4

A	B	C
a_1		a_3
a_1	b_1	b_2
b_3	b_1	a_3

Figure 11.69

Proof Let Q_1 be a tableau query in $\mathbf{Q}_1$. Since $\mathbf{Q}_1 \sqsubseteq \mathbf{Q}_2$, there is a tableau query $Q_2 \in \mathbf{Q}_2$ such that $Q_1 \sqsubseteq Q_2$. Similarly, since $Q_1 \sqsupseteq Q_2$, there is a tableau query Q_3 in $\mathbf{Q}_1$ such that $Q_3 \sqsupseteq Q_2$. We see $Q_1 \sqsubseteq Q_3$, which means $Q_1 = Q_3$, or else $\mathbf{Q}_1$ is not nonredundant. We conclude $Q_1 \equiv Q_2$. If there is another tableau query Q_4 in $\mathbf{Q}_2$ such that $Q_1 \equiv Q_4$, then $Q_2 \equiv Q_4$ and $\mathbf{Q}_2$ is not nonredundant. Hence Q_2 is the only tableau query in $\mathbf{Q}_2$ equivalent to Q_1.

Corollary If $\mathbf{Q}$ is a nonredundant tableau set query, there is no tableau set query equivalent to $\mathbf{Q}$ with fewer tableau queries.

Proof Left to the reader (see Exercise 11.43).

Example 11.56 Consider the tableau set query $\mathbf{Q}_1 = \{Q_1, Q_2, Q_3\}$ in Example 11.55. $\mathbf{Q}_1$ is not nonredundant, since $Q_2 \supseteq Q_1$. If we let $\mathbf{Q}_3 = \{Q_2, Q_3\}$, then $\mathbf{Q}_1 \equiv \mathbf{Q}_3$. $\mathbf{Q}_3$ is nonredundant, since $Q_2 \not\supseteq Q_3$ and $Q_3 \not\supseteq Q_2$.

11.6 OPTIMIZING CONJUNCTIVE QUERIES

Recall that a conjunctive query over a database d is a domain calculus expression E of the form

$$\{x_1(A_1)\, x_2(A_2) \cdots x_n(A_n) | \exists y_1(B_1)\, \exists y_2(B_2) \cdots \exists y_m(B_m) \\ f(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m).\}$$

such that f is the conjunction of atoms of the form

$$r(a_1\, a_2 \cdots a_k)$$

where $r \in d$ and each a_i is one of the x's, one of the y's, or a constant. Each atom in f represents a relation to be joined in evaluating E. We would like to find a conjunctive query equivalent to E that minimizes the number of atoms. The approach to minimizing conjunctive queries is almost exactly that of minimizing tableau queries.

One obvious requirement for equivalence of conjunctive queries is that they define relations over the same scheme. Any conjunctive query E' that is equivalent to E above must have the form

$$\{w_1(A_1)\, w_2(A_2) \cdots w_n(A_n) | \exists z_1(C_1)\, \exists z_2(C_2) \cdots \exists z_p(C_p) \\ g(w_1, w_2, \ldots, w_n, z_1, z_2, \ldots, z_p)\}.$$

A *folding* ψ from E to E' is a mapping of the domain variables and constants of E to the domain variables and constants of E' such that

c1. $\psi(x_i) = w_i$, $1 \le i \le n$;
c2. $\psi(c) = c$ for any constant c; and
c3. If $r(a_1\, a_2 \cdots a_k)$ is any atom in f, then $r(\psi(a_1)\, \psi(a_2) \cdots \psi(a_k))$ is an atom in g.

Foldings are quite similar to containment mappings of tagged tableau queries.

Example 11.57 Let d be the database $\{q(ABC), r(BCD), s(AD)\}$. Assume $dom(C) = dom(D)$. Let E be the conjunctive query

$$\{x_1(A)\, x_2(D) \mid \exists y_1(B)\ \exists y_2(C)\ \exists y_3(D)\ \exists y_4(D)$$
$$q(x_1\, y_1\, y_2) \wedge r(y_1\, y_2\, x_2) \wedge r(y_1\, x_2\, y_3) \wedge s(x_1\, y_4)\}$$

and let E' be the conjunctive query

$$\{w_1(A)\, w_2(D) \mid \exists z_1(B)\ q(w_1\, z_1\, w_2) \wedge r(z_1\, w_2\, w_2) \wedge s(w_1\, 6)\}.$$

The mapping ψ in Figure 11.70 is a folding from E to E'. There is no folding from E' to E, since the atom $s(w_1\ 6)$ in E' cannot map to any atom of E.

$$\begin{array}{ll} \psi(x_1) = w_1 & \psi(y_1) = z_1 \\ \psi(x_2) = w_2 & \psi(y_2) = w_2 \\ & \psi(y_3) = w_2 \\ & \psi(y_4) = 6 \end{array}$$

Figure 11.70

For conjunctive queries E_1 and E_2, let $E_1 \sqsupseteq E_2$ mean the obvious thing. The proof of the following theorem is similar to that of Theorem 11.1, and is left as Exercise 11.44.

Theorem 11.10 Let E_1 and E_2 be conjunctive queries that define relations over the same scheme. $E_1 \sqsupseteq E_2$ if and only if there is a folding from E_1 to E_2.

Definition 11.15 A conjunctive query E is *minimum* if there is no conjunctive query E' equivalent to E with fewer atoms in its formula.

From Theorem 11.10 we can show that any minimum equivalent conjunctive query for a given conjunctive query E is unique up to a one-to-one renaming of domain variables. That is, if E_1 and E_2 are minimum equivalent conjunctive queries for E, then there is a folding ψ from E_1 to E_2 whose inverse is a folding from E_2 and E_1.

Let E_1 be

$$\{x_1(A_1)\, x_2(A_2) \cdots x_n(A_n) \mid \exists y_1(B_1)\ \exists y_2(B_2) \cdots \exists y_m(B_m)$$
$$f(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)\}$$

and let E_2 be

$$\{w_1(A_1)\, w_2(A_2) \cdots w_n(A_n) \mid \exists z_1(C_1)\ \exists z_2(C_2) \cdots \exists z_p(C_p)$$
$$g(w_1, w_2, \ldots, w_n, z_1, z_2, \ldots, z_p)\}.$$

By Theorem 11.10, there is a folding ψ_{12} from E_1 to E_2 and a folding ψ_{21} from E_2 to E_1. If ψ_{12} maps two atoms of f to the same atom of g, then $\psi_{21} \circ \psi_{12}$ is a folding from the atoms of E_1 to a proper subset of those atoms. The identity mapping is a folding from this subset to all the atoms. We conclude that we can remove some atoms (and possibly some of the y's) from E_1 to get an equivalent conjunctive query, a contradiction. Thus ψ_{12} must be one-to-one on the atoms of f. By a similar argument, we can show ψ_{12} is one-to-one as a mapping of variables and constants. (If not, there must be some y_i that is not mentioned in f.) Thus, the inverse of ψ_{12} is indeed a function. The inverse of ψ_{12} must also be defined on all variables and constants in E_2, or else there is some atom of g not mapped to by ψ_{12}, which implies E_2 is not minimum.

To show that ψ_{12}^{-1} is indeed a folding, we must show that conditions c1-c3 in the definition of folding hold. We have $\psi_{12}^{-1}(w_i) = x_i$, since $\psi_{12}(x_i) = w_i$. If $r(b_1\ b_2\ \cdots\ b_k)$ is an atom in g, it must be mapped to by some atom $r(a_1\ a_2\ \cdots\ a_k)$ in f, or else E_2 would not be minimum. Therefore, $r(\psi_{12}^{-1}(b_1)\ \psi_{12}^{-1}(b_2)\ \cdots\ \psi_{12}^{-1}(b_k)) = r(a_1\ a_2\ \cdots\ a_k)$ is in f. The only condition left to check is c2: $\psi_{12}^{-1}(c) = c$ for any constant c. The only way a problem can arise is if $\psi_{12}(y_i) = c$ for some $1 \leq i \leq m$.

For an atom $r(a_1\ a_2\ \cdots\ a_k)$ in f and a folding ψ, let $\psi(r(a_1\ a_2\ \cdots\ a_k))$ be $r(\psi(a_1\ \psi(a_2)\ \cdots\ \psi(a_k))$. Consider the folding $\psi_{11} = \psi_{21} \circ \psi_{12}$ from E_1 to itself. Everything we have shown so far about ψ_{12} applies to ψ_{21} by symmetry. The folding ψ_{11} must be a one-to-one mapping on atoms of E_1, since it is the composition of two such mappings. Let α_1 be an atom of E_1 that contains y_i in the j^{th} position. Consider the sequence $\alpha_1, \alpha_2, \alpha_3, \ldots$ of atoms from E_1 such that $\psi_{11}(\alpha_l) = a_{l+1}$, $l \geq 1$. The sequence must eventually repeat an atom. Suppose $\alpha_q = \alpha_{q'}$ for $q < q'$. If $q > 1$ then $\psi_{11}(\alpha_{q-1}) = \alpha_q = \psi_{11}(\alpha_{q'-1})$ and ψ_{11} is not one-to-one on atoms. Hence $q = 1$. But since $\psi_{12}(y_i) = c$ and $\psi_{21}(c) = c$, $\psi_{11}(y_i) = c$. Also, $\psi_{11}(c) = c$, so each of $\alpha_2, \alpha_3, \alpha_4, \ldots$ must have c in the j^{th} position, which contradicts $\alpha_1 = \alpha_{q'}$. The premise that $\psi_{12}(y_i) = c$ must be incorrect.

We summarize our argument in the following theorem.

Theorem 11.11 Let E_1 and E_2 be two minimum conjunctive queries. $E_1 \equiv E_2$ if and only if there is a folding ψ from E_1 to E_2 such that ψ^{-1} is a folding from E_2 to E_1.

As a consequence of Theorem 11.11, it is not hard to show that for any conjunctive query E there is a minimum equivalent conjunctive query E' that is E with some atoms and quantified variables removed. Minimizing a conjunctive query E reduces to finding a folding from E to E' that maps the atoms of E to a proper subset of themselves.

Example 11.58 Let d be the same database as used in Example 11.57. Consider the conjunctive query

$$\begin{aligned} \mathrm{E} = \{x_1(A)\, x_2(D) | \exists y_1(C)\ \exists y_2(B)\ \exists y_3(C)\ \exists y_4(C)\ \exists y_5(D) \\ q(x_1\, 3\, y_1) \wedge q(x_1\, y_2\, y_3) \wedge r(3\, y_1\, x_2) \wedge \\ r(y_2\, y_4\, x_2) \wedge s(x_1\, 5) \wedge s(x_1\, y_5)\} \end{aligned}$$

over database d. The folding ψ_1 for E that maps y_2 to 3, y_3 and y_4 to y_1, and everything else to itself shows that the conjunctive query

$$\begin{aligned} E' = \{x_1(A)\, x_2(A) | \exists y_1(C)\ \exists y_5(D) \\ q(x_1\, 3\, y_1) \wedge r(3\, y_1\, x_2) \wedge s(x_1\, 5) \wedge s(x_1\, y_5)\} \end{aligned}$$

is equivalent to E. The folding ψ_2 for E' that maps y_5 to 5 and everything else to itself shows that

$$E'' = \{x_1(A)\, x_2(A) | \exists y_1(C)\ q(x_1\, 3\, y_1) \wedge r(3\, y_1\, x_2) \wedge s(x_1\, 5)\}$$

is equivalent to E' and hence to E. E'' is clearly minimum (why?), so E'' is a minimum equivalent conjunctive query for E.

Determining whether a given conjunctive query E is minimum is an NP-complete problem. No known method for testing minimality is significantly better than examining all foldings from E to itself to see if one maps the atoms of E to a proper subset of themselves. Nevertheless, if E contains only a few atoms, such a search will not be prohibitive, especially if E is to be evaluated many times.

11.7 QUERY MODIFICATION FOR DISTRIBUTED DATABASES

Some database systems support relational databases whose parts are physically separated. Different relations might reside at different sites, multiple copies of a single relation might be distributed among several sites, or one relation might be broken into pieces and the pieces distributed. In order to evaluate a query posed at one site, it may be necessary to transfer data between various sites. The dominant element in the time required to process such a query will often be the time spent transferring data between sites, rather than the time spent on retrieval from secondary storage or computa-

tion. Efficient query evaluation depends on minimizing communication between sites.

Example 11.59 Suppose that the expression $r \bowtie \sigma_{A=a}(s)$ must be evaluated at site 1 of a distributed database system, where r is stored at site 1, but s is at site 2. Site 1 could ask site 2 to transmit all of relation s, and then compute the selection itself. A more reasonable approach, if s is large, if for site 1 to transmit the selection condition $A = a$ to site 2. Site 2 then performs the selection and transmits the result, which is presumably much smaller than s, to site 1.

In this section we shall look at semijoin, which is a useful operation for computing joins in a distributed system, and fragmented relations, which are relations that have been horizontally decomposed and stored at multiple sites.

11.7.1 Semijoins

Before defining semijoin, we demonstrate its utility with an example.

Example 11.60 Let relations $r(A\ B)$ and $s(B\ C\ D)$, as shown in Figure 11.71, be stored at site 1 and site 2, respectively, of a distributed database. Suppose we wish to compute $r \bowtie s$ at site 1. We could transmit all of s from site 2 to site 1 and compute the join at site 1. There would be 21 values sent in that transmission. Alternatively, we can compute $r' = \pi_B(r)$ at site 1, send r' to site 2, compute $s' = r' \bowtie s$, and send s' to site 1. We can then compute $r \bowtie s$ as $r \bowtie s'$. Relations r' and s' are shown in Figure 11.72. For this method, only 15 values must be transmitted: 6 for r' and 9 for s'.

$r(A$	$B)$
1	4
1	5
1	6
2	4
2	6
3	7
3	8
3	9

$s(B$	C	$D)$
4	13	16
4	14	16
7	13	17
10	14	16
10	15	17
11	15	16
11	15	16
12	15	16

Figure 11.71

$r'(A)$
4
5
6
7
8
9

$s'(B$	C	$D)$
4	13	16
4	14	16
7	13	17

Figure 11.72

Definition 11.16 Let $r(R)$ and $s(S)$ be two relations. The *semijoin of* r *with* s, denoted $r \ltimes s$, is the relation $\pi_R(r \bowtie s)$. That is, $r \ltimes s$ is the portion of r that joins with s.

Recalling some of the transformations from Section 11.3, we have

$$\pi_R(r \bowtie s) = \pi_R(r) \bowtie \pi_{R \cap S}(s) = r \bowtie \pi_{R \cap S}(s).$$

Thus, $r \ltimes s$ can be computed knowning only $\pi_{R \cap S}(s)$; all of s is not necessary. (In Example 11.60, $r' = \pi_B(r)$ and $s' = s \ltimes r$.) The property of semijoin that interests us is that $(r \ltimes s) \bowtie s = r \bowtie s$ (see Exercise 11.46). If r and s are at different sites, computing $r \bowtie s$ as $(r \ltimes s) \bowtie s$ saves transmitting data whenever r is larger than $\pi_{R \cap S}(r)$ and $r \ltimes s$ put together. If r and s join completely, there is no savings, of course. However, even in a database where all relations are projections of a common instance, queries can involve joins between relations that do not join completely, as a result of selections. In such cases, semijoins may be effective. Sometimes semijoins can replace joins completely.

Example 11.61 Let $q(A\ B)$, $r(B\ C)$, and $s(C\ D)$ be the three relations shown in Figure 11.73. Suppose q, r, and s are dispersed at sites 1, 2, and 3, respectively, and that we wish to evaluate the algebraic expression

$$E = \pi_D(\sigma_{A=1}(q \bowtie r \bowtie s))$$

at site 3. First we note that E is equivalent to

$$E' = \pi_D(\sigma_{A=1}(q) \bowtie r \bowtie s).$$

To evaluate E', we first compute $q' = \sigma_{A=1}(q)$ at site 1. We next send $\pi_B(q')$ to site 2 and compute $r' = r \ltimes q'$ there. We then send $\pi_C(r')$ to site

3 and compute $s' = s \ltimes r'$ at site 3. The desired result is $\pi_D(s')$, which we also compute at site 3. Relations q', r', and s' are shown in Figure 11.74. Note that no relation was transmitted in its entirety.

$q(A$	$B)$
1	4
1	5
2	4
2	6
3	6

$r(B$	$C)$
4	7
5	7
5	8
6	9
6	10

$s(C$	$D)$
7	11
8	11
9	12
10	11

Figure 11.73

$q'(A$	$B)$
1	4
1	5

$r'(B$	$C)$
4	7
5	7
5	8

$s'(C$	$D)$
7	11
8	11

Figure 11.74

Frequently during the evaluation of a query in a distributed database, there is a point at which intermediate result relations $r_1, r_2, \ldots, r_p$ exist at different sites, and the next step is to compute $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_p$ at a single site. It is helpful if we can easily compute the portion of each relation that takes part in the join.

Definition 11.17 Given database d and a relation $r(R)$ in d, the *full reduction of* r *relative to* d, denoted $FR(r, d)$, is $\pi_R(\bowtie (d))$. That is, $FR(r, d)$ is the portion of r that takes part in the join with all the other relations in d.

Example 11.62 Let d be the database $\{q(A\ B), r(B\ C), s(A\ C)\}$ for the relations q, r, and s in Figure 11.75. $FR(q, d) = q'$, $FR(r, d) = r'$, and $FR(s, d) = s'$ are shown in Figure 11.76.

$q(A$	$B)$
1	4
1	5
2	4
2	6
3	6

$r(B$	$C)$
4	7
5	7
5	8
6	8

$s(A$	$C)$
1	8
2	7
2	8
3	7

Figure 11.75

$q'(A$	$B)$
1	5
2	4
2	6

$r'(B$	$C)$
4	7
5	8
6	8

$s'(A$	$C)$
1	8
2	7
2	8

Figure 11.76

Definition 11.18 Let $d = \{r_1, r_2, \ldots, r_p\}$ be a database. A *semijoin program* SP for d is a sequence of assignment statements of the form

$$r_i \leftarrow r_i \ltimes r_j$$

We let $SP(r_i, d)$ denote the final value of r_i after executing SP on d.

Definition 11.19 A semijoin program SP for database $d = \{r_1, r_2, \ldots, r_p\}$ is a *full-reducer* if for any state of d,

$$FR(r_i, d) = SP(r_i, d), 1 \leq i \leq p.$$

Note that $FR(r_i, d) \subseteq SP(r_i, d)$ whether or not SP is a full reducer for d (see Exercise 11.48).

Example 11.63 Let d be the database $\{r_1(A_1\ A_2), r_2(A_2\ A_3\ A_4\ A_5), r_3(A_3\ A_4\ A_6), r_4(A_4\ A_5\ A_7)\}$. Figure 11.77 shows a semijoin program SP_1 for d. SP_1 is not a full-reducer for d. Consider the state of d shown in Figure 11.78. $SP_1(r_2, d)$ is shown in Figure 11.79. It contains the tuple $\langle 4\ 5\ 7\ 10\rangle$, which is not in $FR(r_2, d)$, since it joins with no tuple in r_i.

The semijoin program SP_1 in Figure 11.78 is a full-reducer for d. For example, using the state of d in Figure 11.75, $SP_2(r_2, d)$ is the relation in Figure 11.79, which is $FR(r_2, d)$. Proving that SP_2 is indeed a full reducer is not a trivial task, nor is SP_2 the shortest full-reducer for d (see Exercise 11.49). We indicate here why, for example, $SP_2(r_2, d)$ is necessarily $FR(r_2, d)$.

By Exercise 11.48, $SP_2(r_2, d) \supseteq FR(r_2, d)$. We need to show that every tuple in $SP_2(r_2, d)$ will actually join with tuples from the other three relations. Note that of the other three relations, only r_3 and r_4 have intersecting schemes, and the intersection is contained in the scheme of r_2. If a tuple t_2 from r_2 joins individually with tuples t_1, t_3, and t_4 from r_1, r_3, and r_4, it will join with them collectively. Since r_2 is semijoined with r_1, r_3, and r_4 in steps 3, 4, and 5 of SP_2, such tuples t_1, t_3, and t_4 must be present for any tuple t_2 in r_2.

SP_1:

1. $r_2 \leftarrow r_2 \ltimes r_3$;
2. $r_2 \leftarrow r_2 \ltimes r_4$;
3. $r_1 \leftarrow r_1 \ltimes r_1$.

Figure 11.77

$r_1(A_1$	$A_2)$
1	3
2	3

$r_2(A_2$	A_3	A_4	$A_5)$
3	5	7	9
3	6	7	9
4	5	7	10

$r_3(A_3$	A_4	$A_6)$
5	7	11
6	8	12

$r_4(A_4$	A_5	$A_7)$
7	9	13
7	10	13

Figure 11.78

$SP_1(r_2,d)(A_2$	A_3	A_4	$A_5)$
3	5	7	9
4	5	7	10

Figure 11.79

SP_2:

1. $r_3 \leftarrow r_3 \ltimes r_4$;
2. $r_4 \leftarrow r_4 \ltimes r_3$;
3. $r_2 \leftarrow r_2 \ltimes r_3$;
4. $r_2 \leftarrow r_2 \ltimes r_4$;
5. $r_2 \leftarrow r_2 \ltimes r_1$;
6. $r_1 \leftarrow r_1 \ltimes r_2$;
7. $r_3 \leftarrow r_3 \ltimes r_2$;
8. $r_4 \leftarrow r_4 \ltimes r_2$.

Figure 11.80

$SP_2(r_2,d)(A_2$	A_3	A_4	$A_5)$
3	5	7	9

Figure 11.81

Example 11.64 The database d in Example 11.62 has no full-reducer. The state of d shown in Figure 11.75 remains unchanged no matter what semijoins are performed, yet none of the relations in d is fully reduced.

In Chapter 13 we shall return to semijoin programs and characterize those databases for which full-reducers exist.

11.7.2 Fragments of Relations

In a distributed database, a given relation is not necessarily stored in its entirety at any single site. Its tuples may be dispersed among several sites for performance considerations. For example, a database of airline reservations might be broken up to put all tuples for each flight at the site where the flight originates. Sometimes, the same tuple is stored at several sites. We shall call the pieces of a relation that reside at each site *fragments* of the relation. Processing a query at one site may involve retrieving fragments from other sites. If the assignment of tuples to fragments follows some rule, then that rule can be used to reduce the number of fragments accessed in processing a query.

We shall define fragments using selection conditions. A fragment of a relation r will be $\sigma_C(r)$ for some selection condition C.

For the following discussion, let $s_1, s_2, \ldots, s_p$ be the fragments of relation r, where s_i, $1 \leq i \leq p$, is defined as $\sigma_{C_i}(r)$ for some selection condition C_i. We call $\{C_1, C_2, \ldots, C_p\}$ the *fragmentation scheme* for f. One condition we wish always to hold is

$$r = s_1 \cup s_2 \cup \cdots \cup s_p.$$

That is, we can recover r from its fragments. We assume there is also a condition C_0 that r is guaranteed to satisfy: $r = \sigma_{C_0}(r)$. To be certain that r can always be represented as the union of its fragments, the fragmentation scheme must have the property

$$C_0 \rightarrow C_1 \vee C_2 \vee \cdots \vee C_p.$$

That is, any tuple that satisfies C_0 must necessarily satisfy one of the conditions in the fragmentation scheme. We call any fragmentation scheme that has the property above *valid*.

Example 11.65 Let r be a relation on scheme $A\ B\ D$, where the domain of each attribute is the positive integers. Assume r always satisfies the condition C_0 where

$$C_0 \text{ is } (A \leq B \vee B \leq D) \wedge A \neq D.$$

The fragmentation scheme $\{C_1, C_2, C_3\}$ for r where

C_1 is $A = B$
C_2 is $B = D$, and
C_3 is $A \leq D$

is valid for f (why?).

Knowing the fragmentation scheme for r can help in processing queries. Suppose, perhaps as part of a larger query, we want to evaluate $\sigma_C(r)$ at a given site. We could send out a request for $\sigma_C(s_i)$ for each fragment s_i that is not at the site. However, we are guaranteed to get nothing back whenever $C \wedge C_i \equiv \mathit{false}$. Thus, we need only ask for $\sigma_C(s_i)$ whenever $C \wedge C_i \not\equiv \mathit{false}$.

We can actually do better at eliminating fragments from consideration. Suppose that for condition C_1 there is a condition C_j such that

$$C \wedge C_1 \Rightarrow C \wedge C_j.$$

Any tuple t in r that appears in fragment s_1 and satisfies C will also appear in fragment s_j. Thus, fragment s_1 need not be consulted to evaluate $\sigma_C(r)$. In general, if

$$C \wedge C_1 \Rightarrow [(C \wedge C_2) \vee (C \wedge C_3) \vee \cdots \vee (C \wedge C_p)]$$

then fragment s_1 can be removed from consideration. We call this implication the *elimination requirement*.

Example 11.66 Let r be a relation on scheme $A\ B\ D$, where the domain of each attribute is the positive integers. Assume r satisfies the condition C_0 where

$$C_0 \text{ is } A \leq B \wedge B \leq D.$$

Let s_1, s_2, s_3, and s_4 be fragments for r corresponding to the fragmentation scheme $\{C_1, C_2, C_3, C_4\}$ where

C_1 is $A < B$,
C_2 is $B \leq 3 \wedge B < D$,
C_3 is $B > 3$, and
C_4 is $A = D$.

The reader should check that this fragmentation scheme is valid.

Suppose we want to evaluate $\sigma_C(r)$ where

C is $A < \mathrm{B} \wedge B < D$.

Which fragments must we consult? Fragment s_4 is out, since $C \wedge C_4 \equiv \mathit{false}$. Also, since

$$C \wedge C_1 \Rightarrow [(C \wedge C_2) \vee (C \wedge C_3)],$$

s_1 could be eliminated. Alternatively, since

$C \wedge C_2 \Rightarrow C \wedge C_1$ and
$C \wedge C_3 \Rightarrow C \wedge C_1$,

s_2 and s_3 could be eliminated, and only s_1 retained.

In general, when determining which fragment can be eliminated in evaluating $\sigma_C(r)$, $C' = C \wedge C_0$ can be used in place of C. If $C \not\Rightarrow C_0$, this replacement could allow more fragments to be eliminated.

We have not touched on algorithms to test validity or the elimination requirement. The complexity of such algorithms depends heavily on the particular domains for attributes and the permissible forms of selection conditions. As we saw in the last example, there can be more than one possibility for a minimal set of fragments to consult to evaluate $\sigma_C(r)$. The smallest set among the minimal sets might not be the optimal choice for evaluating the query. There are other considerations, such as communications costs (it could cost more to talk to one site than another), whether one of the fragments is already at the site where the query is being evaluated, and the amount of duplication among tuples in the minimal sets.

11.8 EXERCISES

11.1 Let r and s be the relations from Example 11.9. Give the number of disk accesses necessary to compute $r \bowtie s$ by the following methods.

(a) Read the blocks of r into memory 4 at a time, and for each group of 4, read in the blocks of s one at a time.
(b) Read the blocks of s into memory 2 at a time, and for each group of 2, read in the blocks of r three at a time.

11.2 For the relations r and s of Example 11.9, suppose only r is sorted on B. Can you find a more efficient way to compute $r \bowtie s$ than the one where neither r nor s is sorted on B. Suppose each disk block only contains 5 tuples?

11.3 Assume r and s are relations such that $r \subseteq s$. Simplify the following expressions.
(a) $r \bowtie s$
(b) $\sigma_{A=a}(r) - s$
(c) $\pi_X(r) \cap \pi_X(s)$

11.4 Give the expression tree for each of the following expressions, then merge nodes and simplify where possible.
(a) $(r_1 \bowtie r_2 \bowtie r_3) \cup (r_2 \bowtie r_3 \bowtie r_4) \cup (r_1 \bowtie r_3 \bowtie r_4) \cup (r_1 \bowtie r_2 \bowtie r_4)$
(b) $(r \bowtie q) \cap (\sigma_{A=a}(s) - s)$
(c) $\pi_{\text{P\# PN BD DT RG}}(rp \bowtie low)$, for rp and low as given in Example 11.1.

11.5 Give the expression tree for each of the following expressions, then apply algebraic optimization to each. The schemes of q, r, and s are ABD, BDF, and FG.
(a) $\sigma_{D=d}(\pi_{BDF}(q \bowtie r) - \pi_{BDF}(r \bowtie s))$
(b) $\pi_{AD}(q \bowtie r \bowtie \sigma_{G=g}(s))$
(c) $\pi_B(\sigma_{A=a}(\sigma_{D=d}(q)) \bowtie (r - {}_{\pi BDF}(r \bowtie s))))$

11.6 Consider the statement

$$r \leftarrow \pi_X(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_k)$$

where the schemes of $s_1, s_2, \ldots, s_k$ are all disjoint. Suppose X only contains attributes from $s_1, s_2, \ldots, s_p$ for some $p < k$. Show a way to transform the statement to a program containing only $p-1$ **for**-loops, given that you may easily test if a relation is empty.

11.7 Apply the query decomposition algorithm to the initial statement of Example 11.33 using the following alternatives for comparisons C_1-C_6.
(a) C_1, C_2, C_3, C_5, and C_6 are the same, C_4 is $z.\mathrm{G} \leq 3$
(b) C_1, C_2, C_3, and C_6 are the same, C_4 is $z.\mathrm{G} \leq 3$, C_5 is $y.F = 4$

(c) C_1, C_3, C_4, and C_5 are the same, C_2 is $x.A = 5$, C_6 is $x.F \leq 9$
(d) C_2-C_6 are the same, C_1 is $w.B = 7$

11.8 For the statement

$$r \leftarrow \pi_X(\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_k))$$

let P_1 be the program that results from query decomposition where $\{s_1, s_2, \ldots, s_p\}$, $p < k$ is instantiated, then $s_1, s_2, \ldots, s_p$ are all iterated. Compare the structure of P_1 to the program P_2 that results where $s_1, s_2, \ldots, s_p$ are iterated without the instantiation first.

11.9 Consider the tuple calculus expression and the corresponding algebraic expression at the beginning of Section 11.4. Let h be the formula

$$\exists y_1(R_1) \in r_1\ \exists y_2(R_2) \in r_2 \cdots \exists y_m(R_m) \in r_m\ g(y_1, y_2, \ldots, y_m).$$

Let E be the corresponding part of the algebraic expression, namely

$$\sigma_C(s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m).$$

Can you find modifications to h that correspond to applying instantiation and iteration to the statement $r \leftarrow E$? Hint: For instantiation, it is necessary to introduce a new tuple variable into h.

11.10 Consider relations $r_1(A_1\ A_2)$, $r_2(A_2\ A_3)$, $r_3(A_3\ A_4)$, and $r_4(A_4\ A_5)$. Below are two methods to compute $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$. Assuming each relation has n tuples, under what conditions will the second method require less time. Assume time is measured by the number of executions of assignment statements. Single tuples are treated as one tuple relations in the assignment statements.

```
Method 1:
  r ← ∅;
  for each tuple t1 in r1 do
    for each tuple t2 in r2 do
      for each tuple t3 in r3 do
        for each tuple t4 in r4 do
          r ← r ∪ (t1 ⋈ t2 ⋈ t3 ⋈ t4)
```

```
Method 2:
  s1 ← ∅;
  for each tuple t1 in r1 do
    for each tuple t2 in r2 do
      s1 ← s1 ∪ (t1 ⋈ t2);
  s2 ← ∅;
  for each tuple t3 in r3 do
    for each tuple t4 in r4 do
      s2 ← s2 ∪ (t3 ⋈ t4);
  r ← ∅;
  for each tuple u1 in s1 do
    for each tuple u2 in s2 do
      r ← r ∪ (u1 ⋈ u2)
```

11.11 Give an example where option 5 of the query decomposition heuristic would be used. That is, give a connection graph where options 1-4 do not apply.

11.12 Find two tableau queries Q_1 and Q_2 such that Q_1 covers Q_2 but $Q_2 \not\equiv Q_1$.

11.13 Prove Lemma 11.1.

11.14 Find two tableau queries Q_1 and Q_2 such that $Q_1 \equiv Q_2$, Q_2 is Q_1 with a single row removed, but the row is not superseded in Q_1.

11.15 Show that removing all superseded rows from a tableau query does not necessarily guarantee a minimum tableau query.

11.16 Prove Theorem 11.3. Note that the containment mapping can do nothing more than rename nondistinguished variables.

11.17 Give an algorithm to decide if two tableau queries are identical up to a one-to-one renaming of nondistinguished variables.

Definition 11.20 An *expression tableau query* is a tableau query that can be derived from a restricted algebraic expression by the method of Chapter 10.

11.18 Prove the following:

(a) If Q_1 is an expression tableau query, and Q_2 is a subtableau of Q_1 that is minimum and equivalent to Q_1, then Q_2 is an expression tableau query.

(b) Tableau query Q is equivalent to an expression tableau query if and only if every and any minimum equivalent tableau query for Q is an expression query.

(c) There is a tableau query that is equivalent to an expression tableau query but is not itself an expression tableau query.

11.19 Find an efficient method to determine if a simple tableau query is an expression tableau query.

11.20 Let Q be an expression tableau query that is not simple. What is the fewest rows Q can have?

11.21 Show that a minimum equivalent tableau query for a simple tableau query Q is also simple.

11.22 For the tableau query Q in Figure 11.40, compute $COMP_{w_i}(w_j)$ where
(a) $i = 1$ and $j = 2$, (c) $i = 3$ and $j = 6$,
(b) $i = 2$ and $j = 1$, (d) $i = 4$ and $j = 3$.

11.23 Prove Lemma 11.3. The proof depends on the simplicity of Q.

11.24 Find an algorithm to compute $COMP_v(w)$ in $O(k^2 n)$ for a tableau query Q with k rows and n columns.

11.25 Find MINEQ(Q) for tableau query Q in Figure 11.40.

11.26 Let Q be a simple tableau query and let v and w be rows of Q. Show that if w' is a row in $COMP_v(w)$, then $COMP_v(w) = COMP_v(w')$.

11.27 Are any of the following tableau queries equivalent?

Q_1

A_1	A_2	A_3	A_4
a_1	a_2		
a_1	b_1	b_2	4
b_3	a_2	b_2	b_3

Q_2

A_1	A_2	A_3	A_4
a_1	a_2		
b_1	a_2	b_2	b_3
b_4	a_2	b_5	b_6
b_7	b_8	b_2	b_6
a_1	b_9	b_2	4

Q_3

A_1	A_2	A_3	A_4
a_1	a_2		
a_1	b_1	b_2	4
a_1	b_3	b_4	b_5
b_6	a_2	b_2	b_5

Q_4

A_1	A_2	A_3	A_4
a_1	a_2		
a_1	a_2	b_2	4
b_3	a_2	b_2	b_3
b_5	a_2	b_2	4

11.28 Which tableau queries in Exercise 11.27 are equivalent on $SAT(A_3 \rightarrow A_4)$?

11.29 Exhibit a tableau query Q and a set $\mathbf{C}$ of FDs such that Q is simple but $chase_{\mathbf{C}}(Q)$ is not, nor can $chase_{\mathbf{C}}(Q)$ be made simple be removing superseded rows.

11.30 Given a simple tableau query Q and a set $\mathbf{C}$ of FDs and JDs, show that $chase_{\mathbf{C}}(\text{MINEQ}(Q))$ is not necessarily a $\mathbf{C}$-minimum equivalent tableau query for Q.

11.31 If Q is an expression tableau query, and C is a set of FDs and JDs, is $chase_{\mathbf{C}}(Q)$ necessarily an expression tableau query?

11.32 Let $d = \{q(AB), r(BC), s(AC)\}$ be a database that is the projection of a common instance over $A\ B\ C$. Use tableau query optimization to find algebraic expressions equivalent to the following with fewer joins, if such exist.

(a) $\pi_B(q \bowtie r \bowtie s)$

(b) $\pi_{AC}(q \bowtie r)$

(c) $\pi_{AC}(q \bowtie r \bowtie s)$

(d) $\pi_{AC}(\sigma_{A=c_1}(q) \bowtie \sigma_{C=c_2}(r) \bowtie s)$.

11.33 Let d be a database over database scheme $\mathbf{R}$. Assume d is the projection of an instance $r(\mathbf{U})$ in $SAT(F)$ for some set F of FDs. Let $q_1, q_2, \ldots, q_m$ be a sequence of relations such that w_i, $1 \leq i \leq m$ is either

i. a relation in d, or

ii. the q_j-extension of q_k by $Y \rightarrow Z$ for $j < i$, $k < i$ and $Y \rightarrow Z \in F$.

Let S be the scheme of q_m. Show that $q_m = \pi_S(r)$.

11.34 Let F be a set of FDs. Let $\mathbf{R} = \{R_1, R_2, \ldots, R_p\}$ be SYNTHESIZE(F) for the SYNTHESIZE algorithm of Chapter 6. Let $\mathbf{U} = R_1 R_2 \cdots R_p$. Under what condition is $\mathbf{U}$ an $\mathbf{R}_F$-extension?

11.35 Let F be a set of FDs and let $\mathbf{R} = \{R_1, R_2, \ldots, R_p\}$ be a database scheme, where $\mathbf{U} = R_1 R_2 \cdots R_p$. Suppose $F \models *[\mathbf{R}]$. Is $\mathbf{U}$ necessarily an $\mathbf{R}_F$-extension?

11.36 Let F be a set of FDs and let $\mathbf{R}$ be a database scheme over $\mathbf{U}$. Show

that if $R \subseteq \mathbf{U}$ is an $\mathbf{R}_F$-extension, then $echase_F(T_\mathbf{R})$ contains a row distinguished at least on the attributes in R.

11.37 Let F be a set of FDs and let $\mathbf{R}$ be a database scheme. Prove that if F is enforceable on $\mathbf{R}$, then $echase_F(T_\mathbf{R}) = chase_F(T_\mathbf{R})$. Recall that F is enforceable on $\mathbf{R}$ if some cover G of F applies to R.

11.38 Show the ADE is not an $\mathbf{R}_F$-extension for $F = \{A \rightarrow C, B \rightarrow C, CD \rightarrow E\}$ and $\mathbf{R} = \{AD, AB, BDE, CE\}$.

11.39 Show that if F is a set of FDs and Q is an expression tableau query, then $echase_F(Q)$ is an expression tableau query.

11.40 Find the containment relationships between the tableau set queries

$$\mathbf{Q}_1 = \{Q_1, Q_3\}$$
$$\mathbf{Q}_2 = \{Q_2, Q_3\}$$
$$\mathbf{Q}_3 = \{Q_1, Q_2, Q_4\}$$
$$\mathbf{Q}_4 = \{Q_3, Q_4\}$$

where tableau queries Q_1, Q_2, Q_3, and Q_4 are given below. Which of $\mathbf{Q}_1$-$\mathbf{Q}_4$ are nonredundant?

$\mathbf{Q}_1$

A	B	C	D
a_1		a_3	
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_5
b_4	b_6	b_7	b_3

$\mathbf{Q}_2$

A	B	C	D
a_1		a_3	
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_5

$\mathbf{Q}_3$

A	B	C	D
a_1		a_3	
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_3

$Q_4(A$	B	C	$D)$
a_1		a_3	
a_1	5	b_1	b_2
b_3	5	a_3	b_4

11.41 Prove the corollary to Theorem 11.9.

11.42 Prove Theorem 11.10.

11.43 Let d be the database $\{q(A\,B\,C), r(B\,C\,D), s(A\,D)\}$ where $dom(B) = dom(C)$. Find minimum equivalent conjunctive queries for the following conjunctive queries. Give a folding to justify each answer.

(a) $\{x_1(A)\, s_2(B)\, x_3(C) | \exists y_1(D)\ \exists y_2(C)$
$q(x_1\, x_2\, x_3) \wedge r(x_2\, x_2\, y_1) \wedge r(x\, y_2\, y_1) \wedge r(x_2\, x_3\, 6)\}$

(b) $\{x_1(B)\, x_2(D) | \exists y_1(A)\ \exists y_2(C)\ \exists y_3(A)$
$\exists y_4(B)\ \exists y_5(C)\ \exists y_6(A)\ \exists y_7(B)\ \exists y_8(D)$
$q(y_1\, x_1\, y_2) \wedge q(y_3\, y_4\, y_5) \wedge q(y_6\, y_7\, y_5)$
$\wedge s(y_1\, y_8) \wedge s(y_3\, y_8) \wedge s(y_6\, x_2)\}$

(c) $\{x_1(A)\, x_2(D) | \exists y_1(D)\ \exists y_2(A)$
$s(x_1\, y_1) \wedge s(y_2\, y_1) \wedge s(y_2\, x_2)\}$

11.44 For relations r and s, prove $r \bowtie s = (r \ltimes s) \bowtie s$.

11.45 For relations r and s, show that

$$r' = r \ltimes s \text{ and}$$
$$s' = s \ltimes r'$$

always join completely, and that $r \bowtie s = r' \bowtie s'$.

11.46 If SP is a semijoin program for database d, and r is a relation in d, show that $SP(r,d) \supseteq FR(r,d)$.

11.47 (a) Show that the semijoin program SP_2 in Example 11.63 is a full-reducer for d.

(b) For database d in Example 11.63, give a full-reducer with fewer statements than SP_2.

(c)*Prove: If database $d = \{r_1, r_2, \ldots, r_p\}$ has a full-reducer, it has one with $2p$-2 statements.

11.48 Let r be a relation on scheme $A\ B\ D\ E$, where the domain of each attribute is the positive integers. Assume r always satisfies the condition C_0 where

$$C_0 \text{ is } A \leq B \wedge B \leq D \wedge D \leq E.$$

Let

C_1 be $A < B \wedge B < D$
C_2 be $A = D$
C_3 be $A < D$
C_4 be $A \neq B \vee B \neq D$
C_5 be $B < E$

Which of the following fragmentation schemes are valid for r?
(a) $\{C_1, C_2\}$
(b) $\{C_2, C_3\}$
(c) $\{C_1, C_4, C_5\}$
(d) $\{C_2, C_4, C_5\}$.

11.49 Let r be as in Exercise 11.48. Let s_1, s_2, s_3, s_4 be fragments for r corresponding to the fragmentation scheme $\{C_1, C_2, C_3, C_4\}$ for the conditions C_1-C_4 given in Exercise 11.48. Give minimal sets of fragments that can be used to evaluate $\sigma_C(r)$ for the following choices of C.
(a) $A \neq B \wedge B = D$
(b) $B < D \wedge D < E$
(c) $A < 5 \wedge E < 6$

11.9 BIBLIOGRAPHY AND COMMENTS

Query modification in the INGRES database system is discussed by Stonebraker and Wong [1974], Stonebraker [1975], and Stonebraker and Rubenstein [1976].

Numerous researchers have investigated algorithms and complexity of operators in relational algebra. Gotlieb [1975] and Pecherer [1975] both showed a lower bound on the join of two relations proportional to $n \log n$, where n is the number of tuples in each relation. Blasgen and Eswaren [1977] compare various algorithms for computing a join. Stockmeyer and Wong [1979] showed that computing the intersection of two relations of k columns takes at least $(m+n)\log m + (m+n-1)k$ comparisons in the worst case, where the relations have m and n tuples, $m \leq n$. Kim [1980, 1981b] considers the best way of utilizing main memory in computing joins.

Section 11.2 on simplifications and common subexpressions follows largely from Hall [1976]. The utility of pushing selections down an expression tree was recognized by several researchers; Palermo [1974] seems to have been

first. Section 11.3 on optimizing algebraic expressions follows the treatment given in Ullman [1980]. Smith and Chang [1975] discuss several strategies for efficient relational query evaluation, such as exploiting sort orders of intermediate results and directory information. Yao [1977] gives methods for estimating the cost of different strategies in select-project-join expressions. Selinger, Astrahan, *et al.* [1979] and Chamberlin, Astrahan, *et al.* [1981] describe the queries processing strategies used by IBM's System/R relational data base system. System/R makes extensive use of database statistics to select among different retrieval methods, orders for intermediate results, and types and arrangements of joins. Kim [1981a, 1981b] covers types of nesting and optimizing sets of queries in the System/R environment.

The QUEL decomposition algorithm is due to Wong and Youssefi [1976]. The graphical representation is due to Ullman [1980].

The results on tableau query equivalence and simple query minimization are from Aho, Sagiv, and Ullman [1979a, 1979b], who also show testing tableau query equivalence is NP-complete, even in several very restricted cases. Extension joins are from Honeyman [1980a], who refined the ideas of Osborn [1979b]. Others have looked at the problem of computing a lossless join to cover a specified set of attributes: Kambayashi [1979], Lozinskii [1980], Schenk and Pinkert [1977], Tanaka and Tsuda [1977]. The material on tableau set queries is from Sagiv and Yannakakis [1979]. Chandra and Merlin [1976] presented foldings as a means to minimize conjunctive queries, although they show the problem is NP-complete.

Semijoins were introduced in Bernstein and Chiu [1981], where the answer to exercise 11.47c may also be found. We shall treat semijoins further in Chapter 13. The material on fragments is from Maier and Ullman [1981b]. Fragments are used extensively in the SDD-1 distributed database system, described by Goodman, Bernstein, *et al.* [1979] and Rothnie, Bernstein, *et al.* [1981]. Epstein, Stonebraker, and Wong [1978], Ceri and Pelagatti [1980], Paredaens [1980], and Paredaens and DeBra [1981] have examined various aspects of "horizontal" decompositions in relation databases.

Exercises 11.18a and 11.19 are from Aho, Sagiv, Szymanski, and Ullman [1979]. Exercise 11.18b is from Connors and Vianu [1981].

Minker [1975a, 1975b, 1978] studied relational query processing in the framework of inductive inference. Shopiro [1979] has designed a programming language for relational manipulation. Lozinskii [1978] and Schkolnick and Sorenson [1981] examine the trade-offs between storage redundancy and speed of query evaluation. Klug [1980a], Klug and Price [1980], and Tanaka and Kambayashi [1980] show how to calculate constraints on the value of an algebraic expression given the constraints that hold on the component relations.